

## Worksheet 2 Part 1

Worksheet 2 is split into two parts, this is the first part, with a second part to follow. Each part is worth 50% of the total for worksheet 2, which itself is worth 50% of the module assessment, with worksheet 1 making up the other 50%.

### Starting a Tiny OS

#### Marking scheme

- Task 1 - 20%
- Task 2 - 20%
- Task 3 - 40%
- Task 4 - 20%

All code should be submitted via a github repo.

Each task should be documented in a README.md in the main repo, including screen shots of the code running, and details of how your implementation works.

After completing this worksheet you should be familiar with:

- The basics of boot a machine from scratch
- Calling C from assembler
- Developing a framebuffer driver for our Tiny OS

It is assumed that you have completed worksheets 0 and 1 before starting this worksheet.

The deadline for completing this worksheet is 4th December, 2025, by 2pm.

**IMPORTANT:** All code must be submitted via github, failure to provide a link to Github repo will result in a mark of 0. No email or other form of submission is acceptable. There will be in class vivas weeks stating the 4th December, 2025.

You need to have the book [The Little Book of OS Development](#) on hand to help gain a full understanding of what needs to be done to complete the tasks.

#### To proceed

Before continuing accept and clone github repo for worksheet 2 from the link on Blackboard.

For the first part of worksheet 2 we are going to work through chapters 1 - 4 of the [The Little Book of OS Development](#). In particular we are going to develop a Tiny Operating System that boots the machine from scratch, which will require us to boot the machine via a boot loader, something that happens automatically with an OS already installed, and in the process we will learn how to call C from assembler and write output with the framebuffer.

You will need to read the chapters to be able to complete these tasks.

### Task 1

For our OS we will need to have a number of different files, some will be used to build the boot loader, others that will be the kernel, and some will describe how to put it all together. For this it is recommended that you create the following directory structure in the root of the directory created above:

```
drivers
source
iso
boot
    grub
.gitignore
Makefile
README.md
```

As you develop the OS, other files will be added, but this is the minimum. As in chapter 2 of the book we will begin with developing the smallest operating system possible, the one thing this OS will do is write **0xCAFEBAE** to the `eax` register! The cool thing about this is that it actually is the basis of everything else we will do to boot the machine from scratch.

Our first kernel has to be written in assembler as unlike the code in worksheet 1, there is no C stack and no C library to make our lives simpler, we need to provide it all, and in the next task we will develop code to call into C and start working in C.

```
global loader ; the entry symbol for ELF

MAGIC_NUMBER equ 0x1BADB002 ; define the magic number constant
FLAGS        equ 0x0          ; multiboot flags
CHECKSUM     equ -MAGIC_NUMBER ; calculate the checksum
; (magic number + checksum + flags should
;  ↳ equal 0)
```

```
section .text:          ; start of the text (code) section
align 4                ; the code must be 4 byte aligned
dd MAGIC_NUMBER        ; write the magic number to the machine
↪ code,                ; the flags,
    dd FLAGS           ; and the checksum
    dd CHECKSUM

loader:                ; the loader label (defined as entry
↪ point in linker script)
    mov eax, 0xCAFEBAE ; place the number 0xCAFEBAE in the
↪ register eax

.loop:
    jmp .loop          ; loop forever
```

This code can also be found in our book and the only thing it does of interest, is write the number **0xCAFEBAE** in the register **eax**.

Put this in an assembler file, "loader.asm" , and similar to worksheet 1 it can be compiled with the **nasm**:

```
nasm -f elf loader.asm
```

Now in worksheet 1 we linked our assembler files with a C code that had a main function, however, as noted above we no longer have the ability to do that as there is no C library or stack. What are we to do?

As discussed in the book, the code must be linked to produce an executable file, which requires some extra thought compared to when linking most programs, where we just used **gcc** or **clang** to link in our **driver.c**.

Here we are going to have to use a bootloader, in our case GRUB, to load the kernel at a memory address larger than or equal to **0x00100000** (1 megabyte (MB)), because addresses lower than 1 MB are used by GRUB itself, BIOS and memory-mapped I/O. To achieve this we can use a custom linker script and the linker **ld**.

GRUB will transfer control to the operating system by jumping to a position in memory. Before the jump, GRUB will look for a magic number to ensure that it is actually jumping to an OS and not some random code. This magic number is part of the multiboot specification which GRUB adheres to, and explains where we had those magic numbers in our assembler file above. Once

GRUB has made the jump, the OS has full control of the computer.

Therefore, the following linker script is needed (written for GNU LD) and should be added to `link.ld` in our source directory:

```
ENTRY(loader)                      /* the name of the entry label */

SECTIONS {
    . = 0x00100000;                /* the code should be loaded at 1 MB */

    .text ALIGN (0x1000) : /* align at 4 KB */
    {
        *(.text)                  /* all text sections from all files */
    }

    .rodata ALIGN (0x1000) : /* align at 4 KB */
    {
        *(.rodata*)              /* all read-only data sections from all files */
    }

    .data ALIGN (0x1000) : /* align at 4 KB */
    {
        *(.data)                  /* all data sections from all files */
    }

    .bss ALIGN (0x1000) : /* align at 4 KB */
    {
        *(COMMON)                 /* all COMMON sections from all files */
        *(.bss)                   /* all bss sections from all files */
    }
}
```

We are now ready to link our executable, which can be done with the following command:

```
ld -T ./source/link.ld -melf_i386 loader.o -o kernel.elf
```

Ok our kernel is now built. To make life a little simpler we are going to use GRUB Legacy, rather than a more complicated bootloader, but this means we need to copy a file into place that GRUB needs to work correctly. This file can be found in:

```
/opt/os/stage2_eltorito
```

and needs to be copied into the directory:

```
iso/boot/grub
```

A configuration file **menu.lst** for GRUB must be created. This file tells GRUB where the kernel is located and configures some options, it is defined as follows:

```
default=0
timeout=0

title os
kernel /boot/kernel.elf
```

So if everything has gone correctly we should now have a directory structure as follows:

```
iso
|-- boot
    |-- grub
        |-- menu.lst
        |-- stage2_eltorito
    |-- kernel.elf
```

Check this is correct and if so the following command can be used to create the ISO image:

```
genisoimage -R
            -b boot/grub/stage2_eltorito \
            -no-emul-boot \
            -boot-load-size 4 \
            -A os \
            -input-charset utf8 \
            -quiet \
            -boot-info-table \
            -o os.iso \
            iso
```

Ok now the kernel (os.iso) is built and we are ready to run it.

Unlike the book we will be use **qemu**, which is a generic and open source machine emulator and virtualizer. Using an emulator will allow us to not worry about actually booting our machine and not being able to do other stuff, but also we can use it to create a trace the machines execution and thus confirm that eax is indeed written too. To do this we can use the following command:

```
qemu-system-i386 -nographic -boot d -cdrom os.iso -m 32 -d cpu -D logQ.txt
```

This command tells **qemu** to boot a 32-bit (i368) x86 machine, using our ISO image, output the machines execution to the file `logQ.txt`. As the last this our assembler code did was loop over and over, we will need to terminate the run with the command `Ctrl-C`. Once done we can open the `logQ.txt` file and search for the string `0xCAFEBAE`. If you see `EAX` being assigned with it, then you have correctly booted!

To complete this task implement a makefile that builds the kernel, copies it into the GRUB directory structure, and then creates the ISO image. Add another rule, `run`, to your makefile that loads and runs the ISO image.

## Task 2

For this task you should read chapter 3 of the OS book and extend your kernel so that it can call C functions. As well as implementing the function `sum_of_three`, outlined in the chapter. Also, you should implement and test at least two other C functions, although what those functions do is up to you.

Extend your makefile to work with this new structure, in particular, add support to transfer control to C from your `loader.asm`, developed in task 1.

## Task 3

For this task we are going to focus on adding some simple I/O, following chapter 4 of our OS book. An OS, or any program, the lack of IO makes it pretty basic and the ability to interact with the world opens up access to dynamically changing a programs behaviour. The simplest form of I/O is the framebuffer so we can display or output text to the console. In part 2 of the worksheet, we will explore the other basic, but key I/O device, the keyboard.

There are often two different ways to interact with the hardware, memory-mapped I/O and I/O ports.

The simplest kind of I/O device uses memory-mapped I/O, which means you can write to a specific memory address and the hardware will be updated with the new data. One example of this is the

framebuffer. For example, if you write the value **0x410F** to address **0x000B8000**, you will see the letter **A** in white color on a black background.

If the hardware uses I/O ports then the assembly code instructions out and in must be used to communicate with the hardware. The instruction “out” takes two parameters: the address of the I/O port and the data to send. The instruction in takes a single parameter, the address of the I/O port, and returns data from the hardware. One can think of I/O ports as communicating with hardware the same way as you communicate with a server using sockets. The cursor (the blinking rectangle) of the framebuffer is one example of hardware controlled via I/O ports on a PC.

To begin with you should work through the framebuffer section of chapter 4, where you will implement the very basics of implementing a framebuffer driver. Your driver code should be put in the directory **drivers** and you should extend your makefile to compile your additional files, when necessary.

Note, that due to the fact that we are using qemu, which they do not use in the book, we need to make sure that the framebuffer is correctly configured to work with your application. All we need to do is run qemu in Curses mode.

Curses is a terminal control library for Unix-like systems, enabling the construction of text user interface (TUI) applications

We can do this with the following command, assuming that the **os.iso** image is built:

```
qemu-system-i386 -curses -monitor telnet::45454,server,nowait -serial  
    ↵ mon:stdio -boot d -cdrom os.iso -m 32 -d cpu -D logQ.txt
```

Other than putting qemu into curses mode we also told qemu to run a telnet service (on port 45454). This is important because once we put qemu in curses mode it can no longer receive keyboard commands (such as **Ctrl-C**) and thus it is not possible to quit our kernel once it has started running. Now we have to connect to the qemu process using telnet from another terminal window, with the command:

```
telnet 45454
```

Telnet is like SSH, but not encrypted, and once connected to qemu we can send the command:

```
quit
```

to tell qemu to exit.

Don't forget to update your makefile with this new run command.

Finally, to complete this task you should extend your framework driver to a full API, in particular, it should expose it to the user as a 2D API, with the ability to move the cursor to a location, given x and y, and to print strings, numbers, and so on. For example, this might include the following operation to set the cursor position:

```
void fb_move(unsigned short x, unsigned short y);
```

The actual functions of your API are for you to define, but as noted they should include the functions described above, and also support for setting text colour, and clearing the framebuffer.

Write a kernel to test your functionality.

#### Task 4

Document your work in your README.md and submit to Github. The readme should contain code snippets, demonstrating understanding, screen shots, and so on. It should not contain complete files of code, as these will be in the repo itself.