Ralphilou Tatoy
Daniel A. Maciel Manzo
Nathan Vasquez
Saishrithik Reddy Sareddy
Yash Sharma

CSE 150 PROJECT 1

October 9, 2021

Task 1: Thread Join Implementation

Pseudo code:

```
public void join()
{
    Lock = new Semaphore
    Lock .acquire
    if(thread B  ==  thread A) {return and Lock.release}
    Else if(thread A status = = statusFinished){Lock.release then return to thread B}
    Else{append thread B to waiting queue, threadB.sleep()}
    Lock.release
}
```

Explanation:

The primary purpose of this function is to make the thread B that called join to wait until thread A finishes. Thread B will either be put in a queue or not. When thread A is finished and calls the finish function, it will wake any thread in the queue and take it out of the queue. Thread B wakes up and proceeds with the join function.

We will need locks and interrupts manipulation to make sure that our sections of data management manipulations are uninterrupted. Thus we can acquire lock functionality by using a semaphore or a monitor. Also can be used to make sure that join functions do not get called by thread B or any other thread again. This code provides this functionality:

```
Lock = new Semaphore
Lock.acquire
```

```
....

Lock.release
```

If thread B and supposed new thread A are the same, it will disregard to give an error to not do that.

```
if(thread B  ==  thread A) {disregard and Lock.release}
```

Supposed thread A is finished, it will lift the lock and return to thread B where the join was initially called. This would make the code efficient instead of going through each comparison and making thread B wait.

```
Else if(thread A status = = statusFinished){Lock.release then return to thread B}
```

The final condition is that if they are not the same and thread A is operating, thread B will be put into the waiting queue until thread A is finished. After that, we will need to put thread B to sleep. When thread A is finished, the finish function will be called and that is where thread B will be woken up back to the join function and taken out of the queue. If this happens, the join function is operating correctly.

```
Else{append thread B to waiting queue,threadB.sleep()}
```

Correctness:

Join function must be called only once. Another thread does not have to call this function. This is a problem because if we allow thread B to call more than one thread, memory management would be disorganized. For example, if thread B calls to join on thread A and thread C also calls join on A, C and B may share data. The intended purpose of B might change depending on what C does with the shared data. In coding terms, we must have an order of line by line of execution. The queue system would be a problem on which thread will be taken out.

We will want to make sure that thread A and thread B are not the same. This is a problem since if they are the same thread, it will create a loop. The thread will keep calling itself, therefore, calling the join function, and this process repeats.

If all of those are met, then thread B will need to be put in queue to be woken up when thread A calls the finish function.

Testing:

We are going to use server testing. Testing the join function on a thread before the thread is running. The other is testing a join on a thread after the thread is finished.

For our implementation, we can test if after thread A is finished and calls the finish function, that thread B is woken up. We will also use the provided test “joinTest1 and joinTest2 in the self testing files.

Task 2: Condition to Condition2 Conversion

Pseudo code:

```
public void sleep()
{
    Disable machine interrupt
    Put thread in queue
    Put thread to sleep
    Enable machine interrupt
}
public void wake( )
{
    Disable machine interrupt
    if(queue is not empty){
        Get the next thread in queue
        Put thread in readyQUEUE
        Enable machine interrupt
    }
    Else
    {
        Enable machine interrupt
        Return error or prints no thread in queue
    }
}
public void wakeALL( )
{
    if(queue is empty)
    {
        Return error or print no thread in queue.
    }
    Disable machine interrupt
    while(nextTHREAD in queue != null or while queue is not empty)
    {
        Get next thread in queue
        Put thread in readyQUEUE
    }
    Enable machine interrupt
}
```

Explanation:

The code in condition2 class must have the same result as condition class but replace the use of semaphore with interrupt manipulation. The sleep, wake, and wakeALL function must all have the same output in either condition or condition2.

For the Sleep function, instead of using semaphore implementation, we will need to use a machine interrupt to replace it. Using this method, we can make atomic operations that put a thread in a sleep/waiting queue and put it to sleep. After putting it to sleep and in the sleep queue, enable machine interrupt so other threads can gain operation access.

For the Wake function, we use the machine interrupt to secure atomic operation that adds one thread in the ready queue. The if statement checks if the sleep queue has a thread in it. If it has a thread, it will take out a thread and put it in the ready queue. If it doesn't have a thread, it communicates that it doesn't. Enable machine interrupt so other threads can gain operation access.

For the wakeALL function, use the machine interrupt function to secure atomic operation to add all the threads in the sleep queue to the ready queue. Instead of an if statement, we will use a while statement since we have to repeat the operation over and over again until the waiting queue is empty. The condition in the while loop is that it repeats until the queue is empty. Inside, it will get a thread from the queue and put it in the ready queue. Enable machine interrupt so other threads can gain operation access.

Correctness:

The use of the interrupt is key to make sure interruption during memory management is protected by other threads such as putting a thread in queues. Make sure that after the function is done or able to be interrupted that other threads can gain access to the implemented functions. For both wake and wakeALL functions, the queue must not be empty otherwise provide a means to indicate that it is empty.

Testing:

We are going to use server testing. Testing the condition variables using a few threads. Testing the condition variables using many threads

For our implementation, we can test if each function works correctly. Create a test that uses multiple functions in one thread. We are going to use the provided test as well.

Task 3: Alarm Implementation

Pseudo code:

```
public class Alarm
{
    timerInterrupt()
    {
        Enable machine interrupt
        while( queue is not empty and first wake time in queue is <= current time )
        {
            Get thread from next pair in queue
            Ready the thread
        }
        Disable machine interrupt
    }
    waitUntil( long x )
    {
        Calculate wake time
        Enable machine interrupt
        Put (thread, wake time) pair in priority queue ordered by wake time
        Put thread to sleep
        Disable machine interrupt
    }
    WaitQueue : PriorityQueue of pairs with (Thread and Time). Order the priorityqueue with the time
}
```

Explanation:

timerInterrupt - Initially calculate the wake time using the current time and the received variable x. Uses the machine interrupt to secure atomic operation that adds one thread in the wait queue. Then put the thread to sleep. Enable machine interrupt so other threads can gain operation access.

waitUntil - Again, uses the machine interrupt function to secure atomic operation. Use a while loop so we can work our way checking all valid threads. Check to make sure the queue is not empty and that the wake time of the first value in the queue is less than or equal to the current time. Inside, it will get a thread from the queue and ready the thread. Enable machine interrupt so other threads can gain operation access.

WaitQueue - We create a priority queue of pair objects. Each pair contains a thread and a time. We order the priority queue by the time.

Correctness:

The use of the interrupt is key to make sure interruption during memory management is protected by other threads such as putting a thread in queues. Make sure that after the function is done or able to be interrupted that other threads can gain access to the implemented functions. We make use of this in both functions as interaction with the queue is required. For the timerinterupt the queue must not be empty to ready threads and the threads must have completed the required sleep period.

Testing:

We are going to use server testing. Testing the condition variables using a few threads. Testing the condition variables using many threads

For our implementation, we can test if each function works correctly. We can use the given self tests in the catcourse files as well.

Task 4: Communicator Implementation

Pseudo code:

```
private Lock lock = new Lock();

private int word;

private wordToBeHeard = false;

private int AL, WL, AS, WS; // active/waiting Listener/Speaker

private condition okToSpeak, okToListen();

public void speak(int word) {

    lock.acquire();
```

```
        while (!wordToBeHeard or listener queue is empty) {
                Increment waiting speakers
                Go to sleep with the lock
                Decrement waiting speakers
        }
        Increment active speakers
        this.word = word;
        // notes that the buffer is full
        wordToBeHeard = true;
        Wake listener
        Transfer word to listener
        Decrement active speakers
        lock.release();
}
public int listen() {
        lock.acquire();
        while(!wordToBeHeard or speaker queue is empty) {
                Increment waiting listeners
                Go to sleep with the lock
                Decrement waiting listeners
        }
        increment active listeners
        Wake speaker
        int wordToHear = word;
        wordToBeHeard = false;
        //TODO
        Decrement active listeners
        lock.release();
        return wordToHear;
}
```

Explanation:

This code functions similarly to the solution for the readers/writers, only instead of readers and writers we have speakers and listeners. For a speaker, If there is a word to be transferred (wordToBeheard) and there is a listener in the listener queue then we wake the listener. If not then the speaker will go to sleep and wait to be woken up when a listener comes along. A listener will also check for a speaker queue and if there is a word to be heard, if there is then wake the speaker but if not then the listener will go to sleep and wait to be woken by a speaker to come along.

Correctness:

In order for this implementation to be correct then there should be the ability to have multiple speakers/listeners waiting, but there should never be a speaker, listener pair waiting in a queue. By checking whether or not there are waiting speakers/listeners before proceeding, and if there is a thread waiting then it will be woken up and then proceed with the transfer. This ensures that threads only enter the queue IF the other queue is empty (meaning a pair cannot be made). If the other queue is not empty then it means that a pair can be made and it is made.

Testing:

We are going to use server testing. Testing the condition variables using a few threads. Testing the condition variables using many threads

For our implementation, we can test if each function works correctly. We can use the given self tests in the catcourse files as well.

It might also be a good idea to do a few permutations of speakers/listeners and trace through the code by hand on paper during one of the lab sessions.

Task 5: Boat Implementation

Pseudo code:

```
public class Boat()
{
        Initialize Global variables
        public static void begin()
        {
                Instantiate global variables
                Create Sample thread
                Create Child thread
```

```
    Create Adult thread
    Spawn all adult threads
    Spawn all child threads
    While problem is not done:
        Solution calls are made to the Grader
}
static void AdultItinerary()
{
    Set lock.acquire
    While adults are still on Oahu:
        Code here
    Row adult to Molokai
    Set the boat not on Oahu
}
static void ChildItinerary()
{
    Set lock.acquire
    While there are still adults and children on Oahu:
        While boat is not on Oahu:
            Wake child up and sets to Rower
        If no child on boat and boat on Oahu:
            Add child to boat
            Sets child to passenger and waits for Rower
        Else if 1 child on boat and boat on Oahu
            Add child to boat
            Sets child to Rower
            Boat leaves to Molokai
        Else
```

```
                    Boat leaves to Oahu

                    Boat arrives at Oahu and becomes vacant

    }

};
```

Explanation:

The purpose of this implementation is to combine the previous synchronization devices to solve the Boat problem. The problem is to transport all adults and kids from Oahu to Molokai and the restrictions are that the Boat can only hold either 2 kids or 1 adult and there must always be 1 person, either kid or adult, piloting the boat.

selfTest – Initializes problem Grader and tests out variations of the boat problem containing different amounts of children and adults.

begin – Start of solving the boat problem, it stores Grader that was inputted into the function into a local variable while instantiating global variables. Creates Child and Adult threads that will be used to set up the problem based on the number of Childs and Adults inputted. All Adults' threads will be asleep when first spawned while all Child threads are running after being spawned. The function goes into a loop holding the main thread while solutions calls are made to the Grader. The loop will end once all the children and adults are on Molokai.

AdultItinerary – Thread for adults that can only operate atomically. Uses a while loop to send adults one by one to Molokai. Checks to see if there are still adults not asleep on Oahu if there is a loop that keeps going. Adults will then row to Molokai and wake one child to bring the boat back to Oahu for another child or adult to use it. Loop will keep going while the problem is not done and there are adults that need to go to Molokai.

ChildItinerary – Thread for children that can only operate atomically. Uses a while loop to send children in pairs to Molokai and back alone to Oahu. While loop that checks if there are still children asleep on Oahu and the problem is not done yet. Within the loop, checks to see if the boat is not on Oahu, if not the loop will check if 1 child or no child is on the boat and set the role of the child when based on the number already on the boat. Once the boat is full, it will be sent to Molokai, else if the boat is on Molokai, then one child will be piloting back to Oahu.

SampleItinerary – Test thread that will test each function of Adult rowing to Molokai, Child riding and rowing to Molokai.

Correctness:

To keep this implementation correct, there are few constraints to keep track of to make sure the code is working properly. One constraint is that there can be only 2 children or 1 adult at a time and to ensure this we can do checks to make sure that this limit is never broken. Another constraint is that the boat always needs a pilot to take it back to Oahu and this can be done by using a child as a pilot to take the boat back to Oahu. Final constraint is to make sure the flow of information between individual threads are controlled such that a thread in Molokai cannot know how many child and adult threads are on Oahu but the thread can remember how many child and adult threads it saw before it left Oahu. To do this we can have a cache that can store the information and possibly update based on what threads it saw arrive or left.

Testing:

To test our implementation of the Boat problem, we have designed test case scenarios that will be run using our implementation. The results of our boat synchronization implementation will be cross checked with previously stored correct answers. A different series of tests are designed to find other ways our implementation could have synchronization issues, by trying different combinations of randomly chosen # of adults and children in the different islands. Multiple sets of each random variation will be tested to check for unintended results variations.