**Problem #5 Priority Scheduling**

Design:

The priority scheduler implements functions that manage scheduling of the kernel threads to ensure threads with higher priorities execute first. In a priority based scheduler all thread in waiting Queue are sorted by their effective priority. All threads store a sorted map of queues they own and the highest priority thread stored in the Queue(priority cache). The effective priority of a thread is the highest priority stored in its priority cache. When a thread is selected from the Queue to acquire the resource *nextThread()* that thread is removed from the Queue and the new thread is told to acquire *acquire()* the previous owner of the Queue is told to no longer own the Queue, and the cached priority in the Queue is removed. The new owner of the Queue gets the highest priority in the Queue added to its priority cache and its effective priority updated. When a thread is added to the waiting Queue *waitforAccess()* it may have the highest priority of all threads in the Queue, the owner of the Queue then needs to update its effective priority to be greater than or equal the new waiting thread. When a change occurs to a thread’s effective priority, it must reinsert itself into the correct position in the Queue and may result in affecting the effective priority of that Queues owner

**public** KThread nextThread() {

Lib.*assertTrue*(Machine.*interrupt*().disabled());

ThreadState nextThread = pickNextThread();

**if**(nextThread != **null**) {

nextThread.acquire(**this**);

**return** nextThread.thread;

}

**return** **null**;

}

*nextThread()* will check to see if owner has a value, if it does then remove it. It will also check the list of threads on Queue if empty it will reset everything and exit method. It then checks the list of threads on the Queue , if empty it will reset everything and exit the nextThread method. The owner will the obtain the thread that is chosen by pickNextThread and check if it is null which is important to know if an idle thread was created. The owner will receive all the information, priorities and permissions and be removed from threadlist. The owner will then be returned.

**protected** ThreadState pickNextThread() {

**while**(!queue.isEmpty()) {

Integer lastKey = queue.lastKey();

Queue<ThreadState> qlist = queue.get(lastKey);

**if**(qlist == **null** || qlist.isEmpty())

queue.remove(lastKey);

**else** {

**return** qlist.peek();

}

}

**return** **null**;

}

**public** SortedMap<Integer, Queue<ThreadState>> queue = **new** TreeMap<Integer,Queue<ThreadState>>();

**public** **boolean** transferPriority;

**public** ThreadState currentOwner = **null**;

}

}

*pickNextThread()* looks to see what next thread is, if there is nothing on the thread list then exit the method. Check all the priorities of the waitng threads. If there are equivalent prioritiestake the one with the longer waiting time. Need to implement for loop that goes through the threadlist total length. Next check the transfer priority to see if a transfer should be made from waiting thereads.. If so transfer the effective priority, if not just pass in the priority. Check the priority of the current thread to the max priority on the list, if it is greater than, set the new max to the current priority, change the position of the max and update the time. If they have the same priority check to see who was there the longest. Will return the thread that has the highest priority, but not remove it. This is just to see whats next.

**public** **int** getEffectivePriority() {

PriorityDonation donation = priorityCache.last();

**return** donation.priority;

}

**protected** **class** PriorityDonation{

**public** **int** priority;

**public** PriorityQueue queue;

}

*GetEffectivePriority()* returns the priority of the thread after taking into account the donations. It checks if the effective property is the recalculated priority, if so the its been properly updated, otherwise skip this portion of code. If not the same start a loop from 0 to size of the list, the waitQueue Queue will obtain the values in the list each time it increments to the next value and also contain threads from the Queues.

Start the next loop and current priority will get the priority of the j value and check to see if it is the maximum value. If it is greater than the maximum it will set the effective priority to that value.After check if the effective priority has been changed, if not the effective priority will just be the priority of the current thread and the effective priority will be returned.

**public** **void** setPriority(**int** priority) {

**if** (**this**.priority == priority)

**return**;

revokeDonation(**null**);

registerDonation(priority,**null**);

}

*setPriority()* will set the priority of the associated thread to the specified value. In the base code need a check to make sure its not greater than 7 or less than 1, and update the effective priority.

**public** **void** waitForAccess(PriorityQueue waitQueue) {

**assert**(waitQueue != **null**);

//System.out.println(thread+" started waiting for "+waitQueue);

waiting = waitQueue;

requeue(waiting);

}

*waitforAccess()* passes in the waitqueue which does nothing as we just save the current time into the waiting time.

**public** **void** acquire(PriorityQueue waitQueue) {

**assert**(waitQueue != **null**);

//Make sure this thread is completely removed from the queue:

//(before any donation changes are made)

dequeue(waitQueue);

//If there was a previous owner, perform handover:

//(only if there was a donation)

**if**(waitQueue.currentOwner != **null** && waitQueue.transferPriority) {

waitQueue.currentOwner.revokeDonation(waitQueue);

waitQueue.currentOwner = **null**;

}

//Declare ownership of this resource

waitQueue.currentOwner = **this**;

//If we were waiting on this resource then stop:

**if**(waiting == waitQueue){

waiting = **null**;

}

//Perform donation if necessary:

**if**(waitQueue.transferPriority){

registerDonation(waitQueue);

}

}

*Acquire()* add a thread to the acquired list. Reset the effective priority

Reference Code: <https://github.com/lqhl/lqhl-acm-nachos/blob/master/src/nachos/threads/PriorityScheduler.java>

|  |
| --- |
| package nachos.threads; |
|  |  |
|  | import java.util.HashSet; |
|  | import java.util.LinkedList; |
|  |  |
|  | import nachos.machine.Lib; |
|  | import nachos.machine.Machine; |
|  |  |
|  | /\*\* |
|  | \* A scheduler that chooses threads based on their priorities. |
|  | \* |
|  | \* <p> |
|  | \* A priority scheduler associates a priority with each thread. The next thread |
|  | \* to be dequeued is always a thread with priority no less than any other |
|  | \* waiting thread's priority. Like a round-robin scheduler, the thread that is |
|  | \* dequeued is, among all the threads of the same (highest) priority, the thread |
|  | \* that has been waiting longest. |
|  | \* |
|  | \* <p> |
|  | \* Essentially, a priority scheduler gives access in a round-robin fassion to |
|  | \* all the highest-priority threads, and ignores all other threads. This has the |
|  | \* potential to starve a thread if there's always a thread waiting with higher |
|  | \* priority. |
|  | \* |
|  | \* <p> |
|  | \* A priority scheduler must partially solve the priority inversion problem; in |
|  | \* particular, priority must be donated through locks, and through joins. |
|  | \*/ |
|  | public class PriorityScheduler extends Scheduler { |
|  | /\*\* |
|  | \* Allocate a new priority scheduler. |
|  | \*/ |
|  | public PriorityScheduler() { |
|  | } |
|  |  |
|  | /\*\* |
|  | \* Allocate a new priority thread queue. |
|  | \* |
|  | \* @param transferPriority |
|  | \* <tt>true</tt> if this queue should transfer priority from |
|  | \* waiting threads to the owning thread. |
|  | \* @return a new priority thread queue. |
|  | \*/ |
|  | public ThreadQueue newThreadQueue(boolean transferPriority) { |
|  | return new PriorityQueue(transferPriority); |
|  | } |
|  |  |
|  | public int getPriority(KThread thread) { |
|  | Lib.assertTrue(Machine.interrupt().disabled()); |
|  |  |
|  | return getThreadState(thread).getPriority(); |
|  | } |
|  |  |
|  | public int getEffectivePriority(KThread thread) { |
|  | Lib.assertTrue(Machine.interrupt().disabled()); |
|  |  |
|  | return getThreadState(thread).getEffectivePriority(); |
|  | } |
|  |  |
|  | public void setPriority(KThread thread, int priority) { |
|  | Lib.assertTrue(Machine.interrupt().disabled()); |
|  |  |
|  | Lib.assertTrue(priority >= priorityMinimum |
|  | && priority <= priorityMaximum); |
|  |  |
|  | getThreadState(thread).setPriority(priority); |
|  | } |
|  |  |
|  | public boolean increasePriority() { |
|  | boolean intStatus = Machine.interrupt().disable(); |
|  |  |
|  | KThread thread = KThread.currentThread(); |
|  |  |
|  | int priority = getPriority(thread); |
|  | if (priority == priorityMaximum) |
|  | return false; |
|  |  |
|  | setPriority(thread, priority + 1); |
|  |  |
|  | Machine.interrupt().restore(intStatus); |
|  | return true; |
|  | } |
|  |  |
|  | public boolean decreasePriority() { |
|  | boolean intStatus = Machine.interrupt().disable(); |
|  |  |
|  | KThread thread = KThread.currentThread(); |
|  |  |
|  | int priority = getPriority(thread); |
|  | if (priority == priorityMinimum) |
|  | return false; |
|  |  |
|  | setPriority(thread, priority - 1); |
|  |  |
|  | Machine.interrupt().restore(intStatus); |
|  | return true; |
|  | } |
|  |  |
|  | /\*\* |
|  | \* The default priority for a new thread. Do not change this value. |
|  | \*/ |
|  | public static final int priorityDefault = 1; |
|  | /\*\* |
|  | \* The minimum priority that a thread can have. Do not change this value. |
|  | \*/ |
|  | public static final int priorityMinimum = 0; |
|  | /\*\* |
|  | \* The maximum priority that a thread can have. Do not change this value. |
|  | \*/ |
|  | public static final int priorityMaximum = 7; |
|  |  |
|  | /\*\* |
|  | \* Return the scheduling state of the specified thread. |
|  | \* |
|  | \* @param thread |
|  | \* the thread whose scheduling state to return. |
|  | \* @return the scheduling state of the specified thread. |
|  | \*/ |
|  | protected ThreadState getThreadState(KThread thread) { |
|  | if (thread.schedulingState == null) |
|  | thread.schedulingState = new ThreadState(thread); |
|  |  |
|  | return (ThreadState) thread.schedulingState; |
|  | } |
|  |  |
|  | /\*\* |
|  | \* A <tt>ThreadQueue</tt> that sorts threads by priority. |
|  | \*/ |
|  | protected class PriorityQueue extends ThreadQueue { |
|  | PriorityQueue(boolean transferPriority) { |
|  | this.transferPriority = transferPriority; |
|  | } |
|  |  |
|  | public void waitForAccess(KThread thread) { |
|  | Lib.assertTrue(Machine.interrupt().disabled()); |
|  | getThreadState(thread).waitForAccess(this); |
|  | } |
|  |  |
|  | public void acquire(KThread thread) { |
|  | Lib.assertTrue(Machine.interrupt().disabled()); |
|  | getThreadState(thread).acquire(this); |
|  | } |
|  |  |
|  | public KThread nextThread() { |
|  | Lib.assertTrue(Machine.interrupt().disabled()); |
|  |  |
|  | // TASK 1.5 |
|  | if (lockHolder != null) { |
|  | lockHolder.donationQueue.remove(this); |
|  | lockHolder.update(); |
|  | } |
|  | ThreadState threadState = pickNextThread(); |
|  | if (threadState != null) { |
|  | threadState.acquire(this); |
|  | return threadState.thread; |
|  | } |
|  | else |
|  | return null; |
|  | } |
|  |  |
|  | /\*\* |
|  | \* Return the next thread that <tt>nextThread()</tt> would return, |
|  | \* without modifying the state of this queue. |
|  | \* |
|  | \* @return the next thread that <tt>nextThread()</tt> would return. |
|  | \*/ |
|  | protected ThreadState pickNextThread() { |
|  | // TASK 1.5 |
|  | KThread result = null; |
|  | int maxPriority = -1; |
|  | for (KThread thread : waitQueue) |
|  | if (result == null |
|  | || getEffectivePriority(thread) > maxPriority) { |
|  | result = thread; |
|  | maxPriority = getEffectivePriority(thread); |
|  | } |
|  | if (result == null) |
|  | return null; |
|  | return getThreadState(result); |
|  | } |
|  |  |
|  | public void print() { |
|  | Lib.assertTrue(Machine.interrupt().disabled()); |
|  |  |
|  | // TASK 1.5 |
|  | System.out.print("PriorityQueue:"); |
|  | for (KThread thread : waitQueue) |
|  | System.out.print(" " + thread); |
|  | System.out.println(); |
|  | } |
|  |  |
|  | /\*\* |
|  | \* <tt>true</tt> if this queue should transfer priority from waiting |
|  | \* threads to the owning thread. |
|  | \*/ |
|  | public boolean transferPriority; |
|  |  |
|  | LinkedList<KThread> waitQueue = new LinkedList<KThread>(); |
|  |  |
|  | ThreadState lockHolder = null; |
|  | } |
|  |  |
|  | /\*\* |
|  | \* The scheduling state of a thread. This should include the thread's |
|  | \* priority, its effective priority, any objects it owns, and the queue it's |
|  | \* waiting for, if any. |
|  | \* |
|  | \* @see nachos.threads.KThread#schedulingState |
|  | \*/ |
|  | protected class ThreadState { |
|  | /\*\* |
|  | \* Allocate a new <tt>ThreadState</tt> object and associate it with the |
|  | \* specified thread. |
|  | \* |
|  | \* @param thread |
|  | \* the thread this state belongs to. |
|  | \*/ |
|  | public ThreadState(KThread thread) { |
|  | this.thread = thread; |
|  |  |
|  | setPriority(priorityDefault); |
|  | } |
|  |  |
|  | /\*\* |
|  | \* Return the priority of the associated thread. |
|  | \* |
|  | \* @return the priority of the associated thread. |
|  | \*/ |
|  | public int getPriority() { |
|  | return priority; |
|  | } |
|  |  |
|  | /\*\* |
|  | \* Return the effective priority of the associated thread. |
|  | \* |
|  | \* @return the effective priority of the associated thread. |
|  | \*/ |
|  | public int getEffectivePriority() { |
|  | return getEffectivePriority(new HashSet<ThreadState>()); |
|  | } |
|  |  |
|  | private int getEffectivePriority(HashSet<ThreadState> set) { |
|  | // if (effectivePriority != expiredEffectivePriority) |
|  | // return effectivePriority; |
|  |  |
|  | if (set.contains(this)) { |
|  | // System.err.println("Deadlock"); |
|  | return priority; |
|  | } |
|  |  |
|  | effectivePriority = priority; |
|  |  |
|  | for (PriorityQueue queue : donationQueue) |
|  | if (queue.transferPriority) |
|  | for (KThread thread : queue.waitQueue) { |
|  | set.add(this); |
|  | int p = getThreadState(thread) |
|  | .getEffectivePriority(set); |
|  | set.remove(this); |
|  | if (p > effectivePriority) |
|  | effectivePriority = p; |
|  | } |
|  |  |
|  | PriorityQueue queue = (PriorityQueue) thread.waitForJoin; |
|  | if (queue.transferPriority) |
|  | for (KThread thread : queue.waitQueue) { |
|  | set.add(this); |
|  | int p = getThreadState(thread) |
|  | .getEffectivePriority(set); |
|  | set.remove(this); |
|  | if (p > effectivePriority) |
|  | effectivePriority = p; |
|  | } |
|  |  |
|  | return effectivePriority; |
|  | } |
|  |  |
|  | /\*\* |
|  | \* Set the priority of the associated thread to the specified value. |
|  | \* |
|  | \* @param priority |
|  | \* the new priority. |
|  | \*/ |
|  | public void setPriority(int priority) { |
|  | if (this.priority == priority) |
|  | return; |
|  |  |
|  | this.priority = priority; |
|  |  |
|  | // TASK 1.5 |
|  | update(); |
|  | } |
|  |  |
|  | /\*\* |
|  | \* Called when <tt>waitForAccess(thread)</tt> (where <tt>thread</tt> is |
|  | \* the associated thread) is invoked on the specified priority queue. |
|  | \* The associated thread is therefore waiting for access to the resource |
|  | \* guarded by <tt>waitQueue</tt>. This method is only called if the |
|  | \* associated thread cannot immediately obtain access. |
|  | \* |
|  | \* @param waitQueue |
|  | \* the queue that the associated thread is now waiting on. |
|  | \* |
|  | \* @see nachos.threads.ThreadQueue#waitForAccess |
|  | \*/ |
|  | public void waitForAccess(PriorityQueue waitQueue) { |
|  | // TASK 1.5 |
|  | waitQueue.waitQueue.add(thread); |
|  | if (waitQueue.lockHolder == null) |
|  | return; |
|  | waitQueue.lockHolder.update(); |
|  | } |
|  |  |
|  | /\*\* |
|  | \* Called when the associated thread has acquired access to whatever is |
|  | \* guarded by <tt>waitQueue</tt>. This can occur either as a result of |
|  | \* <tt>acquire(thread)</tt> being invoked on <tt>waitQueue</tt> (where |
|  | \* <tt>thread</tt> is the associated thread), or as a result of |
|  | \* <tt>nextThread()</tt> being invoked on <tt>waitQueue</tt>. |
|  | \* |
|  | \* @see nachos.threads.ThreadQueue#acquire |
|  | \* @see nachos.threads.ThreadQueue#nextThread |
|  | \*/ |
|  | public void acquire(PriorityQueue waitQueue) { |
|  | // TASK 1.5 |
|  | waitQueue.waitQueue.remove(thread); |
|  | waitQueue.lockHolder = this; |
|  | donationQueue.add(waitQueue); |
|  | update(); |
|  | } |
|  |  |
|  | public void update() { |
|  | effectivePriority = expiredEffectivePriority; |
|  | getEffectivePriority(); |
|  | } |
|  |  |
|  | /\*\* The thread with which this object is associated. \*/ |
|  | protected KThread thread; |
|  | /\*\* The priority of the associated thread. \*/ |
|  | protected int priority = priorityDefault; |
|  | // TASK 1.5 |
|  | protected int effectivePriority = expiredEffectivePriority; |
|  | protected static final int expiredEffectivePriority = -1; |
|  | protected LinkedList<PriorityQueue> donationQueue = new LinkedList<PriorityQueue>(); |
|  | } |
|  | } |