CMSC412 Operating Systems Project 02 *Prof. William Arbaugh, CMSC412, Fall 2010*

OS Schedulers: Fair-Share Scheduling in the Windows Research Kernel

1 Introduction

The purpose of this experiment is to gain more experience with CPU scheduling inside the operating system, specifically by studying and modifying the Windows? Research Kernel (WRK) scheduler. Prior to your modifications, the WRK implements a general multi-level feedback queue (MLFQ) consisting of 32 priority levels. Real-time priorities (> 15) are assigned statically to threads. The WRK can apply dynamic priority adjustments (boost and decay) under the following situations: I/O completion; wait completion on events or semaphores; when threads in the foreground process complete a wait; when GUI threads wake up for Windows input; and for CPU starvation avoidance. The WRK, as with any MLFQ-based operating system, dynamically manipulates thread priorities in an attempt to minimize interactive response time, maximize CPU utilization, maximize predictability, and in general minimize the time to execute threads.

Although the scheduling algorithm in the WRK in general does a very nice job at balancing these different goals for most normal operating conditions, there is one particular situation where we'd like to improve on the WRK's scheduling capabilities. Currently, the WRK treats all threads equally without regard to the process to which the thread belongs. For example, if one process creates 99 threads and a second process only creates 1 thread, the WRK scheduler essentially behaves as if each processor created 50 threads, or if there were only 1 process that created 100 threads, or if there were 100 processes that each created 1 thread. That is, all other things being equal, the amount of the CPU that a process will get during its lifetime is equal to the number of threads in the process divided by the total number of threads alive in the system. This means that if two processes have the same "amount of work" to do, then the one that is structured with more threads will complete sooner! This hardly seems fair, so we're going to do something about it!

In this experiment, you will address this issue by implementing a variation of fair-share scheduling, in that all processes that have been designated fair-share processes will be given an approximately equal percentage of the CPU, irrespective of the number of threads in each process. For example, let's assume that there are two users, Mary and Jim, who each create 1 process. Mary creates six CPU-bound threads, and Jim creates two CPU-bound threads. With this fair-share scheduling, Mary and Jim should each get approximately 50 percent of the CPU, which means that each of Mary's threads would receive about 8.33 percent of the CPU and Jim's threads would each receive about 25 percent of the CPU.

There are a number of possible algorithmic approaches, each with different costs and benefits. In this experiment, we will implement a relatively simple approach that is reasonably straightforward, yet comes with a heavy overhead cost. You are being asked to implement one particular algorithm (which is described below); however, you are being asked to implement a general-purpose solution that achieves the requirements outlined above. With any approach you implement (including the algorithm discussed below), you should analyze and discuss the overhead incurred by the algorithm in terms of absolute measured overhead as well as general complexity.

The simple approach is this: given sufficient duration, you can approximate fair-share scheduling across n processes by first randomly choosing one of the processes and then randomly choosing one of the threads in that process. That's it!

At this point in the lab, you should convince yourself that this approach does indeed achieve the fair-share goal. Furthermore, you should try to determine the benefits and limitations of such an approach: what's good about this approach? What's bad about this approach?

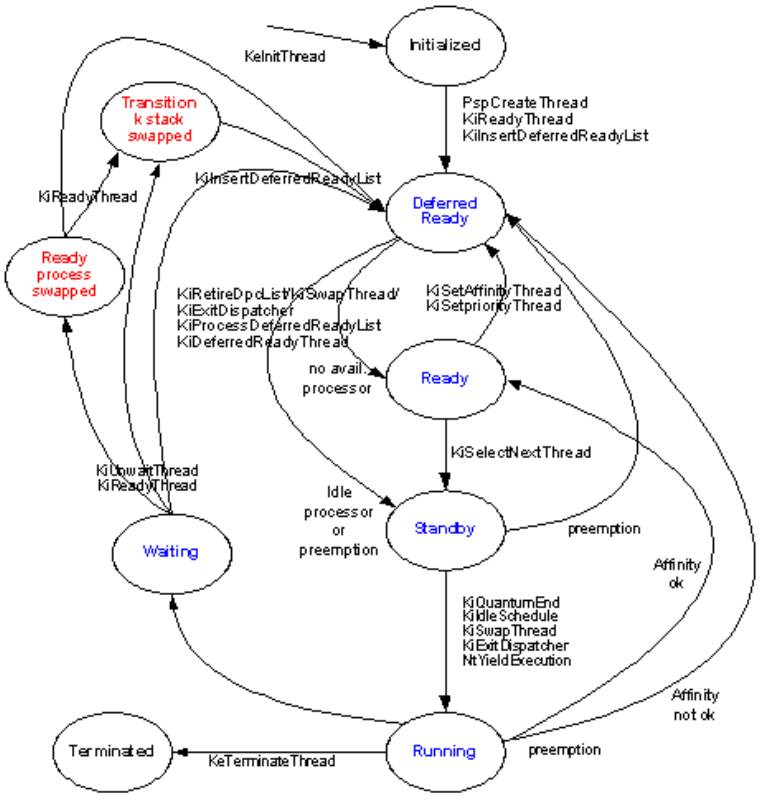
With any approach that you implement, you'll find that there are a number of different design choices-a number of different places in the code in which you might consider making your modifications. For example, you might choose to enforce this fair-share requirement on every scheduling decision, which means applying

it right before dispatch. Or you might choose to modify where in the queue the thread is placed, immediately after completing a quantum (hoping that this is sufficient to implement a fair-share policy). Furthermore, you might choose to reduce the overhead by not engaging your mechanism every single time through the scheduler. It is unlikely that any implementation will perfectly achieve the fair-share requirement, and so tradeoffs are expected (you will need to analyze and discuss the benefits and limitations of your approach). The goal is to get accurate fair-share results without tremendous overhead.

Although the intuition behind fair-share scheduling might lead to the belief that this is a simple experiment, both the complexity of a real operating system scheduler such as the WRK scheduler and the challenges of evaluating the correctness of your implementation make this a challenging exercise.

2 Background: The WRK Scheduler

To help you better understand the WRK scheduler, Figure 1 illustrates the thread state diagram for the WRK:



Additional information about the WRK scheduler is contained in *Windows Internals*, by Russinovich and Solomon (4^{th} Ed., Microsoft Press), which was included as part of your WinXP VM. Here is a summary of the points made in the book:

1. A thread's initial base priority is inherited from the process base priority. A process, by default, inherits its base priority from the process that created it.

2. To make thread-scheduling decisions, the kernel maintains a set of data structures known collectively as the dispatcher database.
3. The dispatcher ready queues (*KiDispatcherReadyListHead*) contain the threads that are in the ready state, waiting to be scheduled for execution. There is one queue for each of the 32 priority levels. To speed up the selection of which thread to run or preempt, Windows maintains a 32-bit bit mask called the ready summary (*KiReadySummary*). Each bit set indicates one or more threads in the ready queue for that priority level. (Bit 0 represents priority 0, and so on.)
4. Each process has a quantum value in the kernel process block. This value is used when giving a thread a new quantum. As a thread runs, its quantum is reduced at each clock interval. If there is no remaining thread quantum, the quantum end processing is triggered. If there is another thread at the same priority that is waiting to run, a context switch occurs to the next thread in the ready queue.
5. The quantum value isn't reset when a thread enters a wait state-when the wait is satisfied, the thread's quantum value is decremented by 1 quantum unit, equivalent to one-third of a clock interval (except for threads running at priority 14 or higher, which have their quantum reset to a full turn after a wait).

Many of the important routines for thread scheduling in the WRK are contained in the file thredsup.c (WRK-v1.2/base/ntos/ke/thredsup.c). Another important file to browse is ki.h.

3 Getting Started (*and Further Simplifications!*)

You can find a zip file with commented versions of some the files you will need to modify at:

http://www.cs.umd.edu/ sandro/412/lab2/fss_students.zip

Download this file and uncompress it over your existing C:
wrk-v1.2 directory. Inside you'll find versions of: ps.h, dpcsup.c, ki.h, and thredsup.c, as well as a test_app directory for testing your scheduler and some updated libs.

This experiment assumes that you have completed the document "Getting Started with the Windows Research Kernel", which explains the basic development cycle: how to modify the WRK source code, how to recompile and create a new kernel, how to install the new kernel on a virtual machine running Windows Server 2003, and how to attach a kernel debugger to the virtual machine.

A general-purpose and comprehensive implementation of fair-share scheduling in WRK would include both the implementation within the kernel and the ability of processes to selectively request (through a system call) to be scheduled according to the fair-share policy. However, to focus our attention on the scheduler only, in this experiment our approach will be to assume that all real-time processes require this behavior. This facilitates a convenient (and pre-existing) mechanism for telling the operating system which processes should be scheduled according to this fair-share policy: if it's a real-time process, then it will be scheduled according to the fair-share policy. This approach further simplifies the experiment, because the real-time class is not subject to the boost/decay behavior described above.

Just as there are many different algorithms for implementing fair-share scheduling, there are a number of different implementation approaches of the algorithm that you choose-in this case, there are many different ways to attempt to implement fair-share scheduling of the real-time processes. We've already decided that we are going to go with the random-process-then-random-thread approach, but where exactly should this be implemented in the WRK? In the remainder of this section, I give a sketch of how you might implement this. I suggest that you take this approach first, and, if time permits, you can redesign and implement a different approach that improves on the one I've outlined here.

Our approach will be to take action whenever a thread stops executing on the CPU:

1. Re-insert the just-ending thread (as normal) into its position in the MLFQ.

2. Manually select the real-time thread that you want to execute next, if one exists (via the random -process-then-random-thread approach).
3. Manually put the selected real-time thread at the front of the appropriate queue.

Furthermore, here's a plan to attempt to accomplish step 2 of the previous procedure:

1. Find all of the real-time threads in the MLFQ of the WRK. Remember, the real-time priorities are greater than 15.
2. Divide the threads up according to the processes to which they belong. Each thread structure has a pointer to its process-this step helps you determine the set of active real-time processes so that you can choose one of them randomly.
3. Print out the process and then the thread you're going to manually select. This step helps you see (via the debugging window) whether your selection mechanism appears to be working.
4. Put this thread at the beginning of the queue. Change the source code to put this thread at the beginning of the queue, comment out your code for step 3, and then rebuild and retest the kernel.

Where should you make the modifications in the code? Good question! First, study the thread state diagram in Figure 1 above, and then start looking through files thredsup.c and ki.h. You're essentially looking for an existing point or points in the code where a thread gives up the CPU, either voluntarily or by force.

Note: The routine or routines that you choose may not necessarily be explicitly identified in Figure 1. Although you are not required to understand all of the gory details about the WRK scheduler code, you will probably need to study this code a lot before you can determine the best place for making your modifications.

4 Evaluating Your Implementation

Testing an operating system scheduler can be tricky. Because the scheduling decisions are so frequent, you need to make sure that your testing methodology does not inadvertently mask the intended behavior of the scheduler. For example, if you insert debugging statements into the scheduler, and have these debugging statements print relatively often, then the very act of this I/O can change the behavior you're trying to study. Similarly, the user-level program can exhibit strange behavior if you do not realize precisely what it's attempting to do.

I have created a simple test program to help you in your evaluation of your implementation of fair-share scheduling (of real-time processes) in the WRK. The program essentially does a slow sorting of numbers and is parameterized by three elements: the number of independent threads performing the sorting, the size of the array (or arrays) to sort, and the clock speed of the processor. This program is available at:

http://www.cs.umd.edu/ sandro/412/lab2/fair_share_user_level_program.exe

http://www.cs.umd.edu/ sandro/412/lab2/fair_share_user_level_program.c

You are certainly free to modify this program and even write your own test program. The purpose of the program I provide is not necessarily to be the perfect test program but rather to give you a feel for how you might write a test program. (Note, for example, that it contains certain invocations to various clock routines.)

To create multiple real-time processes, write a batch file (for example, start_20_then_10.bat) like this:

start /realtime fair_share_user_level_program.exe 20 1.6 29000

start /realtime fair_share_user_level_program.exe 10 1.6 29000

5 Additional Information

A few final notes before you start any modifications:

1. Remember that, while this virtual-machine development cycle is much faster than it would be if we were trying to debug on a physical machine, it is still somewhat time-consuming. That is, you cannot expect to make a new kernel and reboot it as quickly as you'd inevitably like. Therefore, you should be very careful anytime you push a new kernel to the virtual machine-make sure that you have sufficiently studied your anticipated modifications and that they "look right". Get into a defensive state of mind, anticipating errors before they manifest themselves as kernel crashes!

2. Do not attempt to do everything all at once. I suggest putting a few debugging statements in the kernel to better learn how it operates. Then, start making modifications. (I like to use static ints to control and limit the number of times a debugging statement appears.)

3. There are a number of different ways to attempt to compute random numbers (which you use in this lab). One way is to find a hardware clock of sufficient granularity and just look at the low-order bits. Note that PerfGetCycleCount of the WRK might provide such a clock; you'll need to think about this carefully before you use it (for example, does it provide sufficient granularity?).

4. Here's a development setup (which you might or might not follow - it's up to you, but it might save you some time):

   (1). Windows command prompt in c:
   WRK-v1.2: used for grepping files (such as "grep -i RemoveEntryList */*/*/* — less"). Grep can be installed as a Windows binary or as part of the Cygwin package.

   (2). An editor window that is capable of syntax highlighting

   (3). C:/WRK-v1.2/base/ntos/BUILD/EXE open, so that I can easily drag and drop my new kernels onto the virtual machine.

   (4). Windows command prompt in C:
   WRK-v1.2
   base
   ntos, used to "nmake x86=".

   (5). WRK virtual machine running.

   (6). Kernel Debugger window running.

5. Finally, and most importantly, keep in mind that the scheduler is very complicated, and attempting to modify and analyze its behavior can be tricky. I have not attempted to trick you in anything that I put in this experiment, but, on the other hand, you should not assume that anything I've given you works perfectly, including the test programs. Always explicitly predict what will happen before you change code, and then always compare actual behavior with your predictions.

6 Questions for This Lab

In addition to submitting your code modifications via the web interface at http://submit.cs.umd.edu, include a text file answering the following questions:

1. What is the efficiency of the virtual machine as compared to the physical machine on the test program? Explain.

2. What is the complexity of your algorithm? O(1)? O(n)? Worse than O(n)? Explain.

3. What is the (measured) overhead of your approach, and what is your experimental design to determine this? Explain.
4. What is the accuracy of your approach and implementation? That is, how close to "perfect" fair-share scheduling does implementation achieve? Explain.

The answers to these questions will help you gauge the degree to which the random-process-then-random-thread approach is a reasonable approach for implementing fair-share scheduling. If you are not satisfied with this approach, you are strongly encouraged to redesign, implement, and evaluate an approach that is qualitatively and/or quantitatively better. Have fun!