Part 1

Task 1 : Implement Kthread.join()

Nachos source code where changes were made:

- Kthread.join()
- Kthread.finish()

Used Data structure:

- A variable *joinedThread* of **Kthread** type to keep track of the thread joined on this thread.

Basic Idea of Implementation:

1. Inside Kthread.join() at first interrupt was disabled to make sure no context switching occurs. Then checked if any other thread has already called join on this, to make sure this thread to be joined once. So if any other joining thread exists, the function returns without meeting the join request.
2. No thread should be joined on a finished thread,so if the function does not return in (1), it is checked whether this thread is finished. If not,the current thread which called join on this thread, is assigned to the *joinedThread* variable and then the current thread goes to sleep until this thread finishes.
3. Inside Kthread.finish(),it is checked if another thread was waiting for this to finish.If so, the waiting thread which is available in the *joinedThread* variable is made ready so that it can then continue.

Testing of the task:

3 threads thread0,thread1,and thread2 was initiated.
thread0 loops for 10 times
thread1 loops for 5 times
thread2 loops for 5 times
thread1 and thread2 both call join on thread0 but thread2 fails to join thread0 as thread1 has already joined thread0.So it continues and finishes before thread0 finishes. But thread1 successfully joins on thread0 and after thread0 finishes,thread1 continues and finishes.Our testing thread also calls join on thread1 so it continues once thread1 finishes.

Task 2 : Implement condition variable directly

Nachos source code where changes were made:

- Condition2.sleep()
- Condition2.wake()
- Condition2.wakeAll()

Used Data structure:

- A **ThreadQueue** named *waitQueue* (FIFO) to keep the threads waiting on this condition variable .

Basic Idea of Implementation:

1. Inside Condition2.sleep(),at first the condition lock was released so that another thread can acquire it. Then the thread is added to the *watQueue* and goes to sleep until another thread wakes it up via the same condition variable. When it wakes up,it acquires the condition lock of this condition variable again.
2. Inside Condition2.wake(),if there exists any thread in the *waitQueue*, the first thread is made to go to ready state.
3. Inside Condition2.wakeAll(),all the threads(if exist any) in the *waitQueue* are made to go to ready state.

Testing of the task:

Done with Task 4.

Task 3 : Complete the implementation of the Alarm class

Nachos source code where changes were made:

- Alarm.waitUntil()
- Alarm.timerInterrupt()

Used Data structure:

- A **PriorityQueue** of **WaitingThread** class named *waitQueue* to store the threads along with their waking time, which should be woken up after certain ticks. The **WaitingThread** class is written as an inner class of the Alarm class.The instance with early waking time will get priority for this class.

Basic Idea of Implementation:

1. In the previous implementation of Alarm.waitUntil(), busy waiting was used , if the scheduler scheduled it before it’s waiting time was over,it just simply used to yield and go to the ready state again.
2. In our implementation the thread which needs to wait until some certain ticks, its waking time is calculated by summing up the waiting time with the current time.Then a new instance of **WaitingThread** is created with the thread and its waking time and pushed into the *waitQueue.* Then the thread goes to sleep i.e in blocked state.
3. The responsibility of weakening this thread is up to the timerInterrupt() function which is called by the machine's timer periodically (approximately every 500 clock ticks). Before making the current thread to yield, it checks for if there exist threads in the *waitQueue* whose wait time time is overIf so they are made to go to ready state. Since the threads were in blocked state in their full waiting time, the scheduler never scheduled them before their waiting time was over,so no busy waiting here.

Testing of the task:

Three threads t0,t1,t2 was created and made wait until some ticks over the same alarm object (Globally accessible reference to the alarm of ThreadedKernel).
t0 should wait until 1000 ticks
t1 should wait until 500 ticks
t2 should wait until 200 ticks
And the threads wait for at least those mentioned ticks.

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

Nachos source code where changes were made:

- Communicator.speak()
- Communicator.listen()

Used Data structure:

- A **Lock** *waiting* - associated lock of the condition variable
- Three **Condition2** variables *speak,listen and currentSpeaker.*The speakers wait over the *speak* condition variable until it finds a listener and the listeners wait over the *listen* condition variable until it finds a speaker.
- An **int** *toTransfer* to transfer the word from the speaker to the listener. The speaker function sets the variable and the listener function returns it.
- A **boolean** *spoke* to mark if there already exists some speaker to speak.

Basic Idea of Implementation:

- Inside Communicator.speak(), at first the associated lock with the condition variable is acquired since all three condition variable operations can only be used while holding the associated lock. Then it checks if there already exists a speaker via *spoke* flag.If so it goes to sleep on the *speak* condition variable. Next if it is it's turn to speak, it marks the *spoke* flag to be true i.e this is the current speaker now. Then it assigns it's word to the *toTransfer* variable for the listener to listen to and wakes a listener if there exists one which was waiting for a speaker. Then it goes to sleep on the *currentSpeaker* condition variable for some listener to listen it's word and after that it can return, releasing the associated lock.
- Inside Communicator.listen(), at first the associated lock with the condition variable is acquired since all three condition variable operations can only be used while holding the associated lock. Then it checks if some speaker has spoken something, if not it goes to sleep until some speaker speaks something and wakes it up. When it wakes up it listens to the word the speaker has spoken via *toTransfer* variable. Then it wakes up the *currentSpeaker* which has spoken the word so that the speaker can return. It also marks the *spoke* flag to be false and wakes up another speaker if there exists any so that they can now speak. It then returns the word releasing the associated lock.

Testing of the task:

Two listener threads l0,l1 and three speaker threads s0,s1,s2 were created for the same communicator. The speakers spoke some random words and the listeners listened to them.After speaking of one speaker,another speaker couldn’t speak before a listener listened to the first speaker.It was also checked that the speaker and the listener can return once after the transfer is made(commented out the lines to match the format of demonstrated output).
Since communicator uses **Condition2** variable, this test also applies for Task 2.

Part 2

Task 1 : Implement read/write system call

Nachos source code where changes were made:

- UserProcess.handleSyscall()

Used Data structure:

- Two variable stdIn, stdOut of OpenFile type were initialized inside constructor to interact with console.
- Byte array was used to read and write data.

Basic Idea of Implementation:

- Here the read/write function has 3 parameters: fileDescriptor, virtual address of the buffer, the maximum length to write into the file descriptor. As it was mentioned in our assignment, we just implemented the system call for console input/output. So the function call is only valid for fileDescriptor = 0 or 1.
- The write syscall was handled in java inside a method named handleWrite. This method takes the parameter. Then read from the virtual address of the buffer and write directly to console.
- The read syscall was handled in java inside a method named handleRead. This method reads input from the console and then write into the virtual address of the buffer.

Testing of the task:

In out test c code, we wrote to console using printf and read from console using readline. We also tested that the system did not crash when invalid parameters were passed.

Task 2 : Implement support for multiprogramming

Nachos source code where changes were made:

- UserProcess.loadSection()
- UserProcess.readVirtualMemory()
- UserProcess.writeVirtualMemory()

Used Data structure:

- An array of translation entry as page table.
- A LinkedList in UserKernel class to keep track of the available pages.

Basic Idea of Implementation:

- In this task, we enabled our system to run multiple parallel processes. Initially the code was given to us was able to run only the shell program. While loading the program, it was taking physical pages sequentially starting from 0. While reading and writing, it assumed that physical and virtual addresses are equal.
- To implement multiprogramming, We first added a LinkedList to our UserKernel class. The LinkedList was initialized with the kernel. It keeps track of the available pages to prevent overriding. When a new process tries to load its sections, it takes physical pages from from the LinkedList. Then our method fills

up the page table entries which maps virtual to physical page number. Finally sections are loaded into to appropriate physical pages.

- Then we modified our readVirtualMemory and writeVirtualMemory methods. Initially the method assumed the virtual address as the physical address. In our modification, we first translated the virtual address into physical address then we allowed the methods to start reading/writing.

Task 3 : Implement exec, join and exit system call

Nachos source code where changes were made:

- UserProcess.handleSyscall()

Used Data structure:

- An ArrayList for each UserProcess was initialized to keep track of childs for each process.
- Every time a process is created, a new UserProcess object is initialized.

Basic Idea of Implementation:

- Here exec creates a new object for each user process. Then it calls the UserProcess.execute() method. Exec also processes the command line arguments and passes it to the execute method.
- The join system call enables the parent process to wait until the child finishes. This was handled using UThreas.join() method.

- Exit is called when a user process is finished. It finishes the UThread. If the process is the last running process, then it terminates the kernel.

Testing of the task:

In our test c code we tested multi level process exec. We passed command line arguments to our child process successfully. Then we called join on it so that the parent waits until the child finished its execution. We also tested invalid join call. The OS could handle the invalid calls. Parallel process execution was tested in our c code.