Task 1

In the implementation of KThread.join() function, we define a ThreadQueue joinQueue for each KThread object. For a thread A, A.joinQueue contains all the threads which are waiting for A. When a thread A (currentThread) calls B.join(), B will add A into B.joinQueue and call KThread.sleep() to make A sleep. When thread B finishes, it will add all the threads in its joinQueue to the readyQueue.

To implement the idea, we should modify KThread.join() and KThread.finish().

pseudocode:

```
1. KThread.join():
        if (this.status != statusFinished):
                disable interrupt
                joinQueue.push(currentThread)
                sleep(currentThread)
                restore interrupt

2. KThread.finish():
        // normal finish code...
        for (eachThread in currentThread.joinQueue):
                ready(eachThread)
        sleep(currentThread);
```

To test join() function, we will construct several threads, each thread will print 1 to 100 consecutively. Some thread will join others in the mid of the procedure. And we will check whether they print the numbers in a right manner.

Task 2

In the implementation of Condition2 class, we should define a new ThreadQueue: waitQueue, the waitQueue is used to store the threads that are waiting for the Condition corresponding to the Condition variable to be satisfied. This waitQueue is the replacement for the waitQueue in Condition class which is a LinkedList of semaphore because we can not use semaphore in Condition2 class. The other components are the same as Condition class.

WeWe should add instructions in Condition2.sleep(), Condition2.wake() and Condition2.wakeAll() to make these work fine in multi-thread environment. To realize the atomicity, we should put all the procedure inside the environment that the interrupt is disabled and after the procedure is finished we should enable the interrupt.

pseudocode:

```
1. Condition2.sleep():
        disable interrupt
```

```
        lock.release()
        waitQueue.push(currentThread)
        sleep(currentThread)
        lock.acquire()
        restore interrupt
```

2. Condition2.wake():

```
        disable interrupt
        waitThread = waitQueue.pop()
        if waitThread != null:
                ready(waitThread)
        restore interrupt
```

3. Condition2.wakeAll():

```
        while waitQueue != null:
                wake()
```

To test Condition2 class, we create a set of Condition2 variables cond[1..n], a set of integers a[1..n] and a set of threads t[1..n]. Initially a[i]=0 for all i=1..n. Every Condition2 variable cond[i] is satisfied if and only if a[i] >= 0. When the threads start executing, they randomly select an a[i] and randomly plus or minus a[i] by some number. We just check whether all threads run in a right manner.

Task 3

In the implementation of alarm class, we should complete two functions:

1. timerInterrupt();
2. waitUntil(long x);

To implement task 3 efficiently, we firstly define a new subclass TimePlusTread that inherit from class Comparable. A TimePlusTread object contains two members:

1. KThread Thread;
2. long wakeTime;

In class alarm, we define a set of TimePlusTread, The elements in set is recorded in the ascending order of the wakeTime of them.

Now, we can complete the above functions. In timerInterrupt() function, we check all the elements in TimePlusTread set and move the threads which satisfies the time condition out of the set and add them into readyQueue. In waitUntil(long x) function, we combine the current thread and the wakeTime (wait time plus current time) and put it into the TimePlusTread set.

pseudocode:

```
        1: class TimePlusTread implements Comparable:
```

```
                KThread thread;
                long wakeTime;
                KThread getThread();
                void setThread(KThread thread);
                long getWakeTime();
                void setWakeTime(long wake_time);
                int compareTo(Object p);

        2: add Set<ThreadTime> Threadset into class alarm;

        3: void timerInterrupt():
                for eachElement in Threadset:
                        if (eachElement.getWakeTime() <=
Machine.timer().getTime())
                                ready(eachElement.getThread());
                                eachElement.reomve();

        4: void waitUntil(long x):
                disable interrupt
                long wakeTime = Machine.timer().getTime() + x;
                TimePlusTread tpt = new TimePlusTread();
                threadtime.setThread(KThread.currentThread());
                threadtime.setWakeTime(wakeTime);
                Threadset.add(tpt);
                sleep(KThread.currentThread());
                restore interrupt
```

To test task 3, we just construct several threads and infinitely call waitUntil(randomTime). And we check whether all the threads run in the right manner.

Task 4

To implement Communicator class. We define one lock and two condition variables. One condition variable is satisfied if and only if the communicator can speak, the other is satisfied if and only if the communicator can listen.

For each communicator, the constructor should contain the elements as follows:

```
        int num;
        enum {SPEAK_READY, LISTEN_READY} status;
        Lock lock;
        Condition speakCondition;
        Condition listenCondition;
```

num is the message we send, speakCondition is satisfied if and only if status == SPEAK_READY and listenCondition is satisfied if and only if status == LISTEN_READY.

pseudocode:

```
1. Communicator():
        lock = new Lock();
        speakCondition = new Condition(lock);
        listenCondition = new Condition(lock);
        status = SPEAK_READY;

2. void speak(int word):
        lock.acquire();
        whlie status == LISTEN_READY:
                speakCondition.sleep();
        num = word; // num to be transferred
        status = LISTEN_READY;
        listenCondition.wake();
        lock.release();

3. int listen():
        lock.acquire();
        while status == SPEAK_READY:
                listenCondition.sleep();
        status = SPEAK_READY;
        speakCondition.wake();
        lock.release();
        return num;
```

To test the communicator, we need to create separate threads and run their speak methods several times in a short time period, and then run listen method for several times see if only after the spoken message has been listened will a second speak run successfully, or they interweave which shows that our method is wrong.

Task 5

In the implementation, we add element thread father_thread in Thread class. Thread father_thread records the thread that blocks the original thread. And also, we should new the schedulingState as ThreadState. We also need run setWrong() in function join(). The rest work is all about file PriorityScheduler.java.

The block relation ship between threads forms a forest. For each tree, the root thread always run first. So, the priority of the thread will never be smaller. Add boolean priority_Wrong in Thread means the priority may be wrong.

pseudocode:

```
1. PriorityQueue:
        private LinkedList<KThread> waitQueue = new
LinkedList<KThread>();

        protected ThreadState pickNextThread():
```

```
                ThreadState res=null;
                        for each ThreadState iterrator in
waitQueue
                                if res is null OR iterator has
higher priority than res
                                        res = iterator
                        return res;

                public KThread nextThread()

Lib.assertTrue(Machine.interrupt().disabled());
                        if (waitQueue.isEmpty())
                                return null;
                        ThreadState firstThread =
pickNextThread();
                        waitQueue.removeFirst();
                        firstThread.acquire(this);
                        return (KThread)firstThread;
```

2. ThreadState:

```
                protected int priority_Wrong;

                public void setWrong():
                        if (priority_Wrong==true)
                                return;
                        priority_Wrong=true;
                        father_Thread.setWrong();

                public void setPriority(int priority):
                        if (this.priority == priority)
                                return;
                        this.priority = priority;
                        setWrong();

                public int getEffectivePriority():
                        if (priority_Wrong==true):
                                for each element ele in
join_queue:
                                        priority = MAX(priority,
ele.getEffectivePriority());
                        priority_Wrong=false;
                        return priority;
```

To test the PriorityScheduler class, we need different test cases for different aspects of it. First, we test the functionality of getPriority and setPriority. We construct several threads, call setPriority to change their priority and call getPriority to print their priority. Second, we test the functionality of priority donation, we construct several threads with different priority and use join() function in task 1 to ask the thread with high priority wait for a thread with low priority. If there is a thread with a middle priority. Then we can call getEffectivePriority to test whether it works.

Task 6

We first give a brief introduction to our strategy and then define the variables.

Out strategy is following. If the boat is in Oahu and there are at least 2 children in Oahu, then 2 children go to Molokai. If the boat is in Oahu and there is exactly 1 child in Oahu, then 1 adult goes to Molokai. If the boat is in Molokai, then 1 child goes back to Oahu. Of course, when all children and adults go to Molokai we should terminate our process.

To make sense the strategy is work, one can check the following facts. The strategy will first allow all children to go to Molokai (2 children go to Molokai and then 1 child goes back). When there are only adults in Oahu, the boat must be in Molokai and 1 child goes back, then an adult goes to Molokai, 1 child goes back, 2 children go to Molokai. After a round, one adult goes to Molokai and there are still only adults in Oahu. The cycle will execute several times until there are no adult in Molokai.

To implement the class Boat, we should have several variables to store the state of the system. These variable are member variables in the Class Boat:

```
        childLeft        the number of children in Oahu
        adultLeft        the number of adults in Oahu
        boatPos          0 for boat in Oahu, 1 for boat in Molokai
        pilotGot 1 for a child becomes a pilot from Oahu to Molokai
but the passenger doesn't come, 0 otherwise
```

Certainly, we need a lock of this Boat class to achieve atomicity:

```
        mutex            the lock for Boat object
```

To decide how to move next, we need some condition variables to decide the following conditions (they are description of the strategy above):

```
        adultGo          pilotGot = 0 and childLeft = 1 and boatPos
= 0
        childGo          (childLeft >= 2 or pilotGot = 1) and
boatPos = 0
        childBack        boatPos = 1
        done             adultLeft = 0 and childLeft = 0
```

At the beginning, we should initialize those condition variable and lock in the construction function

```
        Boat::Boat()
                mutex = new Lock()
                adultGo = new Condition(mutex)
```

```
                childGo = new Condition(mutex)
                childBack = new Condition(mutex)
                done = new Condition(mutex)
                begin(adultNum, childNum)
```

In the method begin(), we should fork all child threads and adult threads following the number of adults and children, then the Boat class just checks status done forever:

```
        void Boat::begin(adultNum, childNum)
                adultLeft = adultNum
                childLeft = childNum
                boatPos = 0
                pivotGot = 0
                mutex.acquire()
                for i in range(adultNum):
                        thread adult = new adultThread()
                        adult.fork()
                for i in range(childNum):
                        thread child = new childThread()
                        child.fork()
                while not (adultLeft = 0 and childLeft = 0): //
done condition
                        done.sleep()
                mutex.release()
```

In the adultThread, the adult should wait for the condition that he can go to Molokai and he will never get back then

```
        void Boat::AdultItinerary()
                mutex.acquire()
                while not (pilotGot = 0 and childLeft = 1 and
boatPos = 0): // adultGo condition
                        adultGo.sleep()
                AdultRowToMolokai()
                adultLeft -= 1
                boatPos = 1
                if adultLeft = 0 and childLeft = 0: // update done
condition
                        done.wake()
                if boatPoas = 1: // update childBack condition
                        childBack.wake()
                mutex.release()
```

In the childThread, the child should go to Molokai at once if he finds possible. After he decides to go to Molokai, he should check whether he can be the pilot or the passenger.

```
        void Boat::ChildItinerary()
                myPos = 0
                while True:
                        mutex.acquire()
                        if myPos = 0:
                                while not ((childLeft >= 2 or
```

```
pilotGot = 1) and boatPos = 0): // childGo condition
                                        childGo.sleep()
                                if pilotGot = 0:
                                        // if this child is the
pilot
                                        pivotGot = 1
                                        ChildRowToMolokai()
                                else:
                                        // if this child is the
passenger
                                        ChildRideToMolokai()
                                        pilotGot = 0
                                        boatPos = 1
                                childLeft -= 1
                                myPos = 1
                                if adultLeft = 0 and childLeft =
0: // update done condition
                                        done.wake()
                                if boatPos = 1: // update
childBack condition
                                        childBack.wake()
                        else:
                                while not (boatPos = 1): //
childBack condition
                                        childBack.sleep()
                                ChildRowToOahu()
                                childLeft += 1
                                boatPos = 0
                                myPos = 0
                                if pilotGot = 0 and childLeft = 1
and boatPos = 0: // update adultGo condition
                                        adultGo.wake()
                                if (childLeft >= 2 or pilotGot =
1) and boatPos = 0: // update childGo condition
                                        childGo.wake()
                        mutex.release()
```

The way to test this program is running this program by setting different case of numbers of children and adults, and see whether the program is running in a right manner.

Remark: We think this program has a bug (well, actually we think it is due to the problem) --- If all children and adults go to Molokai, then the parent thread will be waken. However, a child thread will still try to go back to Oahu. These two threads will cause mess. The problem assumes that the people do not have any technology other than a boat! Therefore, the children have no memory skill. They never know there is a person in Oahu or not. Therefore, the parent thread cannot stop them going back to Oahu. And here comes the bug. We wonder how to understand the problem correctly or fix the bug.

Maybe not. You can find some ways to let a child not return to Oahu when unnecessary. For example, when two children gets to Molokai, one of them can sleep and the other one can return to Oahu (if the task is not finished).

It is acceptable that each person knows sth. For example, if I am on Oahu, I should be able to know: how many adults and how many children there are besides me. I also know if the boat is on this side. You can read this —>

So it is possible for the last one to know that he is the terminator of this task.