The Thread Scheduler
and
Concurrency Control Primitives

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

September 26, 2007
Revised: August 19, 2008

The BLITZ Thread Scheduler

This document examines the design of a specific kernel scheduler, the *BLITZ Thread Scheduler*. We'll look at data structures, at algorithms, and even at code fragments in an attempt to explain exactly how a real thread scheduler works.

The BLITZ thread scheduler is part of the *BLITZ System*, which is a collection of software designed to support a university-level course on Operating Systems. Over the course of one or two terms, students will implement a small, but complete, operating system kernel. The BLITZ software provides the framework for these student projects.

Who this Document is Written For

This document is written primarily for students using the BLITZ system in an Operating Systems course, but familiarity with the BLITZ system is not assumed. Although this document is basically a *code walk-through* of the BLITZ thread scheduler code, it can be read by anyone interested in thread schedulers.

Familiarity with the BLITZ system is not assumed and is not needed in order to read this.

This document will be of greatest help to students working on "project 2" in the BLITZ build-your-own-kernel project. Students do not need to read this document in order to do project 2. Nevertheless, this document goes into much greater depth and will provide deeper insights into the concepts of thread scheduling in general and the BLITZ scheduler in particular.

This document may also be of interest to instructors who are using the BLITZ system in their courses.

A Quick Introduction to Multithreading Concepts

At the core of an operating system kernel lies the *thread scheduler*. The scheduler provides the illusion that many CPUs are available, operating in parallel, when only one CPU exists. The scheduler implements *multithreading* by rapidly switching the CPU from one program to the next, so that each program can make progress over time.

In other words, the CPU executes one program for a short time, then switches to another program and executes it for a while. Each individual program is executed for a while and then suspended while other programs are executed. After each program has been given a turn, the process is repeated. And each program is given another turn to execute.

The term multithreading should be contrasted with *multiprocessing*, which refers to a computer system composed of more than one CPU. In a multiprocessing system, two or more programs can truly execute in parallel, since each program can be executing on a different CPU at the same moment.

With multithreading, the switching between one program and the next is done by the scheduler and is done in such a way that each individual program is unaware that the switching is being done. From the perspective of an individual program, it appears that a single CPU is dedicated to its execution. From the program's point of view, it is impossible to determine or distinguish whether every program is executing on its own private CPU or whether a single CPU is being shared by all programs.

When the programmer writes code to be executed in a multithreaded environment, he or she doesn't do anything special. In particular, the programmer doesn't need to think about when the switching will occur or whether the program will be executed in a multithreaded environment or in a multiprocessing environment.

Multithreading is sometimes called *multitasking* or *multiprogramming*; for our purposes, these terms all mean the same thing and we will avoid them in this document. Another term for multithreading is *time-slicing*, which refers to the idea that time is broken into many sequential intervals. Each program is allowed to use the CPU for one of "slice of time" before the program is suspended and the CPU is put to work on another program.

Multithreading and multiprocessing are often combined. It may be the case that the computer is a multiprocessor system which has, say, 10 CPUs. A thread scheduler designed for a multiprocessor system could utilize these 10 real CPUs to execute, say, 50 programs at once. Each of the programs would be suspended for part of the time and each CPU would be time-sliced to execute, on average, 5

programs. In such a scheduler, many complexities must be addressed, such as whether each program will always be executed on the same CPU or whether its time-slices of execution may occur on different CPUs.

The BLITZ Hardware

The scheduler described here runs on the BLITZ computer, which has a single CPU. The architecture of the BLITZ machine is similar to modern architectures, such as the SPARC, but is somewhat simplified.

At all times, the BLITZ CPU is either running in *System Mode* or in *User Mode*. Certain instructions, called *privileged instructions*, can only be executed when the CPU is in System Mode.

The CPU has 16 general purpose *integer registers* (called **r0** through **r15**) with 32 bits each. In addition, there are 16 floating point registers (called **f0** through **f15**) which can each hold a double precision floating point value.

There are 2 sets of the general purpose integer registers, one for System Mode and one for User Mode. There is only one set of the floating point registers.

Additional registers include a program counter (**PC**); a status registers (**SR**), and two registers related to the page table.

The BLITZ machine has an interrupt mechanism. Interrupts can be signaled from the following hardware and external sources:

Timer
Disk
Serial Terminal
Power-on Reset
Hardware Fault

The interrupt mechanism also processes program exceptions arising from the following causes:

Address Exception
Alignment Exception
Arithmetic Exception
Privileged Instruction
Illegal Instruction
Page Fault (invalid page)
Page Fault (update to read-only page)
Syscall Trap Instruction

The *timer interrupt* is generated by the hardware regularly at a fixed periodic rate, providing the necessary "clock tick" needed for a time-slicing scheduler.

The CPU includes a *MMU* (*Memory Management Unit*) to support paging and virtual memory. The I/O devices include a disk for storing files and a terminal for communication with the user.

The scheduler is programmed mostly in a high level language called *KPL (Kernel Programming Language)*, but some critical routines are coded in assembly code. [By the way, KPL should not be confused with something called *Kid's Programming Language*. This KPL is definitely not for kids!]

The BLITZ computer is emulated using a virtual machine. The BLITZ architecture was carefully designed to be realistic and complete enough to execute an operating system kernel. The thread scheduler discussed in this document forms the core of a functional Unix-like kernel, which is elaborate enough to execute a simple shell program.

The BLITZ architecture was designed with this single goal in mind: to facilitate the teaching of operating systems. Consequently, much of the complexity of modern CPU cores was avoided. For example, the BLITZ CPU was designed with the expectation that it would be emulated. Several hardware optimizations—such as pipelined execution, out-of-order execution and instruction/data caching—were not included, since these have little relevance to this goal.

The following documents contain more information about the BLITZ system:

An Overview of the BLITZ System (7 pages)

An Overview of the BLITZ Computer Hardware (8 pages)

The BLITZ Architecture (71 pages)

An Overview of KPL, A Kernel Programming Language (66 pages)

All of these documents and many more may be accessed through the BLITZ webpage:

http://web.cecs.pdx.edu/~harry/Blitz

The Thread Concept

Previously, we used the term "program" loosely in the discussion of multithreading, but technically a program is a static thing, a bunch of code. At a given moment in time, it might not even be loaded into a computer. But once the program is loaded into the computer's memory and execution begins, it becomes a *process*. In other words, a process is a program in execution.

A process is sometimes called a *task*.

A process consists of two things: a memory region containing the code and a *thread* of execution. In the simplest scenario, with no operating system, the program would be loaded into the computer's memory

directly and the CPU would execute the instructions of the program, one after the other. In this simple scenario, the CPU itself provides the thread.

The thread breathes life into a program as it executes instructions; without a thread, a program is nothing more than a bunch of bytes stored in memory. Without instruction execution, the code will never be executed, memory will never change, and nothing will ever happen. The thread brings activity to the instructions stored in the bytes of memory.

Next, imagine a computer with several CPUs, all sharing the same memory. Assume we got a *symmetrical* system. There is only one block of memory and each CPU can freely access all of this memory. Each CPU is identical and any CPU can read or update any byte. Any update will be visible to every other CPU.

If some program has been designed to be executed on such a system with multiple CPUs, it is said to be a *multithreaded program*. Each CPU will implement a single thread; with 10 CPUs we'll have 10 threads. (We're still assuming there is no operating system yet.)

In general, each CPU will be executing at different places in the program, although it is certainly possible that each of the CPUs is executing the same instruction at the same moment. It is even possible that all CPUs are marching in lockstep, doing exactly the same thing. This synchrony of CPUs is possible but would be unusual and occur mostly in fault-tolerant systems with "hot" backup CPUs.

In a more common scenario, each CPU is doing something different. All CPUs are executing instructions from the same program, but each CPU is at a different point in that program. In a typical multithreaded program, the CPUs are all working on some coordinated, shared task, but each is doing something different to contribute to the common goal. There is one program—one set of instructions and one set of shared, global variables stored in the shared memory—and these are shared by all CPUs.

Of course, in order to complete the task correctly, each of the CPUs will need to communicate and coordinate with the others. This is the problem of *concurrency control*. There must be mechanisms whereby each CPU can interact with the others. For example, one CPU may provide some piece of data that another CPU will need. If the second CPU becomes ready for the data before the first CPU has produced it, the CPU will have to wait.

We will discuss concurrency control mechanisms in depth later in this paper.

Now let us return to a single CPU system. In order to execute a multithreaded program with only a single CPU, we need a scheduler. The scheduler implements threads. The scheduler provides the illusion that there are many CPUs available. In particular, the scheduler creates a new thread whenever the process asks for one, providing the illusion that the computer has as many CPUs as the process needs. It is as if there are an infinite number of idle CPUs standing by, available to be called into action whenever the process wants a new thread. Each CPU will have access to the same memory, so each CPU can access the variables and the instructions stored in memory.

In a sense, a thread is nothing more than a *virtual CPU*.

As a starting example, let's look at the small code fragment shown next. This code is written in the BLITZ assembly language:

```
            mov     300,r1
            jmp     MyLabel
            mul     r1,500,r3
            ...
MyLabel:    add     r1,200,r3
            ...
```

The code first moves the number 300 into register **r1** and then jumps to the instruction labeled `MyLabel`. Then the `add` instruction will add 200 to the value in **r1** and store the result (i.e., 500) into register **r3**. The instruction following the jump is `mul` (multiply); apparently it will never be executed.

Next, imagine that our computer also has a "fork" machine instruction to implement multithreading on a machine with multiple CPUs. (In reality, the BLITZ computer has only one CPU and does not have any such "fork" instruction.)

Now let's change the `jmp` instruction to this hypothetical `fork` instruction:

```
            mov     300,r1
            fork    MyLabel
            mul     r1,500,r3
            ...
MyLabel:    add     r1,200,r3
            ...
```

This hypothetical `fork` instruction is a little like a `jmp` (or branch or goto) instruction, but with an important difference: a new thread will be created each time the `fork` instruction is executed.

When the `fork` instruction is executed, it will call into service a second CPU, which we can assume was standing idle waiting to be needed. The original CPU will continue instruction execution just after the `fork`, and the multiply instruction will in fact be executed next by that CPU. However, the second CPU will begin executing instructions, starting at `MyLabel`, with the `add` instruction.

So the `mul` and the `add` instructions will both be executed, but on different CPUs.

Notice that the newly activated CPU, when it executes the `add` instruction will be starting in the middle of a program. Its very first instruction will be the `add` instruction, which makes use of the value previously stored in register **r1**. In order for the newly activated CPU to begin executing in the middle of this program, we must assume that its registers are preloaded with exactly the values the registers had at the moment of the `fork` instruction.

Also note that register **r3** will be immediately changed by both CPUs and, moreover, **r3** will be given different values by each CPU. Since each CPU has its own set of registers, this is fine.

More generally, notice that whenever we start a new thread, it must be started as a "fork" to an existing thread and that the entire state of the previous thread must be copied to create a starting state for the new thread. A copy must be made—it cannot be shared—because the moment the new thread begins executing, its state will change and diverge from the other thread's state.

The OS Kernel

The scheduler to be described here runs on a single CPU. It provides the illusion of multiple CPUs by implementing threads and time-slicing.

We'll discuss the fork operation in detail later, but in short, the fork operation will create a new thread, making a copy of the previous thread's state, and will initiate the execution of the new thread. After the fork operation, there will be one more thread than before. Conceptually, both threads will begin executing simultaneously, in parallel. But since there is only a single CPU, only one of the threads at a time can be given a time-slice while the other thread will have to wait for its turn to run.

The thread scheduler we are about to describe here will go on to be used as the core of an operating system kernel in the BLITZ project. However, this document will concern itself only with the thread scheduler, which can be understood and used in isolation. While this thread scheduler will be embedded within a kernel later, we will describe it here as a stand-alone program. We will not discuss any of the other functions of an operating system, such as user-level processes, virtual memory and paging, or device I/O.

You can have a thread scheduler without any other parts of the operating system, but you really can't have an operating system without a thread scheduler. The code we describe here consists of only the thread scheduler and some concurrency control operations. We'll show how to write multithreaded code using the functionality provided by the scheduler. Later, the other functions of the operating system can be added to the code body discussed here.

An operating system kernel is, in some sense, just another program. However, unlike other programs, the kernel is loaded directly into the computer's memory during the booting process. All other programs are loaded into memory by the operating system and are tightly controlled by the operating system. All other programs assume the presence of an OS. In order to communicate with the outside world—for example, to talk to users or access files on the disk—programs must interact with the OS. On the other hand, the OS will interact directly with the hardware devices, such as the terminal console and the disk.

The scheduler code we'll discuss here, which is meant to form the core of an OS kernel, is a standalone program which will be loaded directly into physical memory, during a sort of "boot" process. The code discussed here will not be run on top of an operating system. In fact, there will be no other code in memory but the program discussed here.

Since there is no operating system below it, this scheduling program will have to include all the functionality it needs. It can't invoke the operating system to do anything. As a consequence, any subroutine or function that might be needed must be included directly and linked into the executable.

The Thread Scheduler

The BLITZ thread scheduler uses the simplest possible scheduling algorithm. It maintains a collection of all the threads—called the *ready list*—and runs them in *round-robin* fashion. There are no priority levels nor are there multiple ready queues.

Each time the scheduler is invoked, the first thread on the ready list is selected and executed on the CPU. The thread is executed until the next timer interrupt, at which time the thread is placed at the tail end of the ready list. Then, the next thread is taken from the front of the ready list and executed. Each thread will get roughly a full slice of time; only the scheduling overhead at the beginning of the time-slice is lost.

Each thread is represented by a data structure called a **Thread**. The KPL language supports classes and object-oriented programming and there is a class called **Thread**. Each instance (or object) in this class represents an individual thread.

For reference, here is the definition of the class called **Thread**:

```
class Thread
  superclass Listable
  fields
    regs: array [13] of int      -- Space for r2..r14
    stackTop: ptr to void        -- Space for r15 (system stack top ptr)
    name: ptr to array of char
    status: int                  -- JUST_CREATED,READY,RUNNING,BLOCKED,UNUSED
    initialFunction: ptr to function (int)    -- The thread's "main" function
    initialArgument: int                      -- The argument to that function
    systemStack: array [SYSTEM_STACK_SIZE] of int
  methods
    Init (n: ptr to array of char)
    Fork (fun: ptr to function (int), arg: int)
    Yield ()
    Sleep ()
    CheckOverflow ()
    Print ()
endClass
```

First, some remarks about the KPL language: In this document, the keywords of the language are underlined. Comments begin with two hyphens (--).

The data types in KPL should be familiar to anyone who knows C++ or Java. Basic types include int, char, double, and bool. KPL also allows pointers and arrays. Some more complicated types from the above example are given below.

```
int                     -- A 32-bit signed integer
array [13] of int       -- An array of 13 ints
```

```
array of char           -- An array of chars, with unspecified size
ptr to array of char    -- A pointer to an array of chars
ptr to void             -- A pointer to anything
function (int)          -- A function taking a single int argument
ptr to function (int)   -- A pointer to such a function
```

In KPL, each class is defined with two things: a *class specification* and a *class implementation*.

The code given above is an example of a class specification.

A class specification tells which fields (i.e., data members) are in the class, after the `fields` keyword). It also tells which methods (i.e., member functions) are implemented in the class, after the `methods` keyword. A class specification also shows where, in the hierarchy of classes and superclasses, the class lives, after the `superclass` keyword.

This specification indicates that **Thread** is a subclass of a class called **Listable**, i.e., the superclass of **Thread** is **Listable**. (We'll discuss **Listable** later.)

A class implementation gives the method bodies (i.e., the code of the methods). The class implementation for **Thread** is not shown here, but we'll look at the code of several of the methods below, as we discuss how the scheduler works.

Let us comment here about each field in a **Thread** object.

The CPU contains 16 general purpose registers, called **r0** through **r15**. Register **r0** always has a value of zero, which provides handy access to a commonly needed constant, namely zero. Instructions may also use **r0** when a result should be discarded. Consequently, we'll never need to save **r0**.

In the BLITZ architecture, register **r15** has a unique and important role: it is the *stack pointer*. During `call` instructions, a return address is pushed onto the stack, using **r15**. During a return (`ret`) instruction, the return address is popped using **r15**.

When a thread is running, the CPU registers will contain values that change as instructions are executed. But when a thread is suspended (e.g., stopped to let another thread use the CPU), we'll need to save the state of the CPU. In particular, we need to put the contents of the registers (as used by the suspended thread) somewhere while other threads use the registers.

The first field in **Thread** is named **regs**. It is an array with enough space to store registers **r2** through **r14**. Later, we'll see why we never need to store register **r1**. By the way, arrays in KPL are numbered from zero, just like in Java or "C". Thus register **r2** will be stored in **regs[0]**, and so on, up to **r14** in **regs[12]**.

The next field **stackTop** will hold the value of register **r15**.

The next field is called **name**. It holds a pointer to a string which, in KPL, is an array of characters. Each thread is given a name, which is useful when printing certain error messages. Conceptually,

threads don't need to have names and in some operating systems threads are referred to by numbers or by the addresses in memory of their **Thread** objects.

The next field is called **status**, which contains an integer code number.

KPL includes an ability to equate a sequence of numbers with names, via a construct called an *enum*. The following KPL code assigns the numbers 1, 2, 3, and so on, to constants named **JUST_CREATED**, **READY**, **RUNNING**, **BLOCKED**, and **UNUSED**:

```
enum JUST_CREATED, READY, RUNNING, BLOCKED, UNUSED
```

The next field is called **initialFunction**, which contains a pointer to a function. This function is the thread's "main" function and is the function that will be executed when the thread begins execution. It takes a single integer as an argument and returns nothing.

The next field is called **initialArgument**. It contains an integer, which will be passed to the initial function when the thread is first started. The **initialFunction** and **initialArgument** fields are only used when starting a thread and are not used after that.

The last field is called **systemStack** and is an array of 4000 bytes. The size of an integer is 4 bytes and the size of the array is determined by the following constant definition:

```
const
  SYSTEM_STACK_SIZE = 1000
```

By the way, all of the material discussed so far is in the file named **Thread.h**. The following files will be discussed in this document:

Thread.h
Thread.c
Runtime.s
Switch.s
System.h
System.c
Synch.h
Synch.c
List.h
List.c

These files can be found at:

http://web.cecs.pdx.edu/~harry/Blitz/OSProject/p2/

Linked Lists

Class **Thread** is a subclass of class **Listable** and consequently **Thread** objects may be placed on linked lists. The **Listable** superclass essentially provides a field called **next**, which points to the next **Thread** object. Therefore, every **Thread** object inherits a **next** field from **Listable**. Since each object has only one **next** pointer, a **Thread** object cannot be placed on more than one linked list.

Linked lists are "singly linked", which means that each element in the list has a "next" pointer but no "prev" pointer.

There is also another class named **List[]**. An instance of this class will represent the list as a whole. A **List[]** object contains two fields called **first** and **last**, which point to the elements at each end of the list.

By maintaining pointers to both first and last elements, adding an element to either the front end or the tail end of the list is efficient. However, since the elements do not contain "prev" pointers, removal of any element besides the first element is not supported by the class.

The class **List[]** is an example of a parameterized class in the KPL language. To define a variable of type **List[]**, the programmer must specify a type, such as **Thread**.

Below is a collection of important variable definitions. These variables are global variables, which means that they are accessible from all methods and functions.

```
var
    readyList: List [Thread]
    currentThread: ptr to Thread
    mainThread: Thread
    idleThread: Thread
    threadsToBeDestroyed:  List [Thread]
    currentInterruptStatus: int
```

We'll describe these variables more later, but notice that both **readyList** and **threadsToBeDestroyed** are lists of **Thread** objects. The type-checking system of KPL is strong and safe enough to assure that only **Thread** objects are placed on these two lists.

Here is the specification of **List[]** and **Listable**, with some simplifications that can safely be ignored.

```
class List [T: Listable]
  superclass Object
  fields
    first: ptr to T
    last: ptr to T
  methods
    AddToFront (p: ptr to T)
    AddToEnd (p: ptr to T)
    IsEmpty () returns bool
    Remove () returns ptr to T
    ApplyToEach (f: ptr to function (ptr to T))
endClass

class Listable
```

```
  superclass Object
  fields
    next: ptr to Listable
endClass
```

Note that we have operations to add a **Thread** to the head of a list (**AddToFront**), to add a **Thread** to the tail of a list (**AddToEnd**), to test whether a list is empty (**IsEmpty**) and to remove and return a **Thread** from the front of the list (**Remove**). These methods work with pointers to **Thread** objects.

There is also a method called **ApplyToEach** which is passed a function—or more precisely, a pointer to a function. **ApplyToEach** will invoke that function once for each element of the list, supplying that element to the function. For example, there is a function called **ThreadPrint** which will print the contents of a **Thread** object in human-readable form. To print the entire **readyList**, use this code:

```
readyList.ApplyToEach (ThreadPrint)
```

The code for the classes **List[]** and **Listable** is in the files **List.h** and **List.c**.

We should note that this approach to handling lists, while adequate for our purposes, is somewhat less complex than what other OS kernels do. For example, in Linux, a single object might participate in several kinds of list and might in fact be on two, unrelated lists simultaneously; our single **next** field is inadequate to allow a **Thread** object to be placed on more than one list. Fortunately, in our system, a **Thread** will never be on more than one list at a time. Also, in Linux, lists are kept doubly linked, with both "next" and "prev" pointers, which allows elements to be removed efficiently from any place in the list, not just the front.

Variable Initialization

In KPL, all global variables, like the ones shown above, will be initialized to zero values, unless a specific initial value is given. For example, the variable named **currentInterruptStatus**—an integer—will be initialized to zero.

The variable named **currentThread** is a pointer; even with no explicit initialization, KPL guarantees that it will be initialized to the "null" value. In KPL it is common to rely on this initialization when a zero or null initial value is desired, so it is common to see code where variables are not explicitly initialized.

The remaining variables (**readyList**, **mainThread**, **idleThread**, **currentInterruptStatus**) will contain objects. Note that these four variables do not contain pointers to objects (like **currentThread** does), but have enough bytes allocated to hold the entire object.

Variables containing objects must always be initialized properly before use. This is because each object contains a hidden field pointing to its class. This hidden "class" pointer is used during method lookup to dynamically select the correct methods, just as in other object-oriented languages.

In the case of the variables named **mainThread** and **idleThread**, we see no explicit initialization, so these variables are not usable until initialized. These variables are initialized with this code in the function called **InitializeScheduler**:

```
mainThread = new Thread
idleThread = new Thread
```

In KPL, the "new" expression creates a new object—of class **Thread** in this case—with all fields initialized to zero values. That object is then copied into the variable and the variable is ready to go.

The variables **readyList** and **threadsToBeDestroyed** also require initialization before use. They are initialized with this code in the same function:

```
readyList = new List [Thread]
threadsToBeDestroyed = new List [Thread]
```

Any attempt to use a variable which contains an object before it is initialized is an error. The runtime system will check for this error and will print the error message "Attempt to use an uninitialized object!" if it occurs.

A similar initialization of arrays is required in KPL. Each array carries its size with it in KPL, unlike in the "C" language. When an array is initialized, the size is changed from zero to the number of elements in the array. After that, the array size cannot be changed.

For example, each **Thread** object contains an field called **systemStack**, defined as:

```
systemStack: array [SYSTEM_STACK_SIZE] of int
```

This array could be initialized with this code:

```
systemStack = new array of int {SYSTEM_STACK_SIZE of -1}
```

This will initialize the size and will initialize each element to -1.

For every array access, the KPL compiler produces code that will perform index bounds checking, so if an attempt is made to access an element that is beyond the end of the array, the system will print something like "This array index is out-of-range!"

In fact, to avoid repetitively copying in the initial values, the actual code—shown below—does something rather tricky: it simply stores an integer into the (normally hidden) array size field. In order to do this, the programmer must explicitly go around the strong, static type-checking done by the KPL compiler. The following code is "type unsafe" and a bit risky; a mistake here could lead to a system crash.

```
*((& systemStack) asPtrTo int) = SYSTEM_STACK_SIZE
```

The States of a Thread

The **status** field in a **Thread** object tells what state the thread is in. A thread that has possession of the CPU and is executing will have a **status** of **RUNNING**. At any time, only one thread will have a status of **RUNNING**.

At some point, a timer interrupt will occur and the current thread's time-slice will end. At that point, the scheduler will suspend that thread and begin executing some other thread. The previously running thread will be given a status of **READY** and the next thread to be scheduled will have its status changed to **RUNNING**. Nothing prevents a thread with a **READY** status from running, other than the fact that the CPU is busy running some other thread.

Sometimes a thread will be forced to wait for some event. For example, a thread may be waiting for some data that will be produced in the future by another thread. Such a waiting thread cannot be run, even if the CPU becomes available. This thread will have a status of **BLOCKED**. A blocked thread will not become **READY** until some other thread takes action to un-block it. **BLOCKED** threads are in some sense frozen and suspended from making progress.

During the process of creating a new thread, the **Thread** object will first be initialized and will then be scheduled to run. As part of the initialization process, before the thread becomes **READY**, it will be given a status of **JUST_CREATED**.

Each **Thread** object normally represents an active thread. But after a thread terminates, it will never run again and its **Thread** object no longer represents a valid, runnable thread. At this point, the **status** field in the object will be changed to **UNUSED**, indicating that the thread has terminated.

Thread objects are a limited resource and must be managed carefully by any kernel. In particular, rather than simply freeing the associated **Thread** object, the BLITZ kernel will maintain a pool of free, unused **Thread** objects. Technically, the **UNUSED** status is redundant and unnecessary; the presence of a **Thread** object on this free list is sufficient to indicate that a **Thread** object is unused. Nevertheless, the **UNUSED** status will help guard against programming errors.

Allocating and Freeing Objects

The KPL language has a facility for creating new objects on a "heap". For example:

```
var p: ptr to Thread
...
p = alloc Thread
...
free p
```

Several aspects of traditional heap allocation and management make heap usage unacceptable for kernel code. First, the heap may be too small and may fill up, causing the kernel itself to fail. But any extra space allocated to a kernel heap to prevent it from filling up results in permanently lost physical memory and a kernel should optimize its use of memory, in order to make more available to user-level programs. Second, an automatic garbage collector can introduce unpredictable and unacceptable pauses in the kernel. Finally, the heap and the objects in the heap are generally accessed by many different threads, further complicating any automatic garbage collection or heap management.

Instead, the approach taken in systems like Linux involves a complex memory management scheme. Linux has something called the "slab allocator". You can request allocations of memory and, when done with the memory, it can be returned to the slab allocator. Linux's slab allocator is quite complex. For each allocation, you must include many flags to specify things like how to handle the situation when memory is limited. (For example, the allocator could wait, or it could free non-essential objects, or it could just return with failure.)

The BLITZ approach is simpler.

We will restrict ourselves to allocating objects on the heap only during initialization and startup. However, after startup, no new objects will be allocated on the heap. By following this convention, an "out of memory" condition cannot occur after startup.

Since there is no risk of overflowing the heap after startup, objects allocated on the heap never need to be freed. KPL has a `free` statement, but we will never use it.

But how shall we deal with the problem of memory resource allocation after startup?

For example, after startup and the BLITZ kernel is running, there will occasionally be a demand for new **Thread** objects as new threads are forked. And as those threads terminate, their **Thread** objects will become free and available for re-use.

The kernel will maintain a *free list* (or *free pool*) of **Thread** objects. When a **Thread** object is no longer needed, it will be returned to this free list. The kernel will startup with a fixed, predetermined number of **Thread** objects in the free list and this number can not be increased. This fixed supply of **Thread** objects will limit the number of active threads our kernel can accommodate at any one time. If an attempt is made by some thread to create a new thread when there are already a lot of threads, it will be forced to wait if there are no **Thread** objects on the free list.

With many threads running and each able to create a new thread at any time, the management of the **Thread** free list will require concurrency control. The free list is an object which is shared among many threads and, as such, it cannot be programmed without careful consideration of synchronization and concurrency control.

The thread scheduler discussed here does not include the free list of **Thread** objects. In this document, we are describing only the scheduler, not the rest of the kernel. This document will describe several

concurrency control structures which may be used to implement control over the free list, but the management of **Thread** objects is outside the scope of the scheduler and is part of the kernel at large.

In KPL, objects may be allocated *dynamically* (on the heap) or allocated *statically* (using variables).

Here is some example code which allocates a **Thread** object on the heap (i.e., dynamically) and then calls a function to print it:

```
var
  p: ptr to Thread
...
p = alloc Thread
...
ThreadPrint (p)
```

Here is a second, similar example which uses a statically allocated **Thread** object stored in a variable named **t**.

```
var
  p: ptr to Thread
  t: Thread = new Thread
...
p = &t
...
ThreadPrint (p)
```

In the last statement, the programmer would probably avoid the variable **p** altogether and simply write:

```
ThreadPrint (&t)
```

The variable **t** could be a *global variable* (which is allocated outside any function) or a *local variable* (which is allocated on the call stack). However, if you are working with pointers to local objects, you must be careful to remember that the object's memory will be reclaimed when the relevant function returns!

In using the scheduler code described here, **Thread** objects may either be allocated on the heap, as in the first example above, or stored directly in variables, as in the second example. Whenever a thread terminates, the scheduler will give the **Thread** object a status of **UNUSED**. The scheduler will do nothing further; in particular, a **Thread** objects that as allocated on the heap will not be freed and the object will continue to exist.

Later, when you build a kernel on top of this scheduler, you'll add code to recycle unused **Thread** objects by managing a free list of **Thread** objects.

The State of the CPU

At all times, the BLITZ processor is running in one of two different modes, called System Mode and User Mode. All code discussed in this document will be running in System Mode, including both the code of the scheduler and the code of all the threads. Since all threads will run in System Mode, we might call them *kernel threads*—as opposed to *user threads* which run in User Mode—but we are discussing the scheduler in isolation, not the kernel, so the concept of user threads is not really even meaningful here.

At any time, the CPU either has interrupts *enabled* or *disabled*. When interrupts are enabled and an interrupt occurs—for example, a timer interrupt—the execution of the current code will be interrupted and a jump will be made to an interrupt handler routine. When an interrupt occurs while interrupts are disabled, this jump will not occur and instruction execution will continue normally. Later, when interrupts are once again enabled, the interrupt (which remained pending the whole time) will occur and the jump to the handler code will occur.

The BLITZ CPU also contains page table hardware and, at any moment, paging is either turned on or off. The page table hardware is used to implement virtual memory in the kernel, but for all code discussed here, paging is always turned off. When paging is turned on, we can make a distinction between virtual addresses and physical addresses. We must also distinguish between virtual address spaces and the installed memory physically present on the machine. Since paging will always be off in the code discussed here, all addresses will refer in a simple and straightforward way to the bytes of memory.

The CPU contains a special register, called the *Status Register*, which contains three bits telling (1) whether the CPU is in System Mode or User Mode, (2) whether interrupts are enabled or disabled, and (3) whether paging is off or on. The Status Register also contains three other bits telling the result of previous comparison instructions.

The entire state of the CPU consists of:

15 general purpose (integer) registers – System Registers
15 general purpose (integer) registers – User Registers
16 floating point registers
Status Register
Program Counter (PC)
Page Table Registers

When we change from one thread to another, we'll need to save the state of the previous thread and re-load the CPU with the saved state of the next thread.

In this code, we will ignore the floating point registers entirely. In other words, we will not save the floating point registers when we switch from thread to thread. Therefore, any threads that use the `double` type will not function correctly.

Also, since paging is always turned off, we can ignore the Page Table Registers. There are two registers (called **PTBR** and **PTLR**) but we will not bother to save them whenever we switch from thread to thread.

The BLITZ CPU contains two copies of the general purpose integer registers. One copy is used when the CPU is running in System Mode and one copy is used when the CPU is in User Mode. Since all the code discussed here runs in System Mode, we will ignore the User Registers.

Therefore, whenever we switch from one thread to another we'll need to save and restore:

15 general purpose (integer) registers – System Registers
Status Register
Program Counter (PC)

Context Switching

Next, let's look at what happens when a timer interrupt occurs.

For clarity, let's call the currently running thread the "previous" thread. The timer interrupt will cause a switch to a new thread, which we will call the "next" thread. The previous thread will be changed to status **READY** and will be placed at the tail of the ready list. The thread at the front of the ready list—the next thread—will be removed from the ready list and will becoming the **RUNNING** thread.

Assume that interrupts are enabled and a timer interrupt occurs.

Whenever an interrupt of any kind occurs, the CPU will complete the current instruction and will then push three words onto the stack. Next, the Status Register will be changed to disable interrupts. Also the mode will be changed to System Mode and Paging will be disabled, but these two changes have no effect since the CPU was already in System Mode with Paging disabled. Disabling interrupts means that the processing to be described next will be able to run to completion without a second interrupt occurring and messing things up before it is ready for them.

After the Status Register bits are modified, the CPU will force a branch to a specific address in low memory, by loading the program counter (PC) with a fixed number that depends on the type of interrupt. For a timer interrupt, this address happens to be `0x00000004`. Since there are 14 types of interrupts, the first 14 words—called the *interrupt vector*—are reserved for this purpose.

The three words pushed onto the stack are (1) a word of all zeros, (2) the current status register, and (3) the program counter. For some other interrupt types, the first word will contain relevant information instead of all zeros.

The scheduler includes some assembly code, which is located in the file named **Runtime.s**. In particular, this assembly file places 14 jump instructions in the interrupt vector in low memory. For

example, at address `0x00000004` there is a jump to the first instruction of an assembly routine called **TimerInterruptHandler**.

Here is the code for **TimerInterruptHandler**:

```
TimerInterruptHandler:
      push   r1          ! Save all int registers on the
      push   r2          ! .  interrupted thread's system stack
      push   r3          ! .
      push   r4          ! .
      push   r5          ! .
      push   r6          ! .
      push   r7          ! .
      push   r8          ! .
      push   r9          ! .
      push   r10         ! .
      push   r11         ! .
      push   r12         ! .
      call   _P_Thread_TimerInterruptHandler    ! Perform up-call
      pop    r12         ! Restore int registers
      pop    r11         ! .
      pop    r10         ! .
      pop    r9          ! .
      pop    r8          ! .
      pop    r7          ! .
      pop    r6          ! .
      pop    r5          ! .
      pop    r4          ! .
      pop    r3          ! .
      pop    r2          ! .
      pop    r1          ! .
      reti               ! Return from interrupt
```

This code starts by saving registers **r1** through **r12** by pushing them onto the stack. (Recall that register **r15** points to the stack top.)

Register **r13** is used by all KPL programs and will contain the line number of the statement being executed. This number is used solely to be included in error messages. Register **r14** contains a pointer to the current stack frame, which contains the local variables of a KPL routine. Both registers **r13** and **r14** will be saved whenever we enter another KPL routine and will be restored before we return, so it is not necessary to push them here; they'll be saved as the first actions of the routine called next.

After saving the registers, this routine calls a KPL routine called **TimerInterruptHandler**. (Note that the assembler routine and the KPL routine happen to have the same name, which is certainly a poor choice. Oh well…)

The KPL compiler adds a prefix to the name of each routine to avoid naming clashes and to make the KPL namespace and scoping conventions work. The name given to the KPL function **TimerInterruptHandler** in the compiler output is **_P_Thread_TimerInterruptHandler**, which is what must be used in the above assembly code. [The compiler always prepends "**_P_***xxxxx*_", where *xxxxx* is the name of the package containing the routine.]

Here is the KPL routine named **TimerInterruptHandler**:

```
function TimerInterruptHandler ()
  currentInterruptStatus = DISABLED
  currentThread.Yield ()
  currentInterruptStatus = ENABLED
endFunction
```

The variable **currentInterruptStatus** can take either of two values, **DISABLED** or **ENABLED**. The idea is that the value of this variable will always mirror the status of the interrupt-enabled bit in the CPU, since it is difficult to query this bit directly.

Since the interrupt-enabled bit was changed from enabled to disabled by the CPU as part of the interrupt processing sequence, this statement is necessary to ensure that the variable **currentInterruptStatus** has a correct up-to-date value.

The next statement invokes the **Yield** method. Notice that if the call to **Yield** were to be ignored and execution were to continue, the process described so far would be reversed; the registers would be restored and, at the end of the assembly language routine, a `reti` machine instruction would be executed.

The `reti` instruction pops three words off the stack. (Recall that during interrupt processing, three words were pushed.) The `reti` will discard the zero word, and it will restore the Status Register and the Program Counter. At this point, the entire state of the interrupted thread will be restored and instruction execution will resume with the next instruction from the interrupted code sequence. The interrupted thread will be none-the-wiser and will behave exactly the same as if no interrupt had occurred.

But the call to **Yield** cannot be ignored. Here is the **Yield** method in class **Thread**.

```
method Yield ()
  var
    nextTh: ptr to Thread
    oldIntStat, junk: int
  oldIntStat = SetInterruptsTo (DISABLED)
  nextTh = readyList.Remove ()
  if nextTh
    status = READY
    readyList.AddToEnd (self)
    Run (nextTh)
```

```
  endIf
  junk = SetInterruptsTo (oldIntStat)
endMethod
```

(In the code fragments given in this document, some simplifications have been made. For example, there are a number of self-check tests that have been removed. Also, the comments have been reduced or eliminated. Nevertheless, the core functionality is unchanged.)

This method includes three local variables, named **nextTh**, **oldIntStat**, and **junk**.

In the KPL language, within a method, the `self` keyword refers to the receiving object, just like `this` in Java or C++. Since this method was invoked on the **currentThread**, `self` points to the **Thread** object representing the thread being interrupted.

The first action in this method is to disable interrupts, which in this case, happen to already be disabled. The **SetInterruptsTo** function will change the interrupt status to whatever it is passed—**DISABLED** in this case—and will return the previous value of **currentInterruptStatus**. This call is necessary, since the **Yield** method may also be called in other circumstances in which interrupts are enabled.

Note that we see this pattern in several places:

```
oldIntStat = SetInterruptsTo (DISABLE)
... some critical region of code ...
junk = SetInterruptsTo (oldIntStat)
```

The idea is to disable interrupts while we do something that should not be interrupted and then to restore interrupts to whatever they were previously. (The variable **junk** is needed because KPL does not allow a returned value to be ignored. The variable is named **junk** because we don't need the value.)

The next action is to remove a thread from the front of the ready list. The goal is to switch to this thread. The variable **nextTh** is set to point to this **Thread** object. The **Remove** function will return null if the ready list was empty, so the next test is to see whether there is a "next" thread to switch to. If not, the **Yield** method will return and, subsequently, we'll make a return back to the thread that was interrupted.

If there truly is a "next" thread, we change the **status** of the receiver object (the current thread) to **READY** and place it at the end of the ready list. Then we call the **Run** function, which is shown next.

```
function Run (nextThread: ptr to Thread)
  var
    prevThread, th: ptr to Thread
  prevThread = currentThread
  prevThread.CheckOverflow ()
  currentThread = nextThread
  nextThread.status = RUNNING
  Switch (prevThread, nextThread)
```

```
  while ! threadsToBeDestroyed.IsEmpty ()
    th = threadsToBeDestroyed.Remove()
    th.status = UNUSED
  endWhile
endFunction
```

Each thread has a fixed sized stack which we hope is large enough. If a thread recurses deeply, it may use up a lot of stack space. The call to **CheckOverflow** is an attempt to catch situations when the previously executing thread has overflowed its stack.

Recall that each **Thread** object contains an array of 1000 words, which is used for its stack. In other words, register **r15** will be pointing somewhere in this array and, as things are pushed onto the thread's stack, they will go into memory locations which lie somewhere in this array.

When the **systemStack** array is first initialized, we store a special word in location 0, at the very bottom of the stack. This value is called a *sentinel*. If the stack grows too much, that location will be overwritten. **CheckOverflow** will check to see if the sentinel value is still unchanged. This would signal a catastrophic failure since arbitrary locations may have been overwritten. Although KPL checks array accesses, the **systemStack** simply provides an area for the stack, which is accessed via **r15** by machine instructions like `call` and `ret`, which do not check for overflow.

The next action in **Run** is to update **currentThread** to point to the next thread and to change its status from **READY** to **RUNNING**.

Then we see a call to an assembly routine called **Switch**, which is from the file **Switch.s**:

```
Switch:
        load    [r15+4],r1      ! Move the prevThread into r1
        add     r1,16,r1        ! Make r1 point to r1.regs
        store   r2,[r1+0]       ! Save r2..r14 in r1.regs
        store   r3,[r1+4]       ! .
        store   r4,[r1+8]       ! .
        store   r5,[r1+12]      ! .
        store   r6,[r1+16]      ! .
        store   r7,[r1+20]      ! .
        store   r8,[r1+24]      ! .
        store   r9,[r1+28]      ! .
        store   r10,[r1+32]     ! .
        store   r11,[r1+36]     ! .
        store   r12,[r1+40]     ! .
        store   r13,[r1+44]     ! .
        store   r14,[r1+48]     ! .
        store   r15,[r1+52]     ! Save r15 in r1.stackTop
        load    [r15+8],r1      ! Move the nextThread into r1
        add     r1,16,r1        ! Make r1 point to r1.regs
        load    [r1+0],r2       ! Restore r2..r14 from r1.regs
        load    [r1+4],r3       ! .
```

```
        load    [r1+8],r4       ! .
        load    [r1+12],r5      ! .
        load    [r1+16],r6      ! .
        load    [r1+20],r7      ! .
        load    [r1+24],r8      ! .
        load    [r1+28],r9      ! .
        load    [r1+32],r10     ! .
        load    [r1+36],r11     ! .
        load    [r1+40],r12     ! .
        load    [r1+44],r13     ! .
        load    [r1+48],r14     ! .
        load    [r1+52],r15     ! Restore r15 from r1.stackTop
        ret
```

Switch is passed two arguments, pointers to the previous and next **Thread** objects, which were called **prevThread** and **nextThread** in the calling routine.

To access arguments passed to an assembly routine, the programmer needs to know how KPL routines pass arguments, i.e., the *calling conventions* of the language's implementation. The KPL compiler generates code to pass arguments on the stack and, to access these arguments, the programmer needs to know exactly where, relative to the stack top **r15** these arguments will be placed. Fortunately, students using the BLITZ system will only need to read (never to write) assembly code like this.

The first `load` instruction copies the **prevThread** pointer argument from the stack into a register. The `add` instruction changes to **r1** to point to the first element in the **regs** array in that **Thread** object. This routine uses explicit knowledge about how KPL will lay out the fields of an object. If a field is added to or removed from the **Thread** class, then **reg[0]** will no longer be located at offset 16 in the object and this routine will break.

Next, **Switch** stores the contents of all registers into the **prevThread** object using 14 `store` instructions.

Next, **Switch** gets a pointer to the **nextThread** object. In very similar code, it copies the saved register values from this object into the registers. Notice that this includes **r15**, the stack top, itself! Therefore, after the last `load` instruction is executed, all pushes and pops will now go to or from the next thread's stack.

Note that everything involving the stack so far, including the initial state saving of the interrupted thread, was done on the system stack of the "previous" thread. From here on, we have switched to a new stack. At the moment the stack pointer **r15** is loaded, we might say that we have officially switched from the "previous" thread to the "next" thread.

Finally, **Switch** executes a return statement. But notice that, since **r15** has been changed, this is not returning to the invocation of **Run** that called **Switch**!

If you search through all the code of the scheduler, you'll see that the routine **Switch** is only called from one place: the **Run** routine. However, **Switch** will be called many times from many different

invocations of **Run**. At this point in execution, this invocation of **Switch** return to a *different invocation* of **Run** than the invocation that called it!

In all "normal" programs, every routine always returns to the routine that called it. But what you're seeing here is something that is very unusual and may take some thinking about.

Notice that we have saved the entire state of the previous thread. The Program Counter and Status Register were saved on that thread's stack as part of the interrupt processing. Then we pushed registers **r1** through **r12** onto its stack. Then, we entered some KPL routines (**TimerInterruptHandler**, **Yield**, and **Run**) which saved **r13**, **r14**, and **r15** as part of the KPL calling sequence. Then, in **Switch**, we again saved all registers (except **r1**). (Register **r1** is a "work register" and all KPL routines assume that every routine will trash it. In particular, there is no need to save **r1** since **Run**, which called **Switch**, will assume that **r1** has been trashed.)

The variable **currentThread** has been updated to point to some new **Thread** object, which has been removed from the ready list and had its status changed to **RUNNING**.

So assume that the "next" thread, which is now about to start running, was suspended at some earlier time in the very same way as the "previous" thread was suspended just now. In other words, assume that some earlier timer interrupt caused the exact same sequence of actions to occur for the "next" thread that we've just described for the "previous" thread.

So now, with the "next" thread's registers restored, we will work back and finish executing the routines we've just discussed until the `reti` instruction is executed. The state of the CPU will be restored to what it was just before some earlier interrupt occurred, and at that point the interrupted instruction sequence in the "next" thread will be returned to. We will resume executing the code in some thread that was interrupted long ago. Many threads may have had a chance to run since then and there may have been many timer interrupts and thread switches since then, but it is finally time to return to this particular thread and give it another time-slice of execution.

In other words, the next few actions will be to complete the routines that have been called and to return from each, in turn, unwinding the calling stack. Although this is now occurring in a new and different thread with a different stack, it may be easier for you to imagine that we are still in the same with the same stack, i.e., that the "previous" thread and the "next" thread are the same. It doesn't really matter, since the actions are the same in either case. Whenever a thread is interrupted, the exact same sequence of events occurs and its state is pushed onto its stack in the exact same way. The only difference is that we are returning from invocations that were suspended a long time ago, not from the invocations that were just entered.

After **Switch** returns, we're back in **Run**. After doing something with **threadsToBeDestroyed**, which we'll discuss in a second, we return to **Yield**. The routine **Yield** will restore interrupts to what they were when **Yield** was first entered. (Recall that whenever **Yield** is called from **TimerInterruptHandler**, it starts by disabling interrupts which were already disabled, i.e., doing nothing, so the action of restoring the *interrupt-enabled* bit to its previous status also has no effect. This code is here since **Yield** can be called from other places, when interrupts may in fact be enabled.)

Then **Yield** returns to the KPL routine **TimerInterruptHandler**, which will then set the variable **currentInterruptStatus** to **ENABLED**. This is appropriate, because interrupts will become enabled within the next few instructions.

Then the KPL routine **TimerInterruptHandler** returns to the assembly routine **TimerInterruptHandler**, which will restore the interrupted code's registers and execute the `RETI` (return-from-interrupt) instruction.

The `RETI` instruction will restore the Status Register, thereby re-enabling interrupts, and restore the Program Counter, which will cause a return to the instruction stream that was interrupted.

Now take a look at the code in **Run** just after the call to **Switch**.

Here, we see that a list called **threadsToBeDestroyed** is checked. This is a list of **Thread** objects which is almost always empty. However, in the case that a thread wishes to terminate itself, it will add its own **Thread** object to that list before calling **Run**. This check in **Run** will be executed as the first thing any "next" thread does. It performs any final cleanup required on the **Thread** object that can only be done after the thread has really finished. Here all we do is change the status to **UNUSED**, but in a complete kernel in which the **Thread** objects are recycled, this is the point where you would need to add this **Thread** object back to the free pool. Actions like this must be done by some other thread, and the obvious candidate to do the work is the very next thread that runs.

Creating a New Thread

To create a new thread, you must have a **Thread** object to work with. You'll need to initialize the object and then invoke the **Fork** method to add it to the ready list.

```
var
  aThread: Thread = new Thread
...
aThread.Init ("My Example")
aThread.Fork (Foo, 123)
```

The **Init** method is not too interesting. It simply fills in the fields in the **Thread** object. It fills in the **name** field from the argument. It sets **status** to **JUST_CREATED**. It initializes the **systemStack** array and writes the sentinel value (used to watch for stack overflow) into the array. Finally, it initializes the **regs** array and returns.

The **Fork** method is more complex. It is passed a pointer to a function—**Foo** in this case—and an integer. This is the function that will be executed by the new thread. In some sense, this is the "main" function of the new thread. **Foo** is a made-up name. In an actual program, the programmer will probably give it a more descriptive name.

This function must take a single integer as an argument and the value provided to **Fork** will be passed to this function. The idea is that many threads may be executing the same code and the integer argument allows each thread to differentiate itself from other threads executing the same code.

```
function Foo (arg: int)
...do some stuff...
endFunction
```

If the thread function ever returns, the thread will terminate. However, **Foo** might legitimately consist of an infinite loop and the thread may never terminate.

Here is the code for **Fork**:

```
method Fork (fun: ptr to function (int), arg: int)
  var
    oldIntStat, junk: int
  oldIntStat = SetInterruptsTo (DISABLED)
  initialFunction = fun
  initialArgument = arg
  stackTop = stackTop - 4
  *(stackTop asPtrTo int) = ThreadStartUp asInteger
  status = READY
  readyList.AddToEnd (self)
  junk = SetInterruptsTo (oldIntStat)
endMethod
```

First, interrupts are disabled, if they are not already disabled, since we will be updating the ready list. The ready list is shared by all threads and a timer interrupt during **Fork** might allow some other thread to access the ready list while we are in the middle of updating it, leading to catastrophe. After updating the ready list, interrupts are restored to whatever they were when **Fork** was entered, and **Fork** returns.

When the new thread reaches the front of the ready list at some later time and finally gets a chance to run, it will be started up by the code we looked at earlier in **Switch**. Recall that **Switch** will restore the registers and blindly execute a return statement. **Switch** executes the same instructions, regardless of whether the thread was previously interrupted or is a brand new thread, so we need to set things up here in **Fork** so that the `RET` instruction in **Switch** will branch to the starting code of this thread.

In other words, the new thread will be started up in exactly the same way (namely by the code in **Switch**) that resumes threads that already underway but which were suspended by timer interrupts and calls to **Yield**.

Next, look at the way **Fork** manipulates **stackTop**. First, **stackTop** is decremented and then something is stored at the top of the stack. So, **Fork** is simply pushing something onto the new thread's stack. And the thing that **Fork** is pushing is the address of a routine called **ThreadStartUp**.

In KPL, the name of a function can be used as the address of that function. The expression

`ThreadStartUp asInteger`

converts this from a pointer to a function to an integer so that it can be stored in the stack, which is an array on integers.

ThreadStartUp is an assembly routine located in the file **Switch.s**. At some point in the future, when **Switch** is called and the thread is finally allowed to begin execution, what will happen?

First, **Switch** will "restore" the thread's registers. Since the register array **regs** has been initialized to zeros, the registers will be "restored" to zero. Then **Switch** will "return" to the code that was executing. To do this, **Switch** ends with a `RET` instruction, which pops a return address off the stack and branches to it.

Since **Fork** has pushed the starting address of the **ThreadStartUp** routine, **Switch** will effectively just jump to this routine. **ThreadStartUp** will take it from there, and we'll look at that code soon.

Notice that the `RET` instruction in **Switch** will cause a jump—not really a "call"—to the routine named **ThreadStartUp**. **ThreadStartUp** will never return. Isn't this interesting: a "return" instruction is executed to "call" a routine!

The **Fork** function also stores into the fields **initialFunction** and **initialArgument**, the address of the thread's "main" function and an argument to pass to it. Later, we'll see how this function gets invoked.

Finally, **Fork** changes the thread's status to **READY** and places it on the ready list. After re-enabling interrupts, **Fork** will return and the current thread will continue executing.

At some time in the future, there will be a timer interrupt and **Switch** will be invoked to switch to a new thread. Other threads may be in front of the **readyList**, but eventually the newly created thread will come to the front of the **readyList**. When this happens, **Switch** will act as we just described. After setting the registers (to zero), **Switch** will "return" to the **ThreadStartUp** routine.

Thread Start Up

At some later time, a context switch will occur and **Switch** will be invoked. **Switch** will save the previous thread's registers, load the registers of this newly forked thread, and execute a return instruction which will branch to the first instruction of the routine called **ThreadStartUp**. **ThreadStartUp** is given next.

```
ThreadStartUp:
        mov     r0,r14          ! Clear the FP register
        mov     r0,r12          ! Clear the Catch Stack pointer
        call    _P_Thread_ThreadStartMain    ! Call ThreadStartMain
ThreadHang:
        debug                   ! Should never reach this point
```

```
        jmp     ThreadHang    ! .
```

All running KPL programs assume that register **r15** points to the top of the stack. They also assume that register **r14** points to the activation stack frame of the currently executing routine. This register is also called the *frame pointer* (FP). Finally, register **r12** is assumed to point to the *catch stack*.

This routine begins by setting **r14** and **r12** to zero. (This is redundant—perhaps unnecessarily cautious—since the method **Thread.Init** initialized the **regs** array to all zeros.)

[The KPL language contains a *try-throw-catch* mechanism, similar to what's in Java. The try-throw-catch mechanism does not need to be used in any of the BLITZ code and the mechanism can be ignored safely. The catch stack is a linked list of <*error-id*, *catch-code-entry-point*> pairs that will used whenever an error is thrown. Although the mechanism is not used, it's a good idea to initialize the stack properly. By setting the catch stack to null, we set it up so that no `try`/`catch` statements are active in this thread. If any error should be thrown with no `try` statement to catch the error, the mechanism will work properly and an error message will be issued.]

By setting the current frame pointer (i.e., **r14**) to zero, we will initialize the stack of activation frames properly. As new routines are invoked, a linked list of activation frames will be built. Starting with FP equal to null is necessary so that this list is properly terminated.

Next **ThreadStartUp** will call the KPL routine **ThreadStartMain**, which is shown next:

```
function ThreadStartMain ()
  var
    junk: int
    mainFun: ptr to function (int)
  junk = SetInterruptsTo (ENABLED)
  mainFun = currentThread.initialFunction
  mainFun (currentThread.initialArgument)
  ThreadFinish ()
  FatalError ("ThreadFinish should never return")
endFunction
```

ThreadStart calls **SetInterruptsTo** to enable interrupts (and set **currentInterruptStatus**) so that the new thread begins execution with interrupts enabled.

Recall that the **initialFunction** field of the thread object contains a pointer to the thread's "main" function. The KPL language does not permit this function to be called with the obvious syntax

```
currentThread.initialFunction (...)
```

since that syntax is used for method invocation. To work around this, we copy the pointer to a temporary variable called **mainFun** and then call it, passing the **initialArgument**.

The thread's main function may never return, but if it does, **ThreadStartMain** will call the KPL routine **ThreadFinish** to terminate the thread. **ThreadFinish** will not return. Of course the thread's main function may call **ThreadFinish** itself directly.

Thread Termination

Whenever a thread wishes to terminate itself, it can call the function **ThreadFinish**, which is shown next:

```
function ThreadFinish ()
  var
    junk: int
  junk = SetInterruptsTo (DISABLED)
  threadsToBeDestroyed.AddToEnd (currentThread)
  currentThread.Sleep ()
endFunction
```

This function will add the currently running thread to the **threadsToBeDestroyed** list and then invoke the method **Sleep**. It first disables interrupts because this list is shared among all threads and because **Sleep** expects interrupts to be disabled on entry. Recall that when the **Run** function is invoked next and a new thread is scheduled, **Run** will check the **threadsToBeDestroyed** list. If there is anything on the list, then **Run** can perform any finalization—such as recycling the **Thread** object—that must be done after the thread has terminated.

The method **Thread.Sleep**, which is shown next, can be called from several places. It can be called by a thread that is about to terminate, as above, but it can also be called by a thread that needs to go to sleep for a while, until some other thread takes some action to wake it up. For example, when we discuss semaphores, we'll see that the method **Semaphore.Down** may put the current thread to sleep.

```
method Sleep ()
  var
    nextTh: ptr to Thread
  status = BLOCKED
  nextTh = readyList.Remove ()
  Run (nextTh)
endMethod
```

The **Sleep** method will always be invoked on the current thread and will always be called with interrupts disabled. It will change the current thread's status to **BLOCKED**. Then, it will remove the next thread from the ready list and call **Run** to switch to it. The **Sleep** method does not put the current thread on any list; the assumption is that the caller will have done that, if necessary.

Note that the ready list will never be empty since the idle thread will always be present and ready to run, so the called to **Remove** will never fail.

The Idle Thread

As part of the initialization of the scheduler, a thread called the *idle thread* is created and added to the ready list with this code:

```
idleThread.Init ("idle-thread")
idleThread.Fork (IdleFunction, 0)
```

The main purpose of the idle thread is to make sure that the ready list is never empty. If the ready list were empty, what would the scheduler do if the currently running thread wanted to sleep?

The code of the idle thread, shown next, essentially contains an infinite loop that calls **Yield**. Whenever the idle thread gets scheduled, it immediately gives up the CPU to the next thread in the ready list.

However, the idle thread first checks to see if there are any threads in the ready list. If the ready list is empty then, instead of calling **Yield**, it calls a function named **Wait**.

```
function IdleFunction (arg: int)
  var
    junk: int
  while true
    junk = SetInterruptsTo (DISABLED)
    if readyList.IsEmpty ()
      Wait ()
    else
      currentThread.Yield ()
    endIf
  endWhile
endFunction
```

The function **Wait** is an assembly language routine that simply invokes the `wait` machine instruction and returns. The `wait` instruction will enable interrupts and halt the CPU execution, putting the machine into a low-power *wait state*. The CPU will stay in this state of suspended animation until the next interrupt occurs. Then the CPU will wake up and resume instruction execution. Of course the first thing that will happen is that the interrupt will be serviced.

In the case that a timer interrupt ended the `wait` instruction, the interrupt handler will invoke **Yield**, which will ultimately return to the idle thread, since there are no other threads. But in the case of other interrupts, such as disk I/O or terminal I/O, the interrupt handler will most likely make another thread runnable. Back in the idle thread, after the return from **Wait**, the function will loop and then call **Yield** to switch over to the newly ready thread.

The Current Interrupt Status

Here is the code for **SetInterruptsTo**:

```
function SetInterruptsTo (newStatus: int) returns int
  var
    oldStat: int
  Cleari ()
  oldStat = currentInterruptStatus
  if newStatus == ENABLED
    currentInterruptStatus = ENABLED
    Seti ()
  else
    currentInterruptStatus = DISABLED
    Cleari ()
  endIf
  return oldStat
endFunction
```

The methods **Cleari** and **Seti** are assembly routines that execute the `cleari` and `seti` instructions (respectively) and return. The `cleari` instruction clears the *interrupts-enabled* bit in the Status Register, thereby disabling interrupts. Similarly, the `seti` instruction sets the bit, which enables interrupts.

There is a global variable named **currentInterruptStatus** that attempts to mirror the state of this bit of the Status Register, so that programs can check to see whether interrupts are, at any point, enabled or disabled.

Since **currentInterruptStatus** is a shared variable, the above code must be very careful when modifying it, in order to prevent other threads from interfering and creating an inconsistent state. So this routine starts with a call to **Cleari** to disable interrupts, thereby ensuring that it can complete without interruption. [The second call to **Cleari** is redundant and unnecessary, but is included for… uh… "aesthetic reasons."]

Semaphores

The BLITZ scheduler includes an implementation of semaphores, which we will discuss next.

Recall that "semaphore" is an *abstract data type* (ADT), which means that it may only be used by invoking the operations defined on the type. An instance of an abstract data type may have internal

state, but this is an implementation detail. Users of the type should only use the operations and should never access the internal state directly. The internal state can only be queried or modified by the code implementing the operations.

In the case of semaphores, there are two key operations, called *Up* and *Down*. One way to understand a semaphore is to think of it as an integer counter which can never go negative. The *Up* operation will increment the integer by one, while the *Down* operation will attempt to decrement the count by one. If the count is already at zero, a *Down* operation will freeze the thread until some other thread has invoked *Up* and the count is no longer zero. Then *Down* can safely decrement the counter and resume execution of the frozen thread.

Semaphores are implemented with a class called **Semaphore**, which is shown next:

```
class Semaphore
  superclass Object
  fields
    count: int
    waitingThreads: List [Thread]
  methods
    Init (initialCount: int)
    Down ()
    Up ()
endClass
```

In general, it is convenient to represent each abstract data type with a KPL class. Fields are used to represent the internal state of the object and methods are used to implement the abstract data type's operations.

KPL—unlike some object-oriented languages—has no mechanism to enforce a barrier between "inside" the object and "outside" the object. For example, Java and C++ attach attributes like "public" and "private" to fields and methods. In KPL, there is nothing to prevent the programmer from writing code that accesses the fields from outside the object, except common sense and self-discipline. KPL is designed for kernel code authors; the assumption is that a simple, flexible language model is well-suited for programmers with a high level of skill. Language restrictions certainly have their place, but can occasionally get in the way in kernel code.

Users of **Semaphore** should only invoke the **Init**, **Down**, and **Up** methods; users should never access or even read the fields.

The implementation of **Semaphore** includes two fields, called **count** and **waitingThreads**. The **count** roughly corresponds to the hidden "count" we referred to above, but in the code below, notice carefully that the correspondence is not exact, since the **count** field can and will go negative!

The field called **waitingThreads** is a list of **Thread** objects, namely those threads that are suspended on this semaphore. When a thread invokes **Down** and the count would go negative, that thread is suspended and is added to the list of **waitingThreads**.

Here is the code for the methods of this class:

```
behavior Semaphore

  method Init (initialCount: int)
    count = initialCount
    waitingThreads = new List [Thread]
  endMethod

  method Down ()
    var
      oldIntStat: int
    oldIntStat = SetInterruptsTo (DISABLED)
    count = count - 1
    if count < 0
      waitingThreads.AddToEnd (currentThread)
      currentThread.Sleep ()
    endIf
    oldIntStat = SetInterruptsTo (oldIntStat)
  endMethod

  method Up ()
    var
      oldIntStat: int
      t: ptr to Thread
    oldIntStat = SetInterruptsTo (DISABLED)
    count = count + 1
    if count <= 0
      t = waitingThreads.Remove ()
      t.status = READY
      readyList.AddToEnd (t)
    endIf
    oldIntStat = SetInterruptsTo (oldIntStat)
  endMethod

endBehavior
```

Look at the **Down** method first. It decrements **count** and, if it went negative, then the thread is suspended. To do this, the **Down** method puts the current thread on the list called **waitingThreads**. Then it calls **Sleep** to suspend execution of the current thread. This thread will only be awakened by some other thread when the count goes non-negative. Since the fields **count** and **waitingThreads** are shared by all threads using this semaphore, this method temporarily disables interrupts while these fields are accessed.

Next look at the **Up** method. First, it increments **count**. Then it checks the **count** and possibly wakes up one of the threads waiting on this semaphore.

One way to think about the **count** field is this: If **count** is positive, its value tells how many more **Up** operations have been executed than **Down** operations. If **count** is negative, then more **Down** operations have occurred. Each thread that called **Down**, for which there was no corresponding **Up** operation, was suspended. The absolute value of **count** simply tells how many threads are waiting.

So if **count** is was negative before **Up** was called, then the **Up** method will take a thread off the list, change its **status** to **READY** and add it to the ready list. And notice that since **count**, the **waitingThreads** list and the **readyList** are all shared variables, this method will disable interrupts while they are being accessed.

Waiting threads are always added to the tail end of the **waitingThreads** list. When awakened, threads are always removed from the front of the list. Therefore, the list is a FIFO queue; any thread that waits will eventually be awakened—i.e., starvation is not a possibility—as long as there are enough calls to **Up**.

Semaphore objects must be initialized and an initial count must be supplied. This is usually zero, but it can be any positive number, which acts as if that many excess **Up** operations had been performed.

```
var
  mySem: Semaphore = new Semaphore
mySem.Init (0)
...
mySem.Up ()
...
mySem.Down ()
```

Mutex Locks

The BLITZ system also includes *mutex locks*. A mutex lock has two important operations, called *Lock* and *Unlock*. Similarly to semaphores, mutex locks are modeled with an abstract data type. As such, there is a class called **Mutex**, with methods called **Lock** and **Unlock**.

Here is the class specification:

```
class Mutex
  superclass Object
  fields
    ...
  methods
    Init ()
    Lock ()
    Unlock ()
endClass
```

As part of the BLITZ operating systems project, the implementation of the **Mutex** class is left as an exercise.

There are several ways to implement the **Mutex** class. One approach is to use a **Semaphore** object, in which case the **Lock** operation is nothing more than **Semaphore.Down**. The **Unlock** operation is implemented with **Semaphore.Up**.

In the project assignment, students are asked to implement the **Mutex** class using the same approach used for class **Semaphore**.

Without giving too much away, the general idea is that **Mutex** will need two fields. One will indicate the state of the lock, either "held" or "free". The second will be a list of threads suspended and waiting on the lock, which might be called **waitingThreads**. If the lock is free, then the list will be empty; if the lock is held, there may or may not be threads waiting.

The **Lock** operation will need to check to see if the lock is currently "free". If so, it will change its state to "held" and return. Otherwise, it will suspend the current thread on the **waitingThreads** list. Of course, the method will need to disable interrupts while the fields are accessed, to prevent concurrency race conditions.

The **Unlock** method will need to remove a thread from the **waitingThreads** list and, if there was a waiting thread, transfer the lock to that thread.

The Monitor Concept

The *monitor* concept is a particularly useful concurrency control structuring technique. Like semaphores and mutex locks, monitors can be used to correctly program multithreaded applications where several threads must interact and synchronize their access to shared data.

A monitor is a little like a class. A monitor will have hidden state, just as objects have fields. And a monitor will have *entry methods* which can be invoked, just as an object's class provides several methods which can be invoked on the object. So, as a first approximation, a monitor is much like an object.

The difference between an object and a monitor lies in what happens when several threads invoke methods concurrently. With a normal object, if two threads try to invoke a method on the object simultaneously, the two methods will execute concurrently. Perhaps the two threads invoke the same method or perhaps they invoke different methods in the class, but the key is that both threads are invoking methods on the same object. These method will be executed concurrently and, without any further control, havoc may occur. For example, the threads may each try to access the same field in the object simultaneously.

With a monitor, the following statement is always true: "Only one thread at a time can execute code within the monitor." This means that if one thread is executing a method of the monitor and a second thread attempts to invoke a method of the same monitor, then the second thread will be forced (somehow!) to wait until the first thread returns and leaves the monitor code.

Some languages provide special support for the monitor concept, but KPL does not. However, monitors can be conveniently coded in KPL, as long as the programmer follows a few rules and guidelines.

We will implement each monitor with a KPL class. More precisely, for each kind of monitor, the programmer will create a class. Then, at runtime, when the monitor is needed, the programmer will use an object of that class. The distinction between an "object" and a "class of objects" is similar to the distinction between a "monitor" and a "kind of monitor." In most problems—such as the *Dining Philosophers Problem*—only one instance of the monitor is needed. So the programmer would create a class for the monitor and then allocate a single object of that class. To be more precise, when we say the *monitor class*, we'll mean a class and when we say the *monitor*, we'll mean an instance of this class.

As an example, let's say we want to create a monitor for the "Dining Philosophers Problem." Let's call the monitor class **ForkMonitor**. Each monitor class must have a method called **Init**, which must be called when the monitor is created. So to create and initialize a monitor, we'll see code like this:

```
var
  myForkMon: ForkMonitor
...
myForkMon = new ForkMonitor
myForkMon.Init ()
...
```

Of course we can also use pointers to monitor objects, or even arrays of pointers to monitors, etc., but for this problem, such a complex data structure is unnecessary. A program may have several kinds of monitors, but it is unusual to see a program with more than one monitor of each kind.

In our implementation of monitors, we need to enforce the invariant about only one thread at a time executing code in the monitor. [We define "code in the monitor" as the code of any method in the monitor class and, if other functions or methods are called from this code, we'll include them, too.]

If the programmer follows some simple conventions consistently, then enforcing the invariant is easy. First, every monitor class should include a field of type **Mutex**. When locked, it indicates that some thread is already executing code within the monitor. When unlocked, there are no threads in the monitor. For simplicity, let's always call this field **monitorLock**.

Next, the programmer needs to make a clear distinction about which methods are "entry methods" and which are local, private methods. Any method called from outside the monitor is an entry method. Viewing the monitor as an abstract data type, entry methods are the externally visible operations; they can be invoked to "get into" the monitor. All other methods in the class are private methods and should only be invoked by entry methods or other private methods. In other words, code outside the monitor should never invoke a private method directly.

```
class ForkMonitor
  superclass Object
  fields
    monitorLock: Mutex
    ...
  methods
    Init ()
    MyEntryMethod_1 (...)
    MyEntryMethod_2 (...)
    MyEntryMethod_3 (...)
    ...
    MyPrivateMethod_1 (...)
    MyPrivateMethod_2 (...)
    MyPrivateMethod_3 (...)
    ...
endClass
```

To enforce the invariant that only one thread at a time is in the monitor, the programmer must remember to always lock the **monitorLock** as the first operation of every entry method. Also, the programmer must remember to unlock the **monitorLock** before returning from an entry method. The **monitorLock** should not be locked or unlocked at any other time.

Of course, if there is a `return` in the middle of an entry method, the monitor lock must still be unlocked.

Here is what all entry methods should look like:

```
method MyEntryMethod
  monitorLock.Lock ()
  ...
      if ...
        monitorLock.Unlock ()
        return
      endIf
  ...
  monitorLock.Unlock ()
endMethod
```

Condition Variables

The monitor concept also includes a related concept called the *condition variable*. A condition variable is similar to a mutex lock or a semaphore, in that a variable is created and there are a couple of operations that can be performed on the condition variable. The operations are named *Signal*, *Wait*, and *Broadcast*.

However, condition variables are different from mutexes and semaphores in that each condition variable is linked to a specific monitor. Each condition variable belongs to a monitor and should be defined as a field of that monitor class.

Some monitors will have only one condition variable, but some monitors—such as the **ForkMonitor** from the Dining Philosophers Problem—will naturally contain several condition variables. A monitor might have no condition variables, but then it becomes an academic question of whether to even call it a monitor, or just a collection of critical section code.

The operations on the condition variables should only be invoked by code within the monitor. In other words, whenever an operation, such as **Signal** or **Wait**, is performed on a condition, it should be done by code that has previously acquired a lock on the monitor's lock.

Each condition variable will be an instance of a class called **Condition**. Here is the specification of class **Condition**:

```
class Condition
  superclass Object
  fields
    waitingThreads: List [Thread]
  methods
    Init ()
    Wait (mutex: ptr to Mutex)
    Signal (mutex: ptr to Mutex)
    Broadcast (mutex: ptr to Mutex)
endClass
```

Notice that each of the operations (except **Init**) requires a pointer to a **Mutex** lock. When these operations are invoked, the caller should pass a pointer to the **monitorLock** of the monitor. Since the **Condition** operation is only used by code in the monitor, it is assumed that the **monitorLock** has already been locked.

The semantics of condition variables is given next. When code executing in a monitor executes a **Wait** operation, that code will be suspended, i.e., the current thread will become blocked, waiting on the condition. At this point the **monitorLock** is unlocked and other threads are now free to enter the monitor. The code executing the **Wait** operations is suspended and another thread is now allowed to enter the monitor.

At some future time, another thread executing within the monitor will invoke a **Signal** operation on the condition variable. At this point the first thread, which was waiting, is awakened and becomes ready to run.

But now we have two threads ready to execute code in the monitor. According to our invariant, this is not allowed. So one thread must be forced to wait until the other thread leaves the monitor. Then the other thread can continue execution.

Which thread is allowed to continue in the monitor? And which thread must now wait for the lock?

Over the years, different authors have proposed different answers. Tony Hoare proposed something now called *Hoare Semantics*, which says that the signaling thread always waits and the previous waiting thread (i.e., the one that called **Wait**) immediately enters the monitor. Furthermore, no other threads can enter the monitor between the execution of the **Signal** operation by one thread and the acquisition of the lock by the other thread.

In a looser specification of condition variables, called *Mesa Semantics*, the guarantee is only that the blocked thread will be awakened. After that it will have to compete against all other threads for the monitor lock. Generally, the signaling thread will continue executing and, when it leaves the monitor, the awakened thread, and perhaps others, will try to acquire the monitor lock. There is no guarantee about which thread will get the lock first, only that the awakened thread will eventually get it.

In either case (either Hoare Semantics or Mesa Semantics), the blocked thread will eventually get the monitor lock and will resume execution of code in the monitor. In particular, it will begin with a return from the call to **Wait**. Also, with both semantics, the guarantee is that each **Signal** operation will wake up exactly one thread, if any threads are waiting on the condition. If there are no threads waiting on the condition, then a **Signal** operation will have no effect.

The implementation of the **Condition** class in BLITZ uses the looser Mesa Semantics. The code is given below.

Note that each **Condition** variable is implemented with a single field called **waitingThreads**, which is a list of the threads that have executed a **Wait** operation on this condition variable, but which have not yet been signaled to resume. The **Init** method simply initializes this list.

Looking at the **Wait** method, you can see that the thread will first unlock the **MonitorLock**, allowing other threads to enter the monitor. Then it will block itself, placing itself on the **waitingThreads** list. Once awakened, the code in the **Wait** method will attempt to relock the **monitorLock**. Only after it acquires the lock, will the method return. Interrupts are disabled the entire time; do you see why interrupts must be disabled between the call to **Unlock** and putting the thread on the **waitingThreads** list?

Looking at the **Signal** method, you can see that it simply takes a thread off the **waitingThreads** list—if there are any threads on the list—changes it back to **READY** and puts it on the ready list. Since **Signal** never unlocks the **monitorLock**, we can see that in this implementation, the thread executing the **Signal** will continue in the monitor, while the newly awakened thread will wait until it leaves the monitor before competing (possibly with other threads) for the **monitorLock** before it continues.

```
behavior Condition

  method Init ()
    waitingThreads = new List [Thread]
  endMethod
```

```
  method Wait (mutex: ptr to Mutex)
    var
      oldIntStat: int
    oldIntStat = SetInterruptsTo (DISABLED)
    mutex.Unlock ()
    waitingThreads.AddToEnd (currentThread)
    currentThread.Sleep ()
    mutex.Lock ()
    oldIntStat = SetInterruptsTo (oldIntStat)
  endMethod

  method Signal (mutex: ptr to Mutex)
    var
      oldIntStat: int
      t: ptr to Thread
    oldIntStat = SetInterruptsTo (DISABLED)
    t = waitingThreads.Remove ()
    if t
      t.status = READY
      readyList.AddToEnd (t)
    endIf
    oldIntStat = SetInterruptsTo (oldIntStat)
  endMethod

  method Broadcast (mutex: ptr to Mutex)
    var
      oldIntStat: int
      t: ptr to Thread
    oldIntStat = SetInterruptsTo (DISABLED)
    while true
      t = waitingThreads.Remove ()
      if t == null
        break
      endIf
      t.status = READY
      readyList.AddToEnd (t)
    endWhile
    oldIntStat = SetInterruptsTo (oldIntStat)
  endMethod

endBehavior
```

As an optional exercise (part of project 2, in the BLITZ operating system assignments), you are asked to consider how you might change the code in the **Condition** class to implement Hoare semantics.

We might also mention in passing that Per Brinch Hansen suggested another semantic approach to condition variables. His idea was to make sure the thread executing the **Signal** immediately leaves the

monitor so the monitor lock is always given to the newly awakened thread immediately. However, this approach has not been adopted since it sometimes places a very awkward constraint on the programmer. For example, sometimes one thread will need to signal several condition variables or to signal one condition variable several times.

Sometimes it is convenient for a thread to wake up, not one, but all threads waiting on a condition variable. To facilitate this, our implementation of **Condition** includes a method called **Broadcast**. It simply executes a while-loop to move every thread on the **waitingThreads** list to the ready list.

Conclusion

In this document, we looked at the thread scheduler used in the BLITZ operating system kernel project. The kernel project is broken into 8 individual project assignments. The thread scheduler is introduced at the beginning (in project 2) because it underpins all other kernel code.

This thread scheduler can run alone and can be understood in isolation from the rest of the kernel. In fact, the scheduler can be used to implement multi-threaded applications that are independent of any kernel. For example, the thread scheduler described here can be used to program solutions to classic concurrency control problems, such as *Producer-Consumer*, *Reader-Writer*, *Dining Philosophers*, *Sleeping Barber*, and so on.

Project 2 is all about understanding multithreaded applications, concurrency control primitives, and time-slicing. If you are a student attempting project 2, we hope this material has put you in a better position to understand and do the project. And if you have a more general interest in thread scheduling or in the BLITZ system, we hope this information has been interesting and useful, as well.