Part 1

Task 1 (Implementation of KThread.join()):
Changes made in source code:

1. KThread class in nachos.threads
1. A new member variable private KThread joinCaller added
2. In public void join()
3. In public static void finish()
2. JoinTest class in nachos.proj1

Used data structures: NA
Basic idea of implementation:
In public void join() function, we disabled the interrupt. If the thread is not finished already, then the current thread gets assigned to joinCaller and called KThread.sleep(). Then we restored the interrupt.
In public static void finish() function, we checked the associated joinCaller. If is not null, we called ready() function on underlying KThread.
Testing of our implementation:
In JoinTest class, we created four new KThread (A, B, C, D) and created the following dependency:
A->B->C->D
D called waitUntil(100000).

Implementing a conditional variable without using semaphores:
Data Structures:

1. A wait queue of KThreads
2. The associated Lock

Location of Code-Base Change:
Nachos.threads.Condition2 has been changed. There were 3 methods: sleep(), wake() and wakeAll(). We implemented all three.

Basic Idea of Implementation:
For synchronization purpose we used disabled interrupt here. As nachos
does not let clock tick till the clock is enabled again, we are safe to do anything atomically as long as we do not call any method that enables the interrupt or call yield() or sleep() or something similar that does context switching.
Any KThread calling any of the conditional's method must hold the lock associated with it.
Sleep:
Sleep method requires that the calling KThread should go into blocked state after giving up the lock as long as any other KThread wakes it up on this conditional variable. The method must return acquiring the lock again.
So, our code order goes as follows:

1. Disable the interrupt
2. Add it to the wait queue of the conditional variable
3. Give up the lock

4. Go to sleep
5. Restore the interrupt
6. Try to acquire the lock and return from the method.

Wake:
Wake should pull out a Kthread from the queue and make its status to ready. Wake does not necessarily give up the lock immediately according to our policy. It is up to the user of the class to release the lock appropriately.
So, our code order goes as follow:

1. Disable the interrupt
2. Pull a KThread from the queue
3. Call ready on the Kthread
4. Enable the interrupt

WakeAll:
WakeAll is almost the same as wake. The only difference is: it just
pulls out all the KThreads and put them into ready state. Of course, the entire operation has to be surrounded by interrupt disabling and enabling.

Testing:
We did not write explicit tests for Conditional2. But the Communicator class uses this. It should be impossible for the Communicator to work despite having a bug in the Conditional2.

Task 3 (Completion of the Alarm class, by implementing the waitUntil(long x) method):
Changes made in source code:

1. Alarm class in nachos.threads
1. In public void timerInterrupt()
2. In public void waitUntil(long x)
3. Added a new private class called PendingKThread
2. WaitUntilTest class in nachos.proj1

Used data structures:

1. Min heap: A min-heap of PendingKThread sorted in non-decreasing order of time (in case of tie, KThread Id).

Basic idea of implementation:
An instance of PendingKThread has two member variable:

1. private KThread kThread: reference to KThread that called waitUntil(long x)
2. private long time: represents when it can be ready to run

In waitUntil(long x), we disabled the machine interrupt, created a new instance of PendingKThread using the calling KThread and time is set to current time added with x. Added that instance to min-heap. Then called KThread.sleep() and restored the machine interrupt.

public void timerInterrupt() function, we checked the min-heap, keep extracting the top element if its time is smaller than or equal the machines current time and called the ready() function on the associated KTHread.

Testing of our implementation:
In WaitUntilTest class we created three new KThread each calling waitUntil(long x) with different x (0, 5000, 1000000). Printed the time when waitUntil gets called and when it exits and checked the elapsed time.
Task 4: Here we implemented synchronous send and receive of one-word messages. To do this, we modified the implementation of nachos/threads/Communicator class and implemented the methods void speak(int word) and int listen(). We used six condition2 variables, six corresponding boolean variables and an integer variable message to pass the message from the speaker to the listener.

In speak(), a speaker will first wait until it gets to the leading position (at leading position, it will be the next speaker to speak). After that, it will wait until a listener comes at the leading position. Then the speaker will speak the word by saving it in message. Next, the speaker will wait until the listener has received the message. After that, the speaker will wake another speaker waiting to get to the leading position and leave.

The implementation of listen() is very similar to that of speak(). A Listener will first wait until it gets to the leading position (at leading position, it will be the next listener to speak). After that, it will wait until a speaker comes at the leading position. Then the lister will wait until the speaker has spoken the message by saving it in message. Next, the listener will receive the word by taking it from message. After that, the listener will wake another listener waiting to get to the leading position and leave.

To test this task, we started three speaker threads and three listener threads. Each speaker thread speaks two times and each listener thread listens also two times. The last listener thread was started after a long delay from the rest of the speaker-listener threads. So, the last two speak() methods had to wait for a long time to give their messages to the last listener and leave. We confirmed that by checking the time of each speak and listen.

Part 2

Task 1 (implementation of the system calls read and write documented in syscall.h):
Changes made in source code:

1. UserProcess class in nachos.userprog
1. private static UserProcess rootProcess
2. private OpenFile stdin;
3. private OpenFile stdout;
4. private int handleRead(int fd, int buffAddress, int count)

5. private int handleWrite(int fd, int buffAddress, int count)
6. public int readVirtualMemory(int vaddr, byte[] data, int offset, int length)
7. public int writeVirtualMemory(int vaddr, byte[] data, int offset, int length)
8. public static int pageFromAddress(int address)
9. public static int offsetFromAddress(int address)
10. public static int makeAddress(int page, int offset)

Used data structures: NA
Basic idea of implementation:
rootProcess keeps the root process. When a process calls halt(), we check if the caller is the root process. If it is, the system halts otherwise ignored.
stdin and stdout gets initialized in UserProcess constructor.
handleRead() reads from stdin and uses writeVertualMemory() to writes to memory. Similarly, handleWrite() reads from memory using readVertualMemory() and writes to stdout.
pageFromAddress(int address) returns the page number specified by the address.
offsetFromAddress(int address) returns the offset for the specified address within corresponding page.
makeAddress(int page, int offset) returns virtual address using the page number and offset.

readVirtualMemory(int vaddr, byte[] data, int offset, int length) tries to read from virtual memory starting from vaddr. It will try to read 'length' amount of bytes. What we did to implement this functionality is following:

1. Dividing the virtual address space into pages
2. Find the corresponding physical pages using process's page table
3. Then read from these physical pages and stored to data. We calculated the amount of portion from a single physical page that falls in out range before reading.

writeVirtualMemory(int vaddr, byte[] data, int offset, int length) tries to write to virtual memory starting from vaddr. It will try to write 'length' amount of bytes. What we did to implement this functionality is following:
Similar to readVirtualMemory except for in step 3, instead of reading from physical memory we write back to it.
Testing of our implementation:
We tested our implementation of read and write functionality using the given echo.c in nachos.test .

Part-2 Task-2:
Implementing Support for Multiprogramming:
Data Structures:
A LinkedList of free pages of main memory(freePagePool)
Location of Code-Base Change:
UserThreadKernel:
We added a static linkedList of free page at the end to maintain free pages of RAM.

UserProcess:
Following methods have been changed:

1. load
2. loadSections
3. ReadVirtualMemory
4. WriteVirtualMemory

Basic Idea of Implementation:
Initially in the kernel we loaded the entire RAM into the freePagePool linkedList. We know the number of pages of the RAM from the processor's numPhysPages attribute.
When executed method is called either by another process or the UserKernel, it will try to load the process into main memory.
Load() method will first add the memory needed for the different sections, for stack (which is assumed constant here, 8 pages) and 1 page for command line arguments. After this calculation, we know precisely the amount of pages the new process will need (as we do not allow malloc and free syscall) in the numPages attribute in the new process.
After knowing the needed pages, at the very beginnnig of the loadSections() method, we check whether our main memory has enough pages to accommodate this new process by checking the freePagePool linkedlist. If the new process cannot be accommodated, the process execed is failed.
If we have enough pages left in the freePagePool, we pop a page (we do not apply any fancy method) from the freePagePool and assign it to the corresponding vpn and updates the pageTable[] array with this new TranslationEntry. After this, we load the sections of the .coff file into the correspoing main memory pages and in the process, we update the readOnly attribute of the sections in the translationEntry accordingly.
After that, the entire .coff file will execute. If any read or write syscall is invoked by the syscall, ReadVirtualMemory and WriteVirtualMemory will use the pageTable information as described in part2 task1.
Finally, when the process will exit either willingly or forcefully, before killing the process, we will loop through the pageTable and put the pages used by this process back into the freePagePool (as this process does not need this any longer).

Testing:
This subtask is tested heavily by the task-1 and task-3 of the part-2.
Task 3: Implementation of the system calls exec, join, and exit, also documented in syscall.h

Changes made in source code:

1. nachos/userprog/UserProcess.java
    a. public int handleSyscall(int syscall, int a0, int a1, int a2, int a3)
    b. public void handleException(int cause)
    c. private int handleJoin (int childProcessID, int statusPointer)
    d. private void handleExit (int status)
    e. private int handleExec (int fileNameVaddr, int argc, int argvVaddr)

f. private void killProcess (int status, boolean normallyExited)

Used variables and data structures:

1. An arraylist of child processes
2. Parent process and the associated Kthread of parent process for each process (variables)
3. Some boolean variables indicating if a process has been called joined on, if it is finished and if it has normally exited
4. The exit status for each process (variable)

Basic idea of implementation:

To implement exec, we implemented a method called handleExec(). It first extracts the new process name and the command line arguments from the system call arguments by using readVirtualMemory(). Then it sets the child process' parent to this and tries to call execute() on the child process. On success, it returns the child processID. Otherwise, it returns -1.

To implement join, we implemented a method called handleJoin(). It first checks if the processID argument is actually the processID of one of its children. If not, it returns -1. Then it checks if the child it is calling joined on is already dead. If not, if set the child's joined to true and goes to sleep. After waking, it saves the child's exit status to it's virtual memory by using writeVirtualMemory(). Then it removes the child from it's child list (so it cannot call join on this child anymore) and returns the appropriate value.

To implement exit, we implemented two methods: handleExit() and killProcess(). killProcess takes the exit status as an argument. It frees all the pages used by this process, wakes up it's parent Kthread (if it called join) and then kills this process by calling Kthread.finish() or kernel.Kernel.terminate() in case it is the last process. killProcess() is called from handleExit(), handleSyscall() and handleException() methods. Each case, the appropriate exit status is passed as argument.

Testing of our implementation:

To test join, we recursively called join() on a process several times and checked that it worked correctly. We also returned random exit status from a process to check exit syscall works properly. And as exec is used in each case, it is also tested.