Lab 1:
Introduction to the BLITZ Tools

Due date: This lab is not marked; but please try to complete the lab by **Tuesday, April 26, 2022**.

Submission deadline: There is nothing to submit in this lab.

Overview and goal

Throughout the labs in this course, you will gradually progress towards building an operating system kernel. You will be using the BLITZ tools, which have been designed and implemented for this task. The goals of this lab are to get you familiar with the use of BLITZ tools.

Step 1: Read the documentation

There are a number of documents describing the BLITZ tools. You may download the documents by going to the course website https://oscourse.org, and download a file called **Documentation.zip**. This archive contains the following documents in Adobe PDF format:

An Overview of the BLITZ System (7 pages)
An Overview of the BLITZ Computer Hardware (8 pages)
The BLITZ Emulator (46 pages)
The BLITZ Architecture (67 pages)
An Overview of KPL, A Kernel Programming Language (66 pages)
The BLITZ Assembler (20 pages)
Example BLITZ Assembly Program (7 pages)
BLITZ Instruction Set (3 pages)
The Thread Scheduler and Concurrency Control Primitives (37 pages)

The list above is ordered: you should try to read them in the recommended order. The documents in **bold font** are required reading for this course, and the first five documents are required reading for this lab. The last document will be needed for Lab 2, so you will be reading it soon.

Step 2: Familiarize yourself with the BLITZ tools

Throughout this course, you will develop your lab solutions using the BLITZ command-line tools, which, among others, includes a CPU emulator and a compiler for a high-level programming language that you will be using, called KPL. There will not be graphical user interfaces in any of the BLITZ tools.

Here are the programs that constitute the BLITZ tool set.

`kpl`: The KPL compiler
`asm`: The BLITZ assembler
`lddd`: The BLITZ linker
`blitz`: The BLITZ machine emulator (the virtual machine) and debugger

diskUtil: A utility to manipulate the simulated the BLITZ "DISK" file
dumpObj: A utility to print BLITZ `.o` and `a.out` files
hexdump: A utility to print any file in hex
check: A utility to run through a file looking for problem ASCII characters
endian: A utility to determine if this machine is Big or Little Endian

These tools are listed in the approximate order they would be used. You will typically only need to use the first four tools (within a `makefile`). The last three tools are only documented by the comments at the beginning of the source code files, which you may read if interested.

Step 3: Installing BLITZ in your own computer

If you prefer to use your own computer and work locally (perhaps due to the network latency of logging into a remote computer), it is most convenient to install **Docker** (community edition) by downloading it from Docker's official website:

To install Docker on Windows 7 and 8:

https://docs.docker.com/toolbox/toolbox_install_windows/

To install Docker on Windows 10:

https://docs.docker.com/docker-for-windows/install/

To install Docker on MacOS:

https://docs.docker.com/docker-for-mac/install/

After you have successfully installed Docker on your own computer, download the Docker image we provided to you for this lab from the course website, called `lab1-docker.tar.gz`.

Now start a terminal window (`/Applications/Utilities/Terminal` in MacOS, for example), and use the following command to load the Docker image that we provided to you:

```
docker load -i lab1-docker.tar.gz
```

Finally, run the Docker image as a Docker container by using the following command:

```
docker run -it blitz
```

After the container is running, you will see a command prompt from within the Linux container. The BLITZ tools have been preinstalled for you in `/usr/local/blitz/bin`, and files needed for Lab 1 can be found in `/lab1`. The source code for building BLITZ tools can be found in `/blitz`. Within the container, the search path environment variable has already been set up for you to use BLITZ command-line tools directly.

There are many other commands that may be useful for you to work with Docker containers. For example, to remove all the Docker containers, you can use the command:

```
docker rm $(docker ps -a -q)
```

To remove all the Docker images (so that you can have a clean slate to start working with something else), you may use the command:

```
docker rmi $(docker images -q)
```

To copy files from the Docker container to your host computer, use the command:

```
docker cp <containerId>:/file/path/within/container /host/path/target
```

The container ID can be found in the command prompt while you are running the container.

The command-line editor `vi` has been pre-installed for you in the Docker container. To install other editor alternatives or any other packages, you can use the following command within the container:

```
apt-get install -y <package_name>
```

To learn more about Docker containers and images, refer to the Docker documentation.

Step 4: Verify that the BLITZ Tools are working

At the Unix prompt (Linux, MacOS, or within the Docker Linux container), type the command:

```
kpl
```

You should see the following:

```
*****  ERROR: Missing package name on command line

**********  1 error detected!  **********
```

If you see this, good. If you see anything else, then something is wrong. (If you are not familiar with environment variables and setting up the `PATH` variable, many tutorials are available online.)

Step 5: Assemble, link, and execute the "`Hello`" Program

In the Docker container image, you can directly use the path `/lab1/`, since we will be using a different Docker container image for each lab. If you are using the Docker image, the files have already been made available for you in `/lab1`.

In this course you will not have to write any assembly language. However, you will be using some interesting routines which can only be written in assembly. All assembly language routines will be provided to you, but you will need to be able to read them to understand some important concepts in operating systems.

Take a look at **`Echo.s`** and **`Hello.s`** to see what BLITZ assembly code looks like. The `lab1/` directory contains an assembly language program called "`Hello.s`". First invoke the assembler (the tool called "asm") to assemble the program. Type:

```
asm Hello.s
```

This should produce no errors and should create a file called **Hello.o**.

The Hello.s program is completely standalone. In other words, it does not need any library functions and does not rely on any operating system. Nevertheless, it must be linked to produce an executable ("a.out" file). The linking is done with the tool called "lddd". (In Unix, the linker is called "ld".)

```
lddd Hello.o –o Hello
```

Normally the executable is called a.out, but the "-o Hello" option will name the executable Hello.

Finally, execute this program, using the BLITZ emulator. Type:

```
blitz –g Hello
```

The "-g" option is the "auto-go" option and it means begin execution immediately. You should see:

```
Beginning execution...
Hello, world!

****  A 'debug' instruction was encountered  *****
Done!  The next instruction to execute will be:
000080: A1FFFFB8       jmp     0xFFFFB8          ! targetAddr = main

Entering machine-level debugger...
=======================================================
=====                                             =====
=====         The BLITZ Machine Emulator          =====
=====                                             =====
=====  Copyright 2001-2007, Harry H. Porter III   =====
=====                                             =====
=======================================================

Enter a command at the prompt.  Type 'quit' to exit or 'help' for
info about commands.
>
```

At the prompt, quit and exit by typing "q" (short for "quit"). You should see this:

```
> q
Number of Disk Reads    = 0
Number of Disk Writes   = 0
Instructions Executed   = 1705
Time Spent Sleeping     = 0
    Total Elapsed Time  = 1705
```

This program terminates by executing the **debug** machine instruction. This instruction will cause the emulator to stop executing instructions and will throw the emulator into command mode. In command mode, you can enter commands, such as **quit**. The emulator displays the character ">" as a prompt.

After the **debug** instruction, the **Hello** program branches back to the beginning. Therefore, if you resume execution (with the **go** command), it will result in another printout of "Hello, world!".

Step 6: Run the "Echo" Program

Type in the following commands:

```
asm Echo.s
lddd Echo.o -o Echo
blitz Echo
```

On the last line, we have left out the auto-go "-g" option. Now, the BLITZ emulator, called "**blitz**", will not automatically begin executing; instead it will enter command mode. When it prompts, type the "g" command (short for "go") to begin execution.

Next type some text. Each time the **Enter** key is pressed, you should see the output echoed. For example:

```
> g
Beginning execution...
abcd
abcd
this is a test
this is a test
q
q
****  A 'debug' instruction was encountered  *****
Done!  The next instruction to execute will be:
                    cont:
0000A4: A1FFFFAC        jmp   0xFFFFAC        ! targetAddr = loop
>
```

(For clarity, the material entered on the input is in *italic*.)

This program watches for the "**q**" character and stops when it is typed. If you resume execution with the **go** command, this program will continue echoing whatever you type.

The **Echo** program is also a standalone program, relying on no library functions and no operating system.

The KPL Programming Language

In our labs, you will write source code in the KPL programming language. We ask you to begin studying the document titled "An Overview of KPL: A Kernel Programming Language" in the BLITZ documentation.

Step 7: Compile and Execute a KPL Program called "`HelloWorld`"

Type the following commands:

```
kpl -unsafe System
asm System.s
kpl HelloWorld
asm HelloWorld.s
asm Runtime.s
lddd Runtime.o System.o HelloWorld.o -o HelloWorld
```

There should be no error messages.

Take a look at the files **`HelloWorld.h`** and **`HelloWorld.c`**. These contain the program code.

The **`HelloWorld`** program makes use of some other code, which is contained in the files **`System.h`** and **`System.c`**. These must be compiled with the "-unsafe" option. Try leaving this out; you will get 17 compiler error messages, such as:

```
System.h:30: *****  ERROR at PTR: Using 'ptr to void' is unsafe;
                    you must compile with the 'unsafe' option
                    if you wish to do this
```

Using the UNIX compiler convention, this means that the compiler detected an error on line 30 of file **`System.h`**.

KPL programs are often linked with routines coded in assembly language. Right now, all the assembly code we need is included in a file called **`Runtime.s`**. Basically, the assembly code takes care of:

Starting up the program
Dealing with runtime errors, by printing a message and aborting
Printing output (There is no mechanism for input at this stage. This system really needs an OS!)

Now execute this program. Type:

```
blitz –g HelloWorld
```

You should see the "`Hello, world...`" message. What happens if you type "g" at the prompt, to resume instruction execution?

The **lab1** directory contains a file called **makefile**, which is used with the UNIX **make** command. Whenever a file in the **lab1** directory is changed, you can type "`make`" to re-compile, re-assemble, and re-link as necessary to rebuild the executables. If you wish to rebuild all executables from scratch, you can type "`make clean`" at the command line.

Notice that the command:

kpl HelloWorld

will be executed whenever the file **System.h** is changed. In KPL, files ending in "`.h`" are called "header files" and files ending in "`.c`" are called "code files." Each package (such as **HelloWorld**) will have both a header file and a code file. The **HelloWorld** package uses the **System** package. Whenever the header file of a package that **HelloWorld** uses is changed, **HelloWorld** must be recompiled. However, if the code file for **System** is changed, you do not need to recompile **HelloWorld**. You only need to re-link (i.e., you only need to invoke **lddd** to produce the executable).

Consult the KPL documentation for more info about the separate compilation of packages.

Step 8: Modify the HelloWorld Program

Modify the **HelloWorld.c** program by un-commenting the line

```
--foo (10)
```

In KPL, comments are **"--"** through the end-of-line. Simply remove the hyphens and recompile as necessary, using "make".

The **foo** function calls **bar**. **bar** does the following things:

Increment its argument
Print the value
Execute a "debug" statement
Recursively call itself

When you run this program it will print a value and then halt. The keyword **debug** is a statement that will cause the emulator to halt execution. In later labs, you will probably want to place **debug** in programs you write when you are debugging, so you can stop execution and look at variables.

If you type the **go** command, the emulator will resume execution. It will print another value and halt again. Type **go** several times, causing **bar** to call itself recursively several times. Then try the **st** command (**st** is short for "stack"). This will print out the execution stack. Try the **fr** command (short for "frame"). You should see the values of the local variables in some activation of **bar**.

Try the **up** and **down** commands. These move around in the activation stack. You can look at different activations of **bar** with the **fr** command.

Step 9: Try Some of the Emulator Commands

Try the following commands to the emulator.

```
quit (q)
help (h)
go (g)
step (s)
t
reset
info (i)
stack (st)
frame (fr)
up
down
```

Abbreviations are shown in parentheses.

The "**step**" command will execute a single machine-language instruction at a time. You can use it to walk through the execution of an assembly language program, line-by-line.

The **t** command will execute a single high-level KPL language statement at a time. Try typing "**t**" several times to walk through the execution of the **HelloWorld** program. See what gets printed each time you enter the "**t**" command.

The **i** command (short for **info**) prints out the entire state of the (emulated) BLITZ CPU. You can see the contents of all the CPU registers. There are other commands for displaying and modifying the registers.

The **h** command (short for **help**) lists all the emulator commands. Take a look at what **help** prints.

The **reset** command re-reads the executable file and fully resets the CPU. This command is useful during debugging. Whenever you wish to re-execute a program (without recompiling anything), you could always **quit** the emulator and then start it back up. The **reset** command does the same thing but is faster.

Make sure you get familiar with each of the commands listed above; you will be using them later. Feel free to experiment with other commands, too.

The "DISK" File

The BLITZ emulator ("**blitz**") simulates a virtual disk. The virtual disk is implemented using a file on the host machine and this file is called "**DISK**". The programs in lab 1 do not use the disk, so this file is not necessary. However, if the file is missing, the emulator will print a warning. We have included a file called "**DISK**" to prevent this warning. For more information, try the "**format**" command in the emulator, and see what may happen with the file "**DISK**".