Jocelyn Chan
Shivneel Chand
Michael Cheung
Duy Do
Kenny Jung
Alberto Valle

CSE 150 - Design Document 1

Task 1. KThread.join()

For a parent thread to join a child thread, these criteria must be met:

1). the parent thread is not the child thread
2). the parent thread is not joined to another child thread
3). the child thread is not finished

If a child thread is finished, the parent thread can immediately execute. The other cases are edge cases that will break the program. First, all interrupts must be disabled to prevent other processes from intervening. When permitted to join a child thread, the parent thread will join the child thread's joinQueue which we will design as a linked list of KThreads. Afterwards, the parent thread will sleep and interrupts are enabled again.

```
// Let currentThread = parent thread and thread = child thread
void KThread::join()
    ensure thread != currentThread
    disable interrupts

    if thread.already_joined or thread.status == statusFinished
        return

    insert currentThread into joinQueue
    make currentThread sleep
    restore interrupts
```

Once a thread is finished, it is set to be destroyed to free up memory space. Next, the program checks whether there are any threads in joinQueue, and if there are any, set all those threads as ready.

```
void KThread::finish()

    toBeDestroyed = currentThread
    for each Kthread (curr) in joinQueue
        curr.ready()
```

```
        currentThread.status = statusFinished

        sleep()
```

As discussed in KThread.join(), we need to initialize joinQueue as a linked list of KThreads

```
    joinQueue = new Linked List of KThreads
```

Testing

- thread B joins thread A while thread A is still executing -> thread B waits for thread A
- thread A finishes, and thread B joins thread A -> thread B immediately executes
- thread B joins thread A while thread A is still executing, then thread B tries to join thread C -> thread B waits for thread A and cannot join thread C
- thread A joins thread A -> thread A cannot join itself

Task 2. Conditions

Condition variables are primitives that enable threads to wait on a condition to occur. We implemented condition variables: sleep, wake, and wakeAll, with the use of interrupt enable and disable to provide atomicity. To keep track of waiting threads, we create a linked list of KThreads to queue in threads.

The sleep method will first disable interrupts so working threads can finish their task and after, atomically release the condition lock before it is put into the waiting queue, then it will go to sleep. The thread will reacquire the lock and enable interrupts after the sleep function returns.

```
lock = new ConditionLock
waitQueue = new Linked List of KThreads

void Condition2::sleep()
      ensure conditionLock.isHeldByCurrentThread == true

      disable interrupts
      conditionLock.release

      //add current thread to the queue
      waitQueue.add(Kthread.currentThread)

      //Make current thread go to sleep
      Kthread.sleep
      conditionLock.acquire
      enable interrupts
```

Similar to sleep(), we disable interrupts to let threads finish, and acquire the lock to allow one thread to be woken up. The wake method will check if the waitQueue is not empty, if it is, wake up a single thread by dequeuing the thread from the waiting queue and put it in the ready queue. The thread must hold the condition lock. Then, we will release the condition lock and enable interrupts.

```
void Condition2::wake()
      ensure conditionLock.isHeldByCurrentThread == true

      disable interrupts
      conditionLock.acquire

      If waitQueue is not empty
            KThread tempThread = waitQueue.pop
            tempThread.ready
```

```
        conditionLock.release
        Enable interrupts
```

The wakeAll method dequeues all threads from the waiting queue and puts them in the ready queue while under a disable interrupt. Instead of checking the size once, we loop through the waitQueue if it is not empty and call the wake method until the waiting queue is empty. After the waitQueue is empty, we enable interrupts.

```
void Condition2::wakeAll()
      ensure conditionLock.isHeldByCurrentThread == true
      disable interrupts

      While waitQueue is not empty
            call wake

      enable interrupts
```

Testing

- Use one condition lock to lock a thread. Wake it up with wake().
- Create multiple condition variables, put them to sleep, wake them up with the same lock.
- Create 4 condition variables, put 2 to sleep, wake one up, put 2 to sleep, wake all up.
- Calling sleep()/wake() on a thread multiple times on an already sleeping/woke thread.

Task 3. Alarm

In this class, the current thread waits for x amount of time in the waitUntil method and becomes active again in the timerInterrupt method. A priority queue is created to handle the waiting threads.

The timerInterrupt method puts the waiting threads on the ready queue. Its first task is to make the current thread yield in the case that a context switch is needed. This function checks the priority queue to see if enough time has passed and puts the head of the priority queue on the ready queue.

```
public Alarm()
    wait_queue = new PriorityQueue of threads

void Alarm::timerInterrupt(){
     make current thread yield
     make a head from wait_queue
     //check if enough time has passed to satisfy the waiting period
     while head <= current time
          disable interrupt

          store the head in ready queue
          remove head from wait_queue
          //update head to take new head value from wait_queue
          head = wait_queue head

          restore interrupt
```

In the waitUntil method, a new thread is created that stores the current thread and the necessary waiting time. The thread is put into a priority queue that keeps track of the waiting threads.

```
void Alarm::waitUntil(long x)
     wakeTime = current time + x
     while wakeTime > current time
          disable interrupt

          // every time we add a new thread into wait_queue, priority
goes up by 1
          create new waitingThread(current thread, current time + x)
          put waitingThread into wait_queue
          put current thread to sleep

          restore interrupt
```

Testing

- Primary test case uses waitUntil method
- Create multiple threads with varying times
- Test if each thread's wakeTime is correct
- Test if each thread's minimum wait time is correct

Task 4. Communicator

In the communicator class we are using part of the condition variables such as wake and sleep with the use of monitors. We are creating a speak method that takes a parameter of word as an int. In addition we are also creating a listen method that returns a word.

```
void Communicator::Communicator()

void Communicator::speak(int word)
      acquire mutex

      while wordToBeHeard
            call listener.wake
            call speaker.sleep

      this.word = word

      // buffer is now full
      wordToBeHeard = true

      //wake the listener and put speaker to sleep
      call listener.wake
      call speaker.sleep
      release mutex
```

Our listening function will acquire lock and while there is no buffer, the speaker is signaled to wake up and the listener will be signaled to sleep. Then the buffer will reset to empty. The speaker will wake and the mutex will be released. Returning the buffer.

```
void Communicator::listen()

      acquire mutex

      // while there is no word in buffer
      While not wordToBeHeard
            call speaker.wake
            call listener.sleep

      int wordToHear = word

      // resets buffer
      wordToBeHeard = false

      //then wake up speaker
```

```
call speaker.wake

release mutex
return wordToHear

create private boolean wordToBeHeard = false;
```

In communicator, we are creating a buffer to pass words, a mutex(lock) for condition variables and to maintain atomicity, and declaring a condition variable for listeners and speakers. The speaker and listener conditions, taking a mutex as a parameter.

```
// buffer to pass word
private int word

// lock for condition variables and to maintain atomicity
private Lock mutex = new Lock

// declare condition variable for listeners here //mutex = lock
private Condition2 listener = new Condition2(mutex)

// declare condition variable for speakers here
private Condition2 speaker = new Condition2(mutex)
```

Test Cases for Communicator:

- Buffer is empty and listener runs
- Buffer is full and speaker runs
- there are 2 listener running
- there are 2 speakers running

Task 5. Boat

First in begin() we create all the children and adults. Each person is a thread that runs their respective itinerary(). All threads will share global variables and locks that will help synchronize adults and children to move from Oahu to Molokai. The pilot and passenger locks will prevent threads from using the boat all at once while variables such as childrenOahuPop and adultOahuPop will be updated as each thread travels in the boat. Finally, the boatLocation will prevent threads from using the boat when it is not on their island.

```
void Boat::begin(adults, children, boatGrader):

      bg = boatGrader

      int childrenOahuPop = 0
      int adultOahuPop = 0
      bool boatLocation = 0
      bool boatPilot = 0

      boatSeats = 3

      populationLock = new Lock
      pilotLock = new Lock
      passengerLock = new Lock
      mutex = new Lock

      Condition2 children = new Condition2(mutex)
      Condition2 adult = new Condition2(mutex)

      Alarm clock

      runChildren = new Runnable { function run{ call ChildItinerary }}
      runAdult = new Runnable { function run{ call AdultItinerary }}

      disable interrupts

      for child in children
            tempThread = new Kthread(run ChildItinerary)
            tempThread.fork

      for adult in adults
            tempThread = new Kthread(run AdultItinerary)
            tempThread.fork

      enable interrupts
```

First the threads update the population of Oahu since they can all see who is on the island with them. Adults check for children on their island. If there are more than one children, the adult will yield to the child. Once all children have been moved, and there is only one child left on Oahu (the child who rowed the boat back) then the adults will begin moving. One adult will move to Molokai and lock themselves with a conditionVariable and wake one child with the conditionVariable to row the boat back.

```
void Boat::AdultItinerary()
      acquire populationLock
      adultOahuPop++
      oahuPop++
      release populationLock

      clock.waitUntil(100)
      yield thread

      while oahuPop != 0
            if childrenPopOahu == 1 and boatLocation == 0
            // acquire both locks since adults travel alone
                  acquire pilotLock
                  acquire passengerLock
                  call bg.AdultRowToMolokai
                  island = 1
                  acquire populationLock
                  adultOahuPop--
                  oahuPop--
                  populationLock.release
                  //wake up child to take boat back
                  call child.wake
                  call adult.sleep
            else
                  yield thread
```

If there are more than one child on Oahu, one child will become the pilot and then join() the thread with another child who will be the passenger. After arriving at Molokai, one child will ride back with the boat and the other child will be locked with a conditionVariable. Once the last child has rowed the boat back, everyone will be locked with a condition variable and the program will terminate.

```
void Boat::ChildItinerary()
      Bool island = 0;
      acquire populationLock
```

```
childrenOahuPop++
oahuPop++
release populationLock

clock.waitUntil(100)
yield thread

if island == boatLocation
//We are on island 0 and childOahuPop > 1 and the boat is
//currently empty
      if island == 0 and childOahuPop > 1
            if boatPilot == 0
                  //Ride to island 2 with another child
                  acquire pilotLock
                  call bg.ChildRowToMolakai

                  /*We need to start the other child thread so
                  that they can run bg.ChildRideToMolakai us on
                  the boat */

                  passenger.join
                  childOahuPop--
                  island = 1
                  release pilotLock
            else
                  acquire passengerLock
                  call bg.ChildRideToMolakai
                  childOahuPop --
                  island = 1
                  release passengerLock
                  call child.sleep
      }

      //We island = 1 && childOahuPop == 0 && adultOahuPop > 0
      //We need to bring boat back to pick up more people
      if island == 1 and (adultOahuPop != 0 or childOahuPop) != 0
            acquire pilotLock
            acquire passengerLock
            bg.ChildRowToOahu
            acquire populationLock
            ChildOahuPop++
            island = 0
            release populationLock
```

```
            release pilotLock
            release passengerLock
      else
            call child.sleep
```

Testing

- Description about how the testing is going to be planned for this project
- Description of the different cases tested(edge cases, etc)
    - Two children
    - Two Children One Adult
    - Multiple Children, Multiple Adult
    - Multiple Children, One Adult
    - Two Children, Multiple Adult