Objective: To implement Thread scheduling on the OSP2 simulator using a version of the Linux constant time strategy. You will implement the module `ThreadCB.java` to further your understanding of CPU scheduling.

Required to turn in: Follow all directions regarding hand-in procedures. Points will be deducted if you do not follow directions precisely. You must submit an electronic copy of your Linux constant time scheduling solution (or your best try) via dropbox. Late assignments will be assessed a 10% penalty per day. You must document your code and provide a one-page explanation of how you accomplished the assignment (or what you have currently and why you could not complete). You should describe your use, creation, and manipulation of data structures to accomplish the assignment.

Building and executing the simulation
As in the previous assignment, I suggest that you create a new directory. Copy your files from the previous cpu scheduling project and starting with that version of `ThreadCB.java` modify it to create the Linux constant time scheduling solution. The only file you should have to modify is `ThreadCB.java`. Modifying the other files will probably "break" OSP2.

The changes you need to make to your code from the previous project are:

1) You will need to create two priority arrays of queues: an active array and an expired array (see slide 49 of chapter 6 ppt). In our case we will simplify and have only priority values of 0 to 4, i.e. the active array and expired array will each be comprised of 5 queues.

2) In `do_create()`, the newly created thread will be assigned the default priority value 2 and be placed in the expired array in the queue for threads having priority 2.

3) In `do_resume()`, when inserting a thread into the readyQueue, if the priority of the thread is not already at the highest level (0), then it is raised by decrementing it. (recall that 0 is the highest priority). The thread is then placed in appropriate queue of the expired array. This is how threads have their priority raised.

4) In `do_dispatch()` when the current thread is preempted, you need to check if the reason that it is being preempted is because its quantum has expired. Do this by calling `HTimer.get()`. This returns the time left until the timer interrupts. If the return value is less than 1, then the timer has interrupted indicating that this thread has exceeded its quantum. If this is the case, then lower the priority of the thread by incrementing its priority value. Then insert it into the appropriate queue of the expired array. This is how threads have their priority lowered. If on the other hand the value returned from `HTimer` is positive nonzero, then the thread
has not exceeded its quantum and should be keep its current priority. In this case, place it in the corresponding queue of the expired array.

5) In `do_dispatch()`, threads with priority values in the range 0-2 should dispatched with a quantum value of 40 and threads with priority values in the range 3-4 should be dispatched with a quantum value of 20.

6) When `do_dispatch()` is invoked, you will look in the active array for a thread to dispatch. Start with the highest priority queue (priority 0). If it is empty, then look at the next highest priority queue. Continue until either you find a thread to dispatch or you determine that the active array is completely empty. If the active array is completely empty, then swap the active array and the expired array and start the search for a thread to launch all over again. Only if both the active array and the expired array are empty should you idle the CPU by setting PTBR to null and returning FAILURE. **Suggestion: write a method to return the next thread from the active array or null if the active array is empty. This will be a lot cleaner than inserting the corresponding code into do_dispatch().**

**Some more hints:**

- The outlined algorithm for the solution to CPU scheduling is derived from a discussion in your OSP2 book regarding the thread scheduling module ThreadCB. Your OSP2 book (and the Silberschatz book) are your best references for conceptual questions on implementation of ThreadCB.java.

- Be very careful that you do not leave a thread in the ready queue when it is dispatched. If you do, and then re-insert it after a block, the queue may become disconnected, making some ready threads inaccessible. These stranded threads can never be re-scheduled, so threads may starve in the ready queue.

**Turning in Your Assignment via dropbox**

When you are satisfied that your CPU scheduling algorithm in ThreadCB works, use dropbox to submit your project.

1. `ThreadCB.java` *(the Linux constant time scheduling version that you have implemented)*

2. One-page explanation of how you accomplished the assignment (or what you have currently and why you could not complete). You should describe your use, creation, and manipulation of data structures to accomplish the assignment.

Remember to turn your assignment in on time. Late assignments will be assessed a 10% penalty per day.