CSCI 447 - Operating Systems, Winter 2021
Assignment 1: Introduction to the BLITZ Tools

Due Date: Tuesday, January 12, 2021
Points: 100

Questions from the book (50 points):

1. Review and answer for yourself 1.1, 1.2, 1.5, 1.6 and 1.10. (The e-book has answers to these questions so you don't need to turn in these answers)
2. Answers the following questions typed in an editor or a word processor. Hand written answers will get no points. It is possible you may need to do some reading outside of the book to answer these questions. Problems: 1.14 (10 points), 1.16 (15 points), 1.17 (5 points), 1.19 (5 points), 1.24 (5 points), 2.10 (5 points), 2.15 (5 points)

1 Overview and Goal

In this course you will be creating a very small operating system kernel. You'll be using the BLITZ software tools, which were written for this task. These were originally developed by Harry H. Porter who teaches at Portland State University. I have modified the tools for use here at WWU. *Do not use the original tools.* I will provide the tools, both executables for the Linux cluster and source code so you can use them on your machine. The goals of this project are to make sure that you can use the BLITZ tools and to help you gain familiarity with them.

2 The Documentation

There are a number of documents describing the BLITZ tools. The original documents are found at http://www.cs.pdx.edu/∼harry/Blitz/index.html. The local copy is found at https://facultyweb.cs.wwu.edu/∼phil/classes/blitz. These sites contain the following documents:

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

3 Read the Overview Document

Read the first document ("An Overview of the BLITZ System") before proceeding with rest of this document.

4 Blitz hosting

You will develop your operating system code on a "host" computer and you will be running the BLITZ tools on that host computer. I will be running your code on Blitz on the department's Ubuntu Linux systems. BLITZ also runs on a variety of other systems. I have installed BLITZ on the departmental Linux machines. If you want to run BLITZ on your computer, 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)

If you want to compile your on copy on your machine, the source code for all the BLITZ tools is available on the Linux machines for you to copy. The tar file is:

/home/phil/public/cs447/BlitzSrc.wwu.tgz.

5 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.)

6 Setup for use

Blitz has been installed on the department Ubuntu Linux machines in Phil Nelson's directory tree at the location /home/phil/blitz. To make them easy to run, add the directory /home/phil/blitz to your PATH variable in your login shell "rc" file. For bash the file name is .profile and may need to create on if you don't have one yet. (Use /home/phil/blitz-19 for the new tools.)

The following steps are for running blitz on your own machine.

- Copy the source tar ball to your machine. (see above for location)
- Extract the tar ball.
- Modify the "makefile" and set the make variable BINDIR to where you want your blitz tools installed. Notice, there is also a Makefile in the toyfs subdirectory that needs BINDIR set properly.
- The command "make install" will make and install the tools.
- Change your PATH environment variable to include tha value you put in the BINDIR definition.

In case you haven't added anything to your path before, edit or create the .profile file in your home directory. (Assuming you are using bash.) A line at the end of the file like:

```
PATH=$PATH:/home/phil/blitz
```

will make your shell look for commands in the blitz bin directory. After you edit the .profile file, you need to logout or just ask your shell to read the file by giving the command "source .profile". You may verify that whatever you did to the PATH variable worked.

At the shell 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.

7 Set up a gitlab project for 447 assignment turn-in

During this quarter, you will be turning in all your assignments via the department's gitlab server. This is the assignment in which you set up the structure. Please do the following steps:

1. Create a new gitlab project (gitlab.cs.wwu.edu) with the name csci447_w21. (Capitalization is important as is the underscore.)
2. Add "Phil Nelson" as a member to your project. I need at least "Reporter" access. You can have my access expire on April 1, 2021.

3. Clone your project to a working directory.
4. Create a README.md file with your name on one line and another line that says: "Winter 2021 – CSCI 447".
5. Commit the change.
6. Push the change back to gitlab.
7. Create a new *a1* branch.
8. Check-out the new *a1* branch.
9. Create a new directory *a1* and copy the files from the assignment to this directory. It should include the files DISK, *.s, *.h, *.k and the makefile.
10. Commit these files as distributed.
11. Push the new branch back to gitlab. This will normally require your first push of a new branch to look something like:

    git push –set-upstream origin a1

    You need to have the *a1* branch checked out when you give this command. You should not have an a1 directory on the branch "master".
12. Do your work on the *a1* branch for this assignment.

8 The BLITZ Assembly Language

In this course you may not have to write any assembly language code for the remaining assignments. (You will have to write some for this assignment.) However, during the course you will be using some interesting routines which can only be written in assembly. Most if not 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.

9 Assemble, Link, and Execute the "Hello" Program

The a1 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!".

10 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
>
```

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.

11 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".

11.1 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.k. These contain the program code. (Note: The original Blitz/KPL system used .c files for code ... but the compiler has been changed at WWU to use the .k suffix so these files are not confused with C programs by tools like emacs.)

The HelloWorld program makes use of some other code, which is contained in the files System.h and System.k. 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?

11.2 The "makefile"

The a1 directory contains a file called makefile, which is used with the UNIX make command. Whenever a file in the a1 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 ".k" 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.

11.3 Modify the HelloWorld Program

Modify the HelloWorld.k 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. Now is the time when you have some time to understand how to use the emulator to debug your KPL programs. Doing it for this assignment means that later you already know and won't feel like you need get the assignment done and don't have time to learn how to debug your program.

11.4 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.

11.5 Add some new code (15 points)

Using Echo.s as an example, add a new function to Runtime.s called GetCh that takes no arguments and reads an input character and returns that input character. Add the "external definition" of GetCh to System.h and make sure you export your new function in Runtime.s Show that your new function works correctly by writing a new function in HelloWorld.k that implements the same "echo" functionality as the Echo.s program. (Hint: Read the code for getCatchStack() in Runtime.s to figure out how to return data as the value of a function.)

DO NOT HAVE YOUR FUNCTION SET R15 LIKE Echo.s does. Also, do not recognize characters in your GetCh function. All it should do is to read a character from the device and return it. Nothing more. All the other functionality to make it do the same as echo must be in the HelloWorld.k file. This is what is called “polling I/O”. You have a loop that asks for a copy of the “control register” over and over again until it says a character is ready in the device. You can then read the character and return it to the calling function. WARNING: Blitz is a “big endian” machine and if you are not careful, you won’t return the correct value to the calling function.

12 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 assignment 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.

13 What to Turn In

First, make sure you have committed all your changes to the *a1* branch of your git repository for this class. Next, create a transcript of a terminal session showing you using the blitz system. You should at least run the echo program, your modified hello world program and your new function. If you do not know about creating a script file, check out the UNIX script command by typing

man script

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. Name your script file “a1-script” and add it to your *a1* branch. Don’t forget to commit it and push your *a1* branch. It is a good idea to use the web interface at gitlab.cs.wwu.edu to verify you have the correct files on the gitlab server as that is where I’ll get them for grading.

All that needs to be turned in via canvas for assignment 1 is your answers to the book questions.

For all assignments this quarter, please turn in your assignments with a cover sheet. The cover sheet must not have any content on it for the assignment. This cover sheet must include the Class (CSCI 447 Winter 2021), your name and the assignment number, for example, the following on an otherwise blank page:

```
Your Name
CSCI 447 Winter 2021
Assignment 1
```

Canvas is wanting a PDF file for the turn-in. There are a number of ways to generate a PDF file. OpenOffice/LibreOffice can produce PDF files. There is also a program called a2ps the creates a Postscript file from a text file and there is a companion program called ps2dpf that will create a PDF from a Postscript file. Taken together you can create a PDF file from a text file. There are many other ways that I won’t detail here.

So, for assignment 1, your PDF turnin should have:

```
1:  your cover sheet
2:  the answers to the questions
```

The turn-in time on canvas will determine if your assignment is late or not. I will also consult your commit times on gitlab to make sure your final commit is before the turn-in time. If you commit after the due date, your assignment is late.