EECE 315 – UBC

Assignment 2

Overview

In this assignment you will be working with the Thread system in the Nachos OS. The overall goal for this assignment is to understand the basic multithreading programming concepts and introduction to the Nachos OS code base.

In this assignment, we will focus on the code given in the **code/threads** folder.

synch.cc

Synch.cc contains routines for synchronizing threads. Three kinds of synchronization routines are defined here: semaphores, locks and condition variables **(the implementation of the last two are left for the assignment).**

Any implementation of a synchronization routine needs some primitive atomic operation. We assume Nachos is running on a uniprocessor, and thus turning off interrupts can provide atomicity. While interrupts are disabled, no context switch can occur, and thus the current thread is guaranteed to hold the CPU throughout, until interrupts are re-enabled.

Because some of these routines might be called with interrupts already disabled (Semaphore::V for one), instead of turning on interrupts at the end of the atomic operation, we always simply reset the interrupt state back to its original value (whether that be disabled or enabled).

thread.cc

Thread.cc contains routines to manage threads. There are four main operations:

Fork -- create a thread to run a procedure concurrentlywith the caller (this is done in two steps – first allocate the Thread object and then call Fork on it)

Finish -- called when the forked procedure finishes, to clean up

Yield -- relinquish control over the CPU to another ready thread

Sleep -- relinquish control over the CPU, but thread is now blocked.In other words, it will not run again, until explicitly put back on the ready queue.

Join– Makes the current thread wait until the execution of this thread is over. **(the implementation of join is left as a part of this assignment).**

threadTest.cc

Contains test cases that we will be using for this assignment.

For further references please follow the link below:
http://people.cs.uchicago.edu/~odonnell/OData/Courses/CS230/NACHOS/code-dispatch.html

For this assignment, we provide a modified version of Nachos that includes the additional test cases and function declarations.

Please download the modified version of Nachos and compile it the same way you did for previous Assignment.

Running test cases:

To execute a test case, use the following command in the **code/threads** folder:

./nachos –q <test_case_num>

Task 1

Part 1: Implement lock variable:

After compiling Nachos, go to the **code/threads**folder and run the following command.

./nachos –q 3

This will run the testcase **ThreadTestLock()** given in the testThread.cc file.

You should be able to see the output similar to **incorrectOutputLock.txt** file. In order to have the correct output, you need to implement two functions declared in the **synch.cc** file.

- **Acquire() –** This function can be used by a thread before entering the critical region. When a thread tries to acquire the lock to enter the critical region, if the lock is available the thread acquires the lock, otherwise the thread is added to the wait queue and sent to sleep mode.

- **Release() –** As soon as the lock is released by any process, use the queue to find the next waiting thread. If there is any thread waiting, assign the thread to the scheduler to continue its execution.

Once you implement these 2 functions correctly, your output should match the **correctOutputLock.txt** file.

Part 2: Implement condition variable:

Code to be implemented:
Implement condition variables using interrupt enabling and disabling to provide atomicity. The class "Condition" should be declared in the file *"code/threads/synch.h"*, and the definition of its member function should be placed in the file *"code/threads/synch.cc"*. The signatures for the member functions have already been given in *"code/threads/synch.h"* and you should use them in your code. You can add member variables if you need.

Problem description:
Your condition variable class should have three interfaces:
Wait(Lock*),Signal(Lock*) and ***Broadcast(Lock*).***

Wait(Lock*)should be triggered when the condition or predicate is not fulfilled in the current thread and let the current thread to sleep until other threads wake it up.

Signal(Lock*)should be called when a condition or predicate is met, and wake up one of the threads waiting for the signal.

Broadcast(Lock*) should be called like *Signal(Lock*)*, but *Broadcast(Lock*)* should wake up all the threads that are waiting for the signal.

To avoid a race condition, usually a lock is used along with condition variables. When checking the condition or predicate, we need to make sure no other thread can update the condition or predicate by using a lock here.

Wait(Lock)* should release the lock before the thread goes to sleep to allow other threads go ahead and change the condition in future. When waking up from sleep, the thread will be in *Wait(Lock*)*, and it should re-acquire the lock to avoid racing with other threads if it is going to change the condition or predicate further.

Signal(Lock)* and *Broadcast(Lock*)* will need to check if the current thread holds the lock. An valid call to *Signal(Lock*)* or *Broadcast(Lock*)* should always hold the lock to ensure there is no racing with other threads. Finally, after a condition being changed and a signal being sent, release the lock to allow other threads to modify and check the condition.

Generally, a lock should be used to guarantee the exclusively checking or modifying of the condition, and condition variable should be used to control the sleeping and waking up of the threads.

Testing your code:
For testing your code, you can use the provided test cases implemented in *"code/threads/threadtest.cc"*. By specifying the argument follows *"-q 4" (e.g., "./nachos –q 4"*), you can choose to run the test for your implementation of condition

variable. Nachos will end up with an **assertion fail**as condition variable has not been implemented yet to prevent infinite loop **(see that in the output messages provided in the release package)**. The correct output is the same as the output from "semaphore" test (**also provided in the package**). Debug information can be enabled with *"-d"* argument appended *(e.g., "./nachos –q 4 -d").*

Hints:

1. Read the code of semaphore carefully, and try to understand how the implementation manipulate the thread queue through the scheduler, and how a queue kept by semaphore being used to pause and continue the related threads.
2. The operations in *Wait(Lock*):*"release the lock, add current thread to the waiting list of this condition variable, let the current thread sleep" should be done atomically. Disabling and enabling the interrupt in the right time to achieve that.
3. Our implementation of condition variable doesn't have an "Event or Condition or Predicate" structure explicitly associated in neither the class definition nor the argument list. You can use a global variable as the condition, test the value of it as checking the condition and change the value as updating the condition.
4. You can find a typical use case of condition variable in the following link, which is basically a monitor model. There is also an explanation about why a mutex lock is not enough for such situations (for saving CPU time). http://en.wikipedia.org/wiki/Monitor_(synchronization)
5. Here is another note about lock and condition variable might be useful. http://www.stanford.edu/class/cs140/cgi-bin/lecture.php?topic=locks

Task 2

After compiling Nachos, go to the **code/threads** folder and run the following command.

./nachos –q 6

The output that you get for this will be similar to what is given in **incorrectOutputJoin.txt.** This output is not correct, as the implementation for the thread join function is pending.

For join usage reference you can view the join usage in **threadtest.cc** file.

Code to be implemented:

In the file **thread.h** we have defined a trivial join function that simply returns without doing any task. Your job is to complete the function code and make sure that join works correctly.

Once you are done with the implementation, compile the code and run the same test case. Your output should be similar to what is given in **correctOutputJoin.txt** file.

Hints:

- You need to transfer the control from current thread to the calling thread.
- You need to make sure that when the calling thread is finished, the control is returned back to the current thread. You will need to modify the **Finish** function given in the thread.cc file for this.
- You can declare private variables within the class if required.
- You can use the scheduler class to find out next available thread in the ready state.
- Follow the code given the Thread::Yieldfunction to see the scheduler usage.

Bonustask - Implement Mailbox

Problem description:
You will implement synchronous send and receive of one word messages (one int value) using locks and condition variables. Create a "Mailbox" class with the *operations* ***intMailbox::Send(int message)*** and ***intMailbox::Receive(int * Message)***.

Send atomically waits until *Receive* is called on the same mailbox, and then return with received message. Once the *Receive* received the message, both can return. Similarly, *Receive* waits until *Send* is called, at which point a new message has beensent.

Your solution should work even if there are multiple senders and receivers for the same mailbox. **Note that you cannot use explicit wait queues, Sleep, or disable/enable interrupts to implement Mailbox**; the **condition variables** will do all of that for you. Also, it is not necessary to "match" sending and receiving threads -- a receiver does not care from which sender it gets a message.

Example:
First of all, let's assume Thread 1 gets the priority to run first.

Thread 1	Thread 2
Send(1)	Receive(int*)
Send(2)	Receive(int*)
Receive(int*)	Send(3)

The correct returning queue of those function calls should be:

Thread 1: Send(1)
Thread 2: Receive() return 1
Thread 1: Send(2)
Thread 2: Receive() return 2
Thread 2: Send(3)
Thread 1: Receive() return 3

Note there is no *Thread::Yield()* called to manipulate the execution queue here to control the execution queue, which means, no matter what context switching methods are used, the returning queue of those sending and receiving requests should remain the same (in test code, we need to use *Yield()* to intertwine the execution order).Note the calling queue can be differentfrom the returning queue shown above.

Code to be implemented:

You should declare your Mailbox class in *"code/threads/synch.h"* (as the signatures have already been provided) and define the member functions in *"code/threads/synch.cc".* The test code in *"code/threads/threadtest.cc"* will instantiate your Mailbox class and call its member functions to simulate the Sending-Receiving situation. You need to use the signatures of the functions provided and keep the test code untouched. You may add member variables as you want. In your implementation of *Send(int)* and *Receive(int*)*, you should return the sent value or the received value(not the pointer to it) when the function returns. Code in *"code/threads/threadtest.cc"* will print the sent and received value to the terminal.

Testing your code:

For testing your implementation, you can execute nachos with following command: *"./nachos –q 5".* Nachos will display with "sent: 0" and "received: 0" as the Mailbox has not been implemented yet (**check both the correct output and this default output in the provided output messages**). If you need the debug information collected by the MACRO "DEBUG", you need to append argument "-d" after this command.

Hints:

1. You can define the condition variables and locks required (or the pointers to them) as member variables of your Mailbox class.
2. Think about how many condition variables are required in multiple senders and multiple receivers situation, which simply means many calls to *Send()* and *Receive()*. And how many locks are required.
3. You can think this as a simple implementation of monitor mentioned here (the two links mentioned in the condition variable part):http://en.wikipedia.org/wiki/Monitor_(synchronization)http://www.stanford.edu/class/cs140/cgi-bin/lecture.php?topic=locks as they share the same mechanism to let the current thread sleep for waiting and wake up a thread to let it go on.

Evaluations

Task 1

Correct output for Lock	30%
Correct output for Conditions	30%

Task 2

Correct output for Join	40%

Bonus

Correct output	20% (extra)

Submission Guidelines

You need to submit 4 files, that you will be modifying for this assignment:

Synch.cc
Synch.h
Thread.cc
Thread.h

Create and submit the zip file as an attachment to the eece315term2@gmail.com with the subject as follows:
"Assignment 2: <Group No.>"

Once you submit the assignment, you will receive an automated reply from the system. You need to show that reply to the TA before they could assess your assignment in the lab, to make sure that you have submitted your assignment correctly.