Project 1:

Introduction to the BLITZ Tools

Due Date: ______________________________
Duration: One Week

Overview and Goal

Throughout the labs in this course, you will gradually progress towards building an operating system kernel. You'll be using the BLITZ software tools, which were written for this task. The goals of this project are to make sure that you can use the BLITZ tools and to help you gain familiarity with them.

Step 1: Read the Documentation

There are a number of documents describing the BLITZ tools. You may obtain the documents from BUPT's Teaching Platform or by going to the Course Repository:

`https://gitee.com/hliang-bupt/os23/labs/Documents.zip`

This archive contains the following documents in Adobe PDF format:

An Overview of the BLITZ System
An Overview of the BLITZ Computer Hardware
The BLITZ Emulator
The BLITZ Architecture
An Overview of KPL, A Kernel Programming Language
The BLITZ Assembler
Example BLITZ Assembly Program
BLITZ Instruction Set
The Thread Scheduler and Concurrency Control Primitives

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 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 probably only need to use the first 4 or 5 tools and you may pretty much ignore the remaining tools. (The last three tools are only documented by the comments at the beginning of the source code files, which you may read if interested.)

You can get the tools from BUPT's Teaching Platform or :
`https://gitee.com/hliang-bupt/os23/labs/blitzTools.zip`

Step 3: Modify Your Search Path and Verify the Tools are Working

You must add the **BlitzTools** directory to your shell's search path so that when you type in the name of a BLITZ tool (such as **kpl** or **blitz**), your shell can locate the executable file and execute it.

The Unix "shell" program maintains a "shell variable" called PATH which it uses to locate an executable whenever a command name is typed. Details of how to change the PATH variable will vary between the different shells.

One approach might be to alter the **.bashrc/.cshrc** file in your home directory.

```
export PATH=${HOME}/BlitzTools:${PATH}
```

After changing your PATH, you'll need to run the below command:

```
source ~/.bashrc
```

Next, verify that whatever you did to the PATH variable worked.

At the UNIX/Linux prompt, 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.

Step 4: Set up a Directory for Project 1

Create a directory in which to place all files concerned with this class. We recommend a name matching your course number, for example: *~YourUserName***/os23**

Create a directory in which to place the files concerned with project 1. We recommend the following name:
*~YourUserName***/os23/p1**

Copy all files from: **https://gitee.com/hliang-bupt/os23/labs/p1/**
to your **os23/p1** directory.

Step 5: Assemble, Link, and Execute the "Hello" Program

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.

Take a look at **Echo.s** and **Hello.s** to see what BLITZ assembly code looks like.

The **p1** 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 stand-alone. 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 virtual machine. (Sometimes the BLITZ virtual machine is referred to as the "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 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/RETURN 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 underlined.)

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 stand-alone program, relying on no library functions and no operating system.

The KPL Programming Language

In this course, you will write code in the "KPL" programming language. Begin studying the document titled "An Overview of KPL: A Kernel Programming Language".

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'll get 17 compiler error messages, such as:

```
System.h:39: *****  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 39 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 "makefile"

The **p1** directory contains a file called **makefile**, which is used with the UNIX **make** command. Whenever a file in the **p1** directory is changed, you can type "make" to re-compile, re-assemble, and re-link as necessary to rebuild the executables.

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 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 projects, 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 (virtual) 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 KPL virtual machine (the emulator tool, called “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 project 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.