www.jamesmolloy.co.uk

[**Home**](http://www.jamesmolloy.co.uk/index.html) » JamesM's kernel development tutorials

**Roll your own toy UNIX-clone OS**

This set of tutorials aims to take you through programming a simple UNIX-clone operating system for the x86 architecture. The tutorial uses C as the language of choice, with liberally mixed in bits of assembler. The aim is to talk you through the design and implementation decisions in making an operating system. The OS we make is monolithic in design (drivers are loaded through kernel-mode modules as opposed to user-mode programs), as this is simpler.

This set of tutorials is very practical in nature. The theory is given in every section, but the majority of the tutorial deals with getting dirty and implementing the abstract ideas and mechanisms discussed everywhere. It is important to note that the kernel implemented is a **teaching kernel**. I **know** that the algorithms used are not the most space efficient or optimal. They normally are chosen for their simplicity and ease of understanding.The aim of this is to get you into the correct mindset, and to give you a grounding upon which you can work. The kernel given is extensible, and good algorithms can easily be plugged in.If you have problems with the theory, there are plenty of sites that would be delighted to help you (most questions on OSDev forums are concerned with implementation - "My gets function doesn't work! help!" - A theory question is a breath of fresh air to many ;) ). Links can be found at the bottom of the page.

**Table of contents**

[1. Environment setup](http://www.jamesmolloy.co.uk/tutorial_html/1.-Environment%20setup.html)  
[2. Genesis](http://www.jamesmolloy.co.uk/tutorial_html/2.-Genesis.html)  
[3. The Screen](http://www.jamesmolloy.co.uk/tutorial_html/3.-The%20Screen.html)  
[4. The GDT and IDT](http://www.jamesmolloy.co.uk/tutorial_html/4.-The%20GDT%20and%20IDT.html)  
[5. IRQs and the PIT](http://www.jamesmolloy.co.uk/tutorial_html/5.-IRQs%20and%20the%20PIT.html)  
[6. Paging](http://www.jamesmolloy.co.uk/tutorial_html/6.-Paging.html)  
[7. The Heap](http://www.jamesmolloy.co.uk/tutorial_html/7.-The%20Heap.html)  
[8. The VFS and the initrd](http://www.jamesmolloy.co.uk/tutorial_html/8.-The%20VFS%20and%20the%20initrd.html)  
[9. Multitasking](http://www.jamesmolloy.co.uk/tutorial_html/9.-Multitasking.html)  
[10. User Mode](http://www.jamesmolloy.co.uk/tutorial_html/10.-User%20Mode.html)

**Prerequisites**

To compile and run the sample code I provide requires just GCC, ld, NASM and GNU Make. NASM is an open-source x86 assembler, and is the assembler-of-choice for many x86 OS-devs.

There is no point, however, in just compiling and running without comprehension. You must *understand* what is being coded, and as such you should have a very strong knowledge of C, especially regarding pointers. You should also know a little bit of assembly (Intel syntax is used in these tutorials), including what the EBP register is used for.

**Resources**

There are plenty of resources out there if you [know where to look](http://www.google.com/). In particular, you should find these useful:

* [RTFM!](http://www.intel.com/products/processor/manuals/index.htm) The intel manuals are a godsend
* [The osdev.org wiki](http://www.osdev.org/wiki) and [forums](http://www.osdev.org/forum).
* [Osdever.net](http://www.osdever.net/) has many good tutorials and papers, and in particular [Bran's kernel development tutorials](http://www.osdever.net/bkerndev/index.php), which some of the earlier code from this tutorial is based off. (I myself used these tutorials to get started, and the code is so good I haven't had to change it over the years
* [alt.os.development](http://groups.google.co.uk/group/alt.os.development/topics) can answer many of your non-n00b questions. N00b questions are better asked on the osdever.net forums.

www.jamesmolloy.co.uk

[**Home**](http://www.jamesmolloy.co.uk/index.html) » JamesM's kernel development tutorials

**[1. Environment setup](http://www.jamesmolloy.co.uk/tutorial_html/1.-Environment%20setup.html)**  
[2. Genesis](http://www.jamesmolloy.co.uk/tutorial_html/2.-Genesis.html)  
[3. The Screen](http://www.jamesmolloy.co.uk/tutorial_html/3.-The%20Screen.html)  
[4. The GDT and IDT](http://www.jamesmolloy.co.uk/tutorial_html/4.-The%20GDT%20and%20IDT.html)  
[5. IRQs and the PIT](http://www.jamesmolloy.co.uk/tutorial_html/5.-IRQs%20and%20the%20PIT.html)  
[6. Paging](http://www.jamesmolloy.co.uk/tutorial_html/6.-Paging.html)  
[7. The Heap](http://www.jamesmolloy.co.uk/tutorial_html/7.-The%20Heap.html)  
[8. The VFS and the initrd](http://www.jamesmolloy.co.uk/tutorial_html/8.-The%20VFS%20and%20the%20initrd.html)  
[9. Multitasking](http://www.jamesmolloy.co.uk/tutorial_html/9.-Multitasking.html)  
[10. User Mode](http://www.jamesmolloy.co.uk/tutorial_html/10.-User%20Mode.html)

# 1. 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.

## 1.1. Directory structure

My directory is laid out as follows:

tutorial  
 |  
 +-- src  
 |  
 +-- docs

All your source files will go in src, and all your documentation (you do write documentation? ;) ) should go in docs.

## 1.2. Compiling

The examples in this tutorial should compile successfully with the GNU toolchain (gcc, ld, gas, etc). The assembly examples are written in intel syntax, which is (my personal opinion) a much more human-readable syntax than the AT&T syntax that GNU AS uses. To assemble these, you will need the [Netwide Assembler](http://nasm.sourceforge.net/).

This tutorial is *not* a bootloader tutorial. We will be using [GRUB](http://www.gnu.org/software/grub) to load our kernel. To do this, we need a floppy disk image with GRUB preloaded onto it. There are [tutorials](http://www.jamesmolloy.co.uk/tutorial_html/tutorials) to do this, but, happily, I have made a standard image, which can be found [here](http://www.jamesmolloy.co.uk/downloads/floppy.img). This goes in your 'tutorial' (or whatever you named it) directory.

## 1.3. Running

There is no alternative for bare hardware as a testbed system. Unfortunately, bare hardware is pretty pants at telling you where things went wrong (but of course, you're going to write completely bug-free code first time, aren't you?). Enter [Bochs](http://www.bochs.sourceforge.net/). Bochs is an open-source x86-64 emulator. When things go completely awry, bochs will tell you, and store the processor state in a logfile, which is 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.

## 1.4. Bochs

In order to run bochs, you are going to need a bochs configuration file (bochsrc.txt). Coincidentally, a sample one is included below!

Take care with the locations of the bios files. These seem to change between installations, and if you made bochs from source it is very likely you don't even have them. Google their filenames, you can get them from the official bochs site among others.

megs: 32  
romimage: file=/usr/share/bochs/BIOS-bochs-latest, address=0xf0000  
vgaromimage: /usr/share/bochs/VGABIOS-elpin-2.40  
floppya: 1\_44=/dev/loop0, status=inserted  
boot: a  
log: bochsout.txt  
mouse: enabled=0  
clock: sync=realtime  
cpu: ips=500000

This will make bochs emulate a 32MB machine with a clock speed similar to a 350MHz PentiumII. The instructions per second can be cranked up - I prefer a slower emulation speed, simply so I can see what is going on if lots of text is being scrolled.

## 1.5. Useful scripts

We are going to be doing several things very often - making (compiling and linking) our project, and transferring the resulting kernel binary to our floppy disk image.

### 1.5.1. Makefile

# Makefile for JamesM's kernel tutorials.  
# The C and C++ rules are already setup by default.  
# The only one that needs changing is the assembler   
# rule, as we use nasm instead of GNU as.  
  
SOURCES=boot.o  
  
CFLAGS=  
LDFLAGS=-Tlink.ld  
ASFLAGS=-felf  
  
all: $(SOURCES) link   
  
clean:  
 »  -rm \*.o kernel  
  
link:  
 »  ld $(LDFLAGS) -o kernel $(SOURCES)  
  
.s.o:  
 »  nasm $(ASFLAGS) $<

This Makefile will compile every file in SOURCES, then link them together into one ELF binary, 'kernel'. It uses a linker script, 'link.ld' to do this:

### 1.5.2. Link.ld

/\* Link.ld -- Linker script for the kernel - ensure everything goes in the \*/  
/\*            Correct place.  \*/  
/\*            Original file taken from Bran's Kernel Development \*/  
/\*            tutorials: http://www.osdever.net/bkerndev/index.php. \*/  
  
ENTRY(start)  
SECTIONS  
{  
  .text 0x100000 :  
  {  
    code = .; \_code = .; \_\_code = .;  
    \*(.text)  
    . = ALIGN(4096);  
  }  
  
  .data :  
  {  
     data = .; \_data = .; \_\_data = .;  
     \*(.data)  
     \*(.rodata)  
     . = ALIGN(4096);  
  }  
  
  .bss :  
  {  
    bss = .; \_bss = .; \_\_bss = .;  
    \*(.bss)  
    . = ALIGN(4096);  
  }  
  
  end = .; \_end = .; \_\_end = .;  
}

This script tells LD how to set up our kernel image. Firstly it tells LD that the start location of our binary should be the symbol 'start'. It then tells LD that the .text section (that's where all your code goes) should be first, and should start at 0x100000 (1MB). The .data (initialised static data) and the .bss (uninitialised static data) should be next, and each should be page-aligned (ALIGN(4096)). Linux GCC also adds in an extra data section: .rodata. This is for read-only initialised data, such as constants. For simplicity we simply bundle this in with the .data section.

### 1.5.3. update\_image.sh

A nice little script that will poke your new kernel binary into the floppy image file (This assumes you have made a directory /mnt). Note: you will need /sbin in your $PATH to use losetup.

#!/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

### 1.5.4. run\_bochs.sh

This script will setup a loopback device, run bochs on it, then disconnect it.

#!/bin/bash  
  
# run\_bochs.sh  
# mounts the correct loopback device, runs bochs, then unmounts.  
  
sudo /sbin/losetup /dev/loop0 floppy.img  
sudo bochs -f bochsrc.txt  
sudo /sbin/losetup -d /dev/loop0

***Copyright James Molloy 2008 - james<at>jamesmolloy.co.uk***

www.jamesmolloy.co.uk

[**Home**](http://www.jamesmolloy.co.uk/index.html) » JamesM's kernel development tutorials

[1. Environment setup](http://www.jamesmolloy.co.uk/tutorial_html/1.-Environment%20setup.html)  
**[2. Genesis](http://www.jamesmolloy.co.uk/tutorial_html/2.-Genesis.html)**  
[3. The Screen](http://www.jamesmolloy.co.uk/tutorial_html/3.-The%20Screen.html)  
[4. The GDT and IDT](http://www.jamesmolloy.co.uk/tutorial_html/4.-The%20GDT%20and%20IDT.html)  
[5. IRQs and the PIT](http://www.jamesmolloy.co.uk/tutorial_html/5.-IRQs%20and%20the%20PIT.html)  
[6. Paging](http://www.jamesmolloy.co.uk/tutorial_html/6.-Paging.html)  
[7. The Heap](http://www.jamesmolloy.co.uk/tutorial_html/7.-The%20Heap.html)  
[8. The VFS and the initrd](http://www.jamesmolloy.co.uk/tutorial_html/8.-The%20VFS%20and%20the%20initrd.html)  
[9. Multitasking](http://www.jamesmolloy.co.uk/tutorial_html/9.-Multitasking.html)  
[10. User Mode](http://www.jamesmolloy.co.uk/tutorial_html/10.-User%20Mode.html)

# 2. 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 we just *must* use assembly for. One of those things 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.  
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.  
  
[GLOBAL mboot]                  ; Make 'mboot' accessible from C.  
[EXTERN code]                   ; Start of the '.text' section.  
[EXTERN bss]                    ; Start of the .bss section.  
[EXTERN end]                    ; End of the last loadable section.  
  
mboot:  
  dd  MBOOT\_HEADER\_MAGIC        ; GRUB will search for this value on each  
                                ; 4-byte boundary in 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                     ; Location of this descriptor  
  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]                  ; Kernel entry point.  
[EXTERN main]                   ; This is the entry point of our C code  
  
start:  
  push    ebx                   ; Load multiboot header location  
  
  ; Execute the kernel:  
  cli                         ; Disable interrupts.  
  call main                   ; call our main() function.  
  jmp $                       ; Enter an infinite loop, to stop the processor  
                              ; executing whatever rubbish is in the memory  
                              ; after our kernel!

## 2.2. Understanding the boot code

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

push ebx  
cli  
call main  
jmp $

The rest of it is all to do 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 4KB 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  
dd MBOOT\_HEADER\_FLAGS  
dd MBOOT\_CHECKSUM  
dd mboot  
dd code  
dd bss  
dd end  
dd start

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.

**mboot**  
    The address of the structure that we are currently writing. GRUB uses  
this to tell if we are expecting to be relocated.

**code,bss,end,start**  
    These symbols are all defined by the linker. We use them to tell GRUB  
where the different sections of our kernel can be located.

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.

### 2.2.2. Back to the code again...

So, immediately on bootup, the asm snippet tells the CPU to push the contents of EBX onto the stack (remember that EBX now contains a pointer to the multiboot information structure), disable interrupts (CLI), call our 'main' C function (which we haven't defined yet), then enter an infinite loop.

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

## 2.3. Adding some C code

Interfacing C code and assembly is dead easy. You just have 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 returned in EAX.

...so the function call:

d = func(a, b, c);

Becomes:

push [c]  
push [b]  
push [a]  
call func  
mov [d], eax

See? nothing to it! So, you can see that in our asm snippet above, that 'push ebx' is actually passing the value of ebx as a parameter to the function main().

### 2.3.1. The C code

// main.c -- Defines the C-code kernel entry point, calls initialisation routines.  
// Made for JamesM's tutorials   
  
int main(struct multiboot \*mboot\_ptr)  
{  
  // All our initialisation calls will go in here.  
  return 0xDEADBABA;  
}

Here's our first incarnation of the main() function. As you can see, we've made it take one parameter - a pointer to a multiboot struct. We'll define that later (we don't actually need to define it for the code to compile!).

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

## 2.4. Compiling, linking and running!

Now that we've added a new file to our project, we have to add it to the makefile also. Edit these lines:

SOURCES=boot.o  
CFLAGS=

To become:

SOURCES=boot.o main.o  
CFLAGS=-nostdlib -nostdinc -fno-builtin -fno-stack-protector

We must stop GCC trying to link your linux C library with our kernel - it won't work at all (yet). That's what those CFLAGS are for.

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

cd src  
make clean  # Ignore any errors here.  
make  
cd ..  
./update\_image.sh  
./run\_bochs.sh  # This may ask your for your root password.

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=deadbaba  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

  
Bochs booting your kernel

Notice what the value of EAX is? 0xDEADBABA - 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](http://www.jamesmolloy.co.uk/downloads/genesis.tar)

***Copyright James Molloy 2008 - james<at>jamesmolloy.co.uk***

www.jamesmolloy.co.uk

[**Home**](http://www.jamesmolloy.co.uk/index.html) » JamesM's kernel development tutorials

[1. Environment setup](http://www.jamesmolloy.co.uk/tutorial_html/1.-Environment%20setup.html)  
[2. Genesis](http://www.jamesmolloy.co.uk/tutorial_html/2.-Genesis.html)  
**[3. The Screen](http://www.jamesmolloy.co.uk/tutorial_html/3.-The%20Screen.html)**  
[4. The GDT and IDT](http://www.jamesmolloy.co.uk/tutorial_html/4.-The%20GDT%20and%20IDT.html)  
[5. IRQs and the PIT](http://www.jamesmolloy.co.uk/tutorial_html/5.-IRQs%20and%20the%20PIT.html)  
[6. Paging](http://www.jamesmolloy.co.uk/tutorial_html/6.-Paging.html)  
[7. The Heap](http://www.jamesmolloy.co.uk/tutorial_html/7.-The%20Heap.html)  
[8. The VFS and the initrd](http://www.jamesmolloy.co.uk/tutorial_html/8.-The%20VFS%20and%20the%20initrd.html)  
[9. Multitasking](http://www.jamesmolloy.co.uk/tutorial_html/9.-Multitasking.html)  
[10. User Mode](http://www.jamesmolloy.co.uk/tutorial_html/10.-User%20Mode.html)

# 3. 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 available to it 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 VESA (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.

http://www.jamesmolloy.co.uk/images/the_screen_word_format.png  
Word format

In 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 to the right.

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

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 magneta, 14: light brown, 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 it's cursor position (the flashy underbar thing that tells you where your next character will go).

## 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, and some typedefs that will make it easier for us to write portable code. They are also the ideal place to put functions such as memcpy/memset etc. I have left them for you to implement! :)

// common.h -- Defines typedefs and some global functions.  
// From JamesM's kernel development tutorials.  
  
#ifndef COMMON\_H  
#define COMMON\_H  
  
// Some nice typedefs, to standardise sizes across platforms.  
// These typedefs are written for 32-bit X86.  
typedef unsigned int   u32int;  
typedef          int   s32int;  
typedef unsigned short u16int;  
typedef          short s16int;  
typedef unsigned char  u8int;  
typedef          char  s8int;  
  
void outb(u16int port, u8int value);  
u8int inb(u16int port);  
u16int inw(u16int port);  
  
#endif

// 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(u16int port, u8int value)  
{  
    asm volatile ("outb %1, %0" : : "dN" (port), "a" (value));  
}  
  
u8int inb(u16int port)  
{  
   u8int ret;  
   asm volatile("inb %1, %0" : "=a" (ret) : "dN" (port));  
   return ret;  
}  
  
u16int inw(u16int port)  
{  
   u16int 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.h  
// 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);  
  
#endif // MONITOR\_H

#### 3.2.2.1. 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.

// Updates the hardware cursor.  
static void move\_cursor()  
{  
   // The screen is 80 characters wide...  
   u16int 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.  
}

#### 3.2.2.2. 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:

// Scrolls the text on the screen up by one line.  
static void scroll()  
{  
  
   // Get a space character with the default colour attributes.  
   u8int attributeByte = (0 /\*black\*/ << 4) | (15 /\*white\*/ & 0x0F);  
   u16int 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;  
   }  
}

#### 3.2.2.3. 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.

// Writes a single character out to the screen.  
void monitor\_put(char c)  
{  
   // The background colour is black (0), the foreground is white (15).  
   u8int backColour = 0;  
   u8int foreColour = 15;  
  
   // The attribute byte is made up of two nibbles - the lower being the  
   // foreground colour, and the upper the background colour.  
   u8int  attributeByte = (backColour << 4) | (foreColour & 0x0F);  
   // The attribute byte is the top 8 bits of the word we have to send to the  
   // VGA board.  
   u16int attribute = attributeByte << 8;  
   u16int \*location;  
  
   // 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 >= ' ')  
   {  
       location = video\_memory + (cursor\_y\*80 + cursor\_x);  
       \*location = 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:

location = video\_memory + (cursor\_y\*80 + cursor\_x);  
\*location = c | attribute;

* Set 'location' to point to the linear address of the word corresponding to the current cursor position (see equation above).
* Set the value at 'location' to be the logical-OR of the character and 'attribute'. Remember that we shifted 'attribute' left 8 bits above, so actually we're just setting 'c' as the lower byte of 'attribute'.

#### 3.2.2.4. Clearing the screen

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

// Clears the screen, by copying lots of spaces to the framebuffer.  
void monitor\_clear()  
{  
   // Make an attribute byte for the default colours  
   u8int attributeByte = (0 /\*black\*/ << 4) | (15 /\*white\*/ & 0x0F);  
   u16int 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();  
}

#### 3.2.2.5. Writing a string

// Outputs a null-terminated ASCII string to the monitor.  
void monitor\_write(char \*c)  
{  
   int i = 0;  
   while (c[i])  
   {  
       monitor\_put(c[i++]);  
   }  
}

## 3.3. 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, eh?

## 3.4. 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(u32int n)  
{  
   // TODO: implement this yourself!  
}  
  
void monitor\_write\_dec(u32int 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!

  
Hello, world!

You could also have a scout at the linux0.1 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 hell of a lot easier when it comes to debugging.

Source code for this tutorial is available [here](http://www.jamesmolloy.co.uk/downloads/the_screen.tar.gz)

***Copyright James Molloy 2008 - james<at>jamesmolloy.co.uk***

www.jamesmolloy.co.uk

[**Home**](http://www.jamesmolloy.co.uk/index.html) » JamesM's kernel development tutorials

[1. Environment setup](http://www.jamesmolloy.co.uk/tutorial_html/1.-Environment%20setup.html)  
[2. Genesis](http://www.jamesmolloy.co.uk/tutorial_html/2.-Genesis.html)  
**[3. The Screen](http://www.jamesmolloy.co.uk/tutorial_html/3.-The%20Screen.html)**  
[4. The GDT and IDT](http://www.jamesmolloy.co.uk/tutorial_html/4.-The%20GDT%20and%20IDT.html)  
[5. IRQs and the PIT](http://www.jamesmolloy.co.uk/tutorial_html/5.-IRQs%20and%20the%20PIT.html)  
[6. Paging](http://www.jamesmolloy.co.uk/tutorial_html/6.-Paging.html)  
[7. The Heap](http://www.jamesmolloy.co.uk/tutorial_html/7.-The%20Heap.html)  
[8. The VFS and the initrd](http://www.jamesmolloy.co.uk/tutorial_html/8.-The%20VFS%20and%20the%20initrd.html)  
[9. Multitasking](http://www.jamesmolloy.co.uk/tutorial_html/9.-Multitasking.html)  
[10. User Mode](http://www.jamesmolloy.co.uk/tutorial_html/10.-User%20Mode.html)

# 3. 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 available to it 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 VESA (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.

http://www.jamesmolloy.co.uk/images/the_screen_word_format.png  
Word format

In 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 to the right.

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

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 magneta, 14: light brown, 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 it's cursor position (the flashy underbar thing that tells you where your next character will go).

## 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, and some typedefs that will make it easier for us to write portable code. They are also the ideal place to put functions such as memcpy/memset etc. I have left them for you to implement! :)

// common.h -- Defines typedefs and some global functions.  
// From JamesM's kernel development tutorials.  
  
#ifndef COMMON\_H  
#define COMMON\_H  
  
// Some nice typedefs, to standardise sizes across platforms.  
// These typedefs are written for 32-bit X86.  
typedef unsigned int   u32int;  
typedef          int   s32int;  
typedef unsigned short u16int;  
typedef          short s16int;  
typedef unsigned char  u8int;  
typedef          char  s8int;  
  
void outb(u16int port, u8int value);  
u8int inb(u16int port);  
u16int inw(u16int port);  
  
#endif

// 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(u16int port, u8int value)  
{  
    asm volatile ("outb %1, %0" : : "dN" (port), "a" (value));  
}  
  
u8int inb(u16int port)  
{  
   u8int ret;  
   asm volatile("inb %1, %0" : "=a" (ret) : "dN" (port));  
   return ret;  
}  
  
u16int inw(u16int port)  
{  
   u16int 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.h  
// 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);  
  
#endif // MONITOR\_H

#### 3.2.2.1. 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.

// Updates the hardware cursor.  
static void move\_cursor()  
{  
   // The screen is 80 characters wide...  
   u16int 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.  
}

#### 3.2.2.2. 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:

// Scrolls the text on the screen up by one line.  
static void scroll()  
{  
  
   // Get a space character with the default colour attributes.  
   u8int attributeByte = (0 /\*black\*/ << 4) | (15 /\*white\*/ & 0x0F);  
   u16int 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;  
   }  
}

#### 3.2.2.3. 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.

// Writes a single character out to the screen.  
void monitor\_put(char c)  
{  
   // The background colour is black (0), the foreground is white (15).  
   u8int backColour = 0;  
   u8int foreColour = 15;  
  
   // The attribute byte is made up of two nibbles - the lower being the  
   // foreground colour, and the upper the background colour.  
   u8int  attributeByte = (backColour << 4) | (foreColour & 0x0F);  
   // The attribute byte is the top 8 bits of the word we have to send to the  
   // VGA board.  
   u16int attribute = attributeByte << 8;  
   u16int \*location;  
  
   // 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 >= ' ')  
   {  
       location = video\_memory + (cursor\_y\*80 + cursor\_x);  
       \*location = 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:

location = video\_memory + (cursor\_y\*80 + cursor\_x);  
\*location = c | attribute;

* Set 'location' to point to the linear address of the word corresponding to the current cursor position (see equation above).
* Set the value at 'location' to be the logical-OR of the character and 'attribute'. Remember that we shifted 'attribute' left 8 bits above, so actually we're just setting 'c' as the lower byte of 'attribute'.

#### 3.2.2.4. Clearing the screen

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

// Clears the screen, by copying lots of spaces to the framebuffer.  
void monitor\_clear()  
{  
   // Make an attribute byte for the default colours  
   u8int attributeByte = (0 /\*black\*/ << 4) | (15 /\*white\*/ & 0x0F);  
   u16int 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();  
}

#### 3.2.2.5. Writing a string

// Outputs a null-terminated ASCII string to the monitor.  
void monitor\_write(char \*c)  
{  
   int i = 0;  
   while (c[i])  
   {  
       monitor\_put(c[i++]);  
   }  
}

## 3.3. 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, eh?

## 3.4. 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(u32int n)  
{  
   // TODO: implement this yourself!  
}  
  
void monitor\_write\_dec(u32int 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!

  
Hello, world!

You could also have a scout at the linux0.1 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 hell of a lot easier when it comes to debugging.

Source code for this tutorial is available [here](http://www.jamesmolloy.co.uk/downloads/the_screen.tar.gz)

***Copyright James Molloy 2008 - james<at>jamesmolloy.co.uk***

www.jamesmolloy.co.uk

[**Home**](http://www.jamesmolloy.co.uk/index.html) » JamesM's kernel development tutorials

[1. Environment setup](http://www.jamesmolloy.co.uk/tutorial_html/1.-Environment%20setup.html)  
[2. Genesis](http://www.jamesmolloy.co.uk/tutorial_html/2.-Genesis.html)  
[3. The Screen](http://www.jamesmolloy.co.uk/tutorial_html/3.-The%20Screen.html)  
[**4. The GDT and IDT**](http://www.jamesmolloy.co.uk/tutorial_html/4.-The%20GDT%20and%20IDT.html)  
[5. IRQs and the PIT](http://www.jamesmolloy.co.uk/tutorial_html/5.-IRQs%20and%20the%20PIT.html)  
[6. Paging](http://www.jamesmolloy.co.uk/tutorial_html/6.-Paging.html)  
[7. The Heap](http://www.jamesmolloy.co.uk/tutorial_html/7.-The%20Heap.html)  
[8. The VFS and the initrd](http://www.jamesmolloy.co.uk/tutorial_html/8.-The%20VFS%20and%20the%20initrd.html)  
[9. Multitasking](http://www.jamesmolloy.co.uk/tutorial_html/9.-Multitasking.html)  
[10. User Mode](http://www.jamesmolloy.co.uk/tutorial_html/10.-User%20Mode.html)

# 4. 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 (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.

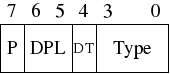
Segmentation is, however, totally in-built into the x86 architecture. It's impossible to get around it. So here we're going to show you how to set up your own Global Descriptor Table - a list of segment descriptors.

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, 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.[[3.2.4]](http://www.jamesmolloy.co.uk/tutorial_html/references.html#3.2.4)

## 4.2. The Global Descriptor Table (practical)

  
Access byte format

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

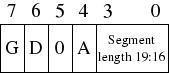
One thing I forgot to mention is that GRUB sets a GDT up for you. The problem is that you don't know where that GDT is, or what's in it. So you could accidentally overwrite it, then your computer would triple-fault and reset. Not clever.

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'.

### 4.2.1. descriptor\_tables.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.  
struct gdt\_entry\_struct  
{  
   u16int limit\_low;           // The lower 16 bits of the limit.  
   u16int base\_low;            // The lower 16 bits of the base.  
   u8int  base\_middle;         // The next 8 bits of the base.  
   u8int  access;              // Access flags, determine what ring this segment can be used in.  
   u8int  granularity;  
   u8int  base\_high;           // The last 8 bits of the base.  
} \_\_attribute\_\_((packed));  
typedef struct gdt\_entry\_struct gdt\_entry\_t;

  
Granularity byte format

Most of those fields should be self-explanatory. The format of the access byte is given on the right above, and the format of the granularity byte is here on the right.

**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 = 1kbyte)  
**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:

struct gdt\_ptr\_struct  
{  
   u16int limit;               // The upper 16 bits of all selector limits.  
   u32int base;                // The address of the first gdt\_entry\_t struct.  
}  
 \_\_attribute\_\_((packed));  
typedef struct gdt\_ptr\_struct 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, descriptor\_tables.h, along with a prototype.

// Initialisation function is publicly accessible.  
void init\_descriptor\_tables();

### 4.2.2. descriptor\_tables.c

In descriptor\_tables.c, we have a few declarations:

//  
// descriptor\_tables.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 "descriptor\_tables.h"  
  
// Lets us access our ASM functions from our C code.  
extern void gdt\_flush(u32int);  
  
// Internal function prototypes.  
static void init\_gdt();  
static void gdt\_set\_gate(s32int,u32int,u32int,u8int,u8int);  
  
gdt\_entry\_t gdt\_entries[5];  
gdt\_ptr\_t   gdt\_ptr;  
idt\_entry\_t idt\_entries[256];  
idt\_ptr\_t   idt\_ptr;

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

// Initialisation routine - zeroes all the interrupt service routines,  
// initialises the GDT and IDT.  
void init\_descriptor\_tables()  
{  
   // Initialise the global descriptor table.  
   init\_gdt();  
}  
  
static void init\_gdt()  
{  
   gdt\_ptr.limit = (sizeof(gdt\_entry\_t) \* 5) - 1;  
   gdt\_ptr.base  = (u32int)&gdt\_entries;  
  
   gdt\_set\_gate(0, 0, 0, 0, 0);                // Null segment  
   gdt\_set\_gate(1, 0, 0xFFFFFFFF, 0x9A, 0xCF); // Code segment  
   gdt\_set\_gate(2, 0, 0xFFFFFFFF, 0x92, 0xCF); // Data segment  
   gdt\_set\_gate(3, 0, 0xFFFFFFFF, 0xFA, 0xCF); // User mode code segment  
   gdt\_set\_gate(4, 0, 0xFFFFFFFF, 0xF2, 0xCF); // User mode data segment  
  
   gdt\_flush((u32int)&gdt\_ptr);  
}  
  
// Set the value of one GDT entry.  
static void gdt\_set\_gate(s32int num, u32int base, u32int limit, u8int access, u8int 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 \* 5 - we have 5 entries. Why 5? well, we have a code and data segment descriptor for the kernel, code and data segment descriptors for user mode, and a null entry. This *must* be present, or bad things will happen.

gdt\_init then sets up the 5 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 4 segment descriptors is the access byte - 0x9A, 0x92, 0xFA, 0xF2. 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 and DPL fields. DPL is the descriptor privilege level - 3 for user code and 0 for kernel code. 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.

[GLOBAL 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

This function takes the first parameter passed to it (in esp+4), loads the value is 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,fd,gs,ss. To change the code segment is slightly different; we must do a far jump. This changes the CS implicitly.

## 4.3. The Interrupt Descriptor Table (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 do 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.

### 4.3.1. 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.

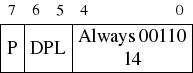
* 0 - Division by zero exception
* 1 - Debug exception
* 2 - Non maskable interrupt
* 3 - Breakpoint exception
* 4 - 'Into detected overflow'
* 5 - Out of bounds exception
* 6 - Invalid opcode exception
* 7 - No coprocessor exception
* 8 - Double fault *(pushes an error code)*
* 9 - Coprocessor segment 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

## 4.4. The Interrupt Descriptor Table (practice)

### 4.4.1. descriptor\_tables.h

We should add some definitions to descriptor\_tables.h:

// A struct describing an interrupt gate.  
struct idt\_entry\_struct  
{  
   u16int base\_lo;             // The lower 16 bits of the address to jump to when this interrupt fires.  
   u16int sel;                 // Kernel segment selector.  
   u8int  always0;             // This must always be zero.  
   u8int  flags;               // More flags. See documentation.  
   u16int base\_hi;             // The upper 16 bits of the address to jump to.  
} \_\_attribute\_\_((packed));  
typedef struct idt\_entry\_struct idt\_entry\_t;  
  
// A struct describing a pointer to an array of interrupt handlers.  
// This is in a format suitable for giving to 'lidt'.  
struct idt\_ptr\_struct  
{  
   u16int limit;  
   u32int base;                // The address of the first element in our idt\_entry\_t array.  
} \_\_attribute\_\_((packed));  
typedef struct idt\_ptr\_struct idt\_ptr\_t;  
  
// These extern directives let us access the addresses of our ASM ISR handlers.  
extern void isr0 ();  
...  
extern void isr31();

  
Flags byte format

See? Very similar to the GDT entry and ptr structs. The flags field format is shown on the right. The lower 5-bits should be constant at 0b0110 - 14 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.

### 4.4.2. descriptor\_tables.c

We need to modify this file to add our new code.

...  
extern void idt\_flush(u32int);  
...  
static void init\_idt();  
static void idt\_set\_gate(u8int,u32int,u16int,u8int);  
...  
idt\_entry\_t idt\_entries[256];  
idt\_ptr\_t   idt\_ptr;  
...  
void init\_descriptor\_tables()  
{  
  init\_gdt();  
  init\_idt();  
}  
...  
static void init\_idt()  
{  
   idt\_ptr.limit = sizeof(idt\_entry\_t) \* 256 -1;  
   idt\_ptr.base  = (u32int)&idt\_entries;  
  
   memset(&idt\_entries, 0, sizeof(idt\_entry\_t)\*256);  
  
   idt\_set\_gate( 0, (u32int)isr0 , 0x08, 0x8E);  
   idt\_set\_gate( 1, (u32int)isr1 , 0x08, 0x8E);  
   ...  
   idt\_set\_gate(31, (u32int)isr32, 0x08, 0x8E);  
  
   idt\_flush((u32int)&idt\_ptr);  
}  
  
static void idt\_set\_gate(u8int num, u32int base, u16int sel, u8int 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 \*/;  
}

This gets added to gdt.s also:

[GLOBAL 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

### 4.4.3. interrupt.s

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) to 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 is pretty crap, so 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.

That's all gravy, but 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  ; define a macro, taking one parameter  
  [GLOBAL isr%1]        ; %1 accesses the first parameter.  
  isr%1:  
    cli  
    push byte 0  
    push byte %1  
    jmp isr\_common\_stub  
%endmacro  
  
%macro ISR\_ERRCODE 1  
  [GLOBAL isr%1]  
  isr%1:  
    cli  
    push byte %1  
    jmp isr\_common\_stub  
%endmacro

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.

; In isr.c  
[EXTERN isr\_handler]  
  
; 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  
  
   call isr\_handler  
  
   pop eax        ; reload the original data segment descriptor  
   mov ds, ax  
   mov es, ax  
   mov fs, ax  
   mov gs, ax  
  
   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

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 - *isr\_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.

### 4.4.4. isr.c

//  
// isr.c -- High level interrupt service routines and interrupt request handlers.  
// Part of this code is modified from Bran's kernel development tutorials.  
// Rewritten for JamesM's kernel development tutorials.  
//  
  
#include "common.h"  
#include "isr.h"  
#include "monitor.h"  
  
// This gets called from our ASM interrupt handler stub.  
void isr\_handler(registers\_t regs)  
{  
   monitor\_write("recieved interrupt: ");  
   monitor\_write\_dec(regs.int\_no);  
   monitor\_put('\n');  
}

Nothing much to explain here - The interrupt handler prints a message out to the screen, along with the interrupt number it handled. It uses a structure registers\_t, which is a representation of all the registers we pushed, and is defined in isr.h:

### 4.4.5. isr.h

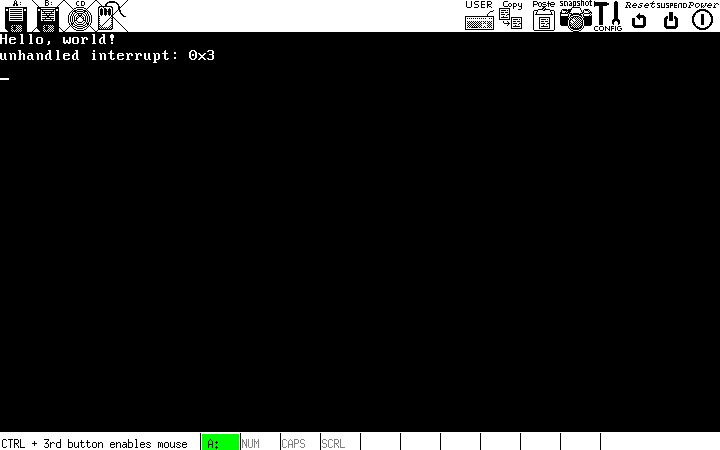
//  
// isr.h -- Interface and structures for high level interrupt service routines.  
// Part of this code is modified from Bran's kernel development tutorials.  
// Rewritten for JamesM's kernel development tutorials.  
//  
  
#include "common.h"  
  
typedef struct registers  
{  
   u32int ds;                  // Data segment selector  
   u32int edi, esi, ebp, esp, ebx, edx, ecx, eax; // Pushed by pusha.  
   u32int int\_no, err\_code;    // Interrupt number and error code (if applicable)  
   u32int eip, cs, eflags, useresp, ss; // Pushed by the processor automatically.  
} registers\_t;

### 4.4.6. Testing it out

Wow, that was a seriously long chapter! Don't get put off, they're not all this length. We just have to do an awful lot here to get anything out of it.

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

asm volatile ("int $0x3");  
asm volatile ("int $0x4");

  
What it should look like

This causes two software interrupts: 3 and 4. You should see the messages printed out just like the screenshot on the right.

Congrats! 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](http://www.jamesmolloy.co.uk/downloads/gdt_idt.tar.gz).

***Copyright James Molloy 2008 - james<at>jamesmolloy.co.uk***

www.jamesmolloy.co.uk

[**Home**](http://www.jamesmolloy.co.uk/index.html) » JamesM's kernel development tutorials

[1. Environment setup](http://www.jamesmolloy.co.uk/tutorial_html/1.-Environment%20setup.html)  
[2. Genesis](http://www.jamesmolloy.co.uk/tutorial_html/2.-Genesis.html)  
[3. The Screen](http://www.jamesmolloy.co.uk/tutorial_html/3.-The%20Screen.html)  
[4. The GDT and IDT](http://www.jamesmolloy.co.uk/tutorial_html/4.-The%20GDT%20and%20IDT.html)  
**[5. IRQs and the PIT](http://www.jamesmolloy.co.uk/tutorial_html/5.-IRQs%20and%20the%20PIT.html)**  
[6. Paging](http://www.jamesmolloy.co.uk/tutorial_html/6.-Paging.html)  
[7. The Heap](http://www.jamesmolloy.co.uk/tutorial_html/7.-The%20Heap.html)  
[8. The VFS and the initrd](http://www.jamesmolloy.co.uk/tutorial_html/8.-The%20VFS%20and%20the%20initrd.html)  
[9. Multitasking](http://www.jamesmolloy.co.uk/tutorial_html/9.-Multitasking.html)  
[10. User Mode](http://www.jamesmolloy.co.uk/tutorial_html/10.-User%20Mode.html)

# 5. IRQs and the PIT

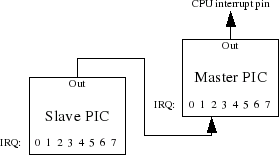
In this chapter we're going to be learning about interrupt requests (IRQs) and the programmable interval timer (PIT).

## 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. 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 slave's output is connected to IRQ2 of the master.

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](http://www.jamesmolloy.co.uk/tutorial_html/4.)). 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](http://www.osdev.org/wiki/PIC).

static 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, (u32int)irq0, 0x08, 0x8E);  
  ...  
  idt\_set\_gate(47, (u32int)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 interrupt.s. Also, though, we need a new macro in interrupt.s - an IRQ stub will have 2 numbers associated with it - it's IRQ number (0-15) and it's 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  
  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*.

; In isr.c  
[EXTERN irq\_handler]  
  
; 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  
  
   call irq\_handler  
  
   pop ebx        ; reload the original data segment descriptor  
   mov ds, bx  
   mov es, bx  
   mov fs, bx  
   mov gs, 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 isr.c):

// This gets called from our ASM interrupt handler stub.  
void irq\_handler(registers\_t 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);  
  
   if (interrupt\_handlers[regs.int\_no] != 0)  
   {  
       isr\_t handler = interrupt\_handlers[regs.int\_no];  
       handler(regs);  
   }  
}

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.

You may also notice that I have added a small custom handler mechanism, allowing you to register custom interrupt handlers. This can be very useful as an abstraction technique, and will neaten up our code nicely.

Some other declarations are needed:

### 5.2.1. isr.h

// A few defines to make life a little easier  
#define IRQ0 32  
...  
#define IRQ15 47  
  
// Enables registration of callbacks for interrupts or IRQs.  
// For IRQs, to ease confusion, use the #defines above as the  
// first parameter.  
typedef void (\*isr\_t)(registers\_t);  
void register\_interrupt\_handler(u8int n, isr\_t handler);

### 5.2.2. isr.c

isr\_t interrupt\_handlers[256];  
  
void register\_interrupt\_handler(u8int n, isr\_t handler)  
{  
  interrupt\_handlers[n] = handler;  
}

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](http://en.wikipedia.org/wiki/Frequency_divider), to modulate the final output frequency. It has 3 channels, each with it's own frequency divider.

* Channel 0 is the most useful. It's output is connected to IRQ0.
* Channel 1 is very un-useful and on modern hardware is no longer implemented. It used to control refresh rates for [DRAM](http://en.wikipedia.org/wiki/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.

## 5.4. 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(u32int frequency);  
  
#endif

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

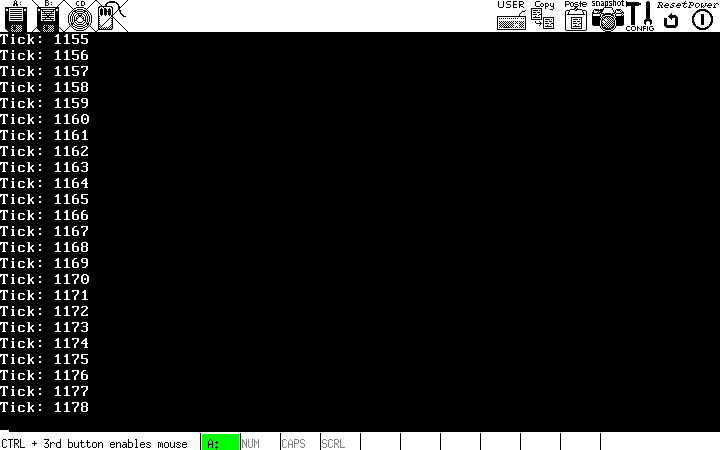
// timer.c -- Initialises the PIT, and handles clock updates.  
// Written for JamesM's kernel development tutorials.  
  
#include "timer.h"  
#include "isr.h"  
#include "monitor.h"  
  
u32int tick = 0;  
  
static void timer\_callback(registers\_t regs)  
{  
   tick++;  
   monitor\_write("Tick: ");  
   monitor\_write\_dec(tick);  
   monitor\_write("\n");  
}  
  
void init\_timer(u32int 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.  
   u32int divisor = 1193180 / frequency;  
  
   // Send the command byte.  
   outb(0x43, 0x36);  
  
   // Divisor has to be sent byte-wise, so split here into upper/lower bytes.  
   u8int l = (u8int)(divisor & 0xFF);  
   u8int h = (u8int)( (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 edit our Makefile, add one line to main.c

init\_timer(50); // Initialise timer to 50Hz

  
A clock!

,compile, and run! 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](http://www.jamesmolloy.co.uk/downloads/irqs_and_the_pit.tar.gz).

***Copyright James Molloy 2008 - james<at>jamesmolloy.co.uk***

www.jamesmolloy.co.uk

[**Home**](http://www.jamesmolloy.co.uk/index.html) » JamesM's kernel development tutorials

[1. Environment setup](http://www.jamesmolloy.co.uk/tutorial_html/1.-Environment%20setup.html)  
[2. Genesis](http://www.jamesmolloy.co.uk/tutorial_html/2.-Genesis.html)  
[3. The Screen](http://www.jamesmolloy.co.uk/tutorial_html/3.-The%20Screen.html)  
[4. The GDT and IDT](http://www.jamesmolloy.co.uk/tutorial_html/4.-The%20GDT%20and%20IDT.html)  
[5. IRQs and the PIT](http://www.jamesmolloy.co.uk/tutorial_html/5.-IRQs%20and%20the%20PIT.html)  
**[6. Paging](http://www.jamesmolloy.co.uk/tutorial_html/6.-Paging.html)**  
[7. The Heap](http://www.jamesmolloy.co.uk/tutorial_html/7.-The%20Heap.html)  
[8. The VFS and the initrd](http://www.jamesmolloy.co.uk/tutorial_html/8.-The%20VFS%20and%20the%20initrd.html)  
[9. Multitasking](http://www.jamesmolloy.co.uk/tutorial_html/9.-Multitasking.html)  
[10. User Mode](http://www.jamesmolloy.co.uk/tutorial_html/10.-User%20Mode.html)

# 6. Paging

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).

## 6.1. Virtual memory (theory)

*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.

jamesmol@aubergine:~/test> 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.

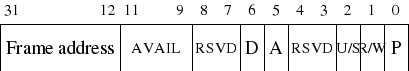
Virtual memory of this type is wholly dependent on hardware support. It cannot be emulated by software. Luckily, the x86 has just such a thing. It's called the MMU (memory management unit), and it handles all memory mappings due to segmentation and paging, forming a layer between the CPU and memory (actually,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

## 6.2. Paging as a concretion of virtual memory

Virtual memory is an abstract principle. As such it requires *concretion* through some system/algorithm. Both segmentation (see [chapter 3](http://www.jamesmolloy.co.uk/tutorial_html/3.)) and paging are valid methods for implementing virtual memory. As mentioned in [chapter 3](http://www.jamesmolloy.co.uk/tutorial_html/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.

### 6.2.1. Page entries

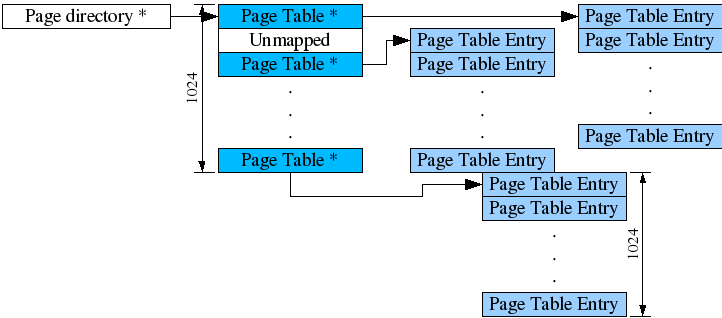
  
Page table entry format

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.

### 6.2.2. Page directories/tables

  
2-tier layout

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. Perhaps, perhaps not - but it's true.

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.

### 6.2.3. Enabling 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.

## 6.3. 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](http://www.jamesmolloy.co.uk/tutorial_html/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 hander itself causes another page fault exception this register will be overwritten - so save it early!

## 6.4. Putting it into practice

We're almost ready to start implementing. We will, however, need a few assistant functions first, the most important of which are memory management functions.

### 6.4.1. Simple memory management with placement malloc

If you come from a C++ background, you may have heard of 'placement new'. This is a version of new that takes a parameter. Instead of calling malloc, as it normally would, it creates the object at the address specified. We are going to use a very similar concept.

When the kernel is sufficiently booted, we will have a kernel heap active and operational. The way we code heaps, though, usually requires that virtual memory is enabled. So we need a simple alternative to allocate memory before the heap is active.

As we're allocating quite early on in the kernel bootup, we can make the assumption that nothing that is kmalloc()'d will ever need to be kfree()'d. This simplifies things greatly. We can just have a pointer (*placement address*) to some free memory that we pass back to the requestee then increment. Thus:

u32int kmalloc(u32int sz)  
{  
  u32int tmp = placement\_address;  
  placement\_address += sz;  
  return tmp;  
}

That will actually suffice. However, we have another requirement. When we allocate page tables and directories, they *must be page-aligned*. So we can build that in:

u32int kmalloc(u32int sz, int align)  
{  
  if (align == 1 && (placement\_address & 0xFFFFF000)) // If the address is not already page-aligned  
  {  
    // Align it.  
    placement\_address &= 0xFFFFF000;  
    placement\_address += 0x1000;  
  }  
  u32int tmp = placement\_address;  
  placement\_address += sz;  
  return tmp;  
}

Now, unfortunately, we have one more requirement, and I can't really explain to you why it is required until later in the tutorials. It has to do with when we clone a page directory (when fork()ing processes). At this point, paging will be fully enabled, and kmalloc will return a virtual address. But, we also (bear with me, you'll be glad we did later) need to get the *physical* address of the memory allocated. Take it on faith for now - it's not much code anyway.

u32int kmalloc(u32int sz, int align, u32int \*phys)  
{  
  if (align == 1 && (placement\_address & 0xFFFFF000)) // If the address is not already page-aligned  
  {  
    // Align it.  
    placement\_address &= 0xFFFFF000;  
    placement\_address += 0x1000;  
  }  
  if (phys)  
  {  
    \*phys = placement\_address;  
  }  
  u32int tmp = placement\_address;  
  placement\_address += sz;  
  return tmp;  
}

Great. This is all we need for simple memory management. In my code I have actually (for aesthetic purposes) renamed *kmalloc* to *kmalloc\_int* (for kmalloc\_internal). I then have several wrapper functions:

u32int kmalloc\_a(u32int sz);  // page aligned.  
u32int kmalloc\_p(u32int sz, u32int \*phys); // returns a physical address.  
u32int kmalloc\_ap(u32int sz, u32int \*phys); // page aligned and returns a physical address.  
u32int kmalloc(u32int sz); // vanilla (normal).

I just feel this interface is nicer than specifying 3 parameters for every kernel heap allocation! These definitions should go in kheap.h/kheap.c.

### 6.4.2. Required definitions

*paging.h* should contain some structure definitions that will make our life easier.

#ifndef PAGING\_H  
#define PAGING\_H  
  
#include "common.h"  
#include "isr.h"  
  
typedef struct page  
{  
   u32int present    : 1;   // Page present in memory  
   u32int rw         : 1;   // Read-only if clear, readwrite if set  
   u32int user       : 1;   // Supervisor level only if clear  
   u32int accessed   : 1;   // Has the page been accessed since last refresh?  
   u32int dirty      : 1;   // Has the page been written to since last refresh?  
   u32int unused     : 7;   // Amalgamation of unused and reserved bits  
   u32int frame      : 20;  // Frame address (shifted right 12 bits)  
} page\_t;  
  
typedef struct page\_table  
{  
   page\_t pages[1024];  
} page\_table\_t;  
  
typedef struct page\_directory  
{  
   /\*\*      Array of pointers to pagetables.   \*\*/  
   page\_table\_t \*tables[1024];  
   /\*\*      Array of pointers to the pagetables above, but gives their \*physical\*      location, for loading into the CR3 register.   \*\*/  
   u32int tablesPhysical[1024];  
   /\*\*      The physical address of tablesPhysical. This comes into play      when we get our kernel heap allocated and the directory      may be in a different location in virtual memory.   \*\*/  
   u32int physicalAddr;  
} page\_directory\_t;  
  
/\*\*  Sets up the environment, page directories etc and  enables paging.\*\*/  
void initialise\_paging();  
  
/\*\*  Causes the specified page directory to be loaded into the  CR3 register.\*\*/  
void switch\_page\_directory(page\_directory\_t \*new);  
  
/\*\*  Retrieves a pointer to the page required.  If make == 1, if the page-table in which this page should  reside isn't created, create it!\*\*/  
page\_t \*get\_page(u32int address, int make, page\_directory\_t \*dir);  
  
/\*\*  Handler for page faults.\*\*/  
void page\_fault(registers\_t regs);

Note the *tablesPhysical* and *physicalAddr* members of page\_table\_t. What are they doing there?

The physicalAddr member is actually only for when we clone page directories (not until later in the tutorials). Remember that at that point, the new directory will have an address in virtual memory that is not the same as physical memory. We will need the physical address to tell the CPU if we ever want to switch directories.

The tablesPhysical member is similar. It is a solution to a problem: How do you access your page tables? It may seem simple, but remember that a page directory must hold *physical* addresses, not virtual ones. And the only way you can read/write to memory is using *virtual* addresses!

One solution to this problem is to never access your page tables directly, but to map one page table to point back to the page directory, so that by accessing memory at a certain address you can see all your page tables as if they were pages, and all your page table entries as if they were normal integers. The diagram on the right should help to explain. This method is a little counter-intuitive in my opinion and it also wastes 256MB of addressable space, so I prefer another method.

The second method is to, for every page directory, keep 2 arrays. One holding the physical addresses of it's page tables (for giving to the CPU), and the other holding the virtual ones (so we can read/write to them). This only gives us an extra overhead of 4KB per page directory, which is not much.

### 6.4.3. Frame allocation

If we want to map a page to a frame, we need some way of finding a free frame. Of course, we could just maintain a massive array of 1's and 0's, but that would be extremely wasteful - we don't need 32-bits just to hold 2 values, we can do that with 1 bit. So if we use a [bitset](http://en.wikipedia.org/wiki/Bitset), we will be using 32 times less space!

If you don't know what a bitset (also called a bitmap) is, you should read the link above. There are only 3 functions a bitset implements - set, test and clear. I have also implemented a function to efficiently find the first free frame from the bitmap. Have a look at it and work out why it is efficient. My implementation of these is below. I'm not going to go through explaining it - this is a general concept and is not kernel related. If you're confused, search google for bitset implementations, and if worst comes to the worst post on [the osdev.net forums](http://www.osdev.net/forum).

// A bitset of frames - used or free.  
u32int \*frames;  
u32int nframes;  
  
// Defined in kheap.c  
extern u32int placement\_address;  
  
// Macros used in the bitset algorithms.  
#define INDEX\_FROM\_BIT(a) (a/(8\*4))  
#define OFFSET\_FROM\_BIT(a) (a%(8\*4))  
  
// Static function to set a bit in the frames bitset  
static void set\_frame(u32int frame\_addr)  
{  
   u32int frame = frame\_addr/0x1000;  
   u32int idx = INDEX\_FROM\_BIT(frame);  
   u32int off = OFFSET\_FROM\_BIT(frame);  
   frames[idx] |= (0x1 << off);  
}  
  
// Static function to clear a bit in the frames bitset  
static void clear\_frame(u32int frame\_addr)  
{  
   u32int frame = frame\_addr/0x1000;  
   u32int idx = INDEX\_FROM\_BIT(frame);  
   u32int off = OFFSET\_FROM\_BIT(frame);  
   frames[idx] &= ~(0x1 << off);  
}  
  
// Static function to test if a bit is set.  
static u32int test\_frame(u32int frame\_addr)  
{  
   u32int frame = frame\_addr/0x1000;  
   u32int idx = INDEX\_FROM\_BIT(frame);  
   u32int off = OFFSET\_FROM\_BIT(frame);  
   return (frames[idx] & (0x1 << off));  
}  
  
// Static function to find the first free frame.  
static u32int first\_frame()  
{  
   u32int i, j;  
   for (i = 0; i < INDEX\_FROM\_BIT(nframes); i++)  
   {  
       if (frames[i] != 0xFFFFFFFF) // nothing free, exit early.  
       {  
           // at least one bit is free here.  
           for (j = 0; j < 32; j++)  
           {  
               u32int toTest = 0x1 << j;  
               if ( !(frames[i]&toTest) )  
               {  
                   return i\*4\*8+j;  
               }  
           }  
       }  
   }  
}

Hopefully that code shouldn't cause too many surprises. It just fancy bit twiddling. We then come to functions to allocate and deallocate frames. Now that we have an efficient bitset implementation, these functions total just a few lines!

// Function to allocate a frame.  
void alloc\_frame(page\_t \*page, int is\_kernel, int is\_writeable)  
{  
   if (page->frame != 0)  
   {  
       return; // Frame was already allocated, return straight away.  
   }  
   else  
   {  
       u32int idx = first\_frame(); // idx is now the index of the first free frame.  
       if (idx == (u32int)-1)  
       {  
           // PANIC is just a macro that prints a message to the screen then hits an infinite loop.  
           PANIC("No free frames!");  
       }  
       set\_frame(idx\*0x1000); // this frame is now ours!  
       page->present = 1; // Mark it as present.  
       page->rw = (is\_writeable)?1:0; // Should the page be writeable?  
       page->user = (is\_kernel)?0:1; // Should the page be user-mode?  
       page->frame = idx;  
   }  
}  
  
// Function to deallocate a frame.  
void free\_frame(page\_t \*page)  
{  
   u32int frame;  
   if (!(frame=page->frame))  
   {  
       return; // The given page didn't actually have an allocated frame!  
   }  
   else  
   {  
       clear\_frame(frame); // Frame is now free again.  
       page->frame = 0x0; // Page now doesn't have a frame.  
   }  
}

Note that the PANIC macro just calls a global function called *panic*, with arguments of the message given and the \_\_FILE\_\_ and \_\_LINE\_\_ it occurred on. *panic* prints these out and enters an infinite loop, stopping all execution.

### 6.4.4. Paging code, finally!

void initialise\_paging()  
{  
   // The size of physical memory. For the moment we  
   // assume it is 16MB big.  
   u32int mem\_end\_page = 0x1000000;  
  
   nframes = mem\_end\_page / 0x1000;  
   frames = (u32int\*)kmalloc(INDEX\_FROM\_BIT(nframes));  
   memset(frames, 0, INDEX\_FROM\_BIT(nframes));  
  
   // Let's make a page directory.  
   kernel\_directory = (page\_directory\_t\*)kmalloc\_a(sizeof(page\_directory\_t));  
   memset(kernel\_directory, 0, sizeof(page\_directory\_t));  
   current\_directory = kernel\_directory;  
  
   // We need to identity map (phys addr = virt addr) from  
   // 0x0 to the end of used memory, so we can access this  
   // transparently, as if paging wasn't enabled.  
   // NOTE that we use a while loop here deliberately.  
   // inside the loop body we actually change placement\_address  
   // by calling kmalloc(). A while loop causes this to be  
   // computed on-the-fly rather than once at the start.  
   int i = 0;  
   while (i < placement\_address)  
   {  
       // Kernel code is readable but not writeable from userspace.  
       alloc\_frame( get\_page(i, 1, kernel\_directory), 0, 0);  
       i += 0x1000;  
   }  
   // Before we enable paging, we must register our page fault handler.  
   register\_interrupt\_handler(14, page\_fault);  
  
   // Now, enable paging!  
   switch\_page\_directory(kernel\_directory);  
}  
  
void switch\_page\_directory(page\_directory\_t \*dir)  
{  
   current\_directory = dir;  
   asm volatile("mov %0, %%cr3":: "r"(&dir->tablesPhysical));  
   u32int cr0;  
   asm volatile("mov %%cr0, %0": "=r"(cr0));  
   cr0 |= 0x80000000; // Enable paging!  
   asm volatile("mov %0, %%cr0":: "r"(cr0));  
}  
  
page\_t \*get\_page(u32int address, int make, page\_directory\_t \*dir)  
{  
   // Turn the address into an index.  
   address /= 0x1000;  
   // Find the page table containing this address.  
   u32int table\_idx = address / 1024;  
   if (dir->tables[table\_idx]) // If this table is already assigned  
   {  
       return &dir->tables[table\_idx]->pages[address%1024];  
   }  
   else if(make)  
   {  
       u32int tmp;  
       dir->tables[table\_idx] = (page\_table\_t\*)kmalloc\_ap(sizeof(page\_table\_t), &tmp);  
       memset(dir->tables[table\_idx], 0, 0x1000);  
       dir->tablesPhysical[table\_idx] = tmp | 0x7; // PRESENT, RW, US.  
       return &dir->tables[table\_idx]->pages[address%1024];  
   }  
   else  
   {  
       return 0;  
   }  
}

Right, let's analyse that. First of all, the utility functions.

*switch\_page\_directory* does exactly what it says on the tin. It takes a page directory, and switches to it. It does this by moving the address of the tablesPhysical member of that directory into the CR3 register. Remember that the tablesPhysical member is an array of physical addresses. After that it first gets the contents of CR0, then OR-s the *PG* bit (0x80000000), then rewrites it. This enables paging and flushes the page-directory cache as well.

*get\_page* returns a pointer to the page entry for a particular address. It can optionally be passed a parameter - make. If make is 1, and the page table that the requested page entry should reside in hasn't been created, then it will be created. Otherwise, the function would just return 0. So, if the table has already been assigned, it will look up the page entry and return it. If it hasn't (and make == 1), it will attempt to create it.

It uses our *kmalloc\_ap* function to retrieve a memory block which is page-aligned, and *also gets given it's physical location*. The physical location gets stored in 'tablesPhysical' (after several bits have been set telling the CPU that it is present, writeable, and user-accessible), and the virtual location is stored in 'tables'.

*initialise\_paging* firstly creates the frames bitset, and sets everything to zero using memset. Then it allocates space (which is page-aligned) for a page directory. After that, it allocates frames such that any page access will map to the frame with the same linear address, called identity-mapping. This is done for a small section of the address space, so the kernel code can continue to run as normal. It registers an interrupt handler for page faults (below) then enables paging.

### 6.4.5. The page fault handler

void page\_fault(registers\_t regs)  
{  
   // A page fault has occurred.  
   // The faulting address is stored in the CR2 register.  
   u32int faulting\_address;  
   asm volatile("mov %%cr2, %0" : "=r" (faulting\_address));  
  
   // The error code gives us details of what happened.  
   int present   = !(regs.err\_code & 0x1); // Page not present  
   int rw = regs.err\_code & 0x2;           // Write operation?  
   int us = regs.err\_code & 0x4;           // Processor was in user-mode?  
   int reserved = regs.err\_code & 0x8;     // Overwritten CPU-reserved bits of page entry?  
   int id = regs.err\_code & 0x10;          // Caused by an instruction fetch?  
  
   // Output an error message.  
   monitor\_write("Page fault! ( ");  
   if (present) {monitor\_write("present ");}  
   if (rw) {monitor\_write("read-only ");}  
   if (us) {monitor\_write("user-mode ");}  
   if (reserved) {monitor\_write("reserved ");}  
   monitor\_write(") at 0x");  
   monitor\_write\_hex(faulting\_address);  
   monitor\_write("\n");  
   PANIC("Page fault");  
}

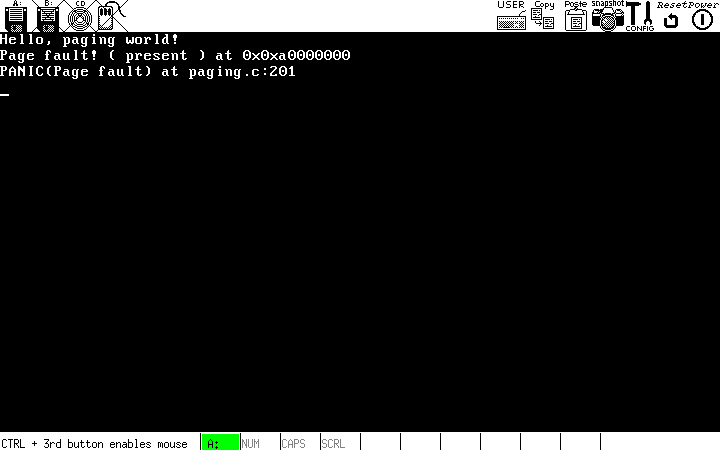
All this handler does is print out a nice error message. It gets the faulting address from CR2, and analyses the error code pushed by the processor to glean some information from it.

### 6.4.6. Testing

Awesome! you now have code that enables paging and handles page faults! Let's just check it actually works, shall we ...?

*main.c*

int main(struct multiboot \*mboot\_ptr)  
{  
   // Initialise all the ISRs and segmentation  
   init\_descriptor\_tables();  
   // Initialise the screen (by clearing it)  
   monitor\_clear();  
  
   initialise\_paging();  
   monitor\_write("Hello, paging world!\n");  
  
   u32int \*ptr = (u32int\*)0xA0000000;  
   u32int do\_page\_fault = \*ptr;  
  
   return 0;  
}

  
Bochs, enabling paging then forcing a page fault

This will, obviously, initialise paging, print a string to make sure it's set up right and not faulting when it shoudn't, and then force a page fault by reading location 0xA0000000.

Congrats! you're all done! you can now move on to the next tutorial - making a working kernel heap :D. The source code for this tutorial is available [here](http://www.jamesmolloy.co.uk/downloads/paging.tar.gz).

***Copyright James Molloy 2008 - james<at>jamesmolloy.co.uk***

www.jamesmolloy.co.uk

[**Home**](http://www.jamesmolloy.co.uk/index.html) » JamesM's kernel development tutorials

[1. Environment setup](http://www.jamesmolloy.co.uk/tutorial_html/1.-Environment%20setup.html)  
[2. Genesis](http://www.jamesmolloy.co.uk/tutorial_html/2.-Genesis.html)  
[3. The Screen](http://www.jamesmolloy.co.uk/tutorial_html/3.-The%20Screen.html)  
[4. The GDT and IDT](http://www.jamesmolloy.co.uk/tutorial_html/4.-The%20GDT%20and%20IDT.html)  
[5. IRQs and the PIT](http://www.jamesmolloy.co.uk/tutorial_html/5.-IRQs%20and%20the%20PIT.html)  
[6. Paging](http://www.jamesmolloy.co.uk/tutorial_html/6.-Paging.html)  
**[7. The Heap](http://www.jamesmolloy.co.uk/tutorial_html/7.-The%20Heap.html)**  
[8. The VFS and the initrd](http://www.jamesmolloy.co.uk/tutorial_html/8.-The%20VFS%20and%20the%20initrd.html)  
[9. Multitasking](http://www.jamesmolloy.co.uk/tutorial_html/9.-Multitasking.html)  
[10. User Mode](http://www.jamesmolloy.co.uk/tutorial_html/10.-User%20Mode.html)

# 7. The Heap

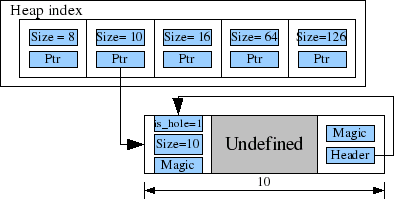
In order to be responsive to situations that you didn't envisage at the design stage, and to cut down the size of your kernel, you will need some kind of dynamic memory allocation. The current memory allocation system (allocation by placement address) is absolutely fine, and is in fact optimal for both 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. It is similar to (though more simple than) [Doug Lea's malloc](http://g.oswego.edu/dl/html/malloc.html) which is used in the GNU C library.

## 7.1. Data structure description

  
The index table with pointers to holes

The algorithm uses two concepts: *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 blocks but their contents are *not* in use. So initially by this concept the entire area of heap space is one large hole.

For every hole there is a corresponding descriptor in an index table. The index table is always ordered ascending by the size of the hole pointed to.

Blocks and holes each contain descriptive data - a header and a footer. The header contains the most information about the block - the footer merely contains a pointer to the header (the reason for the footer will become apparent soon). Pseudocode:

typedef struct  
{  
  u32int magic;  // Magic number, used for error checking and identification.  
  u8int is\_hole; // 1 if this is a hole, 0 if this is a block.  
  u32int size;   // Size of the block, including this and the footer.  
} header\_t;  
  
typedef struct  
{  
  u32int magic;     // Magic number, same as in header\_t.  
  header\_t \*header; // Pointer to the block header.  
} footer\_t;

Notice that each also has a 'magic number' field. This is for error checking, and later will play a part in our 'free' algorithm. This is just a sentinel number - an unusual number that will stand out from others - much like 0xdeadbaba that we used in chapter 2. In the sample code I've gone for 0x123890AB arbitrarily.

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 footer - so within a block of size *x*, there will be x - sizeof(header\_t) - sizeof(footer\_t) user-useable bytes.

## 7.2. Algorithm description

### 7.2.1. Allocation

Allocation is straightforward, if a little long-winded. Most of the steps are error-checking and creating new holes to minimise memory leaks.

1. Search the index table to find the smallest hole that will fit the requested size. As the table is ordered, this just entails iterating through until we find a hole which will fit.
   * If we didn't find a hole large enough, then:
     1. Expand the heap.
     2. If the index table is empty (no holes have been recorded) then add a new entry to it.
     3. Else, adjust the last header's size member and rewrite the footer.
     4. To ease the number of control-flow statements, we can just recurse and call the allocation function again, trusting that this time there will be a hole large enough.
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. If the block should be page-aligned, we must alter the block starting address so that it is and create a new hole in the new unused area.
   * If it is not, we can just delete the hole from the index.
4. Write the new block's header and footer.
5. If the hole was to be split into two parts, do it now and write a new hole into the index.
6. Return the address of the block + sizeof(header\_t) to the user.

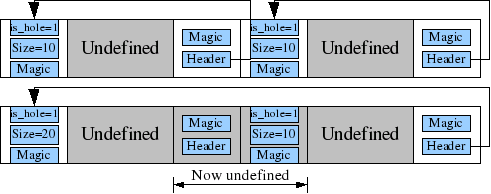
### 7.2.2. Deallocation

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!

  
Unifying the two allocations in the top diagram into one in the lower diagram.

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 that is a footer, which can be discovered from the value of the magic number, then follow the pointer to it's header and query whether it is a hole or a block. If it is a hole, we can modify it's header's size attribute to take into account both it's size and ours, then point *our* footer to *it's* header. 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 the footer. If we find a header there, again identified by it's magic number, we check if it is a hole. We can then use it's size attribute to find it's footer. We rewrite the footer's pointer to point to *our* header. Then, all that needs to be done is to remove it's old entry from the hole index, and add our own.

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. To avoid this happening constantly, in my implementation I have defined a minimum heap size, below which it will not contract.

#### 7.2.2.1. Pseudocode

1. Find the header by taking the given pointer and subtracting the sizeof(header\_t).
2. Sanity checks. Assert that the header and footer's magic numbers remain in tact.
3. Set the is\_hole flag in our header to 1.
4. If the thing immediately to our left is a footer:
   * Unify left. In this case, at the end of the algorithm we shouldn't add our header to the hole index (the header we are unifying with is already there!) so set a flag which the algorithm checks later.
5. If the thing immediately to our right is a header:
   * Unify right.
6. If the footer is the last in the heap ( footer\_location+sizeof(footer\_t) == end\_address ):
   * Contract.
7. Insert the header into the hole array unless the flag described in *Unify left* is set.

## 7.3. Implementing an ordered list

So now we come to the implementation. As usual I'm going to try and explain the utility datatypes and functions first, and finish up with the allocation/free functions themselves.

The first datatype we need it an implementation of an ordered list. This concept will be used multiple times in your kernel (it is a common requirement) so it is probably a good idea to abstract it, so it can be used again.

### 7.3.1. ordered\_array.h

// ordered\_array.h -- Interface for creating, inserting and deleting  
// from ordered arrays.  
// Written for JamesM's kernel development tutorials.  
  
#ifndef ORDERED\_ARRAY\_H  
#define ORDERED\_ARRAY\_H  
  
#include "common.h"  
  
/\*\*  This array is insertion sorted - it always remains in a sorted state (between calls).  It can store anything that can be cast to a void\* -- so a u32int, or any pointer.\*\*/  
typedef void\* type\_t;  
/\*\*  A predicate should return nonzero if the first argument is less than the second. Else  it should return zero.\*\*/  
typedef s8int (\*lessthan\_predicate\_t)(type\_t,type\_t);  
typedef struct  
{  
   type\_t \*array;  
   u32int size;  
   u32int max\_size;  
   lessthan\_predicate\_t less\_than;  
} ordered\_array\_t;  
  
/\*\*  A standard less than predicate.\*\*/  
s8int standard\_lessthan\_predicate(type\_t a, type\_t b);  
  
/\*\*  Create an ordered array.\*\*/  
ordered\_array\_t create\_ordered\_array(u32int max\_size, lessthan\_predicate\_t less\_than);  
ordered\_array\_t place\_ordered\_array(void \*addr, u32int max\_size, lessthan\_predicate\_t less\_than);  
  
/\*\*  Destroy an ordered array.\*\*/  
void destroy\_ordered\_array(ordered\_array\_t \*array);  
  
/\*\*  Add an item into the array.\*\*/  
void insert\_ordered\_array(type\_t item, ordered\_array\_t \*array);  
  
/\*\*  Lookup the item at index i.\*\*/  
type\_t lookup\_ordered\_array(u32int i, ordered\_array\_t \*array);  
  
/\*\*  Deletes the item at location i from the array.\*\*/  
void remove\_ordered\_array(u32int i, ordered\_array\_t \*array);  
  
#endif // ORDERED\_ARRAY\_H

Notice that in the name of abstraction we have made the 'less than' function user-defineable. We will use this in the heap implementation to order by size and not pointer address. Note also we have two methods of defining an ordered\_array. *create\_ordered\_array* will use kmalloc() to get some space. *place\_ordered\_array* will use the given start location. As we want to put our heap in a specific place (and because kmalloc isn't yet working!) we use place\_ordered\_array in our heap code.

### 7.3.2. ordered\_map.c

// ordered\_array.c -- Implementation for creating, inserting and deleting  
// from ordered arrays.  
// Written for JamesM's kernel development tutorials.  
  
#include "ordered\_array.h"  
  
s8int standard\_lessthan\_predicate(type\_t a, type\_t b)  
{  
   return (a<b)?1:0;  
}  
  
ordered\_array\_t create\_ordered\_array(u32int max\_size, lessthan\_predicate\_t less\_than)  
{  
   ordered\_array\_t to\_ret;  
   to\_ret.array = (void\*)kmalloc(max\_size\*sizeof(type\_t));  
   memset(to\_ret.array, 0, max\_size\*sizeof(type\_t));  
   to\_ret.size = 0;  
   to\_ret.max\_size = max\_size;  
   to\_ret.less\_than = less\_than;  
   return to\_ret;  
}  
  
ordered\_array\_t place\_ordered\_array(void \*addr, u32int max\_size, lessthan\_predicate\_t less\_than)  
{  
   ordered\_array\_t to\_ret;  
   to\_ret.array = (type\_t\*)addr;  
   memset(to\_ret.array, 0, max\_size\*sizeof(type\_t));  
   to\_ret.size = 0;  
   to\_ret.max\_size = max\_size;  
   to\_ret.less\_than = less\_than;  
   return to\_ret;  
}  
  
void destroy\_ordered\_array(ordered\_array\_t \*array)  
{  
// kfree(array->array);  
}  
  
void insert\_ordered\_array(type\_t item, ordered\_array\_t \*array)  
{  
   ASSERT(array->less\_than);  
   u32int iterator = 0;  
   while (iterator < array->size && array->less\_than(array->array[iterator], item))  
       iterator++;  
   if (iterator == array->size) // just add at the end of the array.  
       array->array[array->size++] = item;  
   else  
   {  
       type\_t tmp = array->array[iterator];  
       array->array[iterator] = item;  
       while (iterator < array->size)  
       {  
           iterator++;  
           type\_t tmp2 = array->array[iterator];  
           array->array[iterator] = tmp;  
           tmp = tmp2;  
       }  
       array->size++;  
   }  
}  
  
type\_t lookup\_ordered\_array(u32int i, ordered\_array\_t \*array)  
{  
   ASSERT(i < array->size);  
   return array->array[i];  
}  
  
void remove\_ordered\_array(u32int i, ordered\_array\_t \*array)  
{  
   while (i < array->size)  
   {  
       array->array[i] = array->array[i+1];  
       i++;  
   }  
   array->size--;  
}

Hopefully nothing there should surprise you. On *insert* the item is placed at the correct position and all larger other items shifted up one position. As always with these satellite datatypes, any implementation will work. There **are** better implementations of ordered arrays than this (c.f. heap-ordering, binary search trees), but I decided to go with a simple one for teaching purposes.

## 7.4. The heap itself

### 7.4.1. kheap.h

Some #defines and function prototypes are useful:

#define KHEAP\_START         0xC0000000  
#define KHEAP\_INITIAL\_SIZE  0x100000  
#define HEAP\_INDEX\_SIZE   0x20000  
#define HEAP\_MAGIC        0x123890AB  
#define HEAP\_MIN\_SIZE     0x70000  
  
/\*\*  Size information for a hole/block\*\*/  
typedef struct  
{  
   u32int magic;   // Magic number, used for error checking and identification.  
   u8int is\_hole;   // 1 if this is a hole. 0 if this is a block.  
   u32int size;    // size of the block, including the end footer.  
} header\_t;  
  
typedef struct  
{  
   u32int magic;     // Magic number, same as in header\_t.  
   header\_t \*header; // Pointer to the block header.  
} footer\_t;  
  
typedef struct  
{  
   ordered\_array\_t index;  
   u32int start\_address; // The start of our allocated space.  
   u32int end\_address;   // The end of our allocated space. May be expanded up to max\_address.  
   u32int max\_address;   // The maximum address the heap can be expanded to.  
   u8int supervisor;     // Should extra pages requested by us be mapped as supervisor-only?  
   u8int readonly;       // Should extra pages requested by us be mapped as read-only?  
} heap\_t;  
  
/\*\*  Create a new heap.\*\*/  
heap\_t \*create\_heap(u32int start, u32int end, u32int max, u8int supervisor, u8int readonly);  
/\*\*  Allocates a contiguous region of memory 'size' in size. If page\_align==1, it creates that block starting  on a page boundary.\*\*/  
void \*alloc(u32int size, u8int page\_align, heap\_t \*heap);  
/\*\*  Releases a block allocated with 'alloc'.\*\*/  
void free(void \*p, heap\_t \*heap);

I have decided, arbitrarily, to put the kernel heap at 0xC0000000, give it's index a size of 0x20000 bytes, and give it a minimum size of 0x70000 bytes. The header and footer structures are the same as those given at the top of the chapter. We can actually have more than one heap (in my own kernel I have a user-mode heap as well), so for ease of portability I have decided to implement the heap as a datatype itself. heap\_t keeps track of the heap's index, start/end/max addresses and the modifiers to give alloc\_page when requesting more memory.

### 7.4.2. kheap.c

Finding the smallest hole that will fit a certain number of bytes is a common task that gets called on every allocation. It would therefore be nice to wrap this up in a function:

static s32int find\_smallest\_hole(u32int size, u8int page\_align, heap\_t \*heap)  
{  
   // Find the smallest hole that will fit.  
   u32int iterator = 0;  
   while (iterator < heap->index.size)  
   {  
       header\_t \*header = (header\_t \*)lookup\_ordered\_array(iterator, &heap->index);  
       // If the user has requested the memory be page-aligned  
       if (page\_align > 0)  
       {  
           // Page-align the starting point of this header.  
           u32int location = (u32int)header;  
           s32int offset = 0;  
           if ((location+sizeof(header\_t)) & 0xFFFFF000 != 0)  
               offset = 0x1000 /\* page size \*/  - (location+sizeof(header\_t))%0x1000;  
           s32int hole\_size = (s32int)header->size - offset;  
           // Can we fit now?  
           if (hole\_size >= (s32int)size)  
               break;  
       }  
       else if (header->size >= size)  
           break;  
       iterator++;  
   }  
   // Why did the loop exit?  
   if (iterator == heap->index.size)  
       return -1; // We got to the end and didn't find anything.  
   else  
       return iterator;  
}

I feel I should explain two lines:

if ((location+sizeof(header\_t)) & 0xFFFFF000 != 0)  
  offset = 0x1000 /\* page size \*/  - (location+sizeof(header\_t))%0x1000;

It's important to note that when the user requests memory to be page-aligned, he is requesting the memory *that he has access to* to be page-aligned. That means that the header address will actually *not* be page-aligned. The address that we want to fall on a boundary is location + sizeof(header\_t).

Creating a heap is a simple procedure. The only part worthy of note is that we set aside the first HEAP\_INDEX\_SIZE\*sizeof(type\_t) bytes as the index. The index is put there using place\_ordered\_array, and the effective start address is shifted forwards. That is why, when testing your kernel, you will see allocations starting at 0xC0080000 instead of the more obvious 0xC0000000. Also note that we create a custom less\_than function for the index array. This is because with the standard less\_than function the array would be sorted by *pointer address*, instead of by size.

static s8int header\_t\_less\_than(void\*a, void \*b)  
{  
   return (((header\_t\*)a)->size < ((header\_t\*)b)->size)?1:0;  
}  
  
heap\_t \*create\_heap(u32int start, u32int end\_addr, u32int max, u8int supervisor, u8int readonly)  
{  
   heap\_t \*heap = (heap\_t\*)kmalloc(sizeof(heap\_t));  
  
   // All our assumptions are made on startAddress and endAddress being page-aligned.  
   ASSERT(start%0x1000 == 0);  
   ASSERT(end\_addr%0x1000 == 0);  
  
   // Initialise the index.  
   heap->index = place\_ordered\_array( (void\*)start, HEAP\_INDEX\_SIZE, &header\_t\_less\_than);  
  
   // Shift the start address forward to resemble where we can start putting data.  
   start += sizeof(type\_t)\*HEAP\_INDEX\_SIZE;  
  
   // Make sure the start address is page-aligned.  
   if (start & 0xFFFFF000 != 0)  
   {  
       start &= 0xFFFFF000;  
       start += 0x1000;  
   }  
   // Write the start, end and max addresses into the heap structure.  
   heap->start\_address = start;  
   heap->end\_address = end\_addr;  
   heap->max\_address = max;  
   heap->supervisor = supervisor;  
   heap->readonly = readonly;  
  
   // We start off with one large hole in the index.  
   header\_t \*hole = (header\_t \*)start;  
   hole->size = end\_addr-start;  
   hole->magic = HEAP\_MAGIC;  
   hole->is\_hole = 1;  
   insert\_ordered\_array((void\*)hole, &heap->index);  
  
   return heap;  
}

#### 7.4.2.1. Expansion and contraction

At points we will need to alter the size of our heap. If we run out of space, we will need more. If we reclaim space, we may need less.

static void expand(u32int new\_size, heap\_t \*heap)  
{  
   // Sanity check.  
   ASSERT(new\_size > heap->end\_address - heap->start\_address);  
   // Get the nearest following page boundary.  
   if (new\_size&0xFFFFF000 != 0)  
   {  
       new\_size &= 0xFFFFF000;  
       new\_size += 0x1000;  
   }  
   // Make sure we are not overreaching ourselves.  
   ASSERT(heap->start\_address+new\_size <= heap->max\_address);  
  
   // This should always be on a page boundary.  
   u32int old\_size = heap->end\_address-heap->start\_address;  
   u32int i = old\_size;  
   while (i < new\_size)  
   {  
       alloc\_frame( get\_page(heap->start\_address+i, 1, kernel\_directory),  
                    (heap->supervisor)?1:0, (heap->readonly)?0:1);  
       i += 0x1000 /\* page size \*/;  
   }  
   heap->end\_address = heap->start\_address+new\_size;  
}

I think that code is self-explanatory. A few assertions are made, and the new\_size parameter is changed so that it falls on a page boundary. Frames are then allocated one-by-one according to the parameters given when creating the heap (supervisor mode enabled?, read only access?).

static u32int contract(u32int new\_size, heap\_t \*heap)  
{  
   // Sanity check.  
   ASSERT(new\_size < heap->end\_address-heap->start\_address);  
   // Get the nearest following page boundary.  
   if (new\_size&0x1000)  
   {  
       new\_size &= 0x1000;  
       new\_size += 0x1000;  
   }  
   // Don't contract too far!  
   if (new\_size < HEAP\_MIN\_SIZE)  
       new\_size = HEAP\_MIN\_SIZE;  
   u32int old\_size = heap->end\_address-heap->start\_address;  
   u32int i = old\_size - 0x1000;  
   while (new\_size < i)  
   {  
       free\_frame(get\_page(heap->start\_address+i, 0, kernel\_directory));  
       i -= 0x1000;  
   }  
   heap->end\_address = heap->start\_address + new\_size;  
   return new\_size;  
}

Similarly to expand, new\_size is adjusted so it sits on a page boundary. We then check that we're not trying to contract past our minimum size, and free each frame in turn until we reach the desired size.

#### 7.4.2.2. Allocation

We'll talk through the allocation function in parts.

void \*alloc(u32int size, u8int page\_align, heap\_t \*heap)  
{  
  
   // Make sure we take the size of header/footer into account.  
   u32int new\_size = size + sizeof(header\_t) + sizeof(footer\_t);  
   // Find the smallest hole that will fit.  
   s32int iterator = find\_smallest\_hole(new\_size, page\_align, heap);  
     
   if (iterator == -1) // If we didn't find a suitable hole  
   {  
     ... // Will be filled in in a second.  
   }

Here we adjust the requested block size to account for the size of the header and footer. We then request the smallest hole available that will fit using our find\_smallest\_hole function. If we couldn't find one (find\_smallest\_hole() == -1), we go into some error-handling code. It's a bit beefy, so I'll come back to this to explain it.

   header\_t \*orig\_hole\_header = (header\_t \*)lookup\_ordered\_array(iterator, &heap->index);  
   u32int orig\_hole\_pos = (u32int)orig\_hole\_header;  
   u32int orig\_hole\_size = orig\_hole\_header->size;  
   // Here we work out if we should split the hole we found into two parts.  
   // Is the original hole size - requested hole size less than the overhead for adding a new hole?  
   if (orig\_hole\_size-new\_size < sizeof(header\_t)+sizeof(footer\_t))  
   {  
       // Then just increase the requested size to the size of the hole we found.  
       size += orig\_hole\_size-new\_size;  
       new\_size = orig\_hole\_size;  
   }

Here we get the header pointer from the index given us by find\_smallest\_hole. We then save the address and size of this header in case we need to overwrite it later. After this, we decide if it is worth splitting the hole in two (that is, will the free space be able to fit another hole into it?) If not, we increase the requested size to the hole size, so it isn't split.

   // If we need to page-align the data, do it now and make a new hole in front of our block.  
   if (page\_align && orig\_hole\_pos&0xFFFFF000)  
   {  
       u32int new\_location   = orig\_hole\_pos + 0x1000 /\* page size \*/ - (orig\_hole\_pos&0xFFF) - sizeof(header\_t);  
       header\_t \*hole\_header = (header\_t \*)orig\_hole\_pos;  
       hole\_header->size     = 0x1000 /\* page size \*/ - (orig\_hole\_pos&0xFFF) - sizeof(header\_t);  
       hole\_header->magic    = HEAP\_MAGIC;  
       hole\_header->is\_hole  = 1;  
       footer\_t \*hole\_footer = (footer\_t \*) ( (u32int)new\_location - sizeof(footer\_t) );  
       hole\_footer->magic    = HEAP\_MAGIC;  
       hole\_footer->header   = hole\_header;  
       orig\_hole\_pos         = new\_location;  
       orig\_hole\_size        = orig\_hole\_size - hole\_header->size;  
   }  
   else  
   {  
       // Else we don't need this hole any more, delete it from the index.  
       remove\_ordered\_array(iterator, &heap->index);  
   }

If the user wants his memory to be page-aligned, we facilitate that here. The new location for the header to be placed at is calculated by going to the next page boundary then subtracting the size of a header. The attributes of the new hole's header are then filled in, along with the footer. Note that because we are creating a new hole at the old hole's address, we are essentially reusing the old hole, so there is no need to remove it from the hole index.

   // Overwrite the original header...  
   header\_t \*block\_header  = (header\_t \*)orig\_hole\_pos;  
   block\_header->magic     = HEAP\_MAGIC;  
   block\_header->is\_hole   = 0;  
   block\_header->size      = new\_size;  
   // ...And the footer  
   footer\_t \*block\_footer  = (footer\_t \*) (orig\_hole\_pos + sizeof(header\_t) + size);  
   block\_footer->magic     = HEAP\_MAGIC;  
   block\_footer->header    = block\_header;

This should be self-explanatory - we make sure all the header and footer attributes are correct, along with magic numbers.

   // We may need to write a new hole after the allocated block.  
   // We do this only if the new hole would have positive size...  
   if (orig\_hole\_size - new\_size > 0)  
   {  
       header\_t \*hole\_header = (header\_t \*) (orig\_hole\_pos + sizeof(header\_t) + size + sizeof(footer\_t));  
       hole\_header->magic    = HEAP\_MAGIC;  
       hole\_header->is\_hole  = 1;  
       hole\_header->size     = orig\_hole\_size - new\_size;  
       footer\_t \*hole\_footer = (footer\_t \*) ( (u32int)hole\_header + orig\_hole\_size - new\_size - sizeof(footer\_t) );  
       if ((u32int)hole\_footer < heap->end\_address)  
       {  
           hole\_footer->magic = HEAP\_MAGIC;  
           hole\_footer->header = hole\_header;  
       }  
       // Put the new hole in the index;  
       insert\_ordered\_array((void\*)hole\_header, &heap->index);  
   }

If we wanted to split our hole in two, we do it here, creating a new hole.

   // ...And we're done!  
   return (void \*) ( (u32int)block\_header+sizeof(header\_t) );  
}

... And that's our allocation function! The only thing left to do is fill in the error-checking code we missed out earlier:

   if (iterator == -1) // If we didn't find a suitable hole  
   {  
       // Save some previous data.  
       u32int old\_length = heap->end\_address - heap->start\_address;  
       u32int old\_end\_address = heap->end\_address;  
  
       // We need to allocate some more space.  
       expand(old\_length+new\_size, heap);  
       u32int new\_length = heap->end\_address-heap->start\_address;  
  
       // Find the endmost header. (Not endmost in size, but in location).  
       iterator = 0;  
       // Vars to hold the index of, and value of, the endmost header found so far.  
       u32int idx = -1; u32int value = 0x0;  
       while (iterator < heap->index.size)  
       {  
           u32int tmp = (u32int)lookup\_ordered\_array(iterator, &heap->index);  
           if (tmp > value)  
           {  
               value = tmp;  
               idx = iterator;  
           }  
           iterator++;  
       }  
  
       // If we didn't find ANY headers, we need to add one.  
       if (idx == -1)  
       {  
           header\_t \*header = (header\_t \*)old\_end\_address;  
           header->magic = HEAP\_MAGIC;  
           header->size = new\_length - old\_length;  
           header->is\_hole = 1;  
           footer\_t \*footer = (footer\_t \*) (old\_end\_address + header->size - sizeof(footer\_t));  
           footer->magic = HEAP\_MAGIC;  
           footer->header = header;  
           insert\_ordered\_array((void\*)header, &heap->index);  
       }  
       else  
       {  
           // The last header needs adjusting.  
           header\_t \*header = lookup\_ordered\_array(idx, &heap->index);  
           header->size += new\_length - old\_length;  
           // Rewrite the footer.  
           footer\_t \*footer = (footer\_t \*) ( (u32int)header + header->size - sizeof(footer\_t) );  
           footer->header = header;  
           footer->magic = HEAP\_MAGIC;  
       }  
       // We now have enough space. Recurse, and call the function again.  
       return alloc(size, page\_align, heap);  
   }

This code quite simple in function but verbose in code. If a hole big enough couldn't be found (iterator == -1), we must expand the size of the heap (by calling the *expand* function). We then have to account for this expansion in the index. The normal way to do this is to find the endmost hole in the index and adjust it's size. The only time this won't work is when there aren't any holes in the index at all (an unlikely but possible case). In this case we must make one to fill the gap.

#### 7.4.2.3. Freeing

Again, I'll go through this step-by-step.

void free(void \*p, heap\_t \*heap)  
{  
   // Exit gracefully for null pointers.  
   if (p == 0)  
       return;  
  
   // Get the header and footer associated with this pointer.  
   header\_t \*header = (header\_t\*) ( (u32int)p - sizeof(header\_t) );  
   footer\_t \*footer = (footer\_t\*) ( (u32int)header + header->size - sizeof(footer\_t) );  
  
   // Sanity checks.  
   ASSERT(header->magic == HEAP\_MAGIC);  
   ASSERT(footer->magic == HEAP\_MAGIC);

Initially we find the header by subtracting sizeof(header\_t) from *p*, then use this to find the footer. Sanity checks are always a good idea, as they provide an early indication if your code has overwritten crucial data.

   // Make us a hole.  
   header->is\_hole = 1;  
  
   // Do we want to add this header into the 'free holes' index?  
   char do\_add = 1;

This block is being deallocated and so is now a hole. We also create a variable to hold whether we should add the header to the hole index (see algorithm description).

   // Unify left  
   // If the thing immediately to the left of us is a footer...  
   footer\_t \*test\_footer = (footer\_t\*) ( (u32int)header - sizeof(footer\_t) );  
   if (test\_footer->magic == HEAP\_MAGIC &&  
       test\_footer->header->is\_hole == 1)  
   {  
       u32int cache\_size = header->size; // Cache our current size.  
       header = test\_footer->header;     // Rewrite our header with the new one.  
       footer->header = header;          // Rewrite our footer to point to the new header.  
       header->size += cache\_size;       // Change the size.  
       do\_add = 0;                       // Since this header is already in the index, we don't want to add it again.  
   }

This piece of code performs our *left unification*. By subtracting sizeof(header\_t) from the header address, we can get a pointer to a footer. We check if this is actually a valid footer by checking it's magic number, and that it is a hole (not allocated!). If so, we rewrite our footer to point to the test footer's header, change our size, and instruct the algorithm not to add an entry to the hole index (as the header we just unified with was already in the index!).

   // Unify right  
   // If the thing immediately to the right of us is a header...  
   header\_t \*test\_header = (header\_t\*) ( (u32int)footer + sizeof(footer\_t) );  
   if (test\_header->magic == HEAP\_MAGIC &&  
       test\_header->is\_hole)  
   {  
       header->size += test\_header->size; // Increase our size.  
       test\_footer = (footer\_t\*) ( (u32int)test\_header + // Rewrite it's footer to point to our header.  
                                   test\_header->size - sizeof(footer\_t) );  
       footer = test\_footer;  
       // Find and remove this header from the index.  
       u32int iterator = 0;  
       while ( (iterator < heap->index.size) &&  
               (lookup\_ordered\_array(iterator, &heap->index) != (void\*)test\_header) )  
           iterator++;  
  
       // Make sure we actually found the item.  
       ASSERT(iterator < heap->index.size);  
       // Remove it.  
       remove\_ordered\_array(iterator, &heap->index);  
   }

Similarly, this code performs our *right unification*. Again, we test if the header immediately to our right is valid, and is a hole. We rewrite it's footer to point to our header, then remove it's header from the hole index.

   // If the footer location is the end address, we can contract.  
   if ( (u32int)footer+sizeof(footer\_t) == heap->end\_address)  
   {  
       u32int old\_length = heap->end\_address-heap->start\_address;  
       u32int new\_length = contract( (u32int)header - heap->start\_address, heap);  
       // Check how big we will be after resizing.  
       if (header->size - (old\_length-new\_length) > 0)  
       {  
           // We will still exist, so resize us.  
           header->size -= old\_length-new\_length;  
           footer = (footer\_t\*) ( (u32int)header + header->size - sizeof(footer\_t) );  
           footer->magic = HEAP\_MAGIC;  
           footer->header = header;  
       }  
       else  
       {  
           // We will no longer exist :(. Remove us from the index.  
           u32int iterator = 0;  
           while ( (iterator < heap->index.size) &&  
                   (lookup\_ordered\_array(iterator, &heap->index) != (void\*)test\_header) )  
               iterator++;  
           // If we didn't find ourselves, we have nothing to remove.  
           if (iterator < heap->index.size)  
               remove\_ordered\_array(iterator, &heap->index);  
       }  
   }

This is almost the last snippet (I promise!). If we are releasing the last hole in the index (that is, the one closest to the end of memory), then we can contract the heap size. We keep note of the old heap size, then call contract. One of two things can happen here. Either the contract() command will shrink the heap so our hole no longer exists (the 'else' case), or it will either partially contract or not contract at all. In which case our hole still exists, but we need to resize it. We rewrite it's footer with the new size, and exit. If the hole has been removed, we just look ourselves up in the heap index and delete ourselves.

A small one to finish off with:

if (do\_add == 1)  
  insert\_ordered\_array((void\*) header, &heap->index);

If we are suppposed to add ourselves into the index, do it here. And that's it! The next thing to do is initialise the heap when paging is initialised :)

#### 7.4.2.4. paging.c

extern heap\_t \*kheap;

We declare the variable *kheap* as our kernel heap. We define this in kheap.c (you can do that yourself) and reference it here.

   // Map some pages in the kernel heap area.  
   // Here we call get\_page but not alloc\_frame. This causes page\_table\_t's  
   // to be created where necessary. We can't allocate frames yet because they  
   // they need to be identity mapped first below, and yet we can't increase  
   // placement\_address between identity mapping and enabling the heap!  
   int i = 0;  
   for (i = KHEAP\_START; i < KHEAP\_START+KHEAP\_INITIAL\_SIZE; i += 0x1000)  
       get\_page(i, 1, kernel\_directory);

This goes in *initialise\_paging*, before we identity map from 0-placement\_addr. There is a reason for this code. As the kernel heap is up at 0xC0000000, when we write to it some page tables will need to be created (beacause nothing near that area has been accessed before). However, after we finish the identity-mapping loop allocating everything up to placement\_address, we can't use kmalloc any more until our heap is active! So, we need to force the tables to be created **before** we freeze the placement address. That's what this code does.

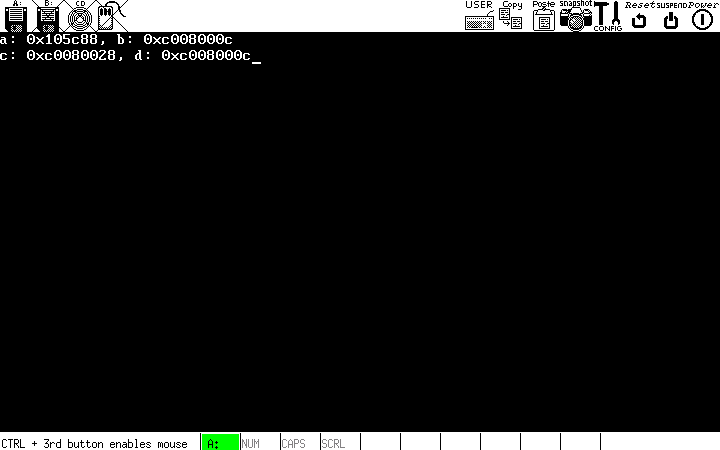
   // Now allocate those pages we mapped earlier.  
   for (i = KHEAP\_START; i < KHEAP\_START+KHEAP\_INITIAL\_SIZE; i += 0x1000)  
       alloc\_frame( get\_page(i, 1, kernel\_directory), 0, 0);  
  
   // Before we enable paging, we must register our page fault handler.  
   register\_interrupt\_handler(14, page\_fault);  
  
   // Now, enable paging!  
   switch\_page\_directory(kernel\_directory);  
  
   // Initialise the kernel heap.  
   kheap = create\_heap(KHEAP\_START, KHEAP\_START+KHEAP\_INITIAL\_SIZE, 0xCFFFF000, 0, 0);

Et voila! you are complete! A nice thing to do (which I have done in my sample code) is to get kmalloc/kfree to pass calls straight through to alloc/free if kheap != 0. I'll leave that to you to do ;)

## 7.5. Testing

*main.c*

u32int a = kmalloc(8);  
initialise\_paging();  
u32int b = kmalloc(8);  
u32int c = kmalloc(8);  
monitor\_write("a: ");  
monitor\_write\_hex(a);  
monitor\_write(", b: ");  
monitor\_write\_hex(b);  
monitor\_write("\nc: ");  
monitor\_write\_hex(c);  
  
kfree(c);  
kfree(b);  
u32int d = kmalloc(12);  
monitor\_write(", d: ");  
monitor\_write\_hex(d);

  
Success!

You can, of course, experiment with the order of allocations and frees here. The code above will allocate one variable, *a*, before initialise\_paging is called, so it'll be allocated via placement address. *b* and *c* both get allocated on the heap, and printed out. They are then both freed and another variable, *d*, created. If the address of *d* is the same as the address of *b*, then the space reclaimed by *b* and *c* has been successfully unified and all is good!

## 7.6. Summary

Dynamic memory allocation is one of the few things that it is very difficult to do without. Without it, you would have to specify an absolute maximum number of processes running (static array of pids), you would have to statically give the size of every buffer - Generally making your OS lacklustre and woefully inefficient.

Sample code, as ever, can be found [here](http://www.jamesmolloy.co.uk/downloads/the_heap.tar.gz).

***Copyright James Molloy 2008 - james<at>jamesmolloy.co.uk***

www.jamesmolloy.co.uk

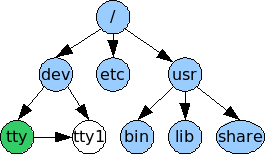
[**Home**](http://www.jamesmolloy.co.uk/index.html) » JamesM's kernel development tutorials

[1. Environment setup](http://www.jamesmolloy.co.uk/tutorial_html/1.-Environment%20setup.html)  
[2. Genesis](http://www.jamesmolloy.co.uk/tutorial_html/2.-Genesis.html)  
[3. The Screen](http://www.jamesmolloy.co.uk/tutorial_html/3.-The%20Screen.html)  
[4. The GDT and IDT](http://www.jamesmolloy.co.uk/tutorial_html/4.-The%20GDT%20and%20IDT.html)  
[5. IRQs and the PIT](http://www.jamesmolloy.co.uk/tutorial_html/5.-IRQs%20and%20the%20PIT.html)  
[6. Paging](http://www.jamesmolloy.co.uk/tutorial_html/6.-Paging.html)  
[7. The Heap](http://www.jamesmolloy.co.uk/tutorial_html/7.-The%20Heap.html)  
[**8. The VFS and the initrd**](http://www.jamesmolloy.co.uk/tutorial_html/8.-The%20VFS%20and%20the%20initrd.html)  
[9. Multitasking](http://www.jamesmolloy.co.uk/tutorial_html/9.-Multitasking.html)  
[10. User Mode](http://www.jamesmolloy.co.uk/tutorial_html/10.-User%20Mode.html)

# 8. The VFS and the initrd

In this chapter we're going to be starting work on our virtual filesystem (VFS). As a baptism of fire, we will also be implementing an initial ramdisk so you can load configuration files or executables to your kernel.

## 8.1. The virtual filesystem

  
Graph of nodes.

A VFS is intended to abstract away details of the filesystem and location that files are stored, and to give access to them in a uniform manner. They are usually implemented as a graph of nodes; Each node representing either a file, directory, symbolic link, device, socket or pipe. Each node should know what filesystem it belongs to and have enough information such that the relavent open/close/etc functions in its driver can be found and executed. A common way to accomplish this is to have the node store function pointers which can be called by the kernel. We'll need a few function pointers:

* *Open* - Called when a node is opened as a file descriptor.
* *Close* - Called when the node is closed.
* *Read* - I should hope this was self explanatory!
* *Write* - Same as above :-)
* *Readdir* - If the current node is a directory, we need a way of enumerating it's contents. Readdir should return the n'th child node of a directory or NULL otherwise. It returns a 'struct dirent', which is compatible with the UNIX readdir function.
* *Finddir* - We also need a way of finding a child node, given a name in string form. This will be used when following absolute pathnames.

So far then our node structure looks something like:

typedef struct fs\_node  
{  
  char name[128];     // The filename.  
  u32int flags;       // Includes the node type (Directory, file etc).  
  read\_type\_t read;   // These typedefs are just function pointers. We'll define them later!  
  write\_type\_t write;  
  open\_type\_t open;  
  close\_type\_t close;  
  readdir\_type\_t readdir; // Returns the n'th child of a directory.  
  finddir\_type\_t finddir; // Try to find a child in a directory by name.  
} fs\_node\_t;

Obviously we need to store the filename, and flags contains the type of the node (directory, symlink etc), but we are still missing things. We need to know what permissions the file has, which user/group it belongs to, and possibly also its length.

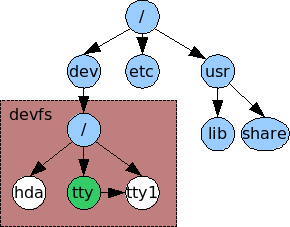
typedef struct fs\_node  
{  
   char name[128];     // The filename.  
   u32int mask;        // The permissions mask.  
   u32int uid;         // The owning user.  
   u32int gid;         // The owning group.  
   u32int flags;       // Includes the node type.  
   u32int length;      // Size of the file, in bytes.  
   read\_type\_t read;  
   write\_type\_t write;  
   open\_type\_t open;  
   close\_type\_t close;  
   readdir\_type\_t readdir;  
   finddir\_type\_t finddir;  
} fs\_node\_t;

Again though, we are still missing things! We need a way for the filesystem driver to track which node is which. This is commonly known as an *inode*. It is just a number assigned by the driver which uniquely represents this file. Not only that, but we may have *multiple instances of the same filesystem type*, so we must also have a variable that the driver can set to track which filesystem instance it belongs to.

Lastly we also need to account for symbolic links (shortcuts in Windows-speak). These are merely pointers or placeholders for other files, and so need a pointer member variable.

typedef struct fs\_node  
{  
   char name[128];     // The filename.  
   u32int mask;        // The permissions mask.  
   u32int uid;         // The owning user.  
   u32int gid;         // The owning group.  
   u32int flags;       // Includes the node type. See #defines above.  
   u32int inode;       // This is device-specific - provides a way for a filesystem to identify files.  
   u32int length;      // Size of the file, in bytes.  
   u32int impl;        // An implementation-defined number.  
   read\_type\_t read;  
   write\_type\_t write;  
   open\_type\_t open;  
   close\_type\_t close;  
   readdir\_type\_t readdir;  
   finddir\_type\_t finddir;  
   struct fs\_node \*ptr; // Used by mountpoints and symlinks.  
} fs\_node\_t;

### 8.1.1. Mountpoints

  
devfs mounted on /dev

Mountpoints are the UNIX way of accessing different filesystems. A filesystem is mounted on a directory - any subsequent access to that directory will actually access the root directory of the new filesystem. So essentially the directory is told that it is a mountpoint and given a pointer to the root node of the new filesystem. We can actually reuse the *ptr* member of fs\_node\_t for this purpose (as it is currently only used for symlinks and they can never be mountpoints).

### 8.1.2. Implementation

#### 8.1.2.1. fs.h

We first need to define the prototypes for our read/write/etc functions. The first four can be gained by looking at the [POSIX specification](http://www.opengroup.org/onlinepubs/009695399/toc.htm). The other two can just be made up :-)

typedef u32int (\*read\_type\_t)(struct fs\_node\*,u32int,u32int,u8int\*);  
typedef u32int (\*write\_type\_t)(struct fs\_node\*,u32int,u32int,u8int\*);  
typedef void (\*open\_type\_t)(struct fs\_node\*);  
typedef void (\*close\_type\_t)(struct fs\_node\*);  
typedef struct dirent \* (\*readdir\_type\_t)(struct fs\_node\*,u32int);  
typedef struct fs\_node \* (\*finddir\_type\_t)(struct fs\_node\*,char \*name);

struct dirent // One of these is returned by the readdir call, according to POSIX.  
{  
  char name[128]; // Filename.  
  u32int ino;     // Inode number. Required by POSIX.  
};

We also need to define what the values in the fs\_node\_t::flags field mean:

#define FS\_FILE        0x01  
#define FS\_DIRECTORY   0x02  
#define FS\_CHARDEVICE  0x03  
#define FS\_BLOCKDEVICE 0x04  
#define FS\_PIPE        0x05  
#define FS\_SYMLINK     0x06  
#define FS\_MOUNTPOINT  0x08 // Is the file an active mountpoint?

Notice that *FS\_MOUNTPOINT* is given the value 8, not 7. This is so that it can be bitwise-OR'd in with FS\_DIRECTORY. The other flags are given sequential values as they are mutually exclusive.

Lastly we need to define the root node of the filesystem and our read/write/etc functions.

extern fs\_node\_t \*fs\_root; // The root of the filesystem.  
  
// Standard read/write/open/close functions. Note that these are all suffixed with  
// \_fs to distinguish them from the read/write/open/close which deal with file descriptors  
// not file nodes.  
u32int read\_fs(fs\_node\_t \*node, u32int offset, u32int size, u8int \*buffer);  
u32int write\_fs(fs\_node\_t \*node, u32int offset, u32int size, u8int \*buffer);  
void open\_fs(fs\_node\_t \*node, u8int read, u8int write);  
void close\_fs(fs\_node\_t \*node);  
struct dirent \*readdir\_fs(fs\_node\_t \*node, u32int index);  
fs\_node\_t \*finddir\_fs(fs\_node\_t \*node, char \*name);

#### 8.1.2.2. fs.c

// fs.c -- Defines the interface for and structures relating to the virtual file system.  
// Written for JamesM's kernel development tutorials.  
  
#include "fs.h"  
  
fs\_node\_t \*fs\_root = 0; // The root of the filesystem.  
  
u32int read\_fs(fs\_node\_t \*node, u32int offset, u32int size, u8int \*buffer)  
{  
  // Has the node got a read callback?  
  if (node->read != 0)  
    return node->read(node, offset, size, buffer);  
  else  
    return 0;  
}

The above code should really be self-explanatory. If the node doesn't have a callback set, just return an error value. You should replicate the above code for open(), close() and write(). The same is true of readdir() and finddir(), although in those there should be an extra check: If the node is actually a directory!

if ((node->flags&0x7) == FS\_DIRECTORY && node->readdir != 0 )

Believe it or not, that is all the code that is needed to make a simple virtual filesystem! With this code as a base we can make our initial ramdisk and maybe later more complex filesystems like FAT or ext2.

## 8.2. The initial ramdisk

An initial ramdisk is just a filesystem that is loaded into memory when the kernel boots. It is useful for storing drivers and configuration files that are needed before the kernel can access the root filesystem (indeed, it usually contains the driver to access that root filesystem!).

An *initrd*, as they are known, usually uses a propriatary filesystem format. The reason for this is that the most complex thing a filesystem has to handle, *deletion of files and reclaimation of space*, isn't necessary. The kernel should try to get the root filesystem up and running as quick as possible - why would it want to delete files from the initrd??

As such you can just make a filesystem format up! I've made one for you as well, if you're not feeling very creative ;)

## 8.3. My own solution

My format does not support subdirectories. It stores the number of files in the system as the first 4 bytes of the initrd file. That is followed by a set number (64) of header structures, giving the names, offsets and sizes of the files contained. The actual file data follows. I have written a small C program to make this for me: it takes two arguments for each file to add: The path to the file from the current directory and the name to give the file in the generated filesystem.

### 8.3.1. Filesystem generator

#include <stdio.h>  
  
struct initrd\_header  
{  
   unsigned char magic; // The magic number is there to check for consistency.  
   char name[64];  
   unsigned int offset; // Offset in the initrd the file starts.  
   unsigned int length; // Length of the file.  
};  
  
int main(char argc, char \*\*argv)  
{  
   int nheaders = (argc-1)/2;  
   struct initrd\_header headers[64];  
   printf("size of header: %d\n", sizeof(struct initrd\_header));  
   unsigned int off = sizeof(struct initrd\_header) \* 64 + sizeof(int);  
   int i;  
   for(i = 0; i < nheaders; i++)  
   {  
       printf("writing file %s->%s at 0x%x\n", argv[i\*2+1], argv[i\*2+2], off);  
       strcpy(headers[i].name, argv[i\*2+2]);  
       headers[i].offset = off;  
       FILE \*stream = fopen(argv[i\*2+1], "r");  
       if(stream == 0)  
       {  
         printf("Error: file not found: %s\n", argv[i\*2+1]);  
         return 1;  
       }  
       fseek(stream, 0, SEEK\_END);  
       headers[i].length = ftell(stream);  
       off += headers[i].length;  
       fclose(stream);  
       headers[i].magic = 0xBF;  
   }  
     
   FILE \*wstream = fopen("./initrd.img", "w");  
   unsigned char \*data = (unsigned char \*)malloc(off);  
   fwrite(&nheaders, sizeof(int), 1, wstream);  
   fwrite(headers, sizeof(struct initrd\_header), 64, wstream);  
     
   for(i = 0; i < nheaders; i++)  
   {  
     FILE \*stream = fopen(argv[i\*2+1], "r");  
     unsigned char \*buf = (unsigned char \*)malloc(headers[i].length);  
     fread(buf, 1, headers[i].length, stream);  
     fwrite(buf, 1, headers[i].length, wstream);  
     fclose(stream);  
     free(buf);  
   }  
     
   fclose(wstream);  
   free(data);  
     
   return 0;  
}

*I'm not going to explain the contents of this file: It is auxiliary and not important. Besides, you should be making your own anyway! ;)*

### 8.3.2. Integrating it in to your own OS

*Even if you are using a different file format to mine, this section may be useful in helping you integrate it into the kernel.*

#### 8.3.2.1. initrd.h

This file just defines the header structure types and gives a function prototype for the *initialise\_initrd* function so the kernel can call it.

// initrd.h -- Defines the interface for and structures relating to the initial ramdisk.  
// Written for JamesM's kernel development tutorials.  
  
#ifndef INITRD\_H  
#define INITRD\_H  
  
#include "common.h"  
#include "fs.h"  
  
typedef struct  
{  
   u32int nfiles; // The number of files in the ramdisk.  
} initrd\_header\_t;  
  
typedef struct  
{  
   u8int magic;     // Magic number, for error checking.  
   s8int name[64];  // Filename.  
   u32int offset;   // Offset in the initrd that the file starts.  
   u32int length;   // Length of the file.  
} initrd\_file\_header\_t;  
  
// Initialises the initial ramdisk. It gets passed the address of the multiboot module,  
// and returns a completed filesystem node.  
fs\_node\_t \*initialise\_initrd(u32int location);  
  
#endif

#### 8.3.2.2. initrd.c

The first thing we need is some static declarations:

// initrd.c -- Defines the interface for and structures relating to the initial ramdisk.  
// Written for JamesM's kernel development tutorials.  
  
#include "initrd.h"  
  
initrd\_header\_t \*initrd\_header;     // The header.  
initrd\_file\_header\_t \*file\_headers; // The list of file headers.  
fs\_node\_t \*initrd\_root;             // Our root directory node.  
fs\_node\_t \*initrd\_dev;              // We also add a directory node for /dev, so we can mount devfs later on.  
fs\_node\_t \*root\_nodes;              // List of file nodes.  
int nroot\_nodes;                    // Number of file nodes.  
  
struct dirent dirent;

The next thing we need is a function to read from a file in our initrd.

static u32int initrd\_read(fs\_node\_t \*node, u32int offset, u32int size, u8int \*buffer)  
{  
   initrd\_file\_header\_t header = file\_headers[node->inode];  
   if (offset > header.length)  
       return 0;  
   if (offset+size > header.length)  
       size = header.length-offset;  
   memcpy(buffer, (u8int\*) (header.offset+offset), size);  
   return size;  
}

That function demonstrates one very annoying thing about writing low level code: 80% of it is error-checking. Unfortunately you can't get away from it - if you leave it out you will spend literally days trying to work out why your code doesn't work.

It would also be quite useful to have some working readdir and finddir functions:

static struct dirent \*initrd\_readdir(fs\_node\_t \*node, u32int index)  
{  
   if (node == initrd\_root && index == 0)  
   {  
     strcpy(dirent.name, "dev");  
     dirent.name[3] = 0; // Make sure the string is NULL-terminated.  
     dirent.ino = 0;  
     return &dirent;  
   }  
  
   if (index-1 >= nroot\_nodes)  
       return 0;  
   strcpy(dirent.name, root\_nodes[index-1].name);  
   dirent.name[strlen(root\_nodes[index-1].name)] = 0; // Make sure the string is NULL-terminated.  
   dirent.ino = root\_nodes[index-1].inode;  
   return &dirent;  
}  
  
static fs\_node\_t \*initrd\_finddir(fs\_node\_t \*node, char \*name)  
{  
   if (node == initrd\_root &&  
       !strcmp(name, "dev") )  
       return initrd\_dev;  
  
   int i;  
   for (i = 0; i < nroot\_nodes; i++)  
       if (!strcmp(name, root\_nodes[i].name))  
           return &root\_nodes[i];  
   return 0;  
}

Last but not least we need to initialise the filesystem:

fs\_node\_t \*initialise\_initrd(u32int location)  
{  
   // Initialise the main and file header pointers and populate the root directory.  
   initrd\_header = (initrd\_header\_t \*)location;  
   file\_headers = (initrd\_file\_header\_t \*) (location+sizeof(initrd\_header\_t));

We assume that the kernel knows where our initrd starts and can convey that location to the initialise function.

   // Initialise the root directory.  
   initrd\_root = (fs\_node\_t\*)kmalloc(sizeof(fs\_node\_t));  
   strcpy(initrd\_root->name, "initrd");  
   initrd\_root->mask = initrd\_root->uid = initrd\_root->gid = initrd\_root->inode = initrd\_root->length = 0;  
   initrd\_root->flags = FS\_DIRECTORY;  
   initrd\_root->read = 0;  
   initrd\_root->write = 0;  
   initrd\_root->open = 0;  
   initrd\_root->close = 0;  
   initrd\_root->readdir = &initrd\_readdir;  
   initrd\_root->finddir = &initrd\_finddir;  
   initrd\_root->ptr = 0;  
   initrd\_root->impl = 0;

Here we make the root directory node. We get some memory from the kernel heap and give the node a name. We really don't need to name this node as the root is never referenced by name, just '/'.

Most of the code initialises pointers to NULL (0), but you'll notice that the node is told it is a directory (flags = FS\_DIRECTORY) and that it has both readdir and finddir functions.

The same is done for the /dev node:

   // Initialise the /dev directory (required!)  
   initrd\_dev = (fs\_node\_t\*)kmalloc(sizeof(fs\_node\_t));  
   strcpy(initrd\_dev->name, "dev");  
   initrd\_dev->mask = initrd\_dev->uid = initrd\_dev->gid = initrd\_dev->inode = initrd\_dev->length = 0;  
   initrd\_dev->flags = FS\_DIRECTORY;  
   initrd\_dev->read = 0;  
   initrd\_dev->write = 0;  
   initrd\_dev->open = 0;  
   initrd\_dev->close = 0;  
   initrd\_dev->readdir = &initrd\_readdir;  
   initrd\_dev->finddir = &initrd\_finddir;  
   initrd\_dev->ptr = 0;  
   initrd\_dev->impl = 0;

Now that they're done we can start actually adding the files in the ramdisk. First we allocate space for them:

   root\_nodes = (fs\_node\_t\*)kmalloc(sizeof(fs\_node\_t) \* initrd\_header->nfiles);  
   nroot\_nodes = initrd\_header->nfiles;

Then we make them:

   // For every file...  
   int i;  
   for (i = 0; i < initrd\_header->nfiles; i++)  
   {  
       // Edit the file's header - currently it holds the file offset  
       // relative to the start of the ramdisk. We want it relative to the start  
       // of memory.  
       file\_headers[i].offset += location;  
       // Create a new file node.  
       strcpy(root\_nodes[i].name, &file\_headers[i].name);  
       root\_nodes[i].mask = root\_nodes[i].uid = root\_nodes[i].gid = 0;  
       root\_nodes[i].length = file\_headers[i].length;  
       root\_nodes[i].inode = i;  
       root\_nodes[i].flags = FS\_FILE;  
       root\_nodes[i].read = &initrd\_read;  
       root\_nodes[i].write = 0;  
       root\_nodes[i].readdir = 0;  
       root\_nodes[i].finddir = 0;  
       root\_nodes[i].open = 0;  
       root\_nodes[i].close = 0;  
       root\_nodes[i].impl = 0;  
   }

And finally return the root node so the kernel can access us:

   return initrd\_root;  
}

## 8.4. Loading the initrd as a multiboot module

Now we need to work out how to get our initrd loaded into memory in the first place. Luckily, the multiboot specification allows for 'modules' to be loaded. We can tell GRUB to load our initrd as a module. You can do this by mounting the floppy.img file as a loopback device, finding the /boot/grub/menu.lst file and adding a 'module (fd0)/initrd' line just below the 'kernel' line.

Alternatively you can download a new and improved image from [here](http://www.jamesmolloy.co.uk/downloads/floppy_module.img).

GRUB communicates the location of this file to us via the multiboot information structure that we declared but never defined in the first tutorial. We have to define it now: This definition is lifted directly from the [Multiboot spec](http://www.gnu.org/software/grub/manual/multiboot/multiboot.html).

*multiboot.h*

#include "common.h"  
  
#define MULTIBOOT\_FLAG\_MEM     0x001  
#define MULTIBOOT\_FLAG\_DEVICE  0x002  
#define MULTIBOOT\_FLAG\_CMDLINE 0x004  
#define MULTIBOOT\_FLAG\_MODS    0x008  
#define MULTIBOOT\_FLAG\_AOUT    0x010  
#define MULTIBOOT\_FLAG\_ELF     0x020  
#define MULTIBOOT\_FLAG\_MMAP    0x040  
#define MULTIBOOT\_FLAG\_CONFIG  0x080  
#define MULTIBOOT\_FLAG\_LOADER  0x100  
#define MULTIBOOT\_FLAG\_APM     0x200  
#define MULTIBOOT\_FLAG\_VBE     0x400  
  
struct multiboot  
{  
   u32int flags;  
   u32int mem\_lower;  
   u32int mem\_upper;  
   u32int boot\_device;  
   u32int cmdline;  
   u32int mods\_count;  
   u32int mods\_addr;  
   u32int num;  
   u32int size;  
   u32int addr;  
   u32int shndx;  
   u32int mmap\_length;  
   u32int mmap\_addr;  
   u32int drives\_length;  
   u32int drives\_addr;  
   u32int config\_table;  
   u32int boot\_loader\_name;  
   u32int apm\_table;  
   u32int vbe\_control\_info;  
   u32int vbe\_mode\_info;  
   u32int vbe\_mode;  
   u32int vbe\_interface\_seg;  
   u32int vbe\_interface\_off;  
   u32int vbe\_interface\_len;  
}  \_\_attribute\_\_((packed));  
  
typedef struct multiboot\_header multiboot\_header\_t;

The interesting fields are the *mods\_addr* and *mods\_count* fields. The *mods\_count* field contains the number of modules loaded. We should check that this is > 0. The *mods\_addr* field is an array of addresses: Each 'entry' consists of the starting address of the module and it's end, each being 4 bytes.

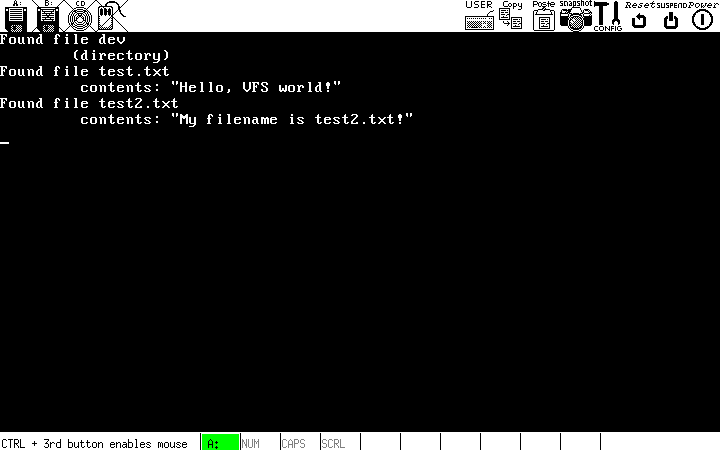
As we are only expecting one module we can just treat the *mods\_addr* field as a pointer and find whatever value lies there. That will be the location of our initrd. The value of the address 4 bytes on from that will be the end address. We can use this to change the memory management placement address so that memory allocations don't accidentally overwrite our ramdisk!

*main.c*

int main(struct multiboot \*mboot\_ptr)  
{  
   // Initialise all the ISRs and segmentation  
   init\_descriptor\_tables();  
   // Initialise the screen (by clearing it)  
   monitor\_clear();  
  
   // Find the location of our initial ramdisk.  
   ASSERT(mboot\_ptr->mods\_count > 0);  
   u32int initrd\_location = \*((u32int\*)mboot\_ptr->mods\_addr);  
   u32int initrd\_end = \*(u32int\*)(mboot\_ptr->mods\_addr+4);  
   // Don't trample our module with placement accesses, please!  
   placement\_address = initrd\_end;  
  
   // Start paging.  
   initialise\_paging();  
  
   // Initialise the initial ramdisk, and set it as the filesystem root.  
   fs\_root = initialise\_initrd(initrd\_location);  
}

Success! That's one VFS and initrd cooked up in no time. Let's test it out.

## 8.5. Testing it out

  
Success!

Firstly let's add some test code to find all files in '/' and print their contents:

*main.c*

// list the contents of /  
int i = 0;  
struct dirent \*node = 0;  
while ( (node = readdir\_fs(fs\_root, i)) != 0)  
{  
  monitor\_write("Found file ");  
  monitor\_write(node->name);  
  fs\_node\_t \*fsnode = finddir\_fs(fs\_root, node->name);  
  
  if ((fsnode->flags&0x7) == FS\_DIRECTORY)  
    monitor\_write("\n\t(directory)\n");  
  else  
  {  
    monitor\_write("\n\t contents: \"");  
    char buf[256];  
    u32int sz = read\_fs(fsnode, 0, 256, buf);  
    int j;  
    for (j = 0; j < sz; j++)  
      monitor\_put(buf[j]);  
  
    monitor\_write("\"\n");  
  }  
  i++;  
}

Make a couple of test files, and build!

./make\_initrd test.txt test.txt test2.txt test2.txt  
cd src  
make clean  
make  
cd ..  
./update\_image.sh  
./run\_bochs.sh

The code for this tutorial can be found [here](http://www.jamesmolloy.co.uk/downloads/the_vfs_and_initrd.tar.gz).

***Copyright James Molloy 2008 - james<at>jamesmolloy.co.uk***

www.jamesmolloy.co.uk

[**Home**](http://www.jamesmolloy.co.uk/index.html) » JamesM's kernel development tutorials

[1. Environment setup](http://www.jamesmolloy.co.uk/tutorial_html/1.-Environment%20setup.html)  
[2. Genesis](http://www.jamesmolloy.co.uk/tutorial_html/2.-Genesis.html)  
[3. The Screen](http://www.jamesmolloy.co.uk/tutorial_html/3.-The%20Screen.html)  
[4. The GDT and IDT](http://www.jamesmolloy.co.uk/tutorial_html/4.-The%20GDT%20and%20IDT.html)  
[5. IRQs and the PIT](http://www.jamesmolloy.co.uk/tutorial_html/5.-IRQs%20and%20the%20PIT.html)  
[6. Paging](http://www.jamesmolloy.co.uk/tutorial_html/6.-Paging.html)  
[7. The Heap](http://www.jamesmolloy.co.uk/tutorial_html/7.-The%20Heap.html)  
[8. The VFS and the initrd](http://www.jamesmolloy.co.uk/tutorial_html/8.-The%20VFS%20and%20the%20initrd.html)  
**[9. Multitasking](http://www.jamesmolloy.co.uk/tutorial_html/9.-Multitasking.html)**  
[10. User Mode](http://www.jamesmolloy.co.uk/tutorial_html/10.-User%20Mode.html)

# 9. Multitasking

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'.

## 9.1. Tasking theory

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.

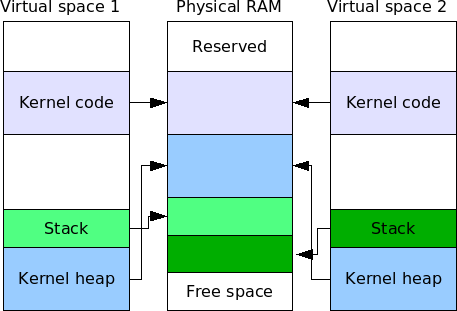
*It should be noted that in more advanced operating systems a process' timeslice will normally also be terminated when it performs a synchronous I/O operation, and in such operating systems (all but the most trivial) this is the normal case.*

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.

This relies on several things:

1. *All the general purpose registers are already saved.* This happens in the IRQ handler, so is automatic.
2. *The task switch code can be run seamlessly when changing address spaces.* The task switch code should be able to change address spaces and then continue executing as if nothing happened. This means that the kernel code must be mapped in at the same place in all address spaces.

### 9.1.1. Some notes about address spaces

  
Address space layout

A lot of the complication in implementing multitasking is not just the context switching - a new address space must be created for each task. The complication is that some parts of the address space must be copied, and others must be linked. That is, two pages point to the same frame in physical memory. Take the example layout on the right - The picture shows two virtual address spaces and how areas are mapped to an example physical RAM layout.

The stack is indicative of most areas in a virtual address space: it is copied when a new process is forked, so that if the new process changes data the old process doesn't see the change. When we load executables, this will also be the case for the executable code and data.

The kernel code and heap areas are slightly different - both areas in both virtual memory spaces map to the same two areas of physical memory. Firstly there is no point in copying the kernel code as it will never change, and secondly it is important that the kernel heap is consistent in all address spaces - if task 1 does a system call and causes some data to be changed the kernel must be able to pick that up in task 2's address space.

## 9.2. Cloning an address space

So, as mentioned above, one of the most complex things we need to do is to create a copy of an address space - so let's get that over with first.

### 9.2.1. Cloning a directory

First off we need to create a new directory. We use our kmalloc\_ap function to obtain an address aligned on a page boundary and to also retrieve the physical address. We then have to ensure that it is completely blank (each entry is initially zero).

page\_directory\_t \*clone\_directory(page\_directory\_t \*src)  
{  
   u32int phys;  
   // Make a new page directory and obtain its physical address.  
   page\_directory\_t \*dir = (page\_directory\_t\*)kmalloc\_ap(sizeof(page\_directory\_t), &phys);  
   // Ensure that it is blank.  
   memset(dir, 0, sizeof(page\_directory\_t));

We now have a new page directory, and the physical address at which it is located. However, for loading into the CR3 register, we need the physical address of the *tablesPhysical* member (remember that the physical address of a directory's page tables are held in *tablesPhysical*. See chapter 6). In order to do this, we perform a simple calculation. We get the offset of the *tablesPhysical* member from the start of the page\_directory\_t struct, then add that to the obtained physical address.

   // Get the offset of tablesPhysical from the start of the page\_directory\_t structure.  
   u32int offset = (u32int)dir->tablesPhysical - (u32int)dir;  
  
   // Then the physical address of dir->tablesPhysical is:  
   dir->physicalAddr = phys + offset;

Now we're ready to copy each page table. If the page table is zero, we don't need to bother copying anything.

   int i;  
   for (i = 0; i < 1024; i++)  
   {  
       if (!src->tables[i])  
           continue;

Now we need a method of working out whether we should *link* a page table or *copy* it. Remember that we want to link the kernel code and heap, and copy everything else. Luckily, we already have a very simple method of finding out. The global variable *kernel\_directory* is the first page directory we create. We identity map the kernel code and data, and map in the kernel heap all in this directory. Up until now, we've finished off the *initialise\_paging* function with this:

current\_directory == kernel\_directory;

But, if instead we set current\_directory to a *clone* of the kernel\_directory, kernel\_directory will remain constant, just containing the kernel code/data and the kernel heap. All modifications will be made to the clone, and not the original. This means that in our *clone* function we can compare page tables with the kernel\_directory. If a page table in the directory we are cloning is *also in the kernel\_directory*, we can assume that that page table should be linked. If not, it should be copied. Simple!

        if (kernel\_directory->tables[i] == src->tables[i])  
        {  
           // It's in the kernel, so just use the same pointer.  
           dir->tables[i] = src->tables[i];  
           dir->tablesPhysical[i] = src->tablesPhysical[i];  
        }  
        else  
        {  
           // Copy the table.  
           u32int phys;  
           dir->tables[i] = clone\_table(src->tables[i], &phys);  
           dir->tablesPhysical[i] = phys | 0x07;  
        }

Let's quickly go through that code segment. If the current page table is the same in the kernel directory and in the current directory, we link it - that is, in the new directory, we set the page table pointer to be the same as in the source directory. We also copy the physical address of this page table (very important - this is the address that matters to the processor). If, instead, we need to copy the table, we use an (as yet) undefined function called *clone\_table*, which returns a virtual pointer to a page table, and stores its physical address in a passed-in argument. When setting the tablesPhysical pointer, we bitwise-OR the physical address with 0x07, which means "Present, Read-write, user-mode".

Let's just quickly end this function, then we can go on to define *clone\_table*.

   }  
   return dir;

}

### 9.2.2. Cloning a table

To clone a page table, we have to do something similar to above, with some changes. We never have to choose whether to copy or link table entries - we always copy. We also have to copy the data in the page table entries.

static page\_table\_t \*clone\_table(page\_table\_t \*src, u32int \*physAddr)  
{  
   // Make a new page table, which is page aligned.  
   page\_table\_t \*table = (page\_table\_t\*)kmalloc\_ap(sizeof(page\_table\_t), physAddr);  
   // Ensure that the new table is blank.  
   memset(table, 0, sizeof(page\_directory\_t));  
  
   // For every entry in the table...  
   int i;  
   for (i = 0; i < 1024; i++)  
   {  
     if (!src->pages[i].frame)  
       continue;

The preamble for this function is exactly the same as in *clone\_directory*.

So, for every page table entry in the table, we need to:

* Allocate ourselves a new frame to hold the copied data.
* Copy the flags - read/write, present, user-mode etc.
* Physically copy the data.

       // Get a new frame.  
       alloc\_frame(&table->pages[i], 0, 0);  
       // Clone the flags from source to destination.  
       if (src->pages[i].present) table->pages[i].present = 1;  
       if (src->pages[i].rw)      table->pages[i].rw = 1;  
       if (src->pages[i].user)    table->pages[i].user = 1;  
       if (src->pages[i].accessed)table->pages[i].accessed = 1;  
       if (src->pages[i].dirty)   table->pages[i].dirty = 1;  
       // Physically copy the data across. This function is in process.s.  
       copy\_page\_physical(src->pages[i].frame\*0x1000, table->pages[i].frame\*0x1000);

All fairly simple. We use a function which (again) is as yet undefined, called copy\_page\_physical. We'll define that in a second, just after we end this function.

   }  
   return table;  
}

### 9.2.3. Copying a physical frame

*copy\_page\_physical* is really a misnomer. What we actually want to do is copy the contents of one *frame* into another *frame*. This, unfortunately, involves disabling paging (so we can access all of physical RAM), so we write this as a pure assembler function. This should go in a file called 'process.s'.

[GLOBAL copy\_page\_physical]  
copy\_page\_physical:  
   push ebx              ; According to \_\_cdecl, we must preserve the contents of EBX.  
   pushf                 ; push EFLAGS, so we can pop it and reenable interrupts  
                         ; later, if they were enabled anyway.  
   cli                   ; Disable interrupts, so we aren't interrupted.  
                         ; Load these in BEFORE we disable paging!  
   mov ebx, [esp+12]     ; Source address  
   mov ecx, [esp+16]     ; Destination address  
  
   mov edx, cr0          ; Get the control register...  
   and edx, 0x7fffffff   ; and...  
   mov cr0, edx          ; Disable paging.  
  
   mov edx, 1024         ; 1024\*4bytes = 4096 bytes to copy  
  
.loop:  
   mov eax, [ebx]        ; Get the word at the source address  
   mov [ecx], eax        ; Store it at the dest address  
   add ebx, 4            ; Source address += sizeof(word)  
   add ecx, 4            ; Dest address += sizeof(word)  
   dec edx               ; One less word to do  
   jnz .loop  
  
   mov edx, cr0          ; Get the control register again  
   or  edx, 0x80000000   ; and...  
   mov cr0, edx          ; Enable paging.  
  
   popf                  ; Pop EFLAGS back.  
   pop ebx               ; Get the original value of EBX back.  
   ret

Hopefully the comments should make it clear what is happening. Anyway, that's a directory successfully cloned! We should now add a call to this in *initialise\_paging*, as mentioned above.

void initialise\_paging()  
{  
   // The size of physical memory. For the moment we  
   // assume it is 16MB big.  
   u32int mem\_end\_page = 0x1000000;  
  
   nframes = mem\_end\_page / 0x1000;  
   frames = (u32int\*)kmalloc(INDEX\_FROM\_BIT(nframes));  
   memset(frames, 0, INDEX\_FROM\_BIT(nframes));  
  
   // Let's make a page directory.  
   u32int phys; // \*\*\*\*\*\*\*\*\*\* ADDED \*\*\*\*\*\*\*\*\*\*\*  
   kernel\_directory = (page\_directory\_t\*)kmalloc\_a(sizeof(page\_directory\_t));  
   memset(kernel\_directory, 0, sizeof(page\_directory\_t));  
   // \*\*\*\*\*\*\*\*\*\*\* MODIFIED \*\*\*\*\*\*\*\*\*\*\*\*  
   kernel\_directory->physicalAddr = (u32int)kernel\_directory->tablesPhysical;  
  
   // Map some pages in the kernel heap area.  
   // Here we call get\_page but not alloc\_frame. This causes page\_table\_t's  
   // to be created where necessary. We can't allocate frames yet because they  
   // they need to be identity mapped first below, and yet we can't increase  
   // placement\_address between identity mapping and enabling the heap!  
   int i = 0;  
   for (i = KHEAP\_START; i < KHEAP\_END; i += 0x1000)  
       get\_page(i, 1, kernel\_directory);  
  
   // We need to identity map (phys addr = virt addr) from  
   // 0x0 to the end of used memory, so we can access this  
   // transparently, as if paging wasn't enabled.  
   // NOTE that we use a while loop here deliberately.  
   // inside the loop body we actually change placement\_address  
   // by calling kmalloc(). A while loop causes this to be  
   // computed on-the-fly rather than once at the start.  
   // Allocate a lil' bit extra so the kernel heap can be  
   // initialised properly.  
   i = 0;  
   while (i < placement\_address+0x1000)  
   {  
       // Kernel code is readable but not writeable from userspace.  
       alloc\_frame( get\_page(i, 1, kernel\_directory), 0, 0);  
       i += 0x1000;  
   }  
  
   // Now allocate those pages we mapped earlier.  
   for (i = KHEAP\_START; i < KHEAP\_START+KHEAP\_INITIAL\_SIZE; i += 0x1000)  
       alloc\_frame( get\_page(i, 1, kernel\_directory), 0, 0);  
  
   // Before we enable paging, we must register our page fault handler.  
   register\_interrupt\_handler(14, page\_fault);  
  
   // Now, enable paging!  
   switch\_page\_directory(kernel\_directory);  
  
   // Initialise the kernel heap.  
   kheap = create\_heap(KHEAP\_START, KHEAP\_START+KHEAP\_INITIAL\_SIZE, 0xCFFFF000, 0, 0);  
  
   // \*\*\*\*\*\*\*\* ADDED \*\*\*\*\*\*\*\*\*  
   current\_directory = clone\_directory(kernel\_directory);  
   switch\_page\_directory(current\_directory);  
}  
  
void switch\_page\_directory(page\_directory\_t \*dir)  
{  
   current\_directory = dir;  
   asm volatile("mov %0, %%cr3":: "r"(dir->physicalAddr)); // \*\*\*\*\*\*\*\* MODIFIED \*\*\*\*\*\*\*\*\*  
   u32int cr0;  
   asm volatile("mov %%cr0, %0": "=r"(cr0));  
   cr0 |= 0x80000000; // Enable paging!  
   asm volatile("mov %0, %%cr0":: "r"(cr0));  
}

(It should be noted that a function prototype for *clone\_directory* should be put in the 'paging.h' header file.)

## 9.3. Creating a new stack

Currently, we have been using an undefined stack. What does that mean? well, GRUB leaves us, stack-wise, in an undefined state. The stack pointer could be anywhere. In all practical situations, GRUB's default stack location is large enough for our startup code to run without problems. However, it is in lower memory (somewhere around 0x7000 physical), which causes us problems as it'll be 'linked' instead of 'copied' when a page directory is changed (because the area from 0x0 - approx 0x150000 is mapped in the kernel\_directory). So, we really need to move the stack.

Moving the stack is not particularly difficult. We just memcpy() the data in the old stack over to where the new stack should be. However, there is a problem. When a new stack frame is created (for example, when entering a function) the EBP register is pushed onto the stack. This base pointer is used by the compiler to work out how to reference local variables. If we plainly copy the stack over, these pushed EBP values will point to locations on the *old* stack, not the new one! So we need to change them manually.

Unfortunately, first, we need to know exactly where the current stack starts! To do this, we have to add an instruction right at the start, in boot.s:

 ; Add this just before "push ebx".  
 push esp

This passes another parameter to main() - the initial stack pointer. We need to modify main() to take this extra parameter also:

u32int initial\_esp; // New global variable.

int main(struct multiboot \*mboot\_ptr, u32int initial\_stack)  
{  
   initial\_esp = initial\_stack;

Good. Now we have what we need to start moving the stack. The following function should be in a new file, "task.c".

void move\_stack(void \*new\_stack\_start, u32int size)  
{  
  u32int i;  
  // Allocate some space for the new stack.  
  for( i = (u32int)new\_stack\_start;  
       i >= ((u32int)new\_stack\_start-size);  
       i -= 0x1000)  
  {  
    // General-purpose stack is in user-mode.  
    alloc\_frame( get\_page(i, 1, current\_directory), 0 /\* User mode \*/, 1 /\* Is writable \*/ );  
  }

Now, we've changed a page table. So we need to inform the processor that a mapping has changed. This is called "Flushing the TLB (translation lookaside buffer)". It can be done partially, using the "invlpg" instruction, or fully, by simply writing to cr3. We choose the simpler latter option.

  // Flush the TLB by reading and writing the page directory address again.  
  u32int pd\_addr;  
  asm volatile("mov %%cr3, %0" : "=r" (pd\_addr));  
  asm volatile("mov %0, %%cr3" : : "r" (pd\_addr));

Next, we read the current stack and base pointers, and calculate an offset to get from an address on the old stack to an address on the new stack, and use it to calculate the new stack/base pointers.

 // Old ESP and EBP, read from registers.  
 u32int old\_stack\_pointer; asm volatile("mov %%esp, %0" : "=r" (old\_stack\_pointer));  
 u32int old\_base\_pointer;  asm volatile("mov %%ebp, %0" : "=r" (old\_base\_pointer));

 u32int offset            = (u32int)new\_stack\_start - initial\_esp;

 u32int new\_stack\_pointer = old\_stack\_pointer + offset;  
 u32int new\_base\_pointer  = old\_base\_pointer  + offset;

Great. Now we can actually copy the stack.

// Copy the stack.  
memcpy((void\*)new\_stack\_pointer, (void\*)old\_stack\_pointer, initial\_esp-old\_stack\_pointer);

Now we try and go through the new stack, looking for base pointers to change. Here we use an algorithm which is not fool-proof. We assume that any value on the stack which is in the range of the stack (old\_stack\_pointer < x < initial\_esp) is a pushed EBP. This will, unfortunately, completely trash any value which isn't an EBP but just happens to be in this range. Oh well, these things happen.

// Backtrace through the original stack, copying new values into  
// the new stack.  
for(i = (u32int)new\_stack\_start; i > (u32int)new\_stack\_start-size; i -= 4)  
{  
   u32int tmp = \* (u32int\*)i;  
   // If the value of tmp is inside the range of the old stack, assume it is a base pointer  
   // and remap it. This will unfortunately remap ANY value in this range, whether they are  
   // base pointers or not.  
   if (( old\_stack\_pointer < tmp) && (tmp < initial\_esp))  
   {  
     tmp = tmp + offset;  
     u32int \*tmp2 = (u32int\*)i;  
     \*tmp2 = tmp;  
   }  
}

Lastly we just need to actually change the stack and base pointers.

  // Change stacks.  
  asm volatile("mov %0, %%esp" : : "r" (new\_stack\_pointer));  
  asm volatile("mov %0, %%ebp" : : "r" (new\_base\_pointer));  
}

## 9.4. Actual multitasking code

Now that we've got the necessary support functions written, we can actually start writing some tasking code.

Firstly in task.h, we'll need some definitions.

//  
// task.h - Defines the structures and prototypes needed to multitask.  
// Written for JamesM's kernel development tutorials.  
//  
  
#ifndef TASK\_H  
#define TASK\_H  
  
#include "common.h"  
#include "paging.h"  
  
// This structure defines a 'task' - a process.  
typedef struct task  
{  
   int id;                // Process ID.  
   u32int esp, ebp;       // Stack and base pointers.  
   u32int eip;            // Instruction pointer.  
   page\_directory\_t \*page\_directory; // Page directory.  
   struct task \*next;     // The next task in a linked list.  
} task\_t;  
  
// Initialises the tasking system.  
void initialise\_tasking();  
  
// Called by the timer hook, this changes the running process.  
void task\_switch();  
  
// Forks the current process, spawning a new one with a different  
// memory space.  
int fork();  
  
// Causes the current process' stack to be forcibly moved to a new location.  
void move\_stack(void \*new\_stack\_start, u32int size);  
  
// Returns the pid of the current process.  
int getpid();  
  
#endif

We define a task structure, which contains a task ID (known as a PID), some saved registers, a pointer to a page directory, and the next task in the list (this is a singly linked list).

In task.c, we'll need some global variables, and we have a small initialise\_tasking function that just creates one, blank, task.

//  
// task.c - Implements the functionality needed to multitask.  
// Written for JamesM's kernel development tutorials.  
//  
  
#include "task.h"  
#include "paging.h"  
  
// The currently running task.  
volatile task\_t \*current\_task;  
  
// The start of the task linked list.  
volatile task\_t \*ready\_queue;  
  
// Some externs are needed to access members in paging.c...  
extern page\_directory\_t \*kernel\_directory;  
extern page\_directory\_t \*current\_directory;  
extern void alloc\_frame(page\_t\*,int,int);  
extern u32int initial\_esp;  
extern u32int read\_eip();  
  
// The next available process ID.  
u32int next\_pid = 1;  
  
void initialise\_tasking()  
{

   asm volatile("cli");  
  
   // Relocate the stack so we know where it is.  
   move\_stack((void\*)0xE0000000, 0x2000);  
  
   // Initialise the first task (kernel task)  
   current\_task = ready\_queue = (task\_t\*)kmalloc(sizeof(task\_t));  
   current\_task->id = next\_pid++;  
   current\_task->esp = current\_task->ebp = 0;  
   current\_task->eip = 0;  
   current\_task->page\_directory = current\_directory;  
   current\_task->next = 0;  
  
   // Reenable interrupts.  
   asm volatile("sti");  
}

Right. We only have two more functions to write - fork(), and switch\_task(). Fork() is a UNIX function to create a new process. It clones the address space and starts the new process running at the same place as the original process is currently at.

int fork()  
{  
   // We are modifying kernel structures, and so cannot be interrupted.  
   asm volatile("cli");  
  
   // Take a pointer to this process' task struct for later reference.  
   task\_t \*parent\_task = (task\_t\*)current\_task;  
  
   // Clone the address space.  
   page\_directory\_t \*directory = clone\_directory(current\_directory);

So firstly we disable interrupts, because we're changing kernel structures and could cause problems if we're interrupted half way through. We then clone the current page directory.

   // Create a new process.  
   task\_t \*new\_task = (task\_t\*)kmalloc(sizeof(task\_t));  
   new\_task->id = next\_pid++;  
   new\_task->esp = new\_task->ebp = 0;  
   new\_task->eip = 0;  
   new\_task->page\_directory = directory;  
   new\_task->next = 0;  
  
   // Add it to the end of the ready queue.  
   // Find the end of the ready queue...  
   task\_t \*tmp\_task = (task\_t\*)ready\_queue;  
   while (tmp\_task->next)  
       tmp\_task = tmp\_task->next;  
   // ...And extend it.  
   tmp\_task->next = new\_task;

Here we create a new process, just like in *initialise\_tasking*. We add it to the end of the ready queue (the queue of tasks that are ready to run). If you don't understand this code, I suggest you look up a tutorial on working with Singly Linked Lists.

We have to tell the task where it should start executing. For this, we need to read the current instruction pointer. We need a quick read\_eip() function to do this - this is in process.s:

[GLOBAL read\_eip]  
read\_eip:  
  pop eax  
  jmp eax

This is a rather clever way of reading the current instruction pointer. When read\_eip is called, the current instruction location is pushed onto the stack. Normally, we use "ret" to return from a function. This instruction pops the value from the stack and jumps to it. Here, however, we pop the value ourselves, into EAX (remember that EAX is the 'return value' register for the \_\_cdecl calling convention), then jump to it.

// This will be the entry point for the new process.  
u32int eip = read\_eip();

Important to note is that because (later) we set the new task's starting address to "eip", after the call to read\_eip we could be in one of two states.

1. We just called read\_eip, and are the parent task.
2. We are the child task, and just started executing.

To try and distinguish between the two cases, we check if "current\_task == parent\_task". In switch\_task(), we will add code which updates "current\_task" to always point to the currently running task. So, if we are the child task, current\_task will not be the same as parent\_task, else, it will.

   // We could be the parent or the child here - check.  
   if (current\_task == parent\_task)  
   {  
       // We are the parent, so set up the esp/ebp/eip for our child.  
       u32int esp; asm volatile("mov %%esp, %0" : "=r"(esp));  
       u32int ebp; asm volatile("mov %%ebp, %0" : "=r"(ebp));  
       new\_task->esp = esp;  
       new\_task->ebp = ebp;  
       new\_task->eip = eip;  
       // All finished: Reenable interrupts.  
       asm volatile("sti");

       return new\_task->id;  
   }  
   else  
   {  
       // We are the child - by convention return 0.  
       return 0;  
   }  
}

Let's just run through that code. If we are the parent task, we read the current stack pointer and base pointer values and store them into the new task's task\_struct. We also store the instruction pointer we found earlier in there, and reenable interrupts (because we've finished). Fork(), by convention, returns the PID of the child task if we are the parent, or zero if we are the child.

### 9.4.1. Switching tasks

Firstly we need to get the timer callback to call our scheduling function.

*In timer.c*

static void timer\_callback(registers\_t regs)  
{  
   tick++;  
   switch\_task();  
}

Now we just need to write it! ;)

void switch\_task()  
{  
   // If we haven't initialised tasking yet, just return.  
   if (!current\_task)  
       return;

Because this function will be called whenever the timer fires, it is very possible that it will be called before *initialise\_tasking* has been called. So we check that here - if the current task is NULL, we haven't set up tasking yet, so just return.

Next, lets just quickly grab the stack and base pointers - we'll need them in a minute.

// Read esp, ebp now for saving later on.  
u32int esp, ebp, eip;  
asm volatile("mov %%esp, %0" : "=r"(esp));  
asm volatile("mov %%ebp, %0" : "=r"(ebp));

Now it's time for some cunning logic. *Make sure you understand this piece of code. It's very important*. We read the instruction pointer, using our read\_eip function again. We'll put this value into the current task's "eip" field, so the next time it is scheduled, it picks up again at exactly the same location. However, just like in fork(), after the call we could be in one of two states:

1. We just called read\_eip, and it returned us the current instruction pointer.
2. We just switched tasks, and execution started just after the read\_eip function.

How do we distinguish between the two? Well, we can cheat. When we actually do the assembly to switch tasks (in a minute), we can plant a dummy value (I've used 0x12345) into EAX. Because C uses EAX as the return value of a function, in the second case the return value of read\_eip will *seem* to be 0x12345! So we can use that to distinguish between them.

   // Read the instruction pointer. We do some cunning logic here:  
   // One of two things could have happened when this function exits -  
   // (a) We called the function and it returned the EIP as requested.  
   // (b) We have just switched tasks, and because the saved EIP is essentially  
   // the instruction after read\_eip(), it will seem as if read\_eip has just  
   // returned.  
   // In the second case we need to return immediately. To detect it we put a dummy  
   // value in EAX further down at the end of this function. As C returns values in EAX,  
   // it will look like the return value is this dummy value! (0x12345).  
   eip = read\_eip();  
  
   // Have we just switched tasks?  
   if (eip == 0x12345)  
       return;

Next, we write the new ESP, EBP and EIP into the current task's task struct.

   // No, we didn't switch tasks. Let's save some register values and switch.  
   current\_task->eip = eip;  
   current\_task->esp = esp;  
   current\_task->ebp = ebp;

Then, we switch tasks! Advance through the current\_task listed list. If we fall off the end (if current\_task ends up being zero, we just start again).

   // Get the next task to run.  
   current\_task = current\_task->next;  
   // If we fell off the end of the linked list start again at the beginning.  
   if (!current\_task) current\_task = ready\_queue;

   esp = current\_task->esp;  
   ebp = current\_task->ebp;

The last three lines are just to make the assembly that follows a bit easier to understand.

The comments in this function really should explain everything. We change all the registers we need, then jump to the new instruction location.

   // Here we:  
   // \* Stop interrupts so we don't get interrupted.  
   // \* Temporarily put the new EIP location in ECX.  
   // \* Load the stack and base pointers from the new task struct.  
   // \* Change page directory to the physical address (physicalAddr) of the new directory.  
   // \* Put a dummy value (0x12345) in EAX so that above we can recognise that we've just  
   // switched task.  
   // \* Restart interrupts. The STI instruction has a delay - it doesn't take effect until after  
   // the next instruction.  
   // \* Jump to the location in ECX (remember we put the new EIP in there).  
   asm volatile("         \   
     cli;                 \   
     mov %0, %%ecx;       \   
     mov %1, %%esp;       \   
     mov %2, %%ebp;       \   
     mov %3, %%cr3;       \   
     mov $0x12345, %%eax; \   
     sti;                 \   
     jmp \*%%ecx           "  
                : : "r"(eip), "r"(esp), "r"(ebp), "r"(current\_directory->physicalAddr));  
}

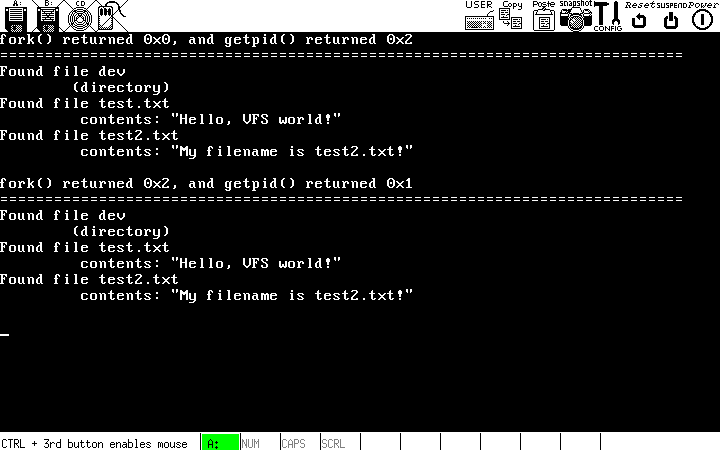
Sorted! That's us finished! Let's test it out!

## 9.5. Testing

Let's change our main() function:

int main(struct multiboot \*mboot\_ptr, u32int initial\_stack)  
{  
   initial\_esp = initial\_stack;  
   // Initialise all the ISRs and segmentation  
   init\_descriptor\_tables();  
   // Initialise the screen (by clearing it)  
   monitor\_clear();  
   // Initialise the PIT to 100Hz  
   asm volatile("sti");  
   init\_timer(50);  
  
   // Find the location of our initial ramdisk.  
   ASSERT(mboot\_ptr->mods\_count > 0);  
   u32int initrd\_location = \*((u32int\*)mboot\_ptr->mods\_addr);  
   u32int initrd\_end = \*(u32int\*)(mboot\_ptr->mods\_addr+4);  
   // Don't trample our module with placement accesses, please!  
   placement\_address = initrd\_end;  
  
   // Start paging.  
   initialise\_paging();  
  
   // Start multitasking.  
   initialise\_tasking();  
  
   // Initialise the initial ramdisk, and set it as the filesystem root.  
   fs\_root = initialise\_initrd(initrd\_location);  
  
   // Create a new process in a new address space which is a clone of this.  
   int ret = fork();  
  
   monitor\_write("fork() returned ");  
   monitor\_write\_hex(ret);  
   monitor\_write(", and getpid() returned ");  
   monitor\_write\_hex(getpid());  
   monitor\_write("\n============================================================================\n");  
  
   // The next section of code is not reentrant so make sure we aren't interrupted during.  
   asm volatile("cli");  
   // list the contents of /  
   int i = 0;  
   struct dirent \*node = 0;  
   while ( (node = readdir\_fs(fs\_root, i)) != 0)  
   {  
       monitor\_write("Found file ");  
       monitor\_write(node->name);  
       fs\_node\_t \*fsnode = finddir\_fs(fs\_root, node->name);  
  
       if ((fsnode->flags&0x7) == FS\_DIRECTORY)  
       {  
           monitor\_write("\n\t(directory)\n");  
       }  
       else  
       {  
           monitor\_write("\n\t contents: \"");  
           char buf[256];  
           u32int sz = read\_fs(fsnode, 0, 256, buf);  
           int j;  
           for (j = 0; j < sz; j++)  
               monitor\_put(buf[j]);  
  
           monitor\_write("\"\n");  
       }  
       i++;  
   }  
   monitor\_write("\n");  
  
   asm volatile("sti");  
  
   return 0;  
}

## 9.6. Summary

  
Success!

Multitasking is really one of the final hurdles to creating a "proper" kernel. Giving the user the appearance of being able to run multiple things concurrently is essential to any modern OS.

Full source code is available [here](http://www.jamesmolloy.co.uk/downloads/multitasking.tar.gz).

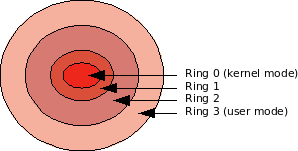
***Copyright James Molloy 2008 - james<at>jamesmolloy.co.uk***

www.jamesmolloy.co.uk

[**Home**](http://www.jamesmolloy.co.uk/index.html) » JamesM's kernel development tutorials

[1. Environment setup](http://www.jamesmolloy.co.uk/tutorial_html/1.-Environment%20setup.html)  
[2. Genesis](http://www.jamesmolloy.co.uk/tutorial_html/2.-Genesis.html)  
[3. The Screen](http://www.jamesmolloy.co.uk/tutorial_html/3.-The%20Screen.html)  
[4. The GDT and IDT](http://www.jamesmolloy.co.uk/tutorial_html/4.-The%20GDT%20and%20IDT.html)  
[5. IRQs and the PIT](http://www.jamesmolloy.co.uk/tutorial_html/5.-IRQs%20and%20the%20PIT.html)  
[6. Paging](http://www.jamesmolloy.co.uk/tutorial_html/6.-Paging.html)  
[7. The Heap](http://www.jamesmolloy.co.uk/tutorial_html/7.-The%20Heap.html)  
[8. The VFS and the initrd](http://www.jamesmolloy.co.uk/tutorial_html/8.-The%20VFS%20and%20the%20initrd.html)  
[9. Multitasking](http://www.jamesmolloy.co.uk/tutorial_html/9.-Multitasking.html)  
**[10. User Mode](http://www.jamesmolloy.co.uk/tutorial_html/10.-User%20Mode.html)**

# 10. User mode (and syscalls)

  
The x86's protection system.

Your kernel, at the moment, is running with the processor in "kernel mode", or "supervisor mode". Kernel mode makes available certain instructions that would usually be denied a user program - like being able to disable interrupts, or halt the processor.

Once you start running user programs, you'll want to make the jump from kernel mode to user mode, to restrict what instructions are available. You can also restrict read or write access to areas of memory. This is often used to 'hide' the kernel's code and data from user programs.

## 10.1. Switching to user mode

The x86 is strange in that there is no direct way to switch to user mode. The only way one can reach **user mode** is to return from an **exception that *began* in user mode.** The only method of getting there in the first place is to set up the stack as if an exception in user mode had occurred, then executing an exception return instruction (IRET).

The IRET instruction expects, when executed, the stack to have the following contents (starting from the stack pointer - the lowermost address upwards):

  
Stack prior to IRET.

* The instruction to continue execution at - the value of EIP.
* The code segment selector to change to.
* The value of the EFLAGS register to load.
* The stack pointer to load.
* The stack segment selector to change to.

The EIP, EFLAGS and ESP register values should be easy to work out, but the CS and SS values are slightly more difficult.

When we set up our GDT we set up 5 selectors - the NULL selector, a code segment selector for kernel mode, a data segment selector for kernel mode, a code segment selector for user mode, and a data segment selector for user mode.

They are all 8 bytes in size, so the selector indices are:

* 0x00: Null descriptor
* 0x08: Kernel code segment
* 0x10: Kernel data segment
* 0x18: User code segment
* 0x20: User data segment

We're currently using selectors 0x08 and 0x10 - for user mode we want to use selectors 0x18 and 0x20. However, it's not quite that straightforward. Because the selectors are all 8 bytes in size, the two least significant bits of the selector will always be zero. Intel use these two bits to represent the RPL - the *Requested Privilege Level*. These have currently been zero because we were operating in ring 0, but now that we want to move to ring three we must set them to '3'. If you wish to know more about the RPL and segmentation in general, you should read the intel manuals. There is far too much information for me to explain everything here.

So, this means that our code segment selector will be (0x18 | 0x3 = 0x1b), and our data segment selector will be (0x20 | 0x3 = 0x23).

### 10.1.1. task.c

This function should go in your task.c. We'll call it from main.c.

void switch\_to\_user\_mode()  
{  
   // Set up a stack structure for switching to user mode.  
   asm volatile("  \   
     cli; \   
     mov $0x23, %ax; \   
     mov %ax, %ds; \   
     mov %ax, %es; \   
     mov %ax, %fs; \   
     mov %ax, %gs; \   
                   \   
     mov %esp, %eax; \   
     pushl $0x23; \   
     pushl %eax; \   
     pushf; \   
     pushl $0x1B; \   
     push $1f; \   
     iret; \   
   1: \   
     ");  
}

This code firstly disables interrupts, as we're working on a critical section of code. It then sets the ds, es, fs and gs segment selectors to our user mode data selector - 0x23.

Our aim is to return from the switch\_to\_user\_mode() function in user mode, so to do that we need to not change the stack pointer. The next line saves the stack pointer in EAX, for reference later. We push our stack segment selector value (0x23), then push the value that we want the stack pointer to have after the IRET. This is the value of ESP before we changed anything on the stack (stored in EAX).

The pushf instruction pushes the current value of EFLAGS - we then push the CS selector value (0x1b).

The next statement is a little special, and can confuse some people who are not used to AS syntax. we push the value of $1f onto the stack. $1f means "the address of the next label '1:', searching forward". Read the GNU AS manual for more information, but numeric symbols are treated differently by it - you can have as many definitions of "1:", "2:" etc as you like.

After this we execute our IRET, and hopefully we should now be executing code at the "1:" line with the same stack, in user mode.

### 10.1.2. Something to watch out for

You may notice that we disabled interrupts before starting the mode switch. A problem now occurs - how do we re-enable interrupts? You'll find that executing *sti* in user mode will cause a general protection fault, however if we enable interrupts before we do our IRET, we may be interrupted at a bad time.

A solution presents itself if you know how the *sti* and *cli* instructions work - they just set the 'IF' flag in EFLAGS. [Wikipedia](http://en.wikipedia.org/wiki/EFLAGS) tells us that the IF flag has a mask of 0x200, so what you *could* do, is insert these lines just after the 'pushf' in the asm above:

pop %eax ; Get EFLAGS back into EAX. The only way to read EFLAGS is to pushf then pop.  
or %eax, $0x200 ; Set the IF flag.  
push %eax ; Push the new EFLAGS value back onto the stack.

This solution means that interrupts get reenabled atomically as IRET is executing - perfectly safe.

## 10.2. System calls

Code running in user mode cannot run any code which is located in or accesses a supervisor-only area of memory (see the page table entry flags) or any code which uses privileged instructions such as *hlt*. Most kernels therefore provide an interface by which common functions can be executed. A call to the kernel through this interface is called a "system call".

The historical, easy, and still widely used way to implement system calls on x86 is to use software interrupts. The user program will set up one register to indicate which system function it would like to execute, then set up parameters in others. It would then execute a software interrupt to a specific vector - linux uses 0x80. The software interrupt causes a mode change to ring 0 - the kernel will have a handler for this interrupt vector, and dispatch the system call appropriately.

One thing that is important to note is that the kernel, when executing interrupt handling code, requires a valid stack to work with. If it doesn't have one, the processor will double fault (and then eventually triple fault because the double fault handler needs a valid stack too!). This would obviously be a very easy way for a malicious user to bring down your system, so it is normal practice to, on mode change from ring 3 to ring 0, switch to a new stack designed solely for use by the kernel, and which is guaranteed to be valid.

Obviously, if you want your kernel to be preemptible (i.e. you want to be able to task switch while executing code inside the kernel) you'll need one of these kernel stacks per task, or you'll end up overwriting one task's data when executing another task!

### 10.2.1. The task state segment

The X86 architecture has support for hardware-assisted task switching by way of a list of Task State Segments (TSS). In this tutorial set we have (like BSD, linux and most x86 operating systems) decided against using it and opted instead for a software based solution. The main reason for this is that hardware task switching is actually not much faster than software, and software task switching is far more portable between platforms.

With that said, the way the x86 architecture is designed we have no choice but to use at least one TSS. This is because when a program in user mode (ring 3) executes a system call (software interrupt) the processor automatically looks in the current TSS and sets the stack segment (SS) and stack pointer (ESP) to what it finds in the SS0 and ESP0 fields ('0' because it's switching to ring 0) - in essence this switches from the user's stack to your kernel stack.

Normal practice when implementing software task switching is just to have one TSS, and update the ESP0 field of it whenever a task switch takes place - this is the minimum work neccessary to allow system calls to work properly.

#### 10.2.1.1. descriptor\_tables.h

We'll need to add a TSS entry structure into the descriptor\_tables header file:

// A struct describing a Task State Segment.  
struct tss\_entry\_struct  
{  
   u32int prev\_tss;   // The previous TSS - if we used hardware task switching this would form a linked list.  
   u32int esp0;       // The stack pointer to load when we change to kernel mode.  
   u32int ss0;        // The stack segment to load when we change to kernel mode.  
   u32int esp1;       // Unused...  
   u32int ss1;  
   u32int esp2;  
   u32int ss2;  
   u32int cr3;  
   u32int eip;  
   u32int eflags;  
   u32int eax;  
   u32int ecx;  
   u32int edx;  
   u32int ebx;  
   u32int esp;  
   u32int ebp;  
   u32int esi;  
   u32int edi;  
   u32int es;         // The value to load into ES when we change to kernel mode.  
   u32int cs;         // The value to load into CS when we change to kernel mode.  
   u32int ss;         // The value to load into SS when we change to kernel mode.  
   u32int ds;         // The value to load into DS when we change to kernel mode.  
   u32int fs;         // The value to load into FS when we change to kernel mode.  
   u32int gs;         // The value to load into GS when we change to kernel mode.  
   u32int ldt;        // Unused...  
   u16int trap;  
   u16int iomap\_base;  
} \_\_attribute\_\_((packed));  
  
typedef struct tss\_entry\_struct tss\_entry\_t;

#### 10.2.1.2. descriptor\_tables.c

We'll also need code to initialise the TSS. The TSS is actually stored as a pointer inside the GDT, so we'll need another GDT entry too.

// Lets us access our ASM functions from our C code.  
...  
extern void tss\_flush();  
  
// Internal function prototypes.  
...  
static void write\_tss(s32int,u16int,u32int);  
...  
  
tss\_entry\_t tss\_entry;  
  
static void init\_gdt()  
{  
   gdt\_ptr.limit = (sizeof(gdt\_entry\_t) \* 6) - 1;  
   gdt\_ptr.base  = (u32int)&gdt\_entries;  
  
   gdt\_set\_gate(0, 0, 0, 0, 0);                // Null segment  
   gdt\_set\_gate(1, 0, 0xFFFFFFFF, 0x9A, 0xCF); // Code segment  
   gdt\_set\_gate(2, 0, 0xFFFFFFFF, 0x92, 0xCF); // Data segment  
   gdt\_set\_gate(3, 0, 0xFFFFFFFF, 0xFA, 0xCF); // User mode code segment  
   gdt\_set\_gate(4, 0, 0xFFFFFFFF, 0xF2, 0xCF); // User mode data segment  
   write\_tss(5, 0x10, 0x0);  
  
   gdt\_flush((u32int)&gdt\_ptr);  
   tss\_flush();  
}  
  
// Initialise our task state segment structure.  
static void write\_tss(s32int num, u16int ss0, u32int esp0)  
{  
   // Firstly, let's compute the base and limit of our entry into the GDT.  
   u32int base = (u32int) &tss\_entry;  
   u32int limit = base + sizeof(tss\_entry);  
  
   // Now, add our TSS descriptor's address to the GDT.  
   gdt\_set\_gate(num, base, limit, 0xE9, 0x00);  
  
   // Ensure the descriptor is initially zero.  
   memset(&tss\_entry, 0, sizeof(tss\_entry));  
  
   tss\_entry.ss0  = ss0;  // Set the kernel stack segment.  
   tss\_entry.esp0 = esp0; // Set the kernel stack pointer.  
  
   // Here we set the cs, ss, ds, es, fs and gs entries in the TSS. These specify what  
   // segments should be loaded when the processor switches to kernel mode. Therefore  
   // they are just our normal kernel code/data segments - 0x08 and 0x10 respectively,  
   // but with the last two bits set, making 0x0b and 0x13. The setting of these bits  
   // sets the RPL (requested privilege level) to 3, meaning that this TSS can be used  
   // to switch to kernel mode from ring 3.  
   tss\_entry.cs   = 0x0b;  
   tss\_entry.ss = tss\_entry.ds = tss\_entry.es = tss\_entry.fs = tss\_entry.gs = 0x13;  
}

Well define tss\_flush in a second. We'll also need a function to update the TSS entry when we change tasks, so it holds the address of the correct kernel stack;

void set\_kernel\_stack(u32int stack)  
{  
   tss\_entry.esp0 = stack;  
}

#### 10.2.1.3. gdt.s

Here we define our tss\_flush function. In it, we tell the processor where to find our TSS within the GDT.

[GLOBAL tss\_flush]    ; Allows our C code to call tss\_flush().  
tss\_flush:  
   mov ax, 0x2B      ; Load the index of our TSS structure - The index is  
                     ; 0x28, as it is the 5th selector and each is 8 bytes  
                     ; long, but we set the bottom two bits (making 0x2B)  
                     ; so that it has an RPL of 3, not zero.  
   ltr ax            ; Load 0x2B into the task state register.  
   ret

Notice that we have to specify an RPL, just like when we switched to user mode.

### 10.2.2. The system call interface

We're going to create a syscall interface similar to Linux's, in that it uses interrupt vector 0x80. Our defined interrupt handlers don't currently reach that high, so we'll have to add another - a "ISR\_NOERRCODE 128" in interrupt.s, and an extra idt\_set\_gate in descriptor\_tables.c (and of course an extra function prototype in descriptor\_tables.h).

#### 10.2.2.1. syscall.h

Initially, we just need to give an interface for starting the syscall interface...

// syscall.h -- Defines the interface for and structures relating to the syscall dispatch system.  
// Written for JamesM's kernel development tutorials.  
  
#ifndef SYSCALL\_H  
#define SYSCALL\_H  
  
#include "common.h"  
  
void initialise\_syscalls();  
  
#endif

#### 10.2.2.2. syscall.c

... and then implement it. As mentioned previously, the normal way to dispatch syscalls is to have one register contain a number which indexes a table of functions. the given function is then executed.

For the moment, we just have three functions which can be called via syscall - the three monitor output functions. This will enable us to check whether our code works easier, by allowing text output in user mode.

// syscall.c -- Defines the implementation of a system call system.  
// Written for JamesM's kernel development tutorials.  
  
#include "syscall.h"  
#include "isr.h"  
  
#include "monitor.h"  
  
static void syscall\_handler(registers\_t \*regs);  
  
static void \*syscalls[3] =  
{  
   &monitor\_write,  
   &monitor\_write\_hex,  
   &monitor\_write\_dec,  
};  
u32int num\_syscalls = 3;  
  
void initialise\_syscalls()  
{  
   // Register our syscall handler.  
   register\_interrupt\_handler (0x80, &syscall\_handler);  
}  
  
void syscall\_handler(registers\_t \*regs)  
{  
   // Firstly, check if the requested syscall number is valid.  
   // The syscall number is found in EAX.  
   if (regs->eax >= num\_syscalls)  
       return;  
  
   // Get the required syscall location.  
   void \*location = syscalls[regs->eax];  
  
   // We don't know how many parameters the function wants, so we just  
   // push them all onto the stack in the correct order. The function will  
   // use all the parameters it wants, and we can pop them all back off afterwards.  
   int ret;  
   asm volatile (" \   
     push %1; \   
     push %2; \   
     push %3; \   
     push %4; \   
     push %5; \   
     call \*%6; \   
     pop %%ebx; \   
     pop %%ebx; \   
     pop %%ebx; \   
     pop %%ebx; \   
     pop %%ebx; \   
   " : "=a" (ret) : "r" (regs->edi), "r" (regs->esi), "r" (regs->edx), "r" (regs->ecx), "r" (regs->ebx), "r" (location));  
   regs->eax = ret;  
}

So here we have a table of the addresses of our syscall functions. The initialise\_syscalls function merely adds the syscall\_handler function as an interrupt handler for interrupt 0x80.

The syscall\_handler function checks that the given function index is valid, then gets the address of the function to call, and then pushes all the parameters we were given onto the stack, call the function, and pop all the parameters back off the stack.

As is customary it also puts the return value of the function call in EAX, when the interrupt returns.

### 10.2.3. Helper macros

So a syscall from user mode would look something like this:

mov eax, <function to call>  
mov ebx, <first parameter>  
mov ecx, <second parameter>  
mov edx, <third parameter>  
mov esi, <fourth parameter>  
mov edi, <fifth parameter>  
int 0x80 ; execute syscall  
         ; return value of syscall is in EAX. </fifth parameter></fourth parameter></third parameter></second parameter></first parameter></function to

This is, however, a little unwieldy. We can simplify this by creating some helper macros to define stub functions that contain inline assembler that actually does the syscall;

*In syscall.h*

#define DECL\_SYSCALL0(fn) int syscall\_##fn();  
#define DECL\_SYSCALL1(fn,p1) int syscall\_##fn(p1);  
#define DECL\_SYSCALL2(fn,p1,p2) int syscall\_##fn(p1,p2);  
#define DECL\_SYSCALL3(fn,p1,p2,p3) int syscall\_##fn(p1,p2,p3);  
#define DECL\_SYSCALL4(fn,p1,p2,p3,p4) int syscall\_##fn(p1,p2,p3,p4);  
#define DECL\_SYSCALL5(fn,p1,p2,p3,p4,p5) int syscall\_##fn(p1,p2,p3,p4,p5);  
  
#define DEFN\_SYSCALL0(fn, num) \   
int syscall\_##fn() \   
{ \   
 int a; \   
 asm volatile("int $0x80" : "=a" (a) : "0" (num)); \   
 return a; \   
}  
  
#define DEFN\_SYSCALL1(fn, num, P1) \   
int syscall\_##fn(P1 p1) \   
{ \   
 int a; \   
 asm volatile("int $0x80" : "=a" (a) : "0" (num), "b" ((int)p1)); \   
 return a; \   
}  
  
#define DEFN\_SYSCALL2(fn, num, P1, P2) \   
int syscall\_##fn(P1 p1, P2 p2) \   
{ \   
 int a; \   
 asm volatile("int $0x80" : "=a" (a) : "0" (num), "b" ((int)p1), "c" ((int)p2)); \   
 return a; \   
}  
  
...

So we have a macro "DECL\_SYSCALLX", which declares a stub function for a function *fn*, with *X* parameters, they being of type *p1*..*pn*.

The macro "DEFN\_SYSCALLX" actually defines the stub function, which is just a piece of inline assembly. The *num* parameter is the index in the syscall function table to call.

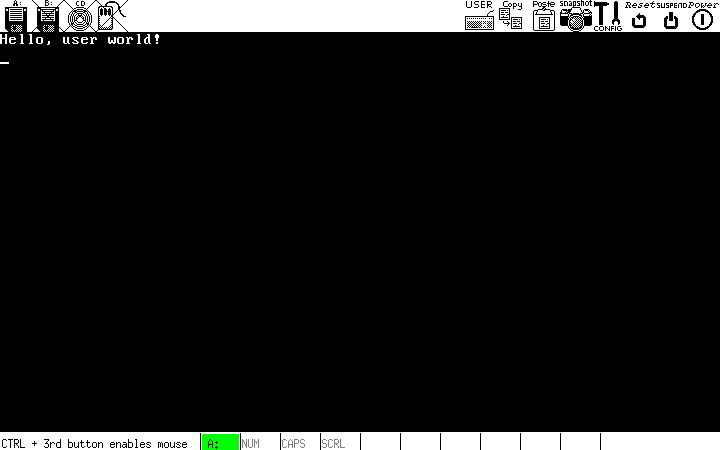
So to define our monitor\_\* functions, we should declare them in syscall.h:

DECL\_SYSCALL1(monitor\_write, const char\*)  
DECL\_SYSCALL1(monitor\_write\_hex, const char\*)  
DECL\_SYSCALL1(monitor\_write\_dec, const char\*)

and define them in syscall.c:

DEFN\_SYSCALL1(monitor\_write, 0, const char\*);  
DEFN\_SYSCALL1(monitor\_write\_hex, 1, const char\*);  
DEFN\_SYSCALL1(monitor\_write\_dec, 2, const char\*);

## 10.3. Testing

  
Hello, user world!

*In main.c*

// Start paging.  
initialise\_paging();  
  
// Start multitasking.  
initialise\_tasking();  
  
// Initialise the initial ramdisk, and set it as the filesystem root.  
fs\_root = initialise\_initrd(initrd\_location);  
  
initialise\_syscalls();  
  
switch\_to\_user\_mode();  
  
syscall\_monitor\_write("Hello, user world!\n");  
  
return 0;

With this test code in main.c, you should have a functional user mode and syscall interface, suitable for running untrusted user programs.

Full source code and image file is available [here](http://www.jamesmolloy.co.uk/downloads/user_mode.tar.gz).

### 10.3.1. Possible problems

If you keep getting page faults when jumping to user mode, make sure that your kernel code/data is set to be user-accessible. When you actually load user programs you won't want this to be the case, however at the moment we merely jump back to the kernel and execute code in main(), so it needs to be accessible in user mode!

***Copyright James Molloy 2008 - james<at>jamesmolloy.co.uk***