Part 1

Task 1: Implement KThread.join()

In this task, we implement a thread join function so that multiple kernel threads can run perfectly. To do that, we modify ***KThread.join()*** & ***KThread.finish()*** functions.

In the ***KThread.join()*** function, at first we check if the thread is still running or not. If it is running & not the current thread, we add the thread into the join queue until the current thread finishes.

In the ***KThread.finish()*** function, at first we check if the thread is still running or not. If it is finished, we remove all the waiting threads from the join queue & call their ***ready()*** method to let them running.

Test: Here we create 4 kernel threads & join them accordingly. They can run synchronously.

Task 2: Implement condition variables directly

In this task, we implement a condition2 class that disables interrupts for synchronization. Here we modify ***Condition2.sleep()***, ***Condition2.wake()*** & ***Condition2.wakeAll()*** functions

In the ***Condition2.sleep()*** function, at first we stop the machine interrupt & lock the current process for synchronization. Then sleep the current thread & add it to the wait queue.

In the ***Condition2.wake()*** function, at first we stop the machine interrupt & then wake the first thread from the wait queue.

In the ***Condition2.wakeAll()*** function, we do the same as ***wake()*** function for all threads in the wait queue.

Task 3: Complete the implementation of the Alarm class

In this task, we implement an alarm class which is responsible for giving an alarm notification after a certain amount of time. Here we modify ***Alarm.waitUntil(x)*** function.

In the ***Alarm.waitUntil(x)*** function, at first we stop the machine interrupt for synchronization. Then set a wake time of *current time + x* & add the current thread to the priority queue which is responsible for sorting the waiting threads based on their wake time.

Test: Here we create 3 alarm threads & join them accordingly. They can give alarm notifications based on their given time.

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

In this task, we implement a communicator class which is responsible for sending & receiving synchronous messages. Here we modify ***Communicator.speak()*** & ***Communicator.listen()*** functions.

In the ***Communicator.speak()*** function, at first we check if any thread already spoke or not. If no thread spoke previously or spoken word is already listened, then we lock the current process & speak the given word. Then awake all the listeners.

In the ***Communicator.listen()*** function, at first we check if any thread spoke any word or not. If any word spoke already & no one listened to that, then we lock the current process & listen to the spoken word. Then awake all the speakers.

Test: Here we create 3 speakers & 3 listeners and join them accordingly. They can speak and listen as described in the assignment task. Communicator class also uses Condition2 class. So both 2 & 4 tasks are tested together.

Part 2

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

In this task-1 we have to implement 2 system calls one is read and another is write system call. Both the functions take 3 arguments.

1. fileDescriptor
2. buffer
3. size

Actually FileDescriptor can refer to Disk or Console. But, for this Offline we are assuming Only Console. So, the value of FileDescriptor can be 0 or 1. If the value is 0 then it means read and 1 for write. Buffer refers to the start address and size refers to how much byte to read.

In the ***Userprocess.handleWrite()*** function, we first check whether fileDescriptor
is 1 or not then size and buffer is less than 0 or not. if this happens we return -1 as it was said to return -1. Then, use readVirtualMemory function and use *stdOut.write()* function where the std is *UserKernel.console.openForWriting()* and it was said in task-1 to use this function.

In the ***Userprocess.handleRead()*** function, we first check whether fileDescriptor is 0 or not, then size and buffer is less than 0 or not. if this happens we return -1 as it was said to return -1. Then first we read using *stdIn.read()* where stdIn is *UserKernel.console.openForReading()* and it was said in task-1 to use this function.After reading then we use the writeVirtualMemory Function.

In Userprocess.handleSyscall() function we add 2 cases, one for *handleRead* and another for *handleWrite*. One case is *syscallWrite* and another is *SyscallRead*.

Test: We use *mypgr.c* file for testing this task. This file includes all necessary read & write statements.

Task 2: Implement support for multiprogramming

In this task-2 we have to implement support for multiprogramming. Here we modify the *UserProcess* & *UserKernel* class.

In the ***UserKernel*** class, we implement ***UserKernel.allocatePage()*** & ***UserKernel.freePage()***

In the ***UserKernel.allocatePage()*** function, we remove the first page from the freePages list & allocate it to the current thread.

In the ***UserKernel.freePage()*** function, we add the removed or allocated page of the current thread to the freePages list.

In the ***UserProcess*** class, we modify ***UserProcess.load(), UserProcess.loadSections(), UserProcess.unloadSections(), UserProcess.readVirtualMemory()*** & ***UserProcess.writeVirtualMemory()*** functions. Here we added a couple of variables related to the physical page as it is not implemented in the previous code.

In the ***UserProcess.readVirtualMemory()***, we copy data from the kernel's space to the virtual data space. Here for multiprogramming support, we copy data one byte at a time.

In the ***UserProcess.writeVirtualMemory()***, we copy data to the kernel's space from the virtual data space. Here for multiprogramming support, we copy data one byte at a time.

In the ***UserProcess.unloadSections()***, we unload all the pages from the thread & also call ***freePage()*** function of the UserKernel class.

In the ***UserProcess.load()*** & ***UserProcess.loadSections(),*** we load the physical page to the virtual page data table & allocate page for each thread. If there is a lack of pages, we unload the loaded pages immediately.

Test: All the test is done by mypgr.c file & after the task-3 implementation.

Task 3: Implement the system calls (exec, join and exit, also documented in syscall.h)

In this task-3, we implement ***UserProcess.handleJoin()***, ***UserProcess.handleExec()*** & ***UserProcess.handleExit()*** functions. All three are for system calls of *join*, *exec* & *exit*.

In the ***UserProcess.handleExit()*** function, we decrement the running process & unload all the allocated pages for running threads.

In the ***UserProcess.handleJoin()*** function, we join all the child processes of the main thread & write their data to using *writeVirtualMemory().*

In the ***UserProcess.handleExec()*** function, we execute the system call & read the memory data to the arguments to run the given program. Here we use *readVirtualMemory()*.

Finally we enable all three of these system calls in the ***handleSyscall()*** function.

Test: Here we use mypgr.c file to test all the tasks. Cause it already has all necessary types of test statements.