

CS 414: Operating Systems
Fall 2007
Assignment #3

Due Thursday Oct 11, 11am

Part 1. (worth 20%) . Short Answer (Answer all 7 questions by yourself).

1. Using the `fork(...)` system call in C, write a program in which the parent creates a child and the child creates a grandchild. Use the `waitpid(...)` system call to synchronize the processes so that the grandchild first displays a message (and then terminates), the child displays a message (and then terminates), and finally the parent displays a message. You should make no assumptions about the relative execution speeds of the processes. You may assume that a simple `printf(...)` statement (in any process) will cause output to be displayed on the standard output device. **Note: you are not required to execute this on a physical machine (but it should be syntactically and semantically correct).**
2. Describe the actions taken by a kernel to context-switch between kernel-level threads. Contrast this to the actions taken by a thread library to context-switch between user-level threads.
3. Assume that you have the following processes to execute with one processor:

<u>Process</u>	<u>Amount of time needed to execute</u>	<u>Arrival Time</u>
0	45	0
1	20	20
2	25	10
3	30	80
4	45	85

Assume that the quantum is 15, and the context switch time is 5 time units with Round Robin scheduling.

- (a) Give a Gantt chart illustrating the execution of these processes.
 - (b) What is the turnaround time for process p3?
 - (c) What is the average response time for all of the processes?
4. The ability of one process to spawn a new process is an important capability, but it is not without its dangers. Consider the consequences of allowing a user to run the process in the following code.

```
int main() {  
    while( true ) {  
        fork();  
    }  
}
```

- a. Assuming that a system allowed such a process to run, what would the consequences be?
- b. Suppose that you as an operating systems designer have been asked to build in safeguards against such processes. We know (from the “Halting Problem” of computability theory) that it is impossible, in the general case, to predict the path of execution a program will take. What are the consequences of this fundamental result from computer science on your ability to prevent processes like the above from running?

- c. Suppose you decide that it is inappropriate to reject certain processes, and that the best approach is to place certain runtime controls on them. What controls might the operating system use to detect processes like the above at runtime?
 - d. Would the controls you propose hinder a process's ability to spawn new processes?
 - e. How would the implementation of the controls you propose affect the design of the system's process handling mechanisms?
5. Choosing the correct quantum size is important to the effective operation of an operating system. Consider a single-processor timesharing system that supports a large number of interactive users. Each time a process gets the processor, the interrupting clock is set to interrupt after the quantum expires. This allows the operating system to prevent any single process from monopolizing the processor and to provide rapid responses to interactive processes. Assume a single quantum for all processes on the system.
- a. What would be the effect of setting the quantum to an extremely large value, say 10 minutes?
 - b. What if the quantum were set to an extremely small value, say a few processor cycles?
 - c. Clearly, an appropriate quantum must be between the values in (a) and (b). Suppose you could turn a dial and vary the quantum, starting with a small value and gradually increasing. How would you know when you had chosen the "right" value?
 - d. What factors make this value right from the user's standpoint?
 - e. What factors make it right from the system's standpoint?
6. The following are common scheduling objectives.
- a. to be fair
 - b. to maximize throughput
 - c. to maximize the number of interactive users receiving acceptable response times
 - d. to be predictable
 - e. to minimize overhead
 - f. to balance resource utilization
 - g. to achieve a balance between response and utilization
 - h. to avoid indefinite postponement
 - i. to obey priorities
 - j. to give preference to processes that hold key resources
 - k. to give a lower grade of service to high overhead processes
 - l. to degrade gracefully under heavy loads

Which of the preceding objectives most directly applies to each of the following?

- i. If a user has been waiting for an excessive amount of time, favor that user.
 - ii. The user who runs a payroll job for a 1000-employee company expects the job to take about the same amount of time each week.
 - iii. The system should admit processes to create a mix that will keep most devices busy.
 - iv. The system should favor important processes.
 - v. An important process arrives but cannot proceed because an unimportant process is holding the resources the important process needs.
 - vi. During peak periods, the system should not collapse from the overhead it takes to manage a large number of processes.
 - vii. The system should favor I/O-bound processes.
 - viii. Context switches should execute as quickly as possible.
7. Prove that the SJF strategy is optimal in the sense that it minimizes average response times. [*Hint*: Consider a list of processes each with an indicated duration. Pick any two processes arbitrarily. Assuming that one is larger than the other, show the effect that placing the smaller process ahead of the longer one has on the waiting time of each process. Draw an appropriate conclusion.]

Part 2. Programming Assignment (Fair-Share Scheduling)

The purpose of this lab 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, WRK implements a general *Multi-Level Feedback Queue (MLFQ)* consisting of 32 priority levels. Realtime priorities (e.g., > 15) are assigned statically to threads. 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. WRK, as with any MLFQ-based operating system, dynamically manipulates thread priorities to attempt to minimize interactive response time, maximize resource (e.g., CPU) utilization, maximize predictability, and in general minimize the time to execute threads.

While the scheduling algorithm in WRK in general does a very nice job at balancing these different goals for most normal operating conditions, there is one particular situation that we'd like to improve on WRK's scheduling capabilities. Currently, 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 will essentially behave as if each processor created 50 threads, or if there was 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 lab, 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, and that Mary creates six CPU-bound threads while Jim creates two CPU-bound threads. With this fair-share scheduling, Mary and Jim should each get approximately 50% of the CPU, which means that each thread of Mary would receive about 8.33% of the CPU and Jim's threads would each receive about 25% of the CPU.

There are a number of different algorithmic approaches, each with different costs and benefits. In this lab, we will implement a relatively simple approach that is reasonably straight-forward but comes with a heavy overhead cost. You are not required to implement *this* particular algorithm that is described below; however, you *are* required to implement a general-purpose solution that achieves the requirements outlined above. With any approach you implement (including the algorithm discussed below), you are required to analyze and discuss the overhead incurred by the algorithm in terms of absolute measured overhead as well as general complexity. (You are encouraged to see your instructor to discuss your design before you implement anything.)

The simple approach is this: given sufficient duration, you can approximate fair-share 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 fair-share. Furthermore, you should try to determine the pros/cons of such an approach – e.g., what's *good* about this approach? What's *bad* about this approach?

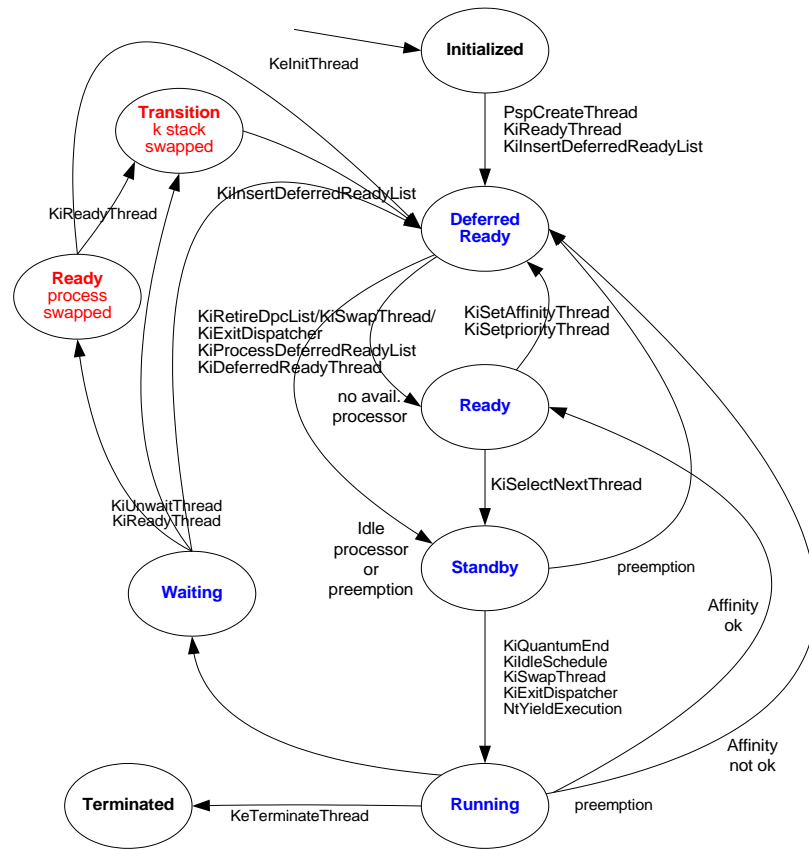
Now, with ANY approach that you implement, you'll find that there are a number of different design choices – e.g., a number of different places in the code in which you might consider making your modifications. For example, you might choose to enforce this fair-share requirement upon every scheduling decision (i.e., right before dispatch). Or you might choose to modify *where* in the queue the thread is placed immediately *after* completing a

quantum (hoping/assuming that this is sufficient to implement a fair-share policy). Furthermore, you might choose to reduce the overhead by not engaging your mechanism *every* time through the scheduler. It is unlikely that any person's implementation will perfectly achieve fair-share, and so trade-offs are expected (you will need to analyze and discuss the pros/cons of your approach). The goal is to get accurate results with regard to fair-share but without tremendous overhead.

While the intuition behind fair-share scheduling might lead to the belief that this is a simple lab, 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 lab. **You are strongly encouraged to do this assignment in a team of two people collaboratively. You can do this assignment by yourself, and no team can have more than two people.**

Background: WRK Scheduler

To help you better understand the WRK scheduler, here's the thread state diagram for WRK:



Kernel Thread Transition Diagram
 DavePr@Microsoft.com
 2003/04/06 v0.4b

Additional information on the WRK scheduler is contained in Windows Internals, 4th Ed., Microsoft Press (authors: Russinovich/Solomon):

- 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.

- To make thread-scheduling decisions, the kernel maintains a set of data structures known collectively as the *dispatcher database*.
- 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.)
- 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 waiting to run, a context switch occurs to the next thread in the ready queue.
- 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 after a wait to a full turn).

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

Getting Started (and further simplifications!)

A general-purpose and comprehensive implementation of fair-share scheduling in WRK would include both the implementation within the kernel as well as the ability of processes to selective request (through a system call) to be scheduled according to the fair-share policy. However, in order to focus our attention on the scheduler only, in this lab our approach will be to assume that ALL “realtime” processes require this behavior. This facilitates a convenient (and pre-existing) mechanism by which to tell the operating system *which* processes should be scheduled according to this fair-share policy: if it's a realtime process, then it will be scheduled according to fair-share. This approach further simplifies the lab, as the realtime class is not subject to the boost/decay behavior described above.

Just as there are many different algorithms to implement 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 realtime 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 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 redesign and implement an 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 realtime thread that you want to execute next, if one exists (via “random process then random thread”)
3. Manually put the selected realtime thread at the front of the appropriate queue

Furthermore, here's a plan to attempt to accomplish Step #2, above:

1. *Find all of the realtime threads in the MLFQ of WRK.* Remember, the realtime priorities are greater than 15.
2. *Divide the threads up according to the processes to which they belong.* That is, each thread structure has a pointer to its process – this step is to help you determine the set of active realtime processes so that you can choose one of them randomly.

3. *Print out which # process and then which # thread you're going to manually select.* This step will help you see (via the debugging window) if your selection mechanism appears to be working.
4. *Put this thread in now at the beginning of the queue.* I suggest that you first build a kernel with steps 1-3 (not this step, step 4) and run it to confirm that your mechanism appears to be working. After this, change the source code to put this thread at the beginning of the queue, comment out your code for step #3, rebuild the kernel, and re-test it.

Where should you make the modifications in the code? Good question! First, study the thread state diagram in the Figure above, and then start looking through *thredsup.c* and *ki.h*. You're essentially looking for existing point(s) in the code where a thread either voluntarily or is forced to give up the CPU. Note: the routine(s) you choose may not necessarily be explicitly identified in the Figure above. While it is not required that you 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 to make your modifications.

Evaluating your Implementation

Testing an operating system scheduler can be very 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 have "strange" behavior if you do not precisely realize 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 realtime processes) in WRK. The program essentially does a slow sorting of numbers and is parameterized by: the number of independent threads performing the sorting, the size of the array(s) to sort, and the clock speed of the processor. This program is available at:

http://www.cs.virginia.edu/~humphrey/cs414/F07/Assignments/Assignment_3/fair_share_user_level_program.exe
http://www.cs.virginia.edu/~humphrey/cs414/F07/Assignments/Assignment_3/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 (i.e., it contains certain invocations to various clock routines).

To create multiple realtime processes, write a batch file (e.g., *start_20_then_10.bat*), like so:

```
start /realtime fair_share_user_level_program.exe 20 1.6 29000
start /realtime fair_share_user_level_program.exe 10 1.6 29000
```

Additional Information

A few final notes before you start any modifications:

- Remember that, while this virtual-machine development cycle is MUCH faster than 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 quick 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 mindset", anticipating errors before they manifest themselves as kernel crashes!

- Do NOT attempt to do everything all at once. I would 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/limit the number of times a debugging statement appears.)
- 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 WRK’s “PerfGetCycleCount” might provide such a clock – you’ll need to think about this carefully before you use it (e.g., does it provide sufficient granularity?)
- Here’s my development setup (which you might or might not follow – it’s up to you, but it might save you some time):
 1. BASH window, pwd is /cygdrive/c/localdata/WRK-v1.2: used for *grepping* files (such as “grep -i RemoveEntryList */*/*/* | less”)
 2. Emacs to edit (with Options → Syntax Highlighting → In This Buffer)
 3. C:\localdata\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:\localdata\WRK-v1.2\base\ntos, used to “nmake x86=”
 5. Virtual machine running
 6. Debugger window (attached to the virtual machine)
- Finally, and most importantly, keep in mind that the scheduler is *very* complicated, and attempting to modify and analyze its behavior can be very tricky. I have not attempted to trick you in anything that I put in this lab, but on the other hand you should not assume that anytime I’ve given you “works perfectly”, including the test program(s). Always explicitly *predict* what will happen before you do something (e.g., change code), and then compare actual behavior with your predictions.

Questions for this Lab

In addition to submitting your code modifications, answer the following four 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? What’s your experimental design to determine this? Explain.
4. What is the accuracy of your approach/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 *random-process-then-random-thread* 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.

What to hand in AND email

You must **both** hand-in your solutions to these questions at the beginning of class on Thursday Oct 11 **AND** email an electronic copy of your solutions to cs414@cs.virginia.edu by 11:00am on Thursday Oct 11. Make sure that both documents clearly contain your name. Each person should submit his/her answers to part 1 individually, and then a team should submit one solution for part 2.