Programming Project 1:
Introduction to the BLITZ Tools

Due Date: _____
Duration: One Week

Overview and Goal

In this course you will be creating 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: Print the Documentation

There are a number of documents describing the BLITZ tools. You may obtain the documents by going to the BLITZ homepage:

`http://www.cs.pdx.edu/~harry/Blitz/index.html`

From there you can access **pdf** versions. Print out the following documents:

An Overview of the BLITZ System (7 pages)
An Overview of the BLITZ Computer Hardware (8 pages)
The BLITZ Architecture (71 pages)
Example BLITZ Assembly Program (7 pages)
BLITZ Instruction Set (4 pages)
The BLITZ Emulator (44 pages)
An Overview of KPL, A Kernel Programming Language (66 pages)
Context-Free Grammar of KPL (7 pages)
BLITZ Tools: Help Information (13 pages)
The Format of BLITZ Object and Executable Files (12 pages)

Step 2: Read the Overview Document

Read the first document ("An Overview of the BLITZ System") before proceeding to Step 3.

Step 3: Choose Your Host Platform

You will develop your operating system code on a "host" computer and you will be running the BLITZ tools on that host computer. You should decide now which host computer you will be using.

The BLITZ tools run on the follow host platforms:

Apple Macintosh, OS X, either PPC-based or Intel-based machines
Sun Solaris
Unix / Linux Systems
Windows, using Cygwin which emulates the Unix POSIX interface (see `www.cygwin.com`)

For the following host platforms, the tools are precompiled and you can simply download the executables:

Apple Macintosh, OS X, Intel-based machines
Apple Macintosh, OS X, PPC-based machines
Sun Solaris

For other systems, you can download the tools (which are written in "C" and "C++". You must then compile them on your computer.

The source code for all the BLITZ tools is available, but you should not need to look at it. Nevertheless, it is available for anyone who is interested.

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 more-or-less in the 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.)

Organization of the Course Material

The BLITZ system is accessible via the following URL:

`http://www.cs.pdx.edu/~harry/Blitz/`

The **`Blitz`** directory contains the following material:

Blitz/
- **BlitzDoc/**
  - ...files containing the documents mentioned above...
- **BlitzBin/**
  - **MacIntel/**
    - ...executables for the Mac/Intel host platform...
  - **MacPPC/**
    - ...executables for the Mac/PPC host platform...
  - **Sun/**
    - ...executables for the Sun/Solaris host platform...
- **BlitzSrc/**
  - ...source code for the BLITZ tools...
- **OSProject/**
  - **p1/**
    - ...files related to project 1...
  - **p2/**
    - ...files related to project 2...
  - ...etc...

You may access this material through the BLITZ Home page. You should also be able to "ftp" directly to these directories.

Step 4A: For Mac Users...

Step 1: Create a directory to put the BLITZ tools into. For example, you may wish to create a directory called **BlitzTools** in your home directory:

`/Users/`*`YourUserName`***`/BlitzTools`**

Then copy all the files from

`www.cs.pdx.edu/~harry/Blitz/BlitzBin/MacIntel`

to your **BlitzTools** directory. These are binary files, not text files.

(I use an application called "Fetch" (`www.fetchsoftworks.com`) to do "ftp" file transfers.)

People using an older PPC-based Mac should use this directory instead:

`www.cs.pdx.edu/~harry/Blitz/BlitzBin/MacPPC`

Step 2: Set the protection bits on these programs to include "executable", with a command such as:

```
chmod  ugo+rx  BlitzTools/*
```

Step 4B: For Portland State University Students...

See section 4C.

Step 4C: For Users in a Shared Environment...

This section applies to users who have an account on a shared system, such as a Sun/Solaris system.

It is assumed that the BLITZ tools have already been downloaded by someone else and are available in some shared directory. All you need to do is modify your PATH variable so that your shell will search the appropriate directory.

For example, students at Portland State University who have an account on the shared Sun/Solaris system can find the executables in this directory:

`~harry/public_html/Blitz/BlitzBin/Sun`

Students at other institutions can find the executables in this directory:

Step 4D: For Unix/Linux Users...

This section applies to users who have a Unix/Linux box and wish to download and re-compile the BLITZ tools for their machine.

Step 1: Create a directory to put the BLITZ source code into. For example, you may wish to create a directory called **BlitzSrc** in your home directory:

~YourUserName/**BlitzSrc**

Then copy all the files from

www.cs.pdx.edu/~harry/Blitz/BlitzSrc

to your **BlitzSrc** directory.

Step 2: Compile the programs in

~YourUserName/**BlitzSrc**

There is a "makefile" so you should be able to execute the following commands to compile the tools.

```
cd ~YourUserName/BlitzSrc
make
```

This will invoke the "C" and "C++" compilers to produce the following executables:

```
kpl
asm
lddd
blitz
diskUtil
dumpObj
hexdump
check
endian
```

Step 3: Create a directory for the executables and move them into it:

```
mkdir ~YourUserName/BlitzTools
cd ~YourUserName/BlitzSrc
mv kpl asm lddd blitz diskUtil dumpObj hexDump check endian \
                    ~YourUserName/BlitzTools
```

Step 4E: For Windows Users...

Consult the document:

www.cs.pdx.edu/~harry/Blitz/OSProject/p1/BlitzOnWindows.pdf
www.cs.pdx.edu/~harry/Blitz/OSProject/p1/BlitzOnWindows.htm

Step 5: 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 **.aliases** file in your home directory.

For example, this file may already contain a line that looks something like this:

```
setenv PATH ${PATH}:${HOME}/bin
```

(Between each colon (:) is a directory specification. The above command sets PATH to whatever it was before followed by the bin directory in your home directory.)

What you need to do is add the BLITZ tools directory in front of whatever else is in the PATH.

Unix / Linux / Mac users who have placed the executables into a subdirectory in their home directory might add the following command to prepend the appropriate directory to the front of the PATH.

```
setenv PATH ${HOME}/BlitzTools:${PATH}
```

Users in a shared Sun/Solaris environment will need to know the shared directory containing the tools. Assume it is:

```
~instructorUserid/BlitzTools
```

Be sure to get the exact directory name, then add the following command after the last place PATH is set.

```
setenv PATH ~instructorUserid/BlitzTools:${PATH}
```

Portland State University students can add the following command after the last place PATH is set.

```
setenv PATH ~harry/public_html/Blitz/BlitzBin/Sun:${PATH}
```

The "bash" shell is a little different; these people should add something like this to `.bashaliases`:

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

The shell builds an internal hash table that speeds up the location of programs whenever you type a command. After changing your PATH, you'll need to restart your shell so that it uses the new PATH when it builds this hash table.

You can do this several ways. A Mac user can quit the "Terminal" application and then restart "Terminal". A Unix / Linux / Solaris user can log out and log back in. In some shells you can simply type the command "source .aliases" instead.

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 6: 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/**cs333**

Create a directory in which to place the files concerned with project 1. We recommend the following name:

~YourUserName/**cs333/p1**

Copy all files from:

http://www.cs.pdx.edu/~harry/Blitz/OSProject/p1/

to your **cs333/p1** directory.

The BLITZ Assembly Language

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.

Step 7: Assemble, Link, and Execute the "Hello" Program

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 8: 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 9: 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 10: 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 11: 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.

What to Hand In

Complete all the above steps.

To verify that you did all this, create a transcript of a terminal session showing what happened. In particular, please include the output associated with the following steps in what you hand in.

Step 7
Step 8
Step 9
Step 11

We do not need to see the other steps.

Hand in a hardcopy print-out showing what happened. If you do not know about creating a script file, check out the UNIX **script** command by typing

```
man script
```

In LARGE BLOCK LETTERS, write your full name.

Note that if you try to use a text editor while running **script**, a bunch of garbage characters may be put into the file. Please do not do this. After you have created your **script** file, it is okay to edit it to remove the entire editing session. We really don't want to see a transcript of your editing session. Alternately, you can start and stop **script**, creating several script files and then concatenate them.

PLEASE STAPLE ALL PAGES!

Grading for this Project

This project will be graded pass/fail.