Notes to Instructors Concerning the BLITZ Projects

Harry H. Porter III, Ph.D.
Department of Computer Science
Portland State University

April 14, 2006
Revised: September 17, 2007

Overview

The BLITZ System includes 8 student programming projects, which you may assign to your students.

The projects are all pieces of a puzzle that, when fully assembled, will be an OS kernel for the BLITZ machine. Many of the pieces of the kernel are provided by us, yet the student will be required to add critical pieces and fully understand all the code.

The kernel that students will build is a simplified version of Unix and will have these features:

- Multiple user-level processes
- Virtual memory using page tables
- A file system (on a simulated disk), containing user-level programs and other files
- A terminal, via which a user can communicate with a shell program
- System calls (including *Fork*, *Join*, *Exec*, *Exit*, *Open*, *Close*, *Read*, *Write*, *Seek*)

By the final project, students will be able to execute a simplified "shell" program, which invokes "fork" and "exec" system calls to execute other programs, and even a sub-shell.

All projects come with the following:

- An assignment document (both in MSWord and PDF formats)
- A directory with all necessary files

In addition, some projects have:

- Example output
- Solution code
- PowerPoint slides, for class discussion
- Grading forms, to facilitate grading

A lot of code is provided by us, but the students will write key portions and will need to read over and understand the entire kernel in order to fit the pieces together. The code we provide is given to the students a little bit at a time, over the 8 projects, rather than all at once, so that they can understand it incrementally.

The projects are designed so they can be assigned as is; you can simply refer your students to the BLITZ website. On the other hand, you are also free to edit the assignment documents any way you want.

Each project is intended to take one week to complete, with the first project being assigned at the first class meeting. Of course, you'll need to let your students know exactly what day you want each project completed.

For a 10 week term, the approach of eight assignments, of one week each, will leave a little time at the end of the term for extensions, etc.

Although many of my students are able to complete all 8 projects in a 10 week term, this is clearly an ambitious undertaking for some students.

The assignment documents are rather lengthy, but are designed to include as much information as necessary to complete the projects. The idea is to err on the side of providing too much information, rather than too little.

Each project is broken into several tasks. Each task serves a specific purpose and the intent is for all tasks to be assigned and completed, including those tasks in the early projects that don't seem to relate directly to building an OS kernel.

If you want to reduce the amount of programming required, I suggest you

- Extend the course from one term to two terms and double the project durations
- Eliminate the last one or two projects

I do not recommend simplifying or eliminating parts the projects, but you are certainly free to use this material in any way you want. For example, you might take a completed version of the entire kernel (i.e., all 8 projects) and just give it to students to read and study.

What Students Hand In: Each project instructs the students to print the code they write and to print the output from running their code. They are asked to hand in hardcopy of the code and its output.

Required Prerequisites: Students are assumed to already know:

- How to use the Unix shell

- How to edit text files
- How to download files (e.g., ftp)
- How to print files
- Assembly language programming

No knowledge of any specific machine architecture is assumed, but familiarity with some real CPU architecture is assumed. The BLITZ system includes some assembly code. The students are encouraged to look over it, but are not required to write any assembly code.

Students will be writing code. No knowledge of any specific programming language is assumed, although knowledge of "C" or "C++" is most appropriate. A language called KPL is used for the BLITZ projects. The first few projects are designed to introduce and teach KPL. Of course there is a learning curve for KPL, but experience has shown it is not a great barrier.

The language used in these projects, which is called KPL, was designed specifically for the BLITZ project. KPL is similar in capabilities to C++. It has pointers, arrays, classes, methods, interfaces, pointers to functions, the same expression syntax, and the ability to link with assembly routines. The syntax (which happens to be LL(1)) was designed to increase the readability of the code, and students will spend more time reading the code than writing it. The error reporting is quite thorough, which really helps student debugging. For example, null pointers and array-out-of-bounds are checked for and reported.

Class Organization Thoughts: My preference is to encourage students to work together—e.g., look at each other's code and even help with debugging—but to require each student to hand in a solution. I ask each student to write his/her own code, not to simply copy someone else's code. A reasonable alternative is to make this a group project.

I also create an email mailing list for my class. I require all students to subscribe and encourage students to post questions and answers to it. There is a lot of BLITZ documentation, and in my experience, most questions will be answered by other students who got the answers from a more careful reading of the documentation.

Managing the Documentation: The BLITZ documentation is available online as PDF files. One possibility is for you to print all the material, deliver it to your local copy shop, and suggest that students purchase a pre-printed, bound copy of everything. I strongly recommend that my students purchase the packet from the copy shop, but if they want to print it themselves, I recommend they print everything at once and then put it all in a single binder.

Additional Support Information: Additional files, such as blank "grading forms" and PowerPoint slides, can be found in the directory

```
www.cecs.pdx.edu/~harry/Blitz/InstructorInfo/
```

Project 1: Intro. to the BLITZ Tools

Duration: 1 week
Document size: 13 pages

Goals: Make sure the students can execute the BLITZ software tools; Make sure students realize the course has a programming component; Allow time for students to obtain computer accounts, if necessary; Allow time for the lectures to introduce important concepts before any substantive challenges are issued.

This project should be assigned at the first class meeting.

This is a warm-up project, which requires the students to perform the following steps:

- Print and begin reading the BLITZ documentation
- Make the tools usable
- Demonstrate the BLITZ framework for future projects
- Startup the BLITZ virtual machine emulator
- Compile, assemble, and execute some demo programs
- Examine some assembly code and some KPL code
- Practice a few commands in the debugger

This assignment is not difficult. There is a script and the students simply follow each step in turn. This project verifies the students have familiarity with the necessary Unix tools. No OS-specific knowledge is needed to complete this project.

This project should be graded pass-fail and should take 1 or 2 hours. The only reason to fail is that the student didn't do it or lacks the prerequisite knowledge.

Project 2: Threads and Interprocess Communication

Duration: 1 week
Document size: 11 pages

Goals: Start to learn the KPL language; Learn about threads (ready-list, idle process, thread switching, thread status, round-robin scheduling) and functions like *sleep* and *yield*; Learn about interprocess communication (*semaphores*, *mutex locks*, and *monitors*); Confront two classic IPC exercises (*Producer-Consumer* and *Dining Philosophers*); Provide motivation to understand the monitor concept.

In this project, students will mostly read code. They will begin to use KPL by filling in the bodies of some methods. Little design freedom is given and much of the code can be written by analogy with existing code, which they will need to study.

In this project, we give the students a completely functional round-robin thread scheduler, an implementation of *semaphores* and an implementation of *condition variables*. The project is divided into these tasks:

- Examine existing code, looking at the thread scheduler, the implementations of *fork*, *yield* and *sleep* and sample multi-threaded code that uses these functions
- Perform several experiments with this existing code to understand thread switching
- Implement *mutex locks* (using the technique of disabling interrupts, invoking *sleep*, and moving threads onto the ready-list)
- Implement a *Producer-Consumer* solution using semaphores
- Implement a *Dining Philosophers* solution using monitors

By implementing mutex locks, students are encouraged to look at and understand how semaphores are implemented. Their implementation of mutex locks will be needed later on. [As part of project 3, two implementations of mutex are distributed (one which disables interrupts and one which uses semaphores) in case the student was unable to complete this step.]

In the Producer-Consumer task, students are asked to implement a solution to the bounded-buffer producer-consumer problem with several producer and several consumer threads. A solution is given in the Tanenbaum textbook and the students are asked to recode this in KPL.

In the Dining Philosophers task, students are asked to implement a solution to the Dining Philosophers task using the monitor construct. This task is much harder. The Tanenbaum text contains a solution using mutexes and semaphores and it is fairly straightforward to re-code this in KPL. However, the students are asked to use monitors in their solution. At this point, students are still grappling with threads, mutex locks and semaphores so the monitor concept is overly challenging. Nonetheless, this difficult task gets the students to play close attention to the solution, which should be discussed in class. In later projects, they will need to implement monitors (e.g., to manage page frames), so this is the first step is really about understanding the importance of the monitor concept.

We provide PowerPoint slides giving a solution to the Dining Philosophers task, which should be covered in lecture after the assignment is collected. The solution to the mutex problem is distributed with the code for the next project, since it is critical that mutex locks are implemented correctly.

A testing framework for mutexes, Producer-Consumer and Dining Philosophers is included. We also provide sample output (as a PDF file) showing students what correct output looks like, which may help in grading.

Coordination with lectures: This project is intended to be given at the beginning of the second week of class and due at the beginning of the third week. In order to give students enough information to complete this project on this schedule, the lectures must cover the following topics within the first two weeks: Concurrency and threads, ready/running/blocked, race conditions, critical sections, mutual exclusion, mutex locks, semaphores (both semantics and implementation), a producer-consumer solution using semaphores, discussion of the dining philosopher scenario, monitors and condition variables.

In my class, I cover all this by the end of the second week, and then give students the weekend to complete the project. In actuality, most students don't fully grasp monitors or condition variables, but are really primed for looking at the solution code for Dining Philosophers at the beginning of the third week.

Project 3: Barbers and Gamblers

Duration: 1 week
Document size: 6 pages

Goals: Reinforce the concepts related to multi-thread synchronization: mutexes, semaphores, condition variables and monitors; Gain additional proficiency in KPL and learn to create entire classes from scratch.

This project is divided into two tasks:

- Implement a solution to the *Sleeping Barber* IPC problem
- Implement a solution to the *Gaming Parlor* IPC problem, using a monitor

Students will use the same thread scheduler as the previous project and really start to understand the concepts introduced in the previous project. This time the students are not given any portion of the solution. They will need to create new classes and new threads on their own.

The Tanenbaum text provides a solution to the Sleeping Barber problem, which the students must translate into KPL. The Tanenbaum text does not discuss the Gaming Parlor problem.

As part of the assignment, we provide sample output and discuss how the students might structure their output.

We also provide a solution to the Gaming Parlor task (as PowerPoint slides), which should be covered in lecture after the assignment is collected. No solution code for the Sleeping Barber task is provided.

We should note that the Gaming Parlor monitor is very close to what the students will need for the next project, in particular the "FrameManager." After working on the previous problems (Producers-Consumers, Dining Philosophers and Sleeping Barbers), some students will be in a position to complete the Gaming Parlor task successfully. But more importantly for the ongoing project, most students should be ready after the Gaming Parlor to correctly implement the monitors needed to manage kernel resources (e.g., process-control-blocks and memory-frames) in the next project.

Project 4: Kernel Resource Management

Duration: 1 week
Document size: 17 pages

Goals: Cement the concepts of thread synchronization, including mutex locks, semaphores, condition variables, and monitors; Introduce concepts like process control block and page frames; Create three monitors which will be used to manage kernel resources; Explore the differences between Hoare semantics and Mesa semantics for monitor condition variables.

This project is divided into these tasks:

- Create a monitor to manage the Thread objects
- Create a monitor to manage the Process objects
- Create a monitor to manage the Frames
- Implement Hoare semantics for monitor condition variables

Each thread in our OS will be represented by an object of class "Thread." These objects are a limited kernel resource which will be managed by a monitor called "ThreadManager."

Likewise, each process will be represented by an object of the "ProcessControlBlock" class. The ProcessControlBlocks are also kernel resources which will be managed by a monitor called "ProcessManager."

Physical memory will be divided into frames and each user-level process will require a virtual address space made of pages. Each page will be stored in a frame. The frames are a critical resource which will be managed by a monitor called "FrameManager."

The last task of this project—to implement "Hoare Semantics" for condition variables—is not strictly necessary and is included as an additional challenge for the better students.

We are including code to test the first three tasks, but students are asked to figure out how to test the last task. Since the first three tasks are critical to the ongoing project, our testing code really hammers on these monitors in an attempt to break them.

Nothing prevents students from extracting the sample output from the PDF files we are distributing and simply submitting that. However, since the thread switching should be occurring at random times, each students' output should differ slightly. Anyone who hands in output that exactly matches the given output has some explaining to do.

Project 5: User-Level Processes

Duration: 1 week
Document size: 34 pages (including 2 pages of sample output)

Goals: Implement a single user-level process running in its own address space; Introduce system calls and the distinction between kernel and user code; Introduce the page table hardware; Introduce the DISK and its device driver and the file system connections.

This project is divided into these tasks:

- Get a user-level process running
- Setup the system-call interface
- Implement system call *Exec*

The operating system we are creating will implement these system calls (although only *Exec* will be implemented in this project).

Exit
Shutdown
Yield
Fork
Join
Exec
Create
Open
Read
Write
Seek
Close

In this project we supply code for a minimal file system, using the (emulated) disk. The user-level programs are written in KPL and compiled on the host platform, just like the kernel itself, and we provide a utility to initialize the emulated disk and copy files from the host to the emulated disk and vice versa.

In the second task, students will put together the system call interface. Although each syscall is an unimplemented stub in the kernel, this step assures that the user-level code can invoke kernel functions.

In the last task, students implement the *Exec* kernel routine, which will read a new executable program from disk and copy it into the address space of the process which invoked the *Exec* and then begin executing it. This task requires the students to understand the virtual memory software and the file system software.

Students are given just enough of the file system to achieve the task without having to modify the file system code; the remainder of the file system will be addressed in Project 7.

The simulated disk will contain some user-level test programs we supply. The students are instructed to run these tests and hand in the results.

Project 6: Multiprogramming With Fork

Duration: 1 week
Document size: 21 pages (including 6 pages of sample output)

Goals: Implement the following system calls: *Fork*, *Join* , *Exit* and *Yield*; Gain additional familiarity with the system call interface and with the semantics of these system calls; Understand the parent-child relationship between processes; Understand multiprogramming; Understand simple synchronization between Unix processes using *Exit* and *Join*; Understand the complexities of managing shared kernel resources.

This project is divided into two tasks:

- Implement *Fork*
- Implement *Join*, *Exit* and *Yield*

In order to implement *Fork*, the students will have to look more carefully at exactly how the system call interface works. There are also a number of deadlock and race conditions that must be avoided.

The simulated disk will contain some user-level test programs we supply. The students are instructed to run these tests and hand in the results.

Project 7: File-Related Syscalls

Duration: 1 week
Document size: 8 pages

Goals: Implement and understand the following system calls: *Open*, *Read*, *Write*, *Seek* and *Close*; Understand the Unix file model; Understand how the kernel buffers information between the disk and processes.

This project asks students to implement these system calls:

- *Open*
- *Read*
- *Write*
- *Seek*
- *Close*

The simulated disk will contain some user-level test programs we supply. The students are instructed to run these tests and hand in the results.

Sample output is provided in a separate file called **DesiredOutput.pdf**.

Project 8: The Serial I/O Device Driver

Duration: 1 week
Document size: 16 pages

Goals: Implement a device driver for a serial character-oriented terminal; Integrate the new device into the file system syscalls; Understand the difference between block and character devices; Understand and execute a shell program.

This project asks the students to implement a device driver for a serial ASCII terminal and integrate it into the existing system call interface (e.g., *Open*, *Read*, *Write*, etc.) This will allow the user to communicate directly with user-level programs by providing a "stdin" and "stdout" for file system calls to use.

With the completion of this project, everything is in place for a "shell" program and a simple shell program is provided by us. This shell is capable of executing commands and of redirecting the stdin/stdout from the terminal to files, using the Unix < and > redirection operators. We also provide a program called "cat," which is essentially the Unix "cat" command, and some other files. The shell is (naturally!) capable of executing script files by recursively invoking itself recursively with I/O redirection.

As an extension, we mention that the students might consider implementing "pipes," but aside from some minimal discussion, no further code is provided.

The simulated disk will contain some user-level test programs we supply. The students are instructed to run these tests and hand in the results.

Sample output is provided in a separate file called **DesiredOutput.pdf**.

Extensions / Additional Project Ideas

Adding Unix "pipes," as discussed in project 8, might constitute a separate project.

In our kernel, each process has exactly one thread. This could be generalized to allow each process to have more than one thread, opening up some design choices about what sort of concurrency control primitives the kernel ought to provide.

The file system used in this project is pretty rudimentary. For example, new files may not be created. (All files are placed on the disk by a BLITZ utility before the kernel begins.) Files may not be enlarged and are allocated in contiguous sectors. There is only a single-level directory. All of this could be improved.

There is no concept of security or userids, and this could be added.

The kernel implements virtual memory using the page tables and a process's address space may be scattered all over physical memory. However there is no swapping of pages out to disk: all the pages of an address space must fit into memory. Paging to disk could be added.

In this kernel, address spaces are copied during *Fork*. Another idea is to add the ability to share pages between processes and/or copy-on-write semantics.

As another idea, it would be interesting to implement the kernel calls needed in a message-passing kernel like Mach.