Project 1 - Build a thread system for kernel processes

Xie Yuanhang 2011012344
Kuang Zhonghong 2011012357
Li Qingyang 2011012360
Yin Mingtian 2011012362
Wang Qinshi 2012011311

Contents

1 Implementation of `KTread.join()`

1.1 Overview

In this task, we need to implement `KThread.join()`, and the method should

- if this is finished return immediately
- be called only once
- store currentThread and wake it later
- assure that the thread finish executing normally

1.2 Correctness Constraints

- Corner cases: when this equals to currentThread or this is already finished, return immediately
- Need to be atomic by disable interruption and restore interruption at last
- When this thread finish, it should wake the stored thread

1.3 Declaration

- Add new state variable `waitQueue` which is a `ThreadQueue(true)` into `KThread`.
- Modification of `join()`: Firstly, disable interruption to make the operation atomic and restore it at last; secondly, prevent joining itself and prevent a finished thread from joining others, otherwise, let the currentThread sleep to wait this method finish.
- Modification of `finish()`: Assure that once this method is finished, wake the thread join it to let the joining thread normally continue executing.

1.4 Description

Shown in pseudocode.

```
procedure JOIN()
    Disable Interruption
    if currentThread == this or this.status == statusFinished then
        Restore Interruption
        return
    else
        add currentThread into waitQueue
        currentThread sleep
    end if
    Restore Interruption
end procedure
procedure FINISH()
    ...
    currentThread.status = statusFinished
    Ready thread in waitQueue
    sleep()
end procedure
```

1.5 Test

Using `selfTest()` method in `KThread`, the main thought is making some `join` example, so we have those tests below.

```
	private static void test1(){
		//Test Case 1
		System.out.println("\n*** Test Case 1 for join ***");
		ToJoin toJoin = new ToJoin();
		KThread toJoinThread = new
		    KThread(toJoin).setName("ToJoin Thread");
		KThread toBeJoinedThread = new KThread(new
		    ToBeJoined(toJoinThread)).setName("ToBeJoined
		    Thread");
		toBeJoinedThread.fork();
		toJoinThread.fork();
		ThreadedKernel.alarm.waitUntil(100000);
	}
	private static void test2(){
		//Test Case 2
		System.out.println("\n*** Test Case 2 for join ***");
		ToJoin toJoin = new ToJoin();
		KThread toJoinThread = new
		    KThread(toJoin).setName("ToJoin Thread");
```

```
        KThread toBeJoinedThread = new KThread(new
            ToBeJoined(toJoinThread)).setName("ToBeJoined
            Thread");
        toJoinThread.fork();
        toBeJoinedThread.fork();
        ThreadedKernel.alarm.waitUntil(100000);
    }
    private static void test3(){
        //Test Case 3
        System.out.println("\n*** Test Case 3 for join ***");
        ToJoin toJoin = new ToJoin();
        KThread toJoinThread = new
            KThread(toJoin).setName("ToJoin Thread");
        KThread toBeJoined1 = new KThread(new
            ToBeJoined(toJoinThread)).setName("ToBeJoined1
            Thread");
        KThread toBeJoined2 = new KThread(new
            ToBeJoined(toJoinThread)).setName("ToBeJoined2
            Thread");
        toBeJoined1.fork();
        toBeJoined2.fork();
        toJoinThread.fork();
        ThreadedKernel.alarm.waitUntil(100000);
    }
    private static void test4(){
        //Test Case 4
        System.out.println("\n*** Test Case 4 for join ***");
        ToJoin toJoin1 = new ToJoin();
        ToJoin toJoin2 = new ToJoin();
        KThread t1 = new KThread(toJoin1).setName("ToJoin Thread
            1");
        KThread t2 = new KThread(toJoin2).setName("ToJoin Thread
            2");
        KThread toBeJoined = new KThread(new
            ToBeJoinedCouple(t1,t2)).setName("ToBeJoined Thread");
        toBeJoined.fork();
        t1.fork();
        t2.fork();
        ThreadedKernel.alarm.waitUntil(100000);
    }
```

And the results of tests are as follows.

```
*** Test Case 1 for join ***
* ToBeJoined starts running
* ToJoin joins
* ToJoin starts running
* ToJoin ends running
* ToBeJoined continues running after ToJoin Thread finishes
```

```
*** Test Case 2 for join ***
* ToJoin starts running
* ToJoin ends running
* ToBeJoined starts running
* ToJoin joins
* ToBeJoined continues running after ToJoin Thread finishes

*** Test Case 3 for join ***
* ToBeJoined starts running
* ToJoin joins
* ToBeJoined starts running
* ToJoin joins
* ToJoin starts running
* ToJoin ends running
* ToBeJoined continues running after ToJoin Thread finishes
* ToBeJoined continues running after ToJoin Thread finishes

*** Test Case 4 for join ***
* ToBeJoinedCouple starts running
* ToJoin Thread 1 joins
* ToJoin starts running
* ToJoin ends running
* ToJoin starts running
* ToJoin ends running
* ToJoin Thread 2 joins
* ToBeJoinedCouple continues running after two joining threads
```

1. For `test1`, we have the thread being joined run first, and have the other thread join the former run later;
2. For `test2`, just change the running order of the threads in the first test;
3. For `test3`, we have a thread join two different threads;
4. For `test4`, we have a thread be joined by two different threads;

And all the results shows that the joining thread finished before the threads being joined continue which is the expected results.

2 Implementation of `Condition2`

2.1 Overview

Implement `Condition2` without using semaphore and `Condition2` must be equivalent implementation as `Condition`.

2.2 Correctness Constraints

- `sleep` method:
  - Atomically release the associated lock and put the current thread to sleep until be waken
  - The associated lock must be held by current thread before the method and re-required after this method.
- `wake` method:
  - Atomically wake up a thread which called `sleep`
  - The associated lock must be held by current thread before the method.
- `wakeAll` method:
  - Atomically wake up all thread which called `sleep` on this condition
  - The associated lock must be held by current thread before the method.

2.3 Declaration

- Add new state variable `waitQueue` which is a `ThreadQueue` into `Condition2`.
- `sleep` method: place currentThread into ThreadQueue and release the lock, then sleep the currentThread and then re-acquire the lock when it return from `sleep`
- `wake` method: remove the thread from the waitQueue and put it on the ready Queue
- `wakeAll` method: remove all threads from the waitQueue and put them on the ready Queue

2.4 Description

2.5 Test

Using `selfTest()` in `Condition2`,

In this session, the test is using `Condition` and `Condition2` to do the same job, and compare their results, and then we finish the test. Here are the codes.

```
private static class Int {
        int value;
        Int(int _value){
                value = _value;
        }
        public void inc(){
                value++;
        }
```

```
procedure SLEEP()
    Lib.assertTrue conditionLock.isHeldByCurrentThread
    Disable Interruption
    Add currentThread into waitQueue
    Release the conditionLock
    currentThread sleep
    Acquire the conditionLock
    Restore Interruption
end procedure
procedure WAKE()
    Lib.assertTrue conditionLock.isHeldByCurrentThread
    Disable Interruption
    if waitQueue is not empty then
        Remove and wake the first thread in waitQueue
    end if
    Restore Interruption
end procedure
procedure WAKEALL()
    Lib.assertTrue conditionLock.isHeldByCurrentThread
    Disable Interruption
    while waitQueue is not empty do
        Remove and wake the first thread in waitQueue
    end while
    Restore Interruption
end procedure
```

```
        public void dec(){
                value--;
        }
        public int val(){
                return value;
        }
}
private static class Producer implements Runnable {
        private Int goods;
        private Condition2 condition;
        private Lock lock;
        private int index;

        Producer(int _index, Int _goods, Condition2 _condition,
            Lock _lock){
                index = _index;
                goods = _goods;
                condition = _condition;
                lock = _lock;
        }

        public void run() {
                lock.acquire();
                System.out.println("Producer " + index + " starts
                    running");
                goods.inc();
                System.out.println("Producer " + index + "
                    produces 1 item (" + goods.val() + " items)");
                condition.wakeAll();
                System.out.println("Producer " + index + " ends
                    running");
                lock.release();
        }
}
private static class Consumer implements Runnable {
        private Int goods;
        private Condition2 condition;
        private Lock lock;
        private int index;

        Consumer(int _index, Int _goods, Condition2 _condition,
            Lock _lock) {
                index = _index;
                goods = _goods;
                condition = _condition;
                lock = _lock;
        }

        public void run() {
                lock.acquire();
```

```
                        System.out.println("Consumer " + index + " starts
                            running");
                        while(goods.val() < 1){
                                System.out.println("Consumer " + index + "
                                    sleeps " + "(" + goods.val() + "
                                    items)");
                                condition.sleep();
                        }
                        goods.dec();
                        System.out.println("Consumer " + index + "
                            consumes " + "" + 1 + " item (" + goods.val()
                            + " items)");
                        System.out.println("Consumer " + index + " ends
                            running");
                        lock.release();
                }
        }
public static void selfTest(){
        Int goods = new Int(0);
        Lock lock = new Lock();
        Condition2 condition = new Condition2(lock);
        KThread consumer1 = new KThread(new Consumer(1, goods,
            condition, lock));
        KThread consumer2 = new KThread(new Consumer(2, goods,
            condition, lock));
        KThread producer1 = new KThread(new Producer(1, goods,
            condition, lock));
        KThread producer2 = new KThread(new Producer(2, goods,
            condition, lock));
        KThread producer3 = new KThread(new Producer(3, goods,
            condition, lock));
        KThread consumer3 = new KThread(new Consumer(3, goods,
            condition, lock));
        consumer1.fork();
        consumer2.fork();
        producer1.fork();
        producer2.fork();
        producer3.fork();
        consumer3.fork();
        ThreadedKernel.alarm.waitUntil(100000);
}
```

And for `Condition`, we just change the test code above. And here are the results.

```
using condition2:
Consumer 1 starts running
Consumer 1 sleeps (0 items)
Consumer 2 starts running
```

```
Consumer 2 sleeps (0 items)
Producer 1 starts running
Producer 1 produces 1 item (1 items)
Producer 1 ends running
Producer 2 starts running
Producer 2 produces 1 item (2 items)
Producer 2 ends running
Producer 3 starts running
Producer 3 produces 1 item (3 items)
Producer 3 ends running
Consumer 3 starts running
Consumer 3 consumes 1 item (2 items)
Consumer 3 ends running
Consumer 1 consumes 1 item (1 items)
Consumer 1 ends running
Consumer 2 consumes 1 item (0 items)
Consumer 2 ends running

using condition:
Consumer 1 starts running
Consumer 1 sleeps (0 items)
Consumer 2 starts running
Consumer 2 sleeps (0 items)
Producer 1 starts running
Producer 1 produces 1 item (1 items)
Producer 1 ends running
Producer 2 starts running
Producer 2 produces 1 item (2 items)
Producer 2 ends running
Producer 3 starts running
Producer 3 produces 1 item (3 items)
Producer 3 ends running
Consumer 3 starts running
Consumer 3 consumes 1 item (2 items)
Consumer 3 ends running
Consumer 1 consumes 1 item (1 items)
Consumer 1 ends running
Consumer 2 consumes 1 item (0 items)
Consumer 2 ends running
```

In this test result, `Consumer` waits for `Producer` rightly when there is no good and `Consumer`, `Producer` consumes and produces good rightly. And `Condition2` works the same as `Condition`.

3 Implementation of `Alarm`

3.1 Overview

Implementation of `Alarm` class, such that after the thread called `waitUtill(x)` at time `t` then `timerInterrupt` wake the thread after time `x+t`.

3.2 Correctness Constraints

- `waitUtill` method: Move the calling thread into waitQueue and block the thread.
- `timerInterrupt` method: The thread should be woken when the interval on the call is over.

3.3 Declaration

- Create a new class `WaitThread` which contains a thread and its wake time. Add a new instance variable `waitQueue` of `WaitThread` which is a `PriorityQueue` with wake time as priority.
- `waitUtill` method: Calculate the wake time by adding x to current time and the `WaitThread` of the thread should be store in waitQueue which is a priority queue(hence is efficient).
- `timerInterrupt` method: use a while loop to wake threads whose wake time is less than current time.

3.4 Description

---

procedure WAITUNTIL(x)
    Disable Interruption
    Calculate the wake time(x + currentTime) and create a waitThread
    Put waitThread into waitQueue
    sleep this thread
    Restore Interruption
end procedure
procedure TIMERINTERRUPT()
    Disable Interruption
    while waitQueue is not empty AND wake time of first thread in waitQueue < currentTime **do**
        wake first thread in waitQueue
    end while
    Restore Interruption
end procedure

3.5 Test

Call `waitUtill` by several threads and print the calling time and the wake time. Find out whether the threads are waken after wake time.

So the codes are as follows.

```
public static void selfTest(){
        KThread[] t = new KThread[10];
        for(int i = 0; i < 10; i++){
                t[i] = new KThread(new AlarmTest((long)((i+1) * 100)));
                t[i].fork();
        }
        ThreadedKernel.alarm.waitUntil(100000);
}
```

And the result is as follows.

```
Thread starts at 120
Thread calls waitUtill with delay 100
Thread starts at 130
Thread calls waitUtill with delay 200
Thread starts at 140
Thread calls waitUtill with delay 300
Thread starts at 150
Thread calls waitUtill with delay 400
Thread starts at 160
Thread calls waitUtill with delay 500
Thread starts at 170
Thread calls waitUtill with delay 600
Thread starts at 180
Thread calls waitUtill with delay 700
Thread starts at 190
Thread calls waitUtill with delay 800
Thread starts at 200
Thread calls waitUtill with delay 900
Thread starts at 210
Thread calls waitUtill with delay 1000
Thread recovers at 500 (500>=120+100)
Thread recovers at 510 (510>=130+200)
Thread recovers at 520 (520>=140+300)
Thread recovers at 1030 (1030>=150+400)
Thread recovers at 1040 (1040>=160+500)
Thread recovers at 1050 (1050>=170+600)
Thread recovers at 1060 (1060>=180+700)
Thread recovers at 1070 (1070>=190+800)
Thread recovers at 1540 (1540>=200+900)
Thread recovers at 1550 (1550>=210+1000)
```

And the test accurately meet the needs, since all threads continues after the delay time.

4 Implementation of Communicator

4.1 Overview

Implementat `Communicator` class. Two methods `speak` and `listener` to implement. The message is passed from exactly one speaker to exactly one listener.

4.2 Correctness Constraints

- Listeners wait when there is no speaker
- Speakers wait when there is no listener

4.3 Declaration

- Add a state variable `lock` which is a `Lock` into `Communicator`. Add four counters `AS, WS, AL, WL` initially as 0. Add three `conditions` for speaker, listener and return with the same lock. Add a state variable `word` to store words from speakers.
- Note that the first speaker or listener is what we call active speaker or listener and is exchanging message and sleep on condition utill other one wakes it up and both return.
- A speaker speaks to only one listener and a listener listens to only one speaker, so the waiting speaker(listener) will be blocked by active speaker(listener).

4.4 Description

4.5 Test

Test a senquence of speakers and listeners in different orders and check the output.

And the test code as follows.

```
public static void selfTest() {
        Communicator com = new Communicator();
        /* case 1
        (new KThread(new Listener(1, com))).fork();
        (new KThread(new Listener(2, com))).fork();
        (new KThread(new Speaker(1, com, 1))).fork();
        (new KThread(new Speaker(2, com, 2))).fork();
        (new KThread(new Speaker(3, com, 3))).fork();
        (new KThread(new Listener(3, com))).fork();
        */
        ///* case 2
        for(int i = 1; i < 6; i++){
                (new KThread(new Listener(i, com))).fork();
        }
```

```
procedure COMMUNICATOR()
    initialize lock
    initialize AS WS AL WL to 0
    initialize conditions with same lock
end procedure
procedure SPEAK(word)
    Acquire the lock
    while AS != 0 do
        WS++
        // sleep on condition speaker
        sleep
        WS−−
    end while
    AS++
    set word
    if AL != 0 then
        Wake active listener
    else
        if WL != 0 then
            wake one listener
        end if
        // in case that the later speaker runs too fast and cover the word
        // sleep on condition return
        sleep till the wait listener to return
        AS–
        AL–
        if WS != 0 then
            wake the wait speaker
        end if
    end if
    Release the lock
    return
end procedure
```

```
procedure LISTENER()
    Acquire the lock
    while AL != 0 do
        WL++
        // sleep on condition listener
        sleep
        WL−−
    end while
    AL++
    if AS != 0 then
        Wake active speaker
    else
        if WS != 0 then
            wake one speaker
        end if
        // in case that the later listener runs too fast and retreive the word
        // sleep on condition return
        sleep till the wait speaker to return
        AL–
        AS–
        if WL != 0 then
            wake the wait listener
        end if
    end if
    retreive word
    Release the lock
    return
end procedure
```

```
        for(int i = 5; i > 0; i--){
                (new KThread(new Speaker(i, com, i))).fork();
        }
        //*/
        ThreadedKernel.alarm.waitUntil(100000);
}
```

And the results is as follows.

```
case 1
Listener 1 starts listening
Listener 2 starts listening
Speaker 1 starts speaking
Speaker 1 speaks 1
Speaker 1 ends speaking
Speaker 2 starts speaking
Speaker 3 starts speaking
Listener 3 starts listening
Listener 1 hears 1 from speaker 1
Listener 1 ends listening
Speaker 2 speaks 2
Speaker 2 ends speaking
Listener 2 hears 2 from speaker 2
Listener 2 ends listening
Speaker 3 speaks 3
Speaker 3 ends speaking
Listener 3 hears 3 from speaker 3
Listener 3 ends listening

case 2
Listener 1 starts listening
Listener 2 starts listening
Listener 3 starts listening
Listener 4 starts listening
Listener 5 starts listening
Speaker 5 starts speaking
Speaker 5 speaks 5
Speaker 5 ends speaking
Speaker 4 starts speaking
Speaker 3 starts speaking
Speaker 2 starts speaking
Speaker 1 starts speaking
Listener 1 hears 5 from speaker 5
Listener 1 ends listening
Speaker 4 speaks 4
Speaker 4 ends speaking
Listener 2 hears 4 from speaker 4
Listener 2 ends listening
Listener 3 hears 3 from speaker 3
Listener 3 ends listening
```

```
Speaker 3 speaks 3
Speaker 3 ends speaking
Speaker 2 speaks 2
Speaker 2 ends speaking
Listener 4 hears 2 from speaker 2
Listener 4 ends listening
Speaker 1 speaks 1
Speaker 1 ends speaking
Listener 5 hears 1 from speaker 1
Listener 5 ends listening
```

And According the results, each `Listener` listens to exactly one `Speaker` and each `Speaker` speaks to exactly one `Listener` which meets the requirements.

5 Implementation of `PriorityScheduler`

5.1 Overview

Implement `PriorityScheduler` class so that the scheduler can properly schedule the threads by its priority and by priority donation we can avoid priority inversion.

5.2 Correctness Constraints

- Waiting thread donates its priority to the thread which is holding the resource to avoid priority inversion.
- Scheduler always retreive the thread with highest priority from the waitingQueue.

5.3 Declaration

- Here we only consider the case `transferPriority` is true, since when it's false, we only have to sort the threads by their o riginal priority and don't need to calculate `EP`.
- In `PriorityQueue`:
  Add `waitingList`: a priority-queue(realized by maximal heap or red-black tree) by EP of `ThreadState` waiting for this resource. Add a `maxEP` denotes maximal EP in the `waitingQueue`. Add a `boolean changed` denotes whether `maxEP` should be updated. Add a `ThreadState master` which is occupying the resource.
- In `ThreadState`:
  Add a list by EP of `PriorityQueue occupiedResources` collects the resources which is now occupied by this thread. Add a list by EP of `PriorityQueue acquiredResources` collects the resources which is being

acquired(but no yet) by this thread. Add a `boolean changed` denotes whether `EP` should be updated. Add a `EP` denotes EP of this thread.

- The EP should be recursively updated when a new waiting thread enter or when a thread quit holding resources. And updated by rules: Waiting thread donates its priority to the thread which is holding the resource to avoid priority inversion.

5.4 Description

5.5 Testing Plan

1. Set up serveral threads with random priorities associated with one condition. And test whether they execute in decreasing order.
2. Set up serveral threads with random priorities associated with mutiple conditions. And test whether they execute in decreasing order. And Test by printing that whether the waiting thread properly donate its priority and whether the quit-holding thread recover its priority.

6 Solution to `Boat` Problem

6.1 Overview

Implement `Boat` class to solve the boat problem.

6.2 Correctness Constraints

- The boat can only hold one or two children or one adult.
- Each time the location of the boat changes, the number of passengers in boat cannot be zero.
- `begin` method finished with both of children and adults is on Molokai.

6.3 Declaration

- In `Boat` class, add `locationOfBoat`, `numOfChildOahu`, `numOfAdultOahu`, `numOfChildMolokai`, `numOfAdultMolokai`, `numOfChildBoat`, `boatLock`, `waitOnOahuCondition`, `waitBoardingCondition`, `waitOnMolokaiCondition`, and `message` a `Communicator`.
- In `begin`, what we need to do is for each children and adults create a thread, and we also use message to see whether we have move all people to Molokai.
- In `AdultItinerary`, the adult can move to Molokai when the boat is at Oahu and empty with child on Molokai can't be zero. Then we move this adult to Molokai and wake waiting on Molokai.

Algorithm 1 PriorityQueue

```
procedure getMaxEP()
    if !transferPriority then
        return minimum priority
    end if
    if changed then
        maxEP ← minimum priority
        for each ThreadState ts in waitingQueue do
            temp ← MAX(maxEP, ts.getEP())
        end forchanged ← false
    end if
    return maxEP
end procedure
procedure waitForAccess(KThread thread)
    Lib.assertTrue(Machine.interrupt.disable()))
    Add (ts ← getThreadState(thread)) into waitingQueue
    ts.waitForAccess(this)
end procedure
procedure acquire(KThread thread)
    Lib.assertTrue(Machine.interrupt.disable()))
    if master != null then
        Remove this from master.occupiedResource
    end if
    getThreadState(thread).acquire(this)
end procedure
procedure nextThread()
    Lib.assertTrue(Machine.interrupt.disable()))
    if master != null then
        Remove this from master.occupiedResource
    end if
    thread ← null
    if waitingQueue is not empty then
        thread ← pickNextThread()
        Remove thread from waitingQueue
        thread.acquire(this)
    end if
    return thread
end procedure
```

```
procedure pickNextThread()
    if waitingQueue is not empty then
        threadState ← first element of waitingQueue
        for each ThreadState ts in waitingQueue do
            if ts has higher priority than threadState then
                threadState ← ts
            end if
        end for
    else
        thread ← null
    end if
    return thread
end procedure
```

Algorithm 2 ThreadState

```
// once new thread enter, all the EP should change
procedure change()
    if !changed then
        changed ← true
        EP ← getEP()
        for each PriorityQueue pq in acquiredResource do
            if pq.master != null && transferPriority then
                pq.changed ← true
                pq.master.change()
            end if
        end for
    end if
end procedure
procedure getEP()
    if changed then
        EP ← priority
        for each PriorityQueue pq in occupiedQueue do
            EP ← MAX(EP, pq.getMaxEP)
        end for
    end if
    return EP
end procedure
```

```
procedure setPriority(int Priority)
    if priority != Priority then
        priority ← Priority
        change()
    end if
end procedure
procedure waitForAccess(PriorityQueue waitQueue)
    Add waitQueue into acquiredResource
    if waitQueue is in occupiedResource then
        Remove waitQueue from occupiedResource
        waitQueue.master ← null
    end if
    if waitQueue.master != null then
        // new thread wait hence EP should change
        if transferPriority then
            waitQueue.changed ← true
            waitQueue.master.change()
        end if
    end if
end procedure
procedure acquire(PriorityQueue waitQueue)
    Add waitQueue into occupiedResource
    if waitQueue is in acquiredResource then
        Remove waitQueue from acquiredResource
        waitQueue.master ← this
    end if
    change()
end procedure
```

- In `ChildItinerary`, add `location` to show the location of this child. For child move to Molokai, the condition is boat is at Oahu and empty or with one child, and also the child can't be himself. If the child is the first in the boat, then it wait for another to board. If the location of the boat is on Molokai, then it must have one child back to Oahu.

6.4 Description

```
procedure BEGIN(int adults, int children, BoatGrader b)
    bg ← b
    for each childs, create a thread of ChildItinerary
    for each adults, create a thread of AdultItinerary
    while message.listen() is not the total of children and adults do
    end while
end procedure
procedure ADULTITINERARY
    numOfAdultOahu++
    boatLock.acquire()
    while locationOfBoat is not Oahu or numOfChildBoat ¿ 0 or numOfChildOahu ¿ 1 do
        if locationOfBoat is on Oahu then
            waitOnOahuCondition.wakeAll()
        end if
        waitOnOahuCondition.sleep()
    end while
    bg.AdultRowToMolokai()
    numOfAdultOahu–, numOfAdultMolokai++
    locationOfBoat is Molokai
    message.speak(numOfAdultMolokai + numOfChildMolokai)
    waitOnMolokaiCondition.wakeAll()
    boat.release()
end procedure
```

6.5 Test

For this problem, the test is easy, just some examples with different numbers of adults or children.

```
procedure ChildItinerary
    location is Oahu
    numOfChildOahu++
    while true do
        boat.acquire()
        if location is Oahu then
            while locationOfBoat is not Oahu or numOfChildBoat is 2 or numOfChildOahu is 1 do
                if locationOfBoat is Oahu then
                    waitOnOahuCondition.wakeAll()
                end if
                waitOnOahuCondition.sleep()
            end while
            numOfChildBoat++
            if numOfChildBoat == 1 then
                waitOnOahuCondition.wakeAll()
                waitBoardingCondition.sleep()
                numOfChildOahu–
                bg.ChildRideToMolokai()
                location is Molokai
                numOfChildBoat ← 0
                numOfChildMolokai++
                message.speak(numOfAdultMolokai + numOfChildMolokai)
                waitOnMolokaiCondition.wakeAll()
                waitOnMolokaiCondition.sleep()
            else
                waitBoardingCondition.wake()
                numOfChildOahu–
                bg.ChildRowToMolokai()
                location is Molokai
                numOfChildMolokai++
                waitOnMolokaiCondition.sleep()
            end if
        else
            while locationOfBoat is not Molokai do
                waitOnMolokaiCondition.sleep()
            end while
            numOfChildMolokai–
            bg.ChildRowToOahu()
            location is Oahu
            locationOfBoat is Oahu
            numOfChildOahu++
            waitOnOahuCondition.wakeAll()
            waitOnOahuCondition.sleep()
        end if
        boat.release()
    end while
end procedure
```