# Machine Problem 1: Getting Started

Due: January $25^{th}$ 11:59 pm

## Introduction

The objective of this machine problem is to test your development environment. You are provided a simple "kernel", which essentially prints a welcome text and goes into an infinite loop. You are to modify the text on the welcome message to print out your name.

The "kernel" source code for this do-nothing kernel consists of a small collection of source files:

`kernel.C`: This file contains the main entry point for the kernel. For this MP, this is where you need to apply the modifications in order to have the kernel print out your name after the welcome message.

`start.asm`: This assembler file contains the multiboot header, some initial setup, and the call to the main entry point in the kernel.

`utils.H/C`: A selection of utility functions, such as memory copy, simple string operations, port I/O operations, and program termination.

`console.H/C`: Access to the console video output.

`makefile`: Used to easily compile everything to generate the `kernel.bin` file. (There are several versions of the makefile, targeting different development environments.)

Your task is to modify the provided kernel, compile it, and generate a floppy disk image that will boot your kernel.

The first step is to setup your development environment, i.e., a place where you will find all tools necessary to build your kernel.

## Step 1: Setup your Development Execution Environment

We will be using *VirtualBox* to run a fully configured image of Ubuntu Linux with all the development environment tools pre-installed. The image is 64-bit, so it is required to have a processor capable of 64-bit guest OS emulation. Note that the 64-bit requirement does not mean your host image has to be 64 bit. You can have Windows 7 32-bit and still have a 64-bit processor. If your machine does not meet the requirements for VirtualBox, you can use machines in the Open Access Labs. *VirtualBox* is available for Windows, Linux, and Mac OS environments.

## Step 2: Setup your Source Revision Control Environment

You will be required to set up a github repository for this class. at `github.tamu.edu`. You will receive information on how to do this. Authentication is done using your netid. If you are not familiar with github, there are plenty of good tutorials out there.

The history of your git actions is a documentation of your development process. In this MP, there is nothing much to document: you start by creating a directory for MP1 and adding all the files you needed there. As you fix problems (e.g., if the provided Makefile did not work in your environment) and develop code (e.g., change `kernel.C` to include your name), you commit new

versions of files to the directory. For each action, use meaningful explanations in your commit messages. Use MP1 as an opportunity to make sure you know how to commit your changes and push them to your github repository, with commit explanations. Introducing lots of code and lots of changes in a single git action is not, in general, a good software practice. For our projects, it is not an acceptable practice. You need to commit your work (and push it to your remote repository) often. If the project submission system on eCampus is not working you won't need to sweat about proving that you finished the project within the deadline: the git history will prove that you had concluded it. **Your git activity will be used to assess your software development practices and it will have an impact on your project grade.** Be aware that the github history may also be used as evidence of insufficient independent work.

### Step 3: Finally – doing the Assignment

Now you should be ready to change `kernel.C`:

- Read the provided information under Project Resources on eCampus.
- Setup your development environment.
- Download and unzip the provided code in file `MP1_Sources.zip` from eCampus to your development environment.
- Look at `README.TXT`
- Work on `kernel.C`

After modifying the provided kernel file, compile it, and copy the resulting `kernel.bin` binary file onto the provided floppy image file, called `dev_kernel_grub.img`. The provided zip file contains a shell command to do this, you invoke it as follows:

```
> ./copykernel.sh
```

This shell command **mounts** the image as a disk, **copies** the file `kernel.bin` onto the mounted disk, and then **unmount** the disk image. In the past some students had to use a lazy umount to get things to work, so as usual, when things don't work, ask around, look around, and find a way to make it work. Remember to check the course discussion forum, and if you see people stuck with problems that you solved, share your knowledge.

Now you are ready to run your own kernel. In order to avoid the difficulties of debugging your code in a bare metal hardware, you are going to run your kernel in an **emulated** or **simulated** environment, on top of your normal development environment. Many emulation or virtualization environments can be used to accomplish running your "small" kernel on top of the "real" kernel managing the machine you are using for this project. In the provided development environment is set up for your to use Bochs. As mentioned before, utilize the makefile to build your kernel binary. Then call the copy kernel script to copy the `kernel.bin` onto the boot drive image.

## The Bochs Environment

Bochs is a highly portable open source IA-32 (x86) PC emulator written in C++, that runs on most popular platforms. It includes emulation of the Intel x86 CPU, common I/O devices, and a custom BIOS. Bochs can be compiled to emulate many different x86 CPUs, from early 386 to the most recent x86-64 Intel and AMD processors, even those not in the market yet.

The ZIP archive that comes with MP1 also contains a set of files that define the Bochs emulation environment:

`dev_kernel_grub.img`: This file contains the image of the boot "floppy disk". It contains the GRUB bootloader and a dummy kernel.The boot loader is the first software that runs when a computer testing and hardware initialization has been done. It loads the kernel and then transfers control to it.

`BIOS-bochs-latest`: This file contains the BIOS.The BIOS (more generally known as firmware) is the code that runs on power-on startup. Its main work is to initialize and test the hardware. Every particular computer has a BIOS specifically designed for it. The BIOS also provides basic operations such as I/O until the operating system is ready to take over.

`VGABIOS-lpgl-latest`: This file contains a basic VGA BIOS to manage the graphics card.

`bochsrc.bxrc`: This text file contains the configuration of the emulated machine. If bochs is correctly installed, double-clicking on this file should start the emulator. You can also run Bochs by invoking the program with the configuration file as an argument, as follows:

```
bochs -f bochsrc.bxrc
```

**Note:** You are not required to use Bochs. If you use a different emulator or a virtual machine monitor (such as *VirtualBox*), the virtual machine will be set up differently, and you will not be using the files described above, except for the floppy image file.

## The Bochs Environment with GDB Integration

In its basic form, Bochs is already part of the provided development environment.

You may consider it helpful to use an external debugger (`gdb`, the GNU debugger) with the Bochs environment. It would allow you to debug your program more efficiently.

The provided basic Bochs installation has no support for remote debugging. The Project Resources folder on eCampus contain detailed descriptions of how to install and use Bochs with support for remote debugging with `gdb`.

## The Assignment

**You are to modify the given "kernel" to print out your name on the welcome screen.** For this, you modify the provided file `kernel.C`. You then compile the source to generate the kernel executable `kernel.bin`. Preferably you do this by invoking the `make` command. You then copy the kernel onto the provided `.img` file. After testing your code with the Bochs emulator, you rename your `.img` file to `mp1.img` and compress it into a ZIP file called `mp1.zip`. You are to turn in the ZIP file.

**What to Hand In**

- You are to hand in one file, with name `mp1.zip`, which contains a single file, named `mp1.img`. The latter is the floppy image file that you obtain by copying (replacing) the file `kernel.bin` on the provided `.img` file.
- Grading of these MPs is a very tedious chore. These handin instructions are meant to mitigate the difficulty of grading, and to ensure that the grader does not overlook any of your efforts.
- **Failure to follow the handing instructions will result in lost points.**

## Project Grading Criteria for MP1

- Your kernel boots: 50%.
- Feature completeness and functional correctness: 50%. (This means your kernel prints the expected welcome message.)
- **If you did not follow the github requirement specified above, the penalty is 50% of the project grade.**

**Extra Bonus for MP1 - 10 points** As you will experience, debugging operating system code may be tricky. You can position yourself for an easier time later on by working now on setting up a debugger in your environment.

The Project Resources folder on eCampus has information on how to set up and use Bochs with support for `gdb`.

If you succeed in this integration, submit as part of your zip file a document with a short description of how you accomplished the integration. Also include a picture of your `gdb` tool in action.

## Helpful Links

[1] `https://www.cs.princeton.edu/courses/archive/fall09/cos318/precepts/bochs_gdb.html`

[2] `https://www.cs.princeton.edu/courses/archive/fall09/cos318/precepts/bochs_setup.html`

[3] `http://bochs.sourceforge.net/doc/docbook/user/debugging-with-gdb.html`

[4] `http://heim.ifi.uio.no/~inf3150/doc/tips_n_tricks/bochsd.html`

[5] `http://wiki.yak.net/746`