Programming Project 2:
Threads and Interprocess Communication

Due Date: Thursday, January 31, 2019 at 10:00 am
Duration: About two weeks

Overview and Goal

In this project you will learn about threads and gain familiarity writing programs involving concurrency control. You will begin by studying the thread package, which implements multi-threading. You will make modifications and additions to the existing code. Then you will use the threads package to solve some traditional concurrency problems.

In addition, you will gain familiarity programming in the KPL language.

If you have difficulties with this project or want to go into more depth, take a look at the document called "The Thread Scheduler and Concurrency Control Primitives":

http://web.cecs.pdx.edu/~harry/Blitz/BlitzDoc/ThreadScheduler.htm
http://web.cecs.pdx.edu/~harry/Blitz/BlitzDoc/ThreadScheduler.pdf

Step 1: Download the Files

Start by creating a directory for the files you'll need for this project. You might call it:

~YourUserName`/cs333/p2`

Then download all the files from:

`http://www.cs.pdx.edu/~harry/Blitz/OSProject/p2/`

into your directory. 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 project, you will modify and hand in the following files:

Main.c
Synch.h
Synch.c

After getting the files, you should be able to compile all the code (as is) with the UNIX **make** command. The program executable we are building will be called "os". You can execute the program by typing:

```
% make
% blitz -g os
```

In the course of your experimentation, you may modify other files besides **Synch.h**, **Synch.c** and **Main.c**, but the code you are required to write and turn in doesn't 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 files, exactly as they are distributed.*

Be sure you copy all files. Even though there are similarities with some of the files used for the previous project, there may be some subtle but important differences.

Step 2: Study the Existing Code

The code provided in this project provides the ability to create and run multiple threads, and to control concurrency through several synchronization methods.

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

print
printInt
printHex
printChar
printBool
nl
MemoryEqual
StrEqual
StrCopy
StrCmp

```
Min
Max
printIntVar
printHexVar
printBoolVar
printCharVar
printPtr
```

Get familiar with the printing functions; you'll be calling them a lot. Some 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 MUST NOT ALLOCATE OBJECTS ON THE HEAP*! Why? Because the heap might fill up and then what is a kernel supposed to do? Crash.

In this project, you should not allocate anything on the heap.

The following functions can be ignored since they concern aspects of the KPL language that we will not be using:

```
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. Take a look at it. Follow what happens when program execution begins at location 0x00000000 (the label "_entry"). The code labeled "_mainEntry" is included in 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 routine called **_P_Thread_TimerInterruptHandler**. This name refers to "a function called **TimerInterruptHandler** in a package called **Thread**." (**_P_Thread_TimerInterruptHandler** is the name the compiler will give this function.)

All the code in this project assumes that no other interrupt types (such as a **DiskInterrupt**) occur. In **Runtime.s**, follow what would happen if another sort of interrupt should ever occur.

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

Next take a look at the **List** package. Read the header file carefully. This package provides code that implements a linked list. We'll use linked lists in this project. 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 look over the **List.c** code file.

The key class in this project is named **Thread**, and it is located in the **Thread** package along with other stuff (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 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 ()
ThreadStartMain ()
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'll call **FatalError** after making some check and finding that things are not as 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 the information printed out by the emulator will pertain to only 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'll 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
initialFunction: ptr to function (int)
initialArgument: int
```

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'll need this in later projects.)

Each thread has a **name**. To create a thread, you'll 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; there is a check to make sure we don't overflow this limited area.)

All threads in this project will run in System mode. Therefore, the stack is called the "system stack". In later projects, we'll see that this stack is used only for kernel routines. User programs will have their own stacks in their virtual address spaces, but we are getting ahead of ourselves.

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 routine named **Switch**.

The **Thread** object also has space to store a pointer to a function (the **initialFunction** field) and an argument for this function (the **initialArgument** field). This pointer will point to the "main" function of this thread; this is the function that will get executed when this thread begins execution. We are storing a pointer to the function because this is a variable: different threads may execute different functions.

We are also able to supply an initial argument to this thread, through the **initialArgument** field. This argument must be an integer. Often there will be several threads executing the same "main" function. The argument is a handy way to let each thread know what its role should be. For example, we might create 10 threads each using the same "main" function, but passing each thread a different integer (say, between 1 and 10) to let it know which thread it is.

After initializing a new **Thread**, we can start it running with the **Fork** method. This doesn't 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**. All **Threads** on the **readyList** have status READY. **ReadyList** is a global variable. 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 thread switch occurs. First, interrupts are disabled; we don't want any interference during a thread switch. The **readyList** and **currentThread** are shared variables and, while switching 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.) Then **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 thread 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'll 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 thread 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 **ThreadStartMain** 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. (No wonder they call it "Thread Death!")

Next, take a look at what happens when a Timer interrupt occurs while some thread is executing. This is an interrupt, 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'll come back here, when this thread gets another chance to run. Then, we'll 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 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 once again be 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.)

Now 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.

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? 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'll see a **behavior** construct for **Mutex**. There are methods for **Init**, **Lock**, **Unlock**, and **IsHeldByCurrentThread**, but these have dummy bodies. You'll 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** to implement the desired functionality.

How can you implement the **Mutex** class? Take a close look at the **Semaphore** class; 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 might 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'll 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'll need to see if any thread already has a lock on this mutex. If so, you'll 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'll 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 longest.

When a mutex lock is released (in the **Unlock** method), you'll 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 doesn't already hold a lock on the mutex. If it does, you can simply invoke **FatalError**. (This would probably indicate a logic 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 not. You'll be using your **Mutex** class later, so these checks will help your debugging in later projects.

The function **TestMutex** in **Main.c** is provided to exercise your implementation of **Mutex**. It creates 7 threads that compete vigorously for a single mutex lock.

Step 6: Implement the Producer-Consumer Solution

The OSTEP textbook contain a discussion of the Producer-Consumer problem in chapter 31, including a solution. A solution implementing their pseudocode will work fine. There's also pseudocode for a solution included in appendix A of this document that uses variable and method names that better match those already in **Main.c**. Implement this in KPL using the classes **Mutex** and **Semaphore**. Deal with multiple producers and multiple consumers, all sharing a single bounded buffer.

The **Main** package contains some code that will serve as a framework. The buffer is called **buffer** and contains up to **BUFFER_SIZE** (e.g., 5) characters. There are 5 producer processes, each modeled by a thread, and 3 consumer processes, each modeled by a thread. Thus, there are 8 threads in addition to the main thread that creates the others.

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.

Step 7: Implement the Dining Philosopher's Solution Using a Monitor

A starting framework for your solution is provided in **Main.c**. Each philosopher is modeled with a thread and the code we've provided sets up these threads. The synchronization will be controlled by a "monitor" called **ForkMonitor**.

The code for each thread/philosopher is provided for you. Look over the **PhilosophizeAndEat** method; you should not need to change this code.

The monitor to control synchronization between the threads is implemented with a class called **ForkMonitor**. The following class specification of **ForkMonitor** is provided:

```
class ForkMonitor
  superclass Object
  fields
    status: array [5] of int      -- For each philosopher: HUNGRY,
                                  -- EATING, or THINKING
  methods
    Init ()
    PickupForks (p: int)
    PutDownForks (p: int)
    PrintAllStatus ()
endClass
```

You'll need to provide the code for the **Init**, **PickupForks** and **PutDownForks** methods. You'll also need to add additional fields and perhaps even add another method.

The code for **PrintAllStatus** is provided. You should call this method whenever you change the status of any philosopher. This method will print a line of output, so you can see what is happening.

How can you proceed? You'll need a mutex to protect the monitor itself. There are two main methods (**PickupForks** and **PutDownForks**) which are called by the philosopher threads. Upon beginning each of these methods, the first thing is to lock the monitor mutex. This will ensure that only one thread at a time is executing within the monitor. Just before each of these methods returns, it must unlock the monitor (by unlocking the monitor's mutex) so that other threads can enter the monitor code.

You'll also need to use the **Condition** class, which is provided in the **Synch** package. (The **Condition** class uses the class **Mutex**, so it is assumed that you've finished and tested the **Mutex** class.)

The BLITZ emulator has a number of parameters and one of these is how often a timer interrupt occurs. The default value is every 5000 instructions. You might try changing this parameter to see how it affects your programs behavior. To change the simulation parameters, type the **sim** command into the emulator. This command will give you the option to create a file called

`.blitzrc`

After creating this file, you can edit it by hand. The next time you run the emulator, it will use this new value. Also note that too small a value—like 1000—will cause the program to hang. What do you suppose causes this effect?

There is no pseudocode provided for this solution, but one approach is discussed in the OSTEP reading for Tuesday 01/22/19.

QUESTION: OSTEP discusses the semantics of signaling a condition variable and mentions "Hoare semantics." The comments in the code in **Synch.c** say that this version implements "Mesa-style" semantics. What is the difference between these two signaling styles?

An Example of Correct Output

The following files contain an example of what correct output should look like:

DesiredOutput1.pdf
DesiredOutput2.pdf
DesiredOutput3.pdf

What to Hand In

Complete all the above steps. In concurrency projects, it's helpful to test it with the same input more than once. As part of evaluating your code, I'll do that as well.

Please submit the following files:

Synch.h
Synch.c
Main.c

As well as pdfs showing the output for steps 5, 6, and 7.

Please print output using a `fixed-width font like this,` in this and all future assignments. Much of the output is designed to look nice when printed with a fixed-width font but is more difficult to read in a standard variable-width font.

Basis for Grading

In this course, the code you submit will be graded on both **correctness** and **style**. Correctness means that the code must work right and not contain bugs. Style means that the code must be neat, well commented, well organized, and easy to read.

In style, your code should match the code we are distributing to you. The indentation should be the same and the degree of commenting should be the same.

Appendix A

Pseudocode solution for Producer Consumer Problem

```
#define N = 100
initialize mutex m
initialize semaphore empty
initialize semaphore full

producer() {
  item = NULL

  while(True) {
    item = do_some_work()
    empty.Down()
    m.Lock()
    insert_item(buffer, item)
    m.Unlock()
    full.Up()
  }
}

consumer() {
    item = NULL

    while(True) {
        full.Down()
        m.Lock()
        item = remove_item(buffer)
        m.Unlock()
        empty.Up()
        consume_item(item)
    }
}
```