**package** nachos.threads;

**import** nachos.machine.\*;

**import** java.util.TreeSet;

**import** java.util.HashSet;

**import** java.util.Iterator;

/\*-------------Andrew-------------\*/

/\*-------------Isaac--------------\*/

**import** java.util.LinkedList;

/\*-------------Andrew-------------\*/

/\*-------------Isaac--------------\*/

/\*\*

\* 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());

// implement me

/\*-------------Andrew-------------\*/

/\*--------------------------------\*/

ThreadState nextThreadstate = pickNextThread();//make Threadstate object

//set object to next thread to get next thread

**if**(nextThreadstate != **null**){//check if there is a next thread

nextThreadstate.acquire(**this**);//if there is acquire it

**return** nextThreadstate.thread;

}

/\*-------------Andrew-------------\*/

/\*--------------------------------\*/

**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() {

// implement me

/\*-------------Andrew-------------\*/

/\*-------------Isaac--------------\*/

KThread nextThread = **null**;

**int** currentMaxPriority = -1;

//we set next priority to -1 to ensure if there is a thread waiting its guaranteed to get picked

**for** (KThread thread : waitQueue)//increment by each thread through waitQueue

**if** (nextThread == **null**|| getEffectivePriority(thread) > currentMaxPriority) {

//Finding the thread with the highest priority to set to nextThread

nextThread = thread;

//set our nextThread to the thread with the highest priority

currentMaxPriority = getEffectivePriority(thread);

}

**if** (nextThread == **null**)

**return** **null**;

**return** getThreadState(nextThread);

//return null was taken out, we need to return the next thread we picked

/\*-------------Andrew-------------\*/

/\*-------------Isaac--------------\*/

}

**public** **void** print() {

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

// implement me (if you want)

/\*-------------Andrew-------------\*/

/\*-------------Isaac--------------\*/

System.***out***.print("Our PriorityQueue:");

**for** (KThread thread : waitQueue)

System.***out***.print(" " + thread);

/\*-------------Andrew-------------\*/

/\*-------------Isaac--------------\*/

}

/\*\*

\* <tt>true</tt> if this queue should transfer priority from waiting

\* threads to the owning thread.

\*/

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

/\*-------------Andrew-------------\*/

/\*-------------Isaac--------------\*/

LinkedList<KThread> waitQueue = **new** LinkedList<KThread>();

/\*-------------Andrew-------------\*/

/\*-------------Isaac--------------\*/

}

/\*\*

\* 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() {

// implement me

/\*-------------Andrew-------------\*/

/\*-------------Landon-------------\*/

**return** getEffectivePriority(**new** HashSet<ThreadState>());

/\*-------------Andrew-------------\*/

/\*-------------Landon-------------\*/

}

/\*-------------Andrew-------------\*/

/\*-------------Landon-------------\*/

**private** **int** getEffectivePriority(HashSet<ThreadState> set) {

**if** (effectivePriority != ***oldEffectivePriority***)

//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.threadsJoinedOnMe;

**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;

/\*-------------Andrew-------------\*/

/\*-------------Landon-------------\*/

}

/\*\*

\* 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;

// implement me

/\*-------------Andrew-------------\*/

/\*-------------Isaac--------------\*/

effectivePriority = ***oldEffectivePriority***;

getEffectivePriority();

/\*-------------Andrew-------------\*/

/\*-------------Isaac--------------\*/

}

/\*\*

\* 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) {

// implement me

/\*-------------Andrew-------------\*/

/\*--------------------------------\*/

waitQueue.waitQueue.add(thread);

/\*if (waitQueue.lockHolder == null)

return;

waitQueue.lockHolder.update();\*/

/\*-------------Andrew-------------\*/

/\*--------------------------------\*/

}

/\*\*

\* 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) {

// implement me

/\*-------------Andrew-------------\*/

/\*--------------------------------\*/

waitQueue.waitQueue.remove(thread);

/\*waitQueue.lockHolder = this;\*/

donationQueue.add(waitQueue);

effectivePriority = ***oldEffectivePriority***;

getEffectivePriority();

/\*-------------Andrew-------------\*/

/\*--------------------------------\*/

}

/\*\* The thread with which this object is associated. \*/

**protected** KThread thread;

/\*\* The priority of the associated thread. \*/

**protected** **int** priority;

/\*-------------Andrew-------------\*/

/\*-------------Isaac--------------\*/

**protected** **int** effectivePriority = ***oldEffectivePriority***;

**protected** **static** **final** **int** ***oldEffectivePriority*** = -1;

**protected** LinkedList<PriorityQueue> donationQueue = **new** LinkedList<PriorityQueue>();

/\*-------------Andrew-------------\*/

/\*-------------Isaac--------------\*/

}

}