Lab 2: Threads

Due date: May 10 2022, Tuesday.

Overview and Goal

In this lab, you will learn about *threads* in BLITZ, and gain familiarity writing programs involving concurrency control. You will begin by studying the **thread** package, which implements multi-threading, and then make some modifications and additions to the existing code to solve some traditional concurrency problems using the provided code in this package. You will also gain familiarity programming in the KPL language while completing this lab.

Before you start this lab, it is required that you carefully read the entire document titled "The Thread Scheduler and Concurrency Control Primitives," by downloading it from the course website (one of the documents in the *documentation.zip* archive).

Step 1: Setting up

In the Docker container image provided to you, the files needed for Lab 2 can be found in `/lab2`. You should get the following files:

```
makefile
DISK
System.h
System.c
Runtime.s
Switch.s
List.h
List.c
Thread.h
Thread.c
Main.h
Main.c
Synch.h
Synch.c
```

In this lab, you will only need to modify and submit the following three files:

```
Main.c
Synch.h
Synch.c
```

You should be able to compile all the source code provided to you with the UNIX `make` command:

```
      % make
```

The program executable we are building will be called "`os`". You can run the program using the BLITZ emulator by typing:

```
      % blitz -g os
```

Feel free to modify other files besides `Synch.h`, `Synch.c` and `Main.c`, but the code you are required to write and submit does not require any changes to the other files. For example, you may wish to uncomment some of the print statements, to see what happens. However, your final versions of `Synch.h`, `Synch.c` and `Main.c` must work with the other provided files, exactly as they are distributed to you.

Do not reuse any of the files from Lab 1, as they are considered out of date.

Working in your Docker container image

You will need to install Docker, as shown in Lab 1. After you have successfully installed Docker on your own computer, download the Docker image we provided to you for this lab from the course website (under the section heading "Lab 2"), called `lab2-docker.tar.gz.` *Please note:* the provided docker image is built on an Intel x86 architecture and does not support an M1 Mac. If you use an M1 Mac, you will need to find an Intel computer to work on labs in this course.

Load the Docker image as you did in Lab 1:

```
docker load -i lab2-docker.tar.gz
```

Then run the Docker image as a Docker container by using the following command:

```
docker run -it blitz
```

After the container is running, you will see a command prompt from within the Linux container. The BLITZ tools have been preinstalled for you in `/usr/local/blitz/bin`, and files needed for Lab 2 can be found in `/lab2.` The source code for building BLITZ tools can be found in `/blitz`. Within the container, the search path environment variable has already been set up for you to use BLITZ command-line tools directly.

As you may have already tried in Lab 1, there are many other commands that may be useful for you to work with Docker containers. For example, to remove all the Docker containers, you can use the command:

```
docker rm $(docker ps -a -q)
```

To remove all the Docker images (so that you can have a clean slate to start working with something else), you may use the command:

```
docker rmi $(docker images -q)
```

To copy files from the Docker container to your host computer, use the command:

```
docker cp <containerId>:/file/path/within/container /host/path/target
```

The container ID can be found in the command prompt while you are running the container.

The command-line editor `vi` has been pre-installed for you in the Docker container. To install other editor alternatives (such as `emacs`) or any other packages, you can use the following command within the container:

```
apt-get install -y <package_name>
```

To learn more about Docker containers and images, refer to the Docker documentation. There was also a mini-tutorial of other Docker commands that you may find useful, distributed to you on the course website when Lab 1 was released.

Step 2: Study the Existing Code

The code you received in this lab provides the ability to create and run multiple threads in the kernel, and to control concurrency through several synchronization methods.

Start by looking over the **`System`** package. Focus on the material toward the beginning of the file **`System.c`**, namely the following functions:

```
print
printInt
printHex
printChar
printBool
nl
MemoryEqual
StrEqual
StrCopy
StrCmp
Min
Max
printIntVar
printHexVar
printBoolVar
printCharVar
printPtr
```

Get familiar with these printing functions, as you may need to call them quite often in your code. Some of these functions are implemented in assembly code, and some are implemented in KPL in the **`System`** package.

The following functions are used to implement the heap in KPL:

```
KPLSystemInitialize
KPLMemoryAlloc
KPLMemoryFree
```

Objects can be allocated on the heap and freed with the **`alloc`** and **`free`** statements. The HEAP implementation is very rudimentary in this implementation. In your kernel, you may allocate objects during start-up but after that, you should **not** allocate objects on the heap. This is because the heap may fill up, and then the kernel may crash.

The following functions can be ignored since they are only related to aspects of the KPL language that we will not be using in this lab:

```
KPLUncaughtThrow
UncaughtThrowError
KPLIsKindOf
KPLSystemError
```

The **`Runtime.s`** file contains a number of routines coded in assembly language. It contains the program entry point and the interrupt vector in low memory. Read it carefully. Follow what happens when program execution begins at location `0x00000000` (the label "`_entry`"). The code labeled "`_mainEntry`" is included in the code the compiler produces. The "`_mainEntry`" code will call the **`main`** function, which appears in the file **`Main.c`**.

In **`Runtime.s`**, follow what happens when a timer interrupt occurs. It makes an "up-call" to a function called **`_P_Thread_TimerInterruptHandler`**. This name implies that it is "a function called **`TimerInterruptHandler`** in a package called **`Thread`**." (It is the name the compiler gives to this function.)

All the code in this lab assumes that no other interrupt types (such as a **`DiskInterrupt`**) occur. When reading **`Runtime.s`**, think about what would happen if another type of interrupt should ever occur.

The KPL language will check for many error conditions, such as the use of a null pointer. Try changing the program to make this error. Follow in **`Runtime.s`** to see what happens when this occurs.

Next, read the **`List`** package. First read the header file carefully. This package provides code that implements a linked list. We will use linked lists in this lab. For example, the threads that are ready to run (and waiting for time on the CPU) will be kept in a linked list called the "ready list." Threads that become `BLOCKED` will sit on other linked lists. Also read the code in **`List.c`** to check out how the linked list is implemented in KPL.

The most important class in this lab is named **`Thread`**, and it is located in the **`Thread`** package along with other code (see **`Thread.h`**, **`Thread.c`**). For each thread, there will be a single **`Thread`** object. **`Thread`** is a subclass of **`Listable`**, which means that each **`Thread`** object contains a **`next`** pointer and can be added to a linked list.

The **`Thread`** package in **`Thread.c`** is central and you should study this code thoroughly. This package contains one class (called **`Thread`**) and several functions related to thread scheduling and time-slicing:

```
InitializeScheduler ()
IdleFunction (arg: int)
Run (nextThread: ptr to Thread)
PrintReadyList ()
ThreadStart ()
ThreadFinish ()
FatalError (errorMessage: ptr to array of char)
SetInterruptsTo (newStatus: int) returns int
TimerInterruptHandler ()
```

`FatalError` is the simplest function. We will call **`FatalError`** whenever we wish to print an error message and abort the program. Typically, we will call **`FatalError`** after making some checks and finding that things are not as we expected. **`FatalError`** will print the name of the thread invoking it, print the message, and then shut down. It will throw us into the BLITZ emulator command line mode. Normally, the next thing to do might be to type the "`st`" command (short for "stack"), to see which functions and methods were active.

(Of course, when multiple threads were concurrently running, the information printed out by the emulator will only pertain to the thread that invoked **`FatalError`**. The emulator does not know about threads, and it is pretty much impossible to extract information about other threads by examining bytes in memory.)

The next function to look at is **`SetInterruptsTo`**, which is used to change the "I" interrupt bit in the CPU. We can use it to disable interrupts with code like this:

```
... = SetInterruptsTo (DISABLED)
```

and we can use it to enable interrupts:

```
... = SetInterruptsTo (ENABLED)
```

This function returns the previous status. This is very useful because we often want to DISABLE interrupts (regardless of what they were before) and then later we want to return the interrupt status to whatever it was before. In our kernel, we will often see code like:

```
var oldIntStat: int
...
oldIntStat = SetInterruptsTo (DISABLED)
...
oldIntStat = SetInterruptsTo (oldIntStat)
```

Next take a look at the **`Thread`** class. Here are the fields of **`Thread`**:

```
name: ptr to array of char
status: int
systemStack: array [SYSTEM_STACK_SIZE] of int
```

```
    regs: array [13] of int
    stackTop: ptr to void
```

Here are the operations (i.e., methods) you can do on a **Thread**:

```
    Init(n: ptr to array of char)
    Fork(fun: ptr to function (int), arg: int)
    Yield()
    Sleep()
    CheckOverflow()
    Print()
```

Each thread is in one of the following states: JUST_CREATED, READY, RUNNING, BLOCKED, and UNUSED, and this is given in the **status** field. (The UNUSED status is given to a **Thread** after it has terminated. We will need this in later labs.)

Each thread has a **name**. To create a thread, you will need a **Thread** variable. First, use **Init** to initialize it, providing a name.

Each thread needs its own stack, and space for this stack is placed directly in the **Thread** object in the field called **systemStack**. Currently, this is an array of 1000 words, which should be enough. (It is conceivable our code could overflow this limit; so there exist code in the implementation to check and make sure that we do not overflow this limited area.)

All threads in this lab are kernel threads, and will run in the Kernel (System) mode. The stack is therefore called the "system stack." In later labs, we will see that this stack is used only for kernel routines. User programs will have their own stacks in their virtual address spaces in later labs.

The **Thread** object also has space to store the state of the CPU, namely the registers. Whenever a thread switch occurs, the registers will be saved in the **Thread** object. These fields (**regs** and **stackTop**) are used by the assembly code function named **Switch**.

After initializing a new **Thread**, we can start it running with the **Fork** method. This does not immediately begin the thread execution; instead it makes the thread READY to run and places it on the **readyList**. The **readyList** is a linked list of **Threads**, and is a global variable. All **Threads** on the **readyList** have status `READY`. There is another global variable named **currentThread**, which points to the currently executing **Thread** object; i.e., the **Thread** whose status is RUNNING.

The **Yield** method should only be invoked on the currently running thread. It will cause a switch to some other thread.

Follow the code in **Yield** closely to see what happens when a context switch between threads occurs. First, interrupts are disabled; we do not want any interference during a context switch. The **readyList** and **currentThread** are shared variables and, while context switching between threads, we want to be able to access and update them safely. Then **Yield** will find the next thread from the **readyList**. (If there is no other thread, then **Yield** is effectively a nop.) After this **Yield** will make the currently running process READY (i.e., no longer RUNNING) and it will add the current thread to the tail end of the **`readyList`**. Finally, it will call the **`Run`** function to do the context switch.

The **`Run`** method will check for stack overflow on the current thread. It will then call **`Switch`** to do the actual **`Switch`**.

`Switch` may be the most fascinating function you ever encounter. It is located in the assembly code file **`Switch.s`**, which you should look at carefully. **`Switch`** does not return to the function that called it. Instead, it switches to another thread. Then it returns. Therefore, the return happens to another function in another thread!

The only place **`Switch`** is called is from the **`Run`** function, so **`Switch`** returns to some invocation of the **`Run`** function in some other thread. That copy (i.e., invocation) of **`Run`** will then return to whoever called it. This could have been some other call to **`Yield`**, so we will return to another **`Yield`** which will return to whoever called it.

And this is exactly the desired functionality of **`Yield`**. A call to **`Yield`** should give up the processor for a while, and eventually return after other threads have had a chance to execute.

`Run` is also called from **`Sleep`**, so we might be returning from a call to **`Sleep`** after a context switch.

How is everything set up when a thread is first created? How can we "return to a function" when we have not ever called it? Take a look at function **`ThreadStart`** in file **`Thread.c`** and look at function **`ThreadStartUp`** in file **`Switch.s`**. What happens when a thread is terminated? Take a look at **`ThreadFinish`** in file **`Thread.c`**. Essentially, the thread is put to sleep with no hope of ever being awakened. Our upcoming lectures will also cover more detailed information about these design choices in the **`Thread`** package.

Next, take a look at what happens when a Timer interrupt occurs while some thread is executing. This is an interrupt from hardware, so the CPU begins by interrupting the current routine's execution and pushing some state onto its system stack. Then it disables interrupts and jumps to the assembly code routine called **`TimerInterruptHandler`** in **`Runtime.s`**, which just calls the **`TimerInterruptHandler`** function in **`Thread.c`**.

In **`TimerInterruptHandler`**, we call **`Yield`**, which then switches to another thread. Later, we will come back here, when this thread gets another chance to run. Then, we will return to the assembly language routine which will execute a "**`reti`**" instruction. This will restore the state to exactly what it was before and the interrupted routine (whatever it was) will get to continue.

Note that this code maintains a variable called **`currentInterruptStatus`**. This is because it is rather difficult to query the "I" bit of the PSW (status register) in the CPU. It is easier to just change the variable whenever a change to the interrupt status changes. We see this occurring in the **`TimerInterruptHandler`** function. Clearly interrupts will be disabled immediately after the interrupt occurs. And the **Yield** function will preserve the interrupt status. So when we return from **Yield**, interrupts will still remain disabled. Before returning to the interrupted thread, we set the **`currentInterruptStatus`** to ENABLED. (They must have been enabled before the interrupt occurred — or else it could not have occurred — so after we execute the "**`reti`**" instruction, the status will revert to what it was before, namely ENABLED.)

It now becomes apparent that you will be reading a lot of code provided to you, before you are ready to start playing with and modifying the code. Please experiment with the code we have just discussed as necessary to understand it better.

Step 3: Run the "SimpleThreadExample" Code

Execute and trace through the output of **`SimpleThreadExample`** in file **`Main.c`**.

In **`TimerInterruptHandler`** there is a statement

```
     printChar('_')
```

which is commented out. Try uncommenting it. Make sure you understand the output.

In **`TimerInterruptHandler`**, there is a call to **`Yield`**. Why is this there? Try commenting this statement out, and see what happens. Make sure you understand how **`Yield`** works here.

Step 4: Run the "MoreThreadExamples" Code

Trace through the output. Try changing this code to see what happens.

Step 5: Implement the "Mutex" Class

In this part, you must implement the class **`Mutex`**. The class specification for **`Mutex`** is given to you in **`Synch.h`**:

```
  class Mutex
    superclass Object
    methods
      Init()
      Lock()
      Unlock ()
      IsHeldByCurrentThread () returns bool
  endClass
```

You will need to provide code for each of these methods. In **`Synch.c`** you will see a **`behavior`** construct for **`Mutex`**. There are methods for **`Init`**, **`Lock`**, **`Unlock`**, and **`IsHeldBy-CurrentThread`**, but these have dummy bodies. You will need to write the code for these four methods. You will also need to add a couple of fields to the **class** specification of **Mutex** in **Synch.h** to implement the desired functionality.

How can you implement the **`Mutex`** class? Take a close look at the **`Semaphore`** class that is provided to you; your implementation of **`Mutex`** will be quite similar.

First consider the **`IsHeldByCurrentThread`** method, which may be invoked by any thread. The code of this method will need to know which thread is holding a lock on the mutex; then it can compare that to the **`currentThread`** to see if they are the same. So, you may consider adding a field (perhaps called **`heldBy`**) to the **`Mutex`** class, which will be a pointer to the thread holding the mutex. Of course, you will need to set it to the current thread whenever the mutex is locked. You might use a null value in this field to indicate that no thread is holding a lock on the mutex.

When a lock is requested on the mutex, you will need to see if any thread already has a lock on this mutex. If so, you will need to put the current process to sleep. For putting a thread to sleep, take a look at the method **`Semaphore.Down`**. At any one time, there may be zero, one, or many threads waiting to acquire a lock on the mutex; you will need to keep a list of these threads so that when an **`Unlock`** is executed, you can wake up one of them. As in the case of **`Semaphore`**s, you should use a FIFO queue, waking up the thread that has been waiting the longest.

When a mutex lock is released (in the **`Unlock`** method), you will need to see if there are any threads waiting to acquire a lock on the mutex. You can choose one and move it back onto the **`readyList`**. Now the waiting thread will begin running when it gets a turn. The code in **`Semaphore.Up`** does something similar.

It is also a good idea to add an error check in the **`Lock`** method to make sure that the current thread asking to lock the mutex does not already hold a lock on the mutex. If it does, you can simply invoke **`FatalError`**. (This would probably indicate a logical error in the code using the mutex. It would lead to a deadlock, with a thread frozen forever, waiting for itself to release the lock.) Likewise, you should also add a check in **`Unlock`** to make sure the current thread really does hold the lock and call **`FatalError`** if it does not. You will be using your **`Mutex`** class later, so these checks will help your debugging in later labs.

The function **`TestMutex`** in **`Main.c`** is provided to exercise your implementation of **`Mutex`**. It creates 7 threads that uses the **`LockTester`** function to compete vigorously for a single mutex lock. The file **`DesiredOutput1.pdf`** that is provided to you contains an example of the correct output from running this function.

Step 6: Implement the Producer-Consumer Solution

In the lectures, we will cover the celebrated Producer-Consumer problem, and introduce a solution that uses a Mutex and two Semaphores. Implement this in KPL using the classes **`Mutex`** and **`Semaphore`**. Your solution needs to deal with multiple producers and multiple consumers, all sharing a single bounded buffer.

At the time when you try to complete this lab, our lectures may have just reached the point of discussing the use of Mutex locks and Semaphores to implement a solution to the producer-consumer problem correctly. In this case, if you wish to complete this part of the lab early, you may need to read ahead a little bit in the “Three Easy Pieces” texbook, from Chapter 26 up to and including Chapter 31.4 (“The Producer-Consumer (Bounded Buffer) Problem”), before the lecture coverage reaches that point. It is fine if you cannot understand the solution completely, as we will present a detailed coverage of this problem in upcoming lectures.

The **`Main`** package contains a part of the solution code that will serve as a framework for your complete solution. The bounded buffer is called **`buffer`** and contains up to **`BUFFER_SIZE`** (e.g., 5) characters. There are 5 producer threads and 3 consumer threads, in addition to the main thread that creates the other ones. You only need to supply the missing portion of the code to support thread synchronization.

Each producer will loop, adding 5 characters to the buffer. The first producer will add five 'A' characters, the second producer will add five 'B's, etc. However, since the execution of these threads will be interleaved, the characters will be added in a somewhat random order. The provided file **`DesiredOutput2.pdf`** provides you with a sample of the correct output.

What to Submit

Complete all the above steps.

Please submit **`Synch.h`**, **`Synch.c`**, **`Main.c.`**

Grading for this Lab

Your submitted solution will also be marked (out of the remaining 5 marks) using test cases, such as the provided function `TestMutex`. The maximum possible mark for this lab assignment is 10.