Part 1

Task 1: Implement KThread.join()

--In **KThread** class,we created a new KThread named previous.

--In **join()** function,we checked the status of the thread.If the status is less than the running state (value is 2) for example in ready state (value is 1) or in new state (value is 0),we assigned the value of previous to the current KThread and then,we put the current KThread in sleep.

-- In **finish()** function, we are setting the current KThread tobeDestroyed and setting the status to statusFinished. Then we are preparing the previous thread of the current KThread to be ready by calling the ready() method.

--In **selfTest()** function,we created multiple kThreads and checked if the join worked or not.

Task 2: Implement condition variables directly

--For this task,we edited and wrote our code in nachos.threads.Condition2.

--We created a lock and created a linked list of KThread.

--In **sleep()** function, we atomically released the associated lock and we added the current thread in that linked list and sent the current thread to sleep.After that the thread will automatically reacquire the lock.

--In **wake()** function,if the linked list is not empty, then we remove the first thread and call ready() method and put that in ready state and add this to the scheduler's ready queue.

--In **wakeAll()** function,we used a while loop to wake up all threads sleeping on this condition variable.

--In **Condition2Test()** function,we created a lock, a linked list and then created a Condition2 type variable and passed that lock in that constructor.Then we created a producer thread and in the run() method,it acquired the lock and added five integers in that linked list and then we called wake() method on the Condition2 variable and then released the lock.After that we created a consumer thread and in the run() method, it acquired the lock.We checked the linked list.If it is empty we called sleep() method on Condition2 variable and if the list is not empty,then we removed all items one by one using while loop.After that we released the lock.After that we joined that threads.

--In **selfTest()** function, we called the above written **Condition2Test()** method to test our code.

Task 3: Complete the implementation of the Alarm class

--In Alarm class we created a HashMap to keep the Kthread and long type values.

--In **timerInterrupt()** function,this is called by the machine's timer periodically (approximately every 500 clock ticks).Causes the current thread to yield,forcing a context switch if there is another thread that should be run.Here we took the current machine time and took the KThread and its related waking time from the map.If the current time is greater or equal to the waking time,then that KThread will call ready() method and put that in ready state and adds this to the scheduler's ready queue and then removed it from the map.After that we called the yield() method.

--In **waitUntil(long x)** function, at first we calculated the wakeTime adding x and the current machine time.Then we put the current thread as key in the map with its wake time as value and put the current thread to sleep.We will wake it up in the timer interrupt handler.

--In **alarmTest()** function,we created an array of different times.Then for every time, we took the initialTime using the method named Machine.timer().getTime() and then we called the above written waitUntil(d) method.After that we calculated the finishing time and printed the difference.

--In **selfTest()** function,we called the above written alarmTest() method to test our code.

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

--In **Communicator class**,we created a lock,an integer type variable named word and created three Condition2 type variables and they are speaker,listener and interCon.

--In Communicator constructor,we initialized all three Condition2 type variables using that lock.

--In **speak(int word)** function,it waits for a thread to listen through this communicator and then transfer word to the listening thread and does not return until this thread is paired up with a listening thread and exactly one listener should receive word.Here we acquired the lock.If there is already a word, then we put speaker to sleep.If there is no word,then we put value to word variable and called wake()
method on listener and put the interCon to sleep and then released the lock.

--In **listen()** function,it acquired the lock.If there is no word to listen to,then we put the listener to sleep.If there is any word to listen to,then we called wake() method on speaker and interCon locks if any are sleeping.We printed the word and released the lock.

--In **CommTest()** function,we created a Communicator object and two arrays for timeCount and message.Then we created three Kthreads named speaker1,speaker2 and speaker3.In every KThread **run()** method,we called the speak(int word) method on the Communicator object and passed some value and calculated the time and put that in timeCount array.After that we created KThread named listener1,listener2 and listener3.In every **run()** method,we called listen() method on the Communicator object and calculated the time and put that in timeCount array and gotten messages from listen() function in message array.

--In **selfTest()** function,we called the above written CommTest() method to test our code.

Part 2

➔ Task 1: Implement the system calls read and write:

- In function **handleRead** - We implemented Syscallread. We read up to size bytes into a buffer from the file or stream (Here Console) - referred to by fileDescriptor at descriptorIndex. It reads the size bytes from the file by calling the file’s **read** method. And writes to process memory with **writeVirtualMemory** function.
- In function **handleWrite** - We implemented SyscallWrite. We write up to size bytes into a buffer to the file or stream (Here Console) - referred to by fileDescriptor at descriptorIndex.It reads process memory with **readVirtualMemory** function. And writes the size bytes to file by calling file's **write** method.
- In function **readVirtualMemory** - We transfer data from this process's virtual memory to the physical memory array. For this purpose we use address translation of Processor class. Here we do the address translation with **Processor.pageFromAddress** and Processor.makeAddress. We continue copying page by page from data to memory using **System.arraycopy** until the virtual address crosses its length.
- In function **WriteVirtualMemory** - We transfer data from the physical memory array to the process's virtual memory. For this purpose we use address translation of Processor class. Here we do the address translation with Processor.pageFromAddress and **Processor.makeAddress**. We continue copying page by page from memory to data using **System.arraycopy** until the virtual address crosses its length.

➔ Task 2: Implement support for multiprogramming.

- In function **load** - We allocated the necessary memory with paging for both code segment and data segment of .coff files. We loop through the sections of Coff by calling **coff.getSection.** For each section we allocate memory of size **section.getLength()** using the function **allocate** Using **UserKernel.newPage().** If the main memory is full it releases the previously allocated memories using **UserKernel.deletePage** in **releaseResource**.

- In function **unloadsections** - We release the allocated memory for the executable program by calling **releaseResource** stated above. We also close the stdin and stdout files (Actually all files in file descriptor).

➔ Task 3: Implement the system calls (exec, join, and exit)

- In function **handleJoin** - We implemented SyscallJoin. It suspends execution of the current process until the child process specified by the callerProcessID has exited. It takes the child's processID as argument and call's **child.thread.join** to join the child process. Then it gets the child exit status by calling childrenExitStatus.get method and writes in the virtual address specified by argument vaddr with method **writeVirtualMemory.**
- In function **handleExit** - We implemented SyscallExit. This function terminates the current process immediately. All the memory used by the process is released by unloadSections(). If it is the master process then its terminates the kernel **Kernel.kernel.terminate**. Otherwise it calls the **Uthread.finish** method to exit.
- In function **handleExec** - We implemented SyscallExec. We execute the program stored in the file with name stored in the virtual memory address vaddr, with the specified arguments, in a new child process. The child process has a new unique process ID, and starts with stdin opened as file descriptor 0, and stdout opened as file descriptor 1. For this we read the filename specified in virtual address vaddr with method **readVirtualMemoryString.** Then we read args specified by argument numberOfArguments in an array of strings. Then we create a new process with **UserProcess.newUserProcess** and execute the coff file with **child.execute.**