

Randy Huang	cs162-be
Ed Liao	cs162-bp
Karl Ni	cs162-em
Alvin AuYoung	cs162-if
Shelley Chen	cs162-ii

Design for Project 1

Part 1: KThread.join()

KThread will be joined successfully after the first call to it, but return immediately for all subsequent calls to it. Because of this, a KThread needs to know if `join()` has been called on it. Add a private member to KThread,

```
private boolean joined;          //initially false. set true on first call.
```

When called, `join()` needs to block until the thread signals that it has finished execution. We use a Lock to implement this wait/signal form of communication. This is also convenient since both joins and locks are supposed to implement priority transfer behavior (Part V) so we would only have to implement this priority transfer once.

Add private members to KThread: locks to signify thread termination.

```
Lock threadHasFinished; //initially locked by owning thread during creation
Lock joinHasFinished; //initially free
```

The thread calling another thread's `join()` will attempt to acquire the `threadHasFinished` Lock. This will not be released until the joined thread enters its `finish()` procedure. At which time it will release the `threadHasFinished` Lock and wait for the thread calling `join` to finish before setting itself up for destruction. The `joinHasFinished` Lock is used to coordinate this.

```
public void join() {
    ...
    disable interrupts
    if this.joined is true
        enable interrupts
        return
    else
        set this.joined to true;
        acquire this.joinHasFinished Lock
        restore interrupts
        attempt to acquire threadHasFinished Lock
        release joinHasFinished Lock
```

To the `finish` procedure, add this signal of releasing the lock.

```
if currentThread.joined is true
    release threadHasFinished Lock;
    attempt to acquire joinHasFinished Lock;
    ...continue executing rest of finish() procedure..
```

In addition, parts of `join()` were made atomic. Interrupts are disabled so that only one thread can join another.

Part II: Condition Variables

`Condition2` class must have a place to store threads waiting for a condition (sleeping). Add a private data structure to hold this:

```
LinkedList waitQueue;
```

The constructor will initialize this `waitQueue`.

The `sleep()` method needs to release the lock and put the thread onto the `waitQueue` so that it is not busy-waiting.

```
public void sleep() {  
    ...  
        disable interrupts  
        release conditionLock  
        add the thread onto a waitQueue.  
        make it go to sleep by calling the thread's sleep method.  
        acquire conditionLock  
        restore interrupts  
}
```

The `wake` method “wakes up” a sleeping thread in its `waitQueue`. For ease of implementation and efficiency, it will wake up the thread at the front of the `waitQueue`. It will accomplish the waking-up of the thread by calling its `KThread.ready()` method, which will put it back onto the ready Queue:

```
public void wake() {  
    ...  
        disable interrupts  
        if waitQueue isn't empty  
            put thread on ready queue [call KThread.ready()]  
        restore interrupts  
}
```

`wakeAll()` is similar to `wake`, except wake up all of the threads on the `waitQueue` by calling `wake` until the `waitQueue` is empty:

```
public void wakeAll() {  
    ...  
        while waitQueue isn't empty  
            call wake  
}
```

Part III: Alarm Class

The `waitUntil(long x)` procedure needs a data structure to keep track of the threads that call it. This data structure will be a private linked list type of structure in the `Alarm` class. A linked list was chosen since the size of the structure is unknown. Add field to `Alarm` class:

```
private LinkedListType waitingToWake;
```

When `waitUntil` adds the thread to the `waitingToWake` structure, it needs to store the thread as well as the time the thread should be woken up. Each node of the `LinkedList` consists of the thread and its wake-up time.

```
public void waitUntil (long x) {
    wakeTime = Machine.timer().getTime() + x
    newNode = createNode(currentThread, wakeTime)
    disable Interrupts
    Insert newNode WaitingToWake in LinkedList by increasing wakeTime
    set currentThread to sleep
    restore Interrupts
}
```

The `timerInterrupt()` procedure should check the current time and see if any threads on the `waitingToWake` list need to be woken up. It will begin at the beginning of the list and wake up all the threads until it finds one that does not need to be woken up. Therefore, `waitingToWake` should be sorted by increasing `wakeTime`.

```
public void timerInterrupt() {
    currentThread yields
    while (waitingToWake not empty && currentTime >= nextThread.wakeTime)
        move nextThread from waitingToWake onto ready queue
}
```

Part IV: Communicator (using condition variables)

The `Communicator` uses a shared buffer to provide shared data. Only one item should ever be on this buffer, so we made this an `Object` type.

```
Object messageQueue;
```

`Communicator` needs a lock to provide mutual exclusion to the `messageQueue`. Add a private member to `Communicator`:

```
Lock CommLock;
```

The `Listeners` need a condition variable to allow it to wait in its critical section for a `Speaker` to produce a message. Also, the `Speakers` need a condition variable to wait for available `Listeners`. Add 2 private condition variable objects for both the `Listener` and `Speaker`:

```
Condition ListenCondition;
Condition SpeakCondition;
```

A listener can either be waiting for a speaker or vice versa. If the listener wakes up a sleeping speaker, he needs to wait for the speaker to place the message on the `messageQueue` before attempting to grab it. An additional Condition variable will be used to synchronize the actual message passing between Listener and Speaker.

```
Condition messageSent;
```

Also, add counters that indicate if there are any Listeners or Speakers waiting:

```
int numberOfListeners, numberOfSpeakers; // initially 0
```

Speaker acquires the `commLock` to get access to the shared resources. It then attempts to put its message on the static `messageQueue` if there are available listeners, if not, it will sleep until awoken by a listener. This is also true for a waiting listener – It will sleep until awoken by a speaker.

Once it has awoken, it will check to make sure that there are still speakers/listeners available. If it is the Speaker who has woken up first, he will place the message on the queue and signal the Listener that he has done so. If it is the Listener who has awoken first, he will wait for the Speaker to signal that he has placed the message on the queue.

Then, it will decrement the number of listeners. This is done instead of decrementing the `numberOfSpeakers` because if you decrement number of speakers, the listener waking up won't know that there is still a speaker there. You can safely decrement the number of listeners, since you just woke one up to listen to your message. After this, you release the `commLock`.

```
public void speak()  
    acquire commLock.  
    increment numberOfSpeakers  
    while (numberOfListeners < 1)  
        SpeakCondition sleeps  
    wake ListenCondition //wake up a listener  
    put message on the messageQueue  
    wake messageSent  
    decrement numberOfListeners  
    release commLock
```

The listener is implemented similarly. It acquires the `commLock` to guard its access to the shared resources, and then it increments the number of Listeners. While there are no speakers, it will put itself to sleep. It can be awoken by a speaker...the while loop here may be unnecessary since at this point, it will have the `commLock` and no other threads could have consumed the message from the speaker that woke it up. It will then consume the message, decrease the `numberOfSpeakers` (since the Speaker decremented the `numOfListeners`)

```
public int listen()  
    acquire commLock.  
    increment numberOfListeners  
    while (numberOfSpeakers < 1)  
        ListenCondition sleeps
```

```

wake SpeakCondition          //wake up a speaker.
messageSent sleeps
item = message on the messageQueue
decrement numberOfSpeakers
release commLock
return item

```

V. Priority Scheduler Class

In each instance of a PriorityQueue, the Queue is guarding access to a particular resource. Since only two types of structures in this implementation of Priority Scheduling are to implement priority transfer, we will only be concerned with these two structures: The ReadyQueue, and the waitQueue of the Locked and Joined threads.

Add private data members in PriorityQueue class so that each Queue has access to the identity and priority of the thread that currently “owns” its resource:

```
private ThreadState Owner
```

The waiting queues will be implemented as a linked list since the size of the list can vary greatly from time to time. We will investigate the possibility of using heaps or tree set data structures later on to increase retrieval and insertion speeds. The logic shouldn’t be affected by the data structure used.

```
private LinkedList waitQueue
```

To pick the next thread to remove from the Queue, the thread with the highest effective priority is chosen. The work of finding this thread is delegated to `pickNextThread()`. There are a few subtle points here though.

First, the picked thread must be removed from the queue since it is no longer waiting. Second, if the thread is giving up its resource (i.e. releasing its lock) then it must also reset its effective priority to its original setting, since it no longer needs to borrow this “higher” priority. Upon leaving the queue, it will reset its effective priority. However, if this is a thread just leaving its lock queue (i.e. receiving its lock for the first time) or thread that holds a lock that is leaving the ready queue, it cannot lose its priority, since in both cases, it may still be blocking a higher priority thread later on. But this isn’t a problem since the effective priority will be updated as it enters a queue again. The exception would be a joined function, which always needs the borrowed priority

```

public KThread nextThread() {
    tempThreadState receives pickNextThread()
    tempThreadState is removed from front of waitQueue
    set owner of Queue to ThreadState
    if Temp isn't joined
        reset effective priority of tempThreadState->thread
    return tempThreadState->thread
}

```

Return the next thread, which should be the one of highest effective priority.

```
protected ThreadState pickNextThread() {
    // can probably add a flag here indicating if it has been changed since
    //we last sorted so we don't have to sort needlessly
    sort the waitQueue if necessary
    return ThreadState from front of waitQueue
}
```

In ThreadState:

ThreadState needs to add fields to bookkeep effective priority and the constructor now needs to set the effective priority of the thread too.

```
protected int effectivePriority;

public ThreadState(KThread thread) {
    ..
    setPriority(priorityDefault);
}
```

Implement accessor and mutator functions for the priorities of the ThreadState objects:

```
public int getEffectivePriority() {
    return effectivePriority;
}

public void setEffectivePriority(int priority) {
    if (this.effectivePriority == priority)
        return;
    this.effectivePriority = priority;
}
```

When a new item is put onto the waitQueue, there could be a need to do a priority transfer. If the waitQueue supports priority transfer (it has a member field indicating this), it will compare the effective priority of the inserted item to the effective priority of the item guarding the queue.

```
public void waitForAccess(PriorityQueue waitQueue) {
    if waitQueue.transferPriority is true
        if (waitQueue.owner's effective priority > current Thread's)
            set waitQueue.owner's effect priority to current thread's.
    insert currentThreadState into waitQueue
}
```

However, the waitQueue's owner could also be in the queue of another resource. (i.e. the owner of a lock is in a join queue) So if we did update the owner of a waitQueue, we will check to see if he is part of another waitQueue, and if so, reinsert him into that waitQueue (thereby updating the owner of that list). This should be a recursive call.

```
[still part of waitForAccess(PriorityQueue waitQueue) ]
    if waitQueue.owner was just updated
```

```

        reinsert waitQueue.owner into waitQueue.owner.myQueue
    }
}

```

We have to make the reinsert function a recursive one such that if a newly updated waitQueue_1 owner is part of another, waitQueue_2, then the owner of waitQueue_2 will also be updated and so on.

```

public void reInsert(PriorityQueue waitQueue, ThreadState updatedNode) {
    if (waitQueue.owner.myQueue is NULL) OR
        (waitQueue.transferPriority is FALSE)
        return
    else
        if (waitQueue.owner's effective priority > current Thread's)
            set waitQueue.owner's effect priority to current thread's.
            reinsert waitQueue.owner into waitQueue.owner.myQueue
        return
}

```

Acquire is called when the associated thread has acquired access to whatever is guarded by waitQueue. So we can set the owner of the queue to be the thread (state) object calling acquire.

```

public void acquire(PriorityQueue waitQueue) {
    set this ( ThreadState) to waitQueue.owner
    put ThreadState->thread into ready queue
}

```

VI. Elevator Controller

The flow of information works like this. The (possibly multiple) riders post events to the Elevator Manager's event queue. The elevator controller removes items from this queue and places them onto the queues of the individual elevators. We will need to implement an elevator class as a thread with holding queues to store its pending events.

```

public elevatorController implements elevatorControllerInterface {

    public void initialize(ElevatorControls control) {
        this.controls = controls;
        eventWait= new Semaphore(0);
        control.setInterruptHandler(new Runnable() {
            public void run() {interrupt();}
        });
    }
}

```

The order of events matters if the elevator is to successfully service all rider requests. An event sequence always starts with a button push from the outside. We insert this button push event to a particular elevator's button queue. This elevator will now sleep until it gets an arrival signal. After which it will open its doors and close its doors. Any further events for this rider will be specific to the elevator, and the controller will send the rider's request to the proper elevator, after which it will bring the rider to his destination.

```

public void run() {
    fork off a thread for all elevators and
    create ElevatorSemaphores for all elevators, initially 0
    Loop through {
        event = getNextEvent()
        if event is AllRidersArrived
            broadcast event to all elevators and join them
            controls.finish()
        else if event is outside button pushed
            cyclically assign event to elevator's button queue
        else if event is not an arrival event
            put event in button queue of proper elevator
        else
            wake arriving elevator with corresponding semaphore
    }
}

private class elevator extends Runnable {

    public queue buttonQueue;
    public void run() {

        while (true) {
            e = buttonQueue.getNextEvent()
            if event is AllRidersArrived
                return;
            Move to given floor
            Sleep by using ElevatorSemaphore.P()
            Sets direction arrow to rider's destination
            Open doors
            waitUntil(time to hold door open)
            close door
        }
    }

    private methods interrupt() {
        eventWait.V();
    }

    private method getNextEvent() {
        ElevatorEvent event;
        while(true) {
            if((event=controls.getNextEvent())!=null)
                break;
            eventWait.P();
        }
        return event;
    }

    private fields controls: ElevatorControls, eventWait: Semaphore
}

```

Each elevator gets events off the buttonQueue, since if there has been no button pressed, the elevator will have no tasks to perform. Once the buttonQueue's event has been gotten, the elevator performs the move, sleeping until it has been notified that it has arrived at the indicated

floor. Once notified, it opens its doors, waits, closes the door, and re-iterates through the while loop.

VII. Elevator Riders

Add private data members:

Need a structure to represent stops that rider will take:

integer array of myStops

Need a reference to a RiderControls Object to use Elevator controls.

RiderControls myControls = null

Use a Semaphore to receive events from Elevator.

Semaphore eventWait

In the initialize method, the fields of the rider need to be initialized.

```
public void initialize(RiderControls controls, int[] stops)
    set myStops to stops parameter
    set myControls to controls parameter
    eventWait = new Semaphore, initially 0
    set myControls.setInterruptHandler to interrupt() method.
```

The run() method, generalizes the procedure the rider will follow for any generic set of Stops. For each stop, there is a series of actions the rider must take and corresponding actions he must wait for before he can proceed. In this sequence, the first action is to press the correct elevator button. This is figured out by comparing your current floor to the floor you intend to visit. The getNextEvent() method in RiderControls interface is used to signal the completion of an action and request a corresponding event. When it arrives, you take the next action, which is to wait for your elevator, and then step inside and push the correct button. After another call to getNextEvent() another event will arrive, the doors should close. After this, you request another event, and the doors should open, after which you step out onto the floor. This process is repeated for each floor in the Stops set until all floors have been visited.

```
public void run()
    set currentFloor to myControls.getFloor()
    initialize counter to 0
while (counter < length of myStops array)
    check floor number
    decide if you need to push up button or down button
    push correct button
    get next event
    if elevator direction arrow == button I pushed
        step inside elevator and push button
    else
        repeat above step
    get next event (doors close)
    get next event (doors open)
```

check floor and step outside elevator