WWU-CS-Idriss /
lab-2-mutex-implementation-nimkicodes

Code Issues Pull requests 1 Actions Projects Wiki Security Insights

csci-509-idriss-w2025-lab-2-mutex-implementation-Lab2KPL created by GitHub Classroom

1 star 20 forks 0 watching Branches Activity Custom properties Tags

Private repository · Forked from WWU-CS-Idriss/csci-509-idriss-w2025-lab-2-mutex-implementation-Lab2KPL

main 3 Branches 0 Tags Go to file t Go to file + Add file Code

This branch is 5 commits ahead of `main` . #1

nimkicodes lab2: Mutex has been updated ab20b11 · 2 months ago

.github	GitHub Classroom Feedback	2 months ago
DISK	Initial commit	2 months ago
List.h	Initial commit	2 months ago
List.k	Initial commit	2 months ago
Main.h	Initial commit	2 months ago
Main.k	lab2: Mutex has been implemented	2 months ago
MutexSampleOutput.pdf	Initial commit	2 months ago
README.md	add online IDE url; add deadline	2 months ago
Runtime.s	Initial commit	2 months ago
Switch.s	Initial commit	2 months ago
Synch.h	lab2: Mutex has been implemented	2 months ago
Synch.k	lab2: Mutex has been updated	2 months ago
System.h	Initial commit	2 months ago
System.k	Initial commit	2 months ago
Thread.h	Initial commit	2 months ago
Thread.k	lab2: Mutex has been updated	2 months ago
lab2-script.txt	lab2: Mutex has been implemented	2 months ago
makefile	Initial commit	2 months ago

Review the assignment due date Open in GitHub Codespaces

CSCI 509 - Operating Systems Internals
Lab 2 : Threads and synchronization

Lab and Homework Assignments

You may work together to complete the labs, but each person must have their own submission. Homework assignments must be completed, debugged, and uploaded individually.

Overview and Goal

This lab introduces you to threads and synchronization and will give you more practice with kpl. This lab is a prelude to homework 2, which will require you to implement concurrency control. Much of the code is being provided to you; you will make modifications and additions to the existing code and use the threads package. **This lab assumes that you've completed and can reproduce all of the setup (exporting `path`, for example) required in lab 1.**

Documentation

The documentation for threads can be found in 'ThreadScheduler.pdf', one of the files in BlitzDocuments.zip.

Git Repository

The files should include the files makefile, DISK, System.h, System.k, Runtime.s, Switch.s, List.h, List.k, Thread.h, Thread.k, Main.h, Main.k, Synch.h, Synch.k..

IMPORTANT : by convention, executable (binary), object (.o), and any other non-text files are not pushed to a git repository. Do not `add` nor `commit` nor `push` such files to your repo! That is because such files are architecture and computer dependent, and are also huge, so pushing them to your git repo will bloat your repository. Except in rare exceptions, only code source files, and plain-text files, are pushed to a git repo.

Compile, run

In this lab, you will modify the following files: Main.k, Synch.h, and Synch.k.. Compile all of the code (as is) with the UNIX `make` command. The program executable you are building is named "os". You can execute the first program by typing:

```
make
blitz -g os
```

Throughout this lab you'll notice several questions. Answers to these you are not asked to hand in, but you should answer them "in your head," to make sure mastery of the lab's concepts.
Question : What output do you notice before the wait instruction?

Threads

The code provided in this lab provides the ability to create and run multiple threads. Several synchronization methods for "controlling" concurrency are also provided.

Start by looking over the *System.h* (header) and .k (source) files. Focus on the material toward the beginning of *System.k*, 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 often in this lab and in homework 2. Some of the print functions are implemented in assembly code and some are implemented in KPL in the System package.

The Heap, Interrupts, Lists

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.

For this lab and homework 2, you should not allocate anything on the heap. Note that some functions you can ignore (for the time being) because they concern aspects of the KPL language that we will not be using for lab2 nor homework 2: `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.k*.

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 lab (and homework 2) assumes that no other interrupt types (such as a DiskInterrupt) occur. Next take a look at the *List.h* and .k files. Read the header file carefully. This package provides code that implements a linked list. You'll use a linked list to keep track of those threads that are ready to run; that linked list you'll name "ready list". Threads that become BLOCKED will be kept in another linked lists.

The Thread Package

The key class in this lab and homework 2 is named `Thread`. For each thread that you create, 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. Several functions related to thread scheduling and time-slicing are provided for you. They are : `InitializeScheduler(), IdleFunction(arg: int), Run(nextThread: ptr to Thread), PrintReadyList(), ThreadStartMain(), ThreadFinish(), FatalError(errorMessage: ptr to array of char), SetInterruptsTo(newStatus: int), TimerInterruptHandler()`. Note that some of the functions receive no arguments, but others require an argument, when invoked. Note also that `SetInterruptsTo` returns an integer. Many of the functions are described in detail for you below. Look at these functions yourself in the code .k files!

- `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.
  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.
- `SetInterruptsTo` is used to change the "I" interrupt bit in the CPU, and returns the previous status. We can use it to disable or enable interrupts in code the following way :
  `... = SetInterruptsTo (DISABLED) ... = SetInterruptsTo (ENABLED)`

  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, you'll often see code like:

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

The Thread class

The `Thread` class contains the following fields :

- `name`: ptr to array of char. Each thread has a name. To create a thread, you'll need a Thread variable. First, use Init to initialize it, providing a name.

- `status` : int. 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. You'll use it in later assignments.)
- `systemStack` : array [SYSTEM_STACK_SIZE] of int. 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.)
- `regs` : array [13] of int. 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.
- `stackTop` : ptr to void.
- `initialFunction` : ptr to function (int). 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.
- `initialArgument` : int. 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.

A thread also has several methods, including :

- `Fork` : After initializing a new thread, you start it using 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.
- `Yield` : 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.
- `Run` : The Run method will check for stack overflow on the current thread. It will then call Switch to do the actual Switch.
- `Switch` : The Switch function is located in the assembly code file *Switch.s*. 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.

Question : 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 the file *Thread.k*, and look at the function `ThreadStartUp` in file *Switch.s*.
Question : What happens when a thread is terminated? Take a look at `ThreadFinish` in the file *Thread.k*. How is the thread put to sleep with no hope of ever being awakened?
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.k*.

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.

Run the "SimpleThreadExample" Code

Execute and trace through the output of `SimpleThreadExample` in the file *Main.k*. In the function `TimerInterruptHandler` (which file is that code in?) the statement `printChar ('_')` is commented out. Uncomment it and make sure you understand the output.
Question : 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.

Run the "MoreThreadExamples" Code

Comment out the `SimpleThreadExample()` invocation in *Main.k*, and uncomment out `MoreThreadExamples()`. Recompile and run. Trace through the output. Try changing this code to see what happens.

The Mutex Class

The coding task for lab2 is implementation of the Mutex class. Refer to the lecture slides for the behavior of 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 need to write the code for each of these methods. In *Synch.k* 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 Semaphores, 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 assignments.

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

The functions `ProducerConsumer()` and `DiningPhilosophers()` are for the coding portion of homework 2, which you can now proceed to (assuming you've completed implementing mutex).

Sample Solution

The file *MutexSampleOutput.pdf*, available on the course website, shows sample output for the Mutex Solution.

The "makefile"

The makefile is being provided for you. As you write (uncomment) the mutex functions you need to write for this lab, run `make`.

What to Hand In & Rubric

Make sure that all of your files needed for this lab are updated and pushed to the repo Use the unix `script` method to capture a sample run of you invoking your Main program which should run the mutex implementation. You can also copy and paste your terminal session. Name your script file *lab2-script* or *lab2-script.txt* and include it in your repo. It is a good idea to use the web interface at github to verify you have the correct files on the server as that is where I'll get them for grading. **Do not push to your branch .o or executable files. Use .gitignore or add files one by one.**

Rubric	
Files have been pushed with no .o nor executables files allowed in your git	2
Access is granted to instructor and TA (access should have been sorted from lab1/hw1) with a link is submitted to canvas.	2
The asked-for Mutex functionality has been added to *Synch.k*	4

Releases

No releases published
Create a new release

Packages

No packages published
Publish your first package

Languages

● Assembly 80.2% ● C++ 17.6% ● Makefile 2.0% ● C 0.2%

Suggested workflows

Based on your tech stack

C/C++ with Make

Build and test a C/C++ project using Make.

Configure