NACHOS OFFLINE-1 REPORT

Submitted By

1605077

1605079

1605084

PART 1

Task 1: Implement KThread.join()

Changes made in nachos source code:

The methods that we changed here for the **KThread.java**:

- join()
- finish()
- selfTest()
- joinTest1()
- joinTest2()

Data Structures used:

No data structures needed.

Implementation:

1. First of all we initialized a parent thread for every thread and set it to null. If the thread calls j**oin()** function then the current thread is set to the parent thread if it is null otherwise it will return a failure.
2. Then we checked if this thread status is finished .Because if the thread is finished then we can't join it as it cannot be taken to a ready or running state.
3. If the status of this thread is not finished then we let the parent thread to sleep until this thread ends execution.
4. We disabled the interrupt status and restored the status after the task to maintain atomicity.
5. Then when this thread ends executing the **finish()** method is called and in that method we let the parent thread of this thread wake up and take it to a ready state.

Test:

We created two methods for testing **join()** method--**joinTest1()** and j**oinTest2()**.

joinTest1:

In this test we tested that a thread cannot join itself and after running it we found assertion failure which indicates our join() method is working correctly.

joinTest2:

In this test we tested if the main thread can join two child threads.So we created two child threads with runnable **PingTest()** which runs a loop for 5 times in the **run()** method.Then we forked the first child thread and calls **join()** method and saw main thread going to sleep until the first child stops running.After execution of first child thread, main thread woke up.Then similarly we tested for the child thread2 and became successful.So **join()** method is working perfectly.

Task 2: Implementing Condition Variables Directly

In this task, we were assigned to implement condition variables directly with the help of enabling and disabling interrupts that will handle atomicity when multiple threads try to access the same function at the same time. In this implementation we used locks that indirectly use semaphores to work with. A “Lock” type object was used here. This was predefined in the project.

Changes made in nachos source code:

The methods that we changed here for the **Conditon2.java**:

- Condition2() (Constructor)
- sleep()
- wake()
- wakeAll()
- Created **CommunicatorTest.java** for testing purposes.

Data Structures Used:

- **Queue<KThread> threadWaitQueue:** The data structure we used in this purpose was a queue implemented as a linked list in java. This was from the java library and the queue was of the type KThread, that was used to track the threads that were sleeping upon this condition variable by calling the sleep() method of the **Condition2** object.

Implementation:

The implementations are described according to the changes we made in the parts of nachos

1. For **Condition2()** (Constructor) aside from instantiating the lock, we instantiated the queue here.

2. In case of **sleep()**, there was an assertion that checked if the lock was held by the current thread or not. At the very start we disable any interrupt that can occur here, this was done to maintain synchronization. Since interrupt causes an auto context switch after 500 ticks, and we need to disable that, we disable interrupt to avoid this effect. Then the lock was released so that the current Thread calling the method of the condition variable may go to sleep. Before we actually call sleep for the current thread, we push the thread to the queue so that we can wake them up in a first come first serve order when another thread calls **wake()** or **wakeall()**. Before the thread returns, the thread will automatically reacquire the lock. And then finally the thread exits the method by restoring the machine interrupt to the previous status that it disabled the machine interrupt to. Again, this is to maintain synchronization behaviour of multiple threads working on the same **Condition2** object.

3. In case of **wake()**, there was an assertion that checked if the lock was held by the current thread or not. At the very start we disable any interrupt that can occur here, this was done to maintain synchronization. Since interrupt causes an auto context switch after

500 ticks, and we need to disable that, we disable interrupt to avoid this effect. Then we check if the threaded queue is empty or not. If it is not empty, then we remove a single Kthread reference that was sleeping on this **Condition2** object. Then we wake the thread up. And then finally restoring the interrupt just like before, we return from the function.

4. In case of **wakeAll()**, there was an assertion that checked if the lock was held by the current thread or not. At the very start we disable any interrupt that can occur here, this was done to maintain synchronization. Since interrupt causes an auto context switch after 500 ticks, and we need to disable that, we disable interrupt to avoid this effect. Since this method is used to wake up all the threads, we just use a loop to check if the queue is empty repeatedly and wake up a single sleeping thread from the front of the queue until all the sleeping threads are woken up. And then finally restoring the interrupt just like before, we return from the function.

Test:

A test for this **Condition2** object was made in another class **ConditionTest2.java**. A static method was made to avoid complexity. In the test, first a single condition lock was taken and a **Condition2** object was made that uses the lock. Then we created two KThreads in the same pattern and using lambda, we overridden the runnable run method. In the first thread's run method, we acquire the lock and then sleep upon the **Condition2** object we created. We returned from the function by releasing the lock. We forked the thread we created, so that it may go to sleep. Similarly we created another thread, overrode the run method in the exact same way using the same **Condition2** object and forked it. Then after that we yielded the "main" thread. The main reason to yield the "main" thread so that this thread should go to the position of the queue behind the first two threads. We intended to wake up the first two sleeping threads by using the main thread and in order to do that, we yielded the main thread. After that we again make two more threads sleep in a similar way, but this time we

intend to wake these two threads by another thread. So we fork the following respective three threads and achieve the desired effect by calling **sleep()** with the **Condition2** object for the first two threads, and **wakeAll()** by the last thread. Thus we tested the **Condition2** cases.

Task 3: Complete the implementation of the Alarm class

Changes Made in Nachos Source Code:

In **Alarm.java** file of **nachos.threads** package:

- **Alarm()** constructor
- **timerInterrupt()** method
- **waitUntil()** method
- **ThreadSleeptimePair** class
- **selfTest()** method

Data Structures Used:

- **PriorityQueue**<ThreadSleepPair>**()**

Implementation:

1. There is only one instance of **Alarm** class associated with every thread. It is a static variable **alarm** in **ThreadedKernel** class. The threads go to sleep until a certain amount of time in ticks, calling the **Threadedkernal.alarm.waitUntil(x)** method. In this method **wakeup-time** is calculated adding current-time and wait-time. Then a **ThreadSleeptimePair** object is created from the **current-thread** and **wakeup-time** and is put to a **PriorityQueue**. Then the Kthread.sleep**()** method is called to put the current thread to sleep. The whole thing is done between **disabling and enabling interrupt** to prevent auto **context-switch**.

2. The **timerInterrupt()** method of Alarm class is called periodically in every 500 ticks. Causes the current thread to yield, forcing a context

switch if there is another thread that should be run. In this method the **PriorityQueue** is checked to see if the wakeup-time of the first thread is greater than the current-time or not. If yes it is put to ready state by calling it's **ready()** method. The whole thing is done between **disabling and enabling interrupt** to prevent auto **context-switch**. After that the current thread forcefully gives away CPU by calling **Kthread.yield()** to allow the popped thread from **PriorityQueue** to execute.

Test:

1. Three threads are created. Each calls **waituntil()** in their run methods.
2. The wait-time for the first one is the largest, then the second one, then the third one.
3. **Kthread.yield()** is called to force the **main thread** to leave the CPU and allow the created threads to execute and call their **waitUntil()** methods.
4. Then **Busy waiting** is done in a **while(){}** loop to let the sleeping threads of the **PriorityQueue** to wake up.
5. In this loop **Kthread.yield()** is called to force the **main thread** to leave the CPU again and again. In each **yield()** there is one disable and enable of interrupt, which measures to 10 ticks. So calling the yield multiple times creates the illusion of passed time.
6. Finally the third thread is seen finishing first, then the second, then the first.

Task 4: Implement synchronous send and receive of one-word messages using condition variables

In this task, we are to create a synchronous communicator that would allow speakers to speak a 32 bit message to a listener. They are both threads and their behaviors work as a pair, that is if a speaker and listener is present, then as soon as the speaker speaks, the listener listens and both quit their respective functions. The implementation was done in such a way that it works for multi speakers and listeners.

Changes made in nachos source code:

The parts that we changed here for the **Communicator.java**:

- Used two new **Condition2** objects, **speaker, listener.**
- Used an int controlFlag
- Used a message variable to transfer data
- Used a **Lock** object
- Communicator() (Constructor)
- speak(int word)
- listen()

Data Structures Used:

No extra data structures were used here. We used **Condition2** objects for speakers and listeners that use a Queue internally that has been described earlier.

Implementation:

1. The speaker will wait for a thread to listen and as soon as a listener is available. The thread using speak() method will sleep and will not return until exactly one listener will receive the word. First we acquire the lock variable and immediately we use an infinite loop. This loop is for trapping any new speaker if they come while another speaker is sleeping. In the loop we check if the controlflag is 0 or not. If it's not we sleep the new speaker thread, because there is already another speaker thread that's sleeping and a listener for that has not arrived yet. This is a way we controlled how multiple speaker scenarios

would work here. We just send a speaker to sleep if there is another speaker that came before it is already sleeping.

2. Now after this loop, we set the message variable to the word we want to speak. Then we increment the control flag, this would help to trap any new speakers. Then we call **wakeAll()** of the **listener** condition2 object upon listener threads so that one wakes up. Now this would call all the listeners if there are multiple ones, but we used a similar type of trapping mechanism in the listener part to trap more than one listeners and that would ensure even if multiple listeners wake up, only one will receive the word and others will again go to sleep eventually. Then we make the speaker thread sleep. In the final statement, we release the lock when we return from the function. Here a speaker thread will only return from a function if a listener has received a word and wakes the speaker up.

3. In case of **listen()** a listener receives a message and returns the received message from the speaker thread. Here a lock is acquired at first. Then we take a variable to hold the received word or message. Then again we check if there is any speaker that has already spoken or not. If the speaker has not spoken yet and the listener came here first, then we need to trap the listener. Inorder to trap or wait the listener by calling sleep, we again used a loop to control this. In the loop we check if the **controlflag** is 1, if it is, we break out of the loop because this is one when there is a speaker speaking. Now if it is not, the listener goes to sleep. This loop also helps to trap or make other listeners wait when there are multiple listeners and one of them is receiving the word that is being spoken by a speaker thread.

4. After this loop, we receive the message from the speaker, and we wake up any speaker thread that has been sleeping so far. Here, we use **wakeAll()** of the **speaker** condition2 object. Although this would wake up all the speakers that have been sleeping, only the speaker that has gone to sleep at first would be in effect here. Because in the next line, we decrement the control flag, that would trap other

speakers even if they have woken up in the loop of the **speak()** function. Then we finally release the lock and return the received message that we obtained from the receiver.

Test:

We created a **CommincatorTest.java** to test the threads working upon a communicator object. We created two classes, one speaker and one listener class that implements a Runnable interface and overrides the **run()** methods. The speaker class has a private communicator object and a message variable. The message is randomly generated in the **Speaker** constructor. then we used **run()** to speak the message.
Similarly in the **Listener** class, we use a private communicator object, which is set in the constructor. Then we simply use **listen()** with this in the **run()** method of this class.

For the case of the test, we created two static functions, **selfTest1()** and **selfTest2()**. In the first test, a single communicator object was used and a random sequence of speaker and listener thread of equal numbers were forked with the class created earlier. A sequence of this kind was created:

Speaker - Listener - Listener - Speaker - Listener - Listener - Speaker - Speaker - Speaker - Listener.

And the result was that we obtained pairs of speakers and listeners speaking and listening to random messages, but it occurred in a total of 5 pairs. Now for the second **selfTest2(),** we created a communicator object. We then used two listener threads and forked them and then two speakers threads and forked them. After that, we again obtained the desired result, the listeners were waiting as expected and when the speakers were speaking, one by one they were listening.

PART 2

Task 1: Implement the system calls read and write documented in syscall.h

Changes Made in Nachos Source Code:

In **UserProcess.java** file of **nachos.userprog** package:

- **UserProcess()** constructor
- **handleHalt()** method
- **handleRead()** method
- **handleWrite()** method
- **handleSyscall()** method

Constants.java interface is added in **nachos.userprog** package.

Data Structures Used:

- An array of **OpenFile**s.

Implementation:

- **UserProcess():** The **ProcessID** is set to the number of processes, stored in a static variable. The **OpenFile** array is initialized. The 1st two elements of the array are set to standard input and output.

- **handleSyscall():** method two more cases are added. One case calls **handleRead()** method to handle **read** system-call. The other case calls **handleWrite()** method to handle **write** system-call.

- **handleHalt():** A condition is added to check if the process calling **halt** system-call is the root process. If not, -1 is returned.

- **handleRead():** At the beginning of the method the validity of the parameters are checked. At this stage the value of the fileDescriptor parameter can be 0 (read) or 1 (write). The **read()** method of

UserKernel.console.openForReading() is used to read the requested size from **standard input** to a read buffer. If the read method returns -1 then an error has occurred and -1 is returned. If the read is successful, the read buffer is written to **virtual memory** of the process starting from the given **virtual address**. If written bytes size is less than the total bytes that have been read then -1 is returned else total bytes is returned.

- **handleWrite():** At the beginning of the method the validity of the parameters are checked. First, the data of requested size is read from the given virtual address of the virtual memory to a write buffer. If total bytes read is less than requested size then -1 is returned. Then the write**()** method of **UserKernel.console.openForWriting()** is used to put the write buffer into standard output.If written bytes size is less than the total bytes that have been read then -1 is returned else total bytes is returned.

Test:

- A simple C file is written with basic **read**, **write** and **halt** system calls. Some invalid read, write calls were also written. The test gave desired output.

Task 2: Implement support for multiprogramming.

Changes Made in Nachos Source Code:

In **UserProcess.java** file of **nachos.userprog** package:

- **UserProcess()** constructor
- **load()** method
- **loadSections()** method
- **allocate()** method
- **releaseResources()** method

- **readVirtualMemory()** method
- **writeVirtualMemory()** method
- **virtualToPhysicalAddress()** method
- **unloadSections()** method

In **UserKernel.java** file of **nachos.userprog** package:

- **initialize()** method
- **getFreePage()** method
- **restorePage()** method

Data Structures Used:

- Static **LinkedList<Integer>** of physical pages

Implementation:

- **initialize():** The lock is initialized here. The linkedlist of physical pages is initialized to hold the number of physical pages provided by the processor.

- **getFreePage():** If there is a physical page available, the page number is returned otherwise -1 is returned. The task is surrounded by a lock.

- **restorePage():** The provided physical page number is added to the free physical page list.

- **UserProcess():** The **ProcessID** is set to the number of processes, stored in a static variable surrounded by a lock to maintain atomicity.

- **load()**: The number of pages required by the process is calculated from the length of the coff sections of the coff file, stack pages and 1 extra page for arguments. Then memory for the required pages are allocated using the **allocate()** method. If allocation is failed, false is returned. After allocation the coff sections are loaded into the physical memory using **loadSections()** method.

- **allocate():** In this method, the number of physical pages required starting from virtual page number 0 to the value of number of pages is mapped to the available physical page numbers using a **TranslationEntry** object. These mappings are stored in the page table for each process. If no page is available for allocation, resources are released and false is returned.

- **releaseResource():** The physical page numbers are restored to the free physical page linked list and the page tables entries are restored to their default state.

- **loadSections():** The section pages are loaded into their mapped physical pages. The status of translation entry of the sections are set to their corresponding status.

- **unloadSections():** The resources are released by the **releaseResource()** method and the coff file is closed.

- **virtualToPhysicalAddress():** This method takes a virtual address and returns the physical address. The virtual page number and the offset is received from the processor by the virtual address. Then the corresponding physical page number of the virtual page number is retrieved from the pagetable of the process. Then finally the physical address is calculated.

- **readVirtualMemory():** At the beginning of the method the validity of the parameters are checked. The required data from physical memory is transferred page by page to the buffer. Until the current virtual address is less than the last virtual address to be read and is less than the last virtual address of the physical memory, a while loop is executed. Inside the loop, the required data from memory to buffer is copied page by page. The amount to be copied is either equal to the page size when within limit or equal to the remaining size of the data to be read. The required physical address to start reading from

is calculated using **virtualToPhysicalAddress()** method using current virtual address. Finally after the page read, the current virtual address is set to the next page's virtual address. The offset and the total amount is incremented by the amount copied.

- **writeVirtualMemory():** At the beginning of the method the validity of the parameters are checked. The required data from the buffer is transferred page by page to the physical memory. Until the current virtual address is less than the last virtual address to be read and is less than the last virtual address of the physical memory, a while loop is executed. Inside the loop, the required data from buffer to memory is copied page by page. The amount to be copied is either equal to the page size when within limit or equal to the remaining size of the data to be written. The required physical address to start writing from is calculated using **virtualToPhysicalAddress()** method using current virtual address. Finally after the page write, the current virtual address is set to the next page's virtual address. The offset and the total amount is incremented by the amount copied.

Test:

- The previous test .C file was successful.