Machine Problem 1: Fair-Share Kernel Thread Scheduler

100 points + 10 bonus points
Due date: To Be Announced

Note: There have been a few modifications with respect to the version previously posted on the web (and discussed in class). These modifications are marked in Small-Cap font.

The objective of this assignment is to add a Weighted Fair-Share (WFS) thread scheduler to the operating system. In this scheduler, each thread is allocated a share of the processor. For example, thread $T_1$ may be allocated a share $S_1$ of 20% of the processor, while both threads $T_2$ and $T_3$ are allocated shares $S_2$ and $S_3$ of 40% each. As a result, thread $T_1$ can use the CPU for roughly half the time of threads in $T_2$ and $T_3$, respectively, while threads $T_2$ and $T_3$ split the time equally. The objective of the fair-share scheduler is to ensure that threads are given their fair share of CPU time by giving less time to threads who have had more than their fair share and more time to threads that have had less than their fair share, respectively. In such a system, each thread appears to be running on a system whose speed is equivalent to the share of the thread. A thread $T_i$ with fair share $S_i$ appears to be running on a processor whose speed is at least a fraction $S_i$ of the speed of the real processor.

Note that the emphasis is on "fair". If a thread is blocked, its allocated share can be re-used – fairly, of course – by the other threads.

Fair-share schedulers can be used in multi-user systems to guarantee a certain level of service to each user. They are also frequently used whenever virtual services are provided to clients, such as web hosting on server farms. Being able to provide a given fair share of any system (be it a web server, a storage farm, network bandwidth, or others) makes it easy for the service provider to formulate service level agreements and to price the service. For the client, buying a "share" of a server capacity is intuitive and isolated from the other workload in the system. It is therefore easy to monitor for compliance: whenever the client has work to do, it will get at least its fair share, independently of how much other work the server has to do.

In principle, a fair-share scheduler could be implemented using a round-robin scheduler. This is very cumbersome, because we would have to probably deal with varying quantum sizes to accommodate differently-sized shares. In addition, the quantum sizes would have to be very short in order to accurately allocate shares to threads, resulting in high context-switch overhead.

Instead, we implement the fair-share scheduler in a so-called greedy fashion. For this, we use the following trick: Assume the CPU bursts that we want to schedule have deadlines. It is well-known that as long as the system is not overloaded (the experts call this "the instantaneous real-time utilization does not exceed 100%") a scheduling algorithm called (preemptive) Earliest-Deadline-First (EDF) can always schedule the bursts so that all their deadlines are met. Now assume that a CPU burst of length 2msec arrives for Thread $T_1$, which is allocated 20% of the CPU. In a CPU with 1/5th the speed, this burst would finish executing 10msec from now. So, as long as we finish within 10msec, Thread $T_1$ gets its fair share. If we do this for all threads, the result is a fair-share schedule. Now we understand how deadlines come into the picture: Whenever a thread becomes ready, we (i) determine the length of the burst, then (ii) assign a deadline based on the burst length and the allocated share for the thread, and finally (iii) schedule the burst in EDF order.

How do we know how long the burst at the head of the queue is? It turns out that one guess is as good as any. Therefore, let’s decide on a WFS-quantum $Q$, and assume that all bursts are of

---

1 Bare with me, this is part of the trick.
length $Q$. If the burst is longer than $Q$, preempt it, assign it a new “lease of life” $Q$, update its deadline, and insert it into the ready queue. If the burst is shorter than $Q$, it simply yields the CPU.

**Side Note:** If the burst length of all threads is an integer multiple of $Q$, everybody gets their fair share. On the other hand, threads with very short burst lengths may receive less than their fair share, because of the following reason: When a thread $T_i$ with fair share $S_i$ becomes ready, it is assigned a deadline equal to $Q/S_i$ into the future. For a small value for $S_i$ the deadline can be rather large. Since many other threads may be ready to run with earlier deadlines, the given thread may not get to run until much later. Its completion time may therefore significantly exceed the “fair share” completion time $C_i/S_i$. In this assignment we do not worry about this source of unfairness.

**Implementation:** In this assignment you are to implement a WFS scheduler, based on the greedy approach described above. The scheduler is based on an preemptive Earliest-Deadline-First and perform the following operations:

1. **Initially, initialize deadline** $d_i$ of thread $T_i$ to be zero.
2. **If thread $T_i$ with fair-share $S_i$ becomes ready at time $t$:** Assign new deadline $d_i = \max(d_i, t) + Q/S_i$. Enqueue thread into preemptive-EDF ready queue.
3. **Whenever thread $T_i$ has been executing for $Q$ time units (possibly with interruptions due to preemption!), preempt $T_i$, assign new deadline, and enqueue in ready queue.
4. **If $T_i$ completes before deadline, update deadline to be** $d_i = d_i - (Q - C_i)/S_i$. This prevents undue penalties for short bursts that cannot fill the WFS quantum $Q$.

The ready thread are served in preemptive EDF order. For this, you need to implement a ready queue that orders threads in increasing deadline. Since deadlines are relative to a – moving – current time, you need to implement a slightly special data structure for this, called a **Delta List**, which is often used to manage timers. The first element in the Delta list contains the time interval to that element. The subsequent elements contain the interval counting from the previous element.

You are to instrument the scheduler to keep statistics on the threads. Keep track of how much CPU time each thread uses and its share of CPU that it received, as well as how much time is spent waiting on the disk. Periodically, print out the time, whether or not the CPU is in use, the number of processes waiting on each queue (individually), and the accumulated statistics for each running thread. These will help in identifying bottlenecks in the system.

Be sure to protect against interference within the scheduler.

**The Assignment**

**Basic level.**

Implement a WFS scheduler. Evaluate the effects on the performance of the system and on fairness of using different values for the WFS quantum. Also determine where the bottlenecks are in the system. What would you change to improve performance?

Evaluate the correctness and the performance of your scheduler implementation on the MIPS emulator. For this you are to use the program `schedtest`, which contains a number of simulated
user threads, some of which are compute bound and some other I/O bound. Since we don’t have I/O implemented at this point, waiting for I/O is implemented simply as waiting for a timer.

Repeatedly run the program schedtest for different values of $Q$.

Advanced level.

For the Advanced Level, you are to submit a class-based WFS scheduler. In many larger systems it is impractical to allocate fair shares of the CPU(s) to individual threads. Instead, shares are allocated on a per-class basis. For example, the threads that deal with incoming requests for a given customer in a web-server farm are grouped into a class and are assigned a share of the server. Threads inside a class are scheduled (in our case at least) in FIFO order. This implementation is not much different from the thread-based WFS scheduler, except that a class may hold on to the CPU even after the CPU is finished: While the remaining portion of the share at the end of the CPU burst in a thread-based WFS scheduler goes wasted, the class can keep holding on to the share until the end of the WFS quantum as long as the class has ready threads.

What to Hand In

- Hand in two files, with names WFSScheduler.H and WFSScheduler.C, which define and implement class WFSScheduler.

- Also, hand in a file (called WFSoutput.txt, in text format) with enough output from a run of schedtest to convince us that your solutions are correct.

This output can be easily generated by redirecting the output of gxemul to a file.

DO NOT hand in the complete output – it is very lengthy. Just hand in enough to show that the system starts up correctly and that each aspect of your algorithm works correctly. Annotate the pertinent lines in the text file and jot down brief notes explaining why your output shows that your solution works. We will not be providing any sample output – we want you to be able to decide on your own if your output is correct.

- Hand in a file (called wfsanalysis.pdf, in PDF format) with the analysis of the effects on the performance of the system when using different values for the WFS quantum. Also determine where the bottlenecks are in the system. What would you change to improve performance? The complete analysis can be made in 500 words or less.

- Make sure that all files contain your name.

- Clearly specify in all files whether you are submitting at the base level or at the advanced level.

- Grading of these MPs is a very tedious chore. These handin instructions are meant to mitigate the difficulty of grading, and to ensure that the grader does not overlook any of your efforts.

- Failure to follow the handing instructions will result in lost points.

Notes about your Implementation

It will be helpful to keep the following points in mind:
The WFS scheduler is realized as an object of class `WFSScheduler`. This class provides a constructor with no arguments, which makes it easy to replace the RRScheduler in the program `schedtest` with your new scheduler.

The main program `schedtest.C` has additional code to set the shares for the threads. This piece of code (and the code to initialize the WFS scheduler) is included whenever the WFS macro is defined. You can either define it in file `schedtest.C` or set the `-DWFS` flag during compilation.

You will have to store thread-specific data (e.g. deadlines, shares, etc.) somewhere. Refrain from modifying the class `Thread`. We have added a field `PObject * cargo` to the `Thread` class, which will allow you to store all the information you need into the `Thread` without having to modify the class `Thread` (or the files `thread.H` and `thread.C` for that matter). You can package all nicely into the class `WFSScheduler`: declare a local class `WFSThreadCargo`: `public PObject with all the data fields you need, and attach it to the cargo pointer in `Thread`.

In the main code in `schedtest.C` some of WFS-specific thread parameters are manipulated using static functions in class `WFSScheduler`. For example, the share of a thread is set using the function

```c
WFSScheduler::set_s_inv(Thread * _thr, int s_inv);
```

You are to implement these functions that manipulate the WFS-specific thread data, which you store in the `cargo` field as described above.

The test program `schedtest` sets up several threads, some of type `ComputeBoundTask`, and some other of type `WebTask`. These threads simulate waiting for IO access or user input by calling a function `wait`, which in turn sets up a timer and waits for it to fire. This gives a very realistic behavior.

The timer subsystem is initialized to assume a 100MHz machine (10 nanosec per instruction).

Look at class `RRScheduler`. Much of what you need for the WFS Scheduler is there.

In implementing the `WFSScheduler`, it may be difficult to inherit much from either `FIFOScheduler` or `RRScheduler`. Maybe you fare better by inheriting from `Scheduler`, and copy whatever code you need from `RRScheduler`.

In terms of code, you are to submit two files only, i.e. `WFSScheduler.H` and `WFSScheduler.C`. These files really should not contain anything that can be found in `scheduler.H` and `scheduler.C` already.

Keep in mind that the WFS scheduler is a priority-driven preemptive scheduler at heart. As a result, whenever a task becomes ready you need to check whether it should preempt the currently running task. If it does, make sure to preempt the currently running thread. (Depending on how you plan to do this, this may be a bit tricky.)
• Finally, keep in mind that this is not a very simple project. It helps if you proceed carefully and step-wise. For example, it may be helpful to first implement only simple non-preemptive EDF functionality into the WFScheduler. This is easy to test. After that, you can add preemption and test again. Finally, add the additional timer handlers that control the WFS scheduler. At this point you don’t have to worry anymore about the correctness of your EDF infrastructure.