CSE 150 - Operating Systems
Documentation #1

Spring 2018 - Lab 04 - Group 1

Avery Berchek, Aleksandr Brodskiy, David Cabral, Christopher DeSoto, Adiam G-Egzabher, Nanditha Embar, Christian Vernikoff

February 27, 2018

Outline

Documentation

Task 1

The implementation of the JOIN function is of significant relevance in building a multi−thread system due to the fact that the joining of multiple threads relies on the preoccupation of the system's resources on the execution of a currently processing thread. In essence a full and complete functional implementation of the JOIN function would mitigate, not ameliorate, the problematic occurence of priority inversion. The logic required for the sucessfully joining of two or more threads is defined in the KTHREAD.JAVA class which makes use of system interrupts as follows:

BOOLEAN MACHINE_START_STATUS = MACHINE.INTERRUPT().DISABLED();
MACHINE.INTERRUPT().DISABLE();

In this manner the current state of the machine is stored before any actions are performed. As such, threads would not be waiting endlessly even if there were to be an unexpected abnormal termination. This functionality is actual achieved through an algorithmic set−up of the JOIN function which is implemented upon conditional checks, IF−statements, and an iterative procedure, WHILE−loop, in order to parse through a priority queue. This logic allows the NachOS machine to terminate a thread upon the invocation of the JOIN functionality. For instance, given two threads; thread_1 & thread_2, the execution of the latter thread would be paused until execution of the former is fully complete, respectively. In this manner, the logic behind the implementation of the JOIN function is demonstrated in the following pseudo-code:

```
join:
        bool status = get interrupt status;
        disable interrupts();
        if (currentThread != this.status && this.status != fininshed)
        {
                joinQueue.waitForAccess(currentThread);
        //Private Static ThreadQueue <>joinQueue = New Queue (provides protection)
                currentThread.sleep();

        }
        restorInterruptState(status);
finish: code here before (modified not implemented)
        KThread waiter;
        while ((waiter = joinQueue.nextThread()) != null)
        {
                waiter.ready();
        }
```

Task 2

The actualization of a direct automation functionality for condition variables was implemented in the CONDITION.JAVA and CONDITION2.JAVA classes. Within this implementation the WAKEALL function, although allows the saving capability in regards to competing resources, is mainly used for synchronization of them. In essence, this design is analogous to the *Token Ring* network protocol, in which the *condition* mirros the utilization of the *token* itself and therefore enables an interface to wake other participants in order to provide a method for their respective synchronization. In this manner establishing a systematic hierarchical automaticity. The pseudo-code demonstrates the functions invoked to achieve this implementation:

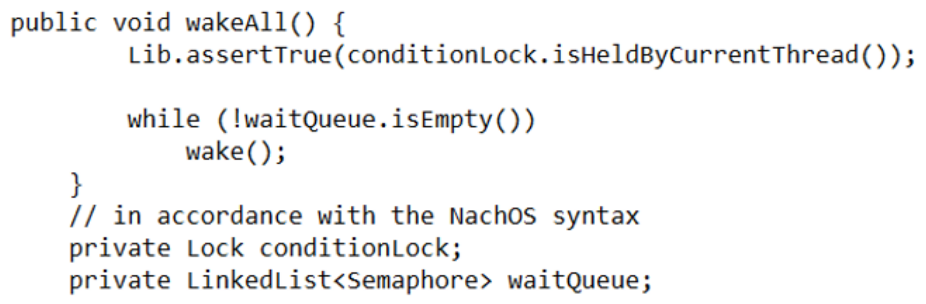
```
1:
        c.sleep();
2:
        c.wakeAll();
```

In this manner the WAKE and WAKEALLSfunctions were composed as follows:

```
void wake()
{
        Lib.assertTrue(conditionLock.isHeldByCurrentThread());

        if (!waitQueue.isEmpty())
            ((Semaphore) waitQueue.removeFirst()).V();
    }
```

Within this implementation it is observable that the priority queue was controlled with a *while*−loop condition to check for emptiness of the waitQueue of semaphores.



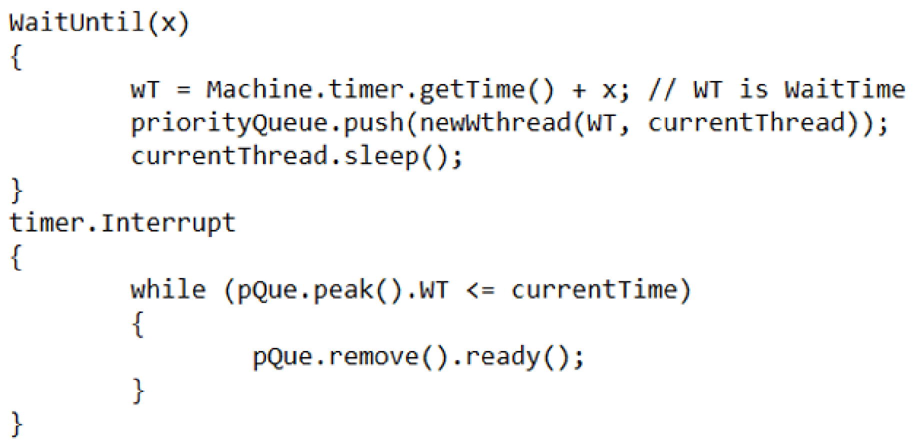
```
public void wakeAll() {
        Lib.assertTrue(conditionLock.isHeldByCurrentThread());

        while (!waitQueue.isEmpty())
            wake();
    }
    // in accordance with the NachOS syntax
    private Lock conditionLock;
    private LinkedList<Semaphore> waitQueue;
```

Although the direct application of *Semaphores* was avoided in the aforementioned design, it is paramount to this documentation and more importantly, to the concerns addressed in the **Design Decisions** section, to emphasize the fact that the functions P and V from the Semaphore class were mapped to the lock.java class in the interest of utilizing the acquire and release functions respectively.

Task III

The consummation of the alarm.java class was achieved by including a waitUntil functionality to the previously established multi−threading system in order to suspend the execution of a thread prior to the completion of a thread released from the priority queue. The value of such a function essentially resides in the dependence real−time thread operations have on hierarchical administration. The implementation involved utilization of the built−in java priority queue class under the pretense that the timer executes only once every 500 clock ticks. In this manner, the current system time was made retrievable through the machine.timer and .gettime function calls repectively. The following pseudo−code demonstrates the algorithmic structure of the logic.



```
WaitUntil(x)
{
        wT = Machine.timer.getTime() + x; // WT is WaitTime
        priorityQueue.push(newWthread(WT, currentThread));
        currentThread.sleep();
}
timer.Interrupt
{
        while (pQue.peak().WT <= currentTime)
        {
                pQue.remove().ready();
        }
}
```

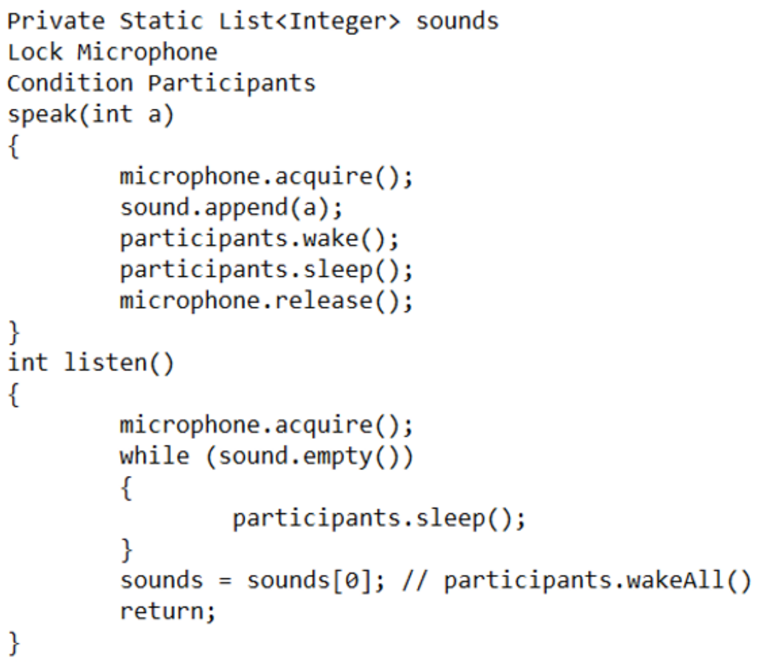
As observable in the pseudo−code, the time elapsed can be calculated by adding the function argument denoted by the variable x to Machine.timer.getTime():

$$\textsc{waittime} = \textsc{Machine.timer.getTime}() + \textsc{x};$$

Where x represents the number of ticks. Consequently, resolution can be achieved by applying a sorted priority queue based on time and storing the thread as well as the corresponding time in the queue. Basically, this resolution allows for any number of threads to invoke the waituntil function and be suspended from execution at any particular time.

Task IV

In order to fully and completely implement the synchronous sending and receiving of one word messages the speak and listen functions were designed with the intent of ensuring that the proper speaker returns an acknowledgment or a desired reciprocated action per consumed word. For instance, given two communicators; T_1 and T_2, if T_1 speaks 1 and T_2 speaks 2 then T_1 should return when 1 is consumed by the listeners and T_2 should return when 2 is consumed. This intention is demonstrated in the following pseudo−code:



```
Private Static List<Integer> sounds
Lock Microphone
Condition Participants
speak(int a)
{
        microphone.acquire();
        sound.append(a);
        participants.wake();
        participants.sleep();
        microphone.release();
}
int listen()
{
        microphone.acquire();
        while (sound.empty())
        {
                participants.sleep();
        }
        sounds = sounds[0]; // participants.wakeAll()
        return;
}
```

As perceivable, reciprocation was achieved by utilization of the lock feature to delineate between threads that are speaking in order to transition the remainder of the threads into the sleep state. This property is true of both the speak and listen functions as they both utilize an iterative procedure, while−loop, in order to establish a sleeping state while the systems remains idle until a thread speaks. This implementation, in essence, allows for potential multiple speakers to broadcast the same word without contingency.

Task V

The implementation of priority scheduling in the PriorityScheduler.java class was achieved through the application of a ThreadQueue *sub−class*. In this manner the initially proposed pseudo−code was designed with the intention of utilizing the following abstractions:

ThreadedKernel.scheduler
nachos.threads.RoundRobinScheduler
nachos.threads.PriorityScheduler

as a priority value for the threadstate.java class. Essentially this would invoke the corresponding priority value as a means of organizing threads in a hierarchical manner to deduce the order of execution. Keeping track of necessary elements requires the addition of a new data structure, *i.e.* treeset, in the priority queue, a hash set and a priority queue within the thread state. The treeset structure inside the priority queue class is intended to hold all the threads that will be managed within the priority queue. The hash set structure in the thread state class is meant to hold all the queues that are owned by the thread to be able to calculate effective priority. The extra priority queue within the thread state class is meant to resemble the queue that the thread is currently waiting in. With these data structures, the developed algorithm becomes fully capable of correctly implementing the priority scheduler. Therefore, if there were to be multiple threads with identical priority values, this issue would be resolved with the utilization of the NACHOS.THREADS.ROUNDROBINSCHEDULER abstraction. In this manner, the logic behind the implementation of the aforementioned functionality is demonstrated in the following excerpt of pseudo-code:

```
protected ThreadState pickNextThread()
{
            Lib.assertTrue(Machine.interrupt().disabled())
            if (threadStates.isEmpty())
                return null;
            return threadStates.last()
}
```

Additionally, the above outlined NEXTTHREAD and two synthesized functions, UPDATE and RECALCULATETHREADSCHEDULING were defined as follows:

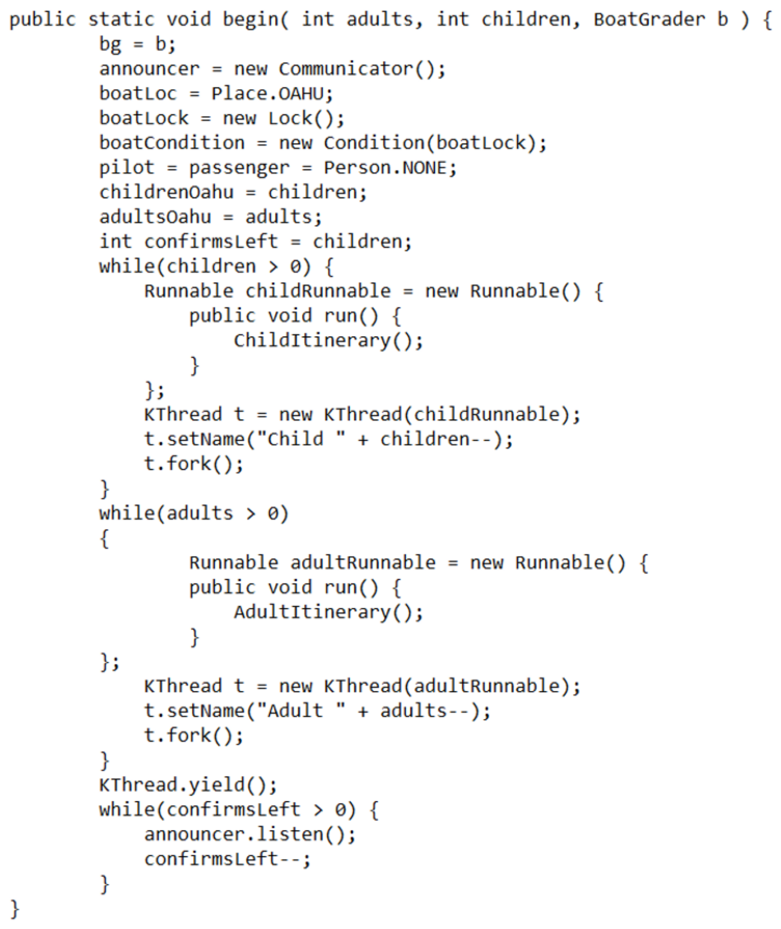
```
public void update()
{
        if (waitingQueue == null)
        else if (waitingQueue.owner == null)
        else if (waitingQueue.pickNextThread() == null)
        if (waitingQueue.transferPriority && waitingQueue.pickNextThread().getWinningPriority() > waitingQueue.owner.getWinningPriority())
        {
                waitingQueue.owner.effectivePriority = waitingQueue.pickNextThread().getWinningPriority();
                waitingQueue.owner.recalculateThreadScheduling();
                waitingQueue.owner.update();
        }
}

public void recalculateThreadScheduling()
        {
        Lib.assertTrue(Machine.interrupt().disabled());
        boolean passed = false;
        if (waitingQueue != null)
        {
                Iterator<ThreadState> it = waitingQueue.threadStates.iterator();
                while(it.hasNext())
                {
                    ThreadState curr = it.next();
                        if (curr.placement == this.placement)
                        {
                                it.remove();
                                passed = true;
                                break;
                        }
                }
        }
}
```

By obersvation it is concludable that in order to be able to implement the correct ordering of threads within the queues, a comparator is implemented that would take into account a thread's priority, time put in the queue and another value which is labeled as placement in the submitted code. This algorithmic functionality would be used for the case in which a thread has the same time and same priority. The placement value would essentially implement the NACHOS.THREADS.ROUNDROBINSCHEDULER when time and priority couldn't be used.

Task VI

In order to develop and implement a complete and efficient solution to the Oahu$\longrightarrow$Molokai problem, a proposed implementation would invoke prioritizing children moving from Oahu to Molokai. As such, CHILDren would occupy Molokai before the ADULTs would and simultaneously function as pilots for the transmitting boat back to Oahu such that the ADULTs can then pilot themselves in the interest of prohibiting CHILDren from commuting with ADULTs on the boat. In this manner unlike CHILDren, ADULTs would refrain from returing to Oahu. Due to the fact that there are at least two CHILDren, there will be at least one child waiting on Molokai who will consequently be able to function as a pilot to bring the boat back to its original state in Oahu, where this process would perpetuate. This method of CHILDren interchangeably switching boat rides with adults will ensure that the adults will eventually occupy Molokai irrespective of the number of ADULTs in relation to the CHILDren. The initializer for the CHILDren and ADULT threads is the BEGIN function which is implemented as follows:



```
public static void begin( int adults, int children, BoatGrader b ) {
    bg = b;
    announcer = new Communicator();
    boatLoc = Place.OAHU;
    boatLock = new Lock();
    boatCondition = new Condition(boatLock);
    pilot = passenger = Person.NONE;
    childrenOahu = children;
    adultsOahu = adults;
    int confirmsLeft = children;
    while(children > 0) {
      Runnable childRunnable = new Runnable() {
        public void run() {
          ChildItinerary();
        }
      };
      KThread t = new KThread(childRunnable);
      t.setName("Child " + children--);
      t.fork();
    }
    while(adults > 0)
    {
        Runnable adultRunnable = new Runnable() {
        public void run() {
          AdultItinerary();
        }
    };
      KThread t = new KThread(adultRunnable);
      t.setName("Adult " + adults--);
      t.fork();
    }
    KThread.yield();
    while(confirmsLeft > 0) {
      announcer.listen();
      confirmsLeft--;
    }
}
```

This function terminates once all CHILDren singal transfer completion. In this manner the CHILD and ADULT itineraries are as follows:

```
static void AdultItinerary() {
        Place location = Place.OAHU; // acquire BOATLOCK here
        while(location is not MOLOKAI) {
            if(boatLoc == location && childrenOahu <= 1 && pilot == Person.NONE)
                pilot = Person.ADULT
                rowToMolokoai()
                location = Place.MOLOKOAI;
            boatCondition.wake(); // and put to sleep
        }
        KThread.finish(); // release BOATLOCK here
    }

static void ChildItinerary() {
        Place location = Place.OAHU; //acquire BOATLOCK here
        while(true) {
            if(boatLoc == location) {
                if(location == Place.MOLOKOAI && pilot == Person.NONE) {
                    pilot = Person.CHILD
                    rowToOahu()
                    location = Place.OAHU
                } else if(location == Place.OAHU) {
                    boolean complete = childrenOahu == 1 && adultsOahu == 0;
                    if(pilot == Person.NONE) {
                        if(childrenOahu > 1) {
                            pilot = Person.CHILD
                            boatCondition.wake()
                            boatCondition.sleep()
                        } else if(childrenOahu == 1 && adultsOahu == 0) {
                            pilot = Person.CHILD
                            rowToMolokoai()
                        }
                    } else if(pilot == Person.CHILD) {
                        passenger = Person.CHILD
                        rowToMolokoai()
                    }
                    location = Place.MOLOKOAI
                    if (complete) DONE
            boatCondition.wake();
            boatCondition.sleep();
        }
        boatLock.release()
        announcer.speak(1)
        KThread.finish()
    }
```

As such, these functions consummate the breadth of the logical infrastructure developed for the Boat.java class

. **Design Decisions**

The primary factors of significant importance to the design considerations for the development of a thread system were the scalability, robustness, and efficiency of the system in accordance with the resources associated with its support. In this manner the utilization of data structures, primitive types included, such as priority queues was preeminently regnant to the functionality of the objects defined. For example priority queues were utilized for the established CONDITION VARIBLES, ALARM objects, as well as the priorities associated with various threads. This dependence on priority queues was paramount to the efficiency of the system. Likewise, an important design decision was the integration of the RELEASE and ACQUIRE functionalities from the SEMAPHORE.JAVA class which were defined there as the P and V functions. This integration was achieved by transcribing the functions and accounting for the variable and object discrepancies between the two classes. This design resulted in the ability to utilize the RELEASE and ACQUIRE functions in order to possess and control a lock with a thread in a non−dormant state. Another design decision formed during the implementation of scheduling in the PRIORITYSCHEDULER.JAVA class was the addition of the following functions:

UPDATE();
RECALCULATETHREADSCHEDULING()

These functions provide abstractions to maintain threads and their corresponding data structures. The UPDATE function is called whenever a thread changes it's priority or effective priority so as to be able to affect the effective priorities of the thread that own the queue of where one of their thread's priorities changed; whether it be effective or regular priority. For the purpose of affecting structures of priority queues, the UPDATE function is called recursively by traversing through the owners of queues and affecting the owners with the potentially changed priority values. Likewise the RECALCULATETHREADSCHEDULING function is a function that is made to keep the data structures updated as their dynamic values and elements change throughout run−time. The underlining discrepancy between these two functions is that the latter is meant to deal with the specific data structure being used by utilizng *treeset* because *treeset* doesn't automatically reorganize the set when one element changes so this function essentially removes the element then adds it again so that the data structure can be in the correct arrangement.
These design decision provided for the redundancy and reusability of the code and as such reduced the necessity of pair programming practices for the purposes of code reviews as the reusability of code ensured proper functionality.

Testing/QA

Task I

Testing the validity of the KThread.join() implementation was performed with multiple threads in the following arrangements; an even number and an odd number of threads. In this manner, the former arrangement was set−up with testing four threads where three of them join to the first. As such another test was executed in which eight threads were set−up where three threads join one and another three can join to another thread. These tests allowed to determine whether or not the join function properly does what is required in accordance with the design specifications. The latter arrangement however, sets−up a test in which three threads are utilizied to check if two would join to one. In order to test to check if a thread doesnt join, nine threads are utilizied; while seven of them join to one thread leaving a single thread without the ability to join to another one. Therefore these two arrangements determine the functionality of the join function by testing how even and odd number of threads work with just one single join. In regards to edge cases however, testing for robustness and scalability was achieved by testing, incrementally, the number of threads being used.

Task II

For the Condition.java and Condition2.java condition variables classes testing was set−up and executed in the following manner; to interrupt enable and to disable in the interest of ensuring that there is no waking up waiting threads. In essence, the control check that the queue storing threads is not empty in conjunction with the interrupt function being is disabled by default served as the predominant *sanity check*.
To stress test these different conditions, a lock variable may be synthesized to make sure that while a thread is invoked the lock remains locked and then put to sleep for each thread inside the lock. After the threads have completed their utilization of the lock, the wake function may be invoked to wake the first waiting thread in the wait queue.

Task III

The Alarm.java test cases were primarily executed with different times for the waitUntil function argument variable. Therefore forming a methodology to account for and essentially test for different timer conditions. These varied tests were performed iteratively to account for as many conditions as possible. In addition to these trivial test case, the corner cases were performed as follows; a wait time of 0 as well as an arbitrarily long wait time of 1000000 were hard coded. This strategy allowed stress testing the implementation in extreme cases to determine robustness and scalability.

Task IV

In Communicator, the main tests needed to be run to test how the implementation handled multiple listeners/speakers were set−up with two discrepant scenarios. These scenarios would naturally be the most demanding and stressful, and so would reveal any hidden errors that might arise under similar, or less stressful loads.
The first part of the test loads up the many listeners test, and checks if the listeners hear the single speaker. The second test creates many threads and designates them to speak to only a sole listener. It then checks if the listener heard the spoken word.

Task V

For general testing, there are two situations that were tested to check for working functionality. They were testing for when priority donation was being used and for when it wasnt. Some estimated corner case testing was also used such as making sure thread states were changed correctly for when a thread lost ownership of a priority queue.
Testing without priority donation was kept simple in the sense that making sure that priority wasnt donated to the owner of the queue. For future test cases, going to mix in queues that have transfer priority and those that dont, that share a common owner to make sure that the owners effective priority considers the relevant queues priorities.
Testing with priority donation involved making three queues and filling them each with threads of various priorities. One thread queue would be filled with the highest priority out of the three queues as so to be able to see that the priority scheduler was able to assign the correct highest priority to the owner of all three thread queues. Thread priorities would be changed through out the program to be able to see that changes were correctly updated throughout the thread queue structure.
Some corner case scenarios that were considered that werent as general as the previous testing was checking the values of thread states in scenarios such as when a thread would lose ownership of a queue when next thread would be called. Reasoning for checking this was because when a thread loses ownership of a queue, then that threads effective priority has a possibility of changing to something lower or staying the same.

Task VI

Although there had been some unsuspected issues with the initializers of the CHILD and ADULT objects in the BOAT.JAVA class, these issues had been mitigated simply with the //*commenting* out of the initializer statements.
The testing process was relatively straight−forward in the sense that a plethora of stress tests were performed all with various, arbitrary combinations of CHILDren and ADULTs. Consequently, the corner cases were performed in which there no ADULTs or CHILDren respectively, per test.

Design Questions

Why is it fortunate that we did not ask you to implement priority donation for SEMAPHORES?
It is fortunate because each time a thread acquires a lock or calls the JOIN *function, the allocator of the resource is acknowledged which enables the further donation of resources in priority to the single, aforementioned, allocator resource. This is true if a thread with a higher−priority rating remains in an idle state however if this functionality were to be implemented with semaphores, it would not be fully and completely utilizable because the last thread to* ACQUIRE *the semaphore will not necessarily be thread of lower/lowest−priority. In this manner the implementation would need amends maintain all threads that are currently utilizing the semaphore.*

A student proposes to solve the boats problem by use of a counter, *AdultsOnOahu*. Since this number isn't known initially, it will be started at zero, and incremented by each adult thread before they do anything else. Is this solution likely to work? Why or why not?
This solution will most likely fail due to the fact that would be no metric such as to delineate and provide the prescription for everyone who would, potentially, increment the counter pro−actively.

Team Member Work-Log

Avery Berchek
DevOps Engineer

- Designed and outlined the pseudo−code for Task I, II, & IV
- Outlined and participated in the design process associated with Tasks I, II, III, IV, V, & VI
- Implemented the algorithmic functionality for Tasks I, II, IV, & VI
- Primarily responsible for implementation of corner−case test cases
- Conducted and performed code reviews
- Performed optimizations and ensured the efficiency of the data structures utilized in the source code

Aleksandr Brodskiy
Project Manager

- Organized the *sprints* and weekly meetings in accordance with the *Agile/Scrum* project management methodology.
- Delegated tasks and assigned roles for the Engineering Team as well as conducted the code reviews
- Created and formatted the Design Documentation
- Managed all progress and operations of the Engineering Team in order to provide an efficient, robust, and optimal solution for a timely and submission.

David Cabral
Design Engineer

- Designed and outlined the pseudo−code for Task V

- Implemented the algorithmic functionality for Tasks V & VI
- Provided insight and proposed solution for *Priority Scheduling*
- Proposed solution and outlined source code structure for Task VI

Christopher DeSoto
Principal Engineer

- Set−up the integrated development environment for the entire Engineering Team.
- Outlined and participated in the design process associated with Tasks I, II, III, IV, V, & VI
- Integrated source code from the *development phase* to the *testing phase* to the *completed phase* to ensure the validity of the solution and provide a general framework for the debugging and testing process for the QA and Software Engineers.

Adiam G-Egzabher
Systems Engineer

- Outlined and participated in the pseudo−code development associated with Task VI
- Participated in the design process and testing/QA phase for Tasks IV, V, & VI

Nanditha Embar
QA Engineer

- Ensured functionality of all features
- Ensured the highest quality attainable with the time and resources provided for submitted source code
- Participated in the design process and testing/QA phase for Tasks I, II, & III

Christian Vernikoff
Software Engineer

- Designed and outlined the pseudo−code for Task I & III
- Implemented the algorithmic functionality for Tasks I, III, & VI
- Outlined and participated in the design process associated with Tasks I, II, III, IV, V, & VI
- Debugged and performed all necessary tests for Task I & III

Conclusion

As a result of completing an efficient and optimized solution to the aforementioned tasks, a foundational component of an operating system, a thread structure, was formed. This thread system will be utilized in the formation of a operating system scheduler for the development of more advanced multi−functional operating system features.