EECS 489 Lab 8: Weighted Fair Queueing

This assignment is due on Wednesday, 15 April 2015, 10 pm.

Introduction

One of the main tool to provide resource isolation, either for performance guarantee or infrastructure virtualization, is *fair queueing* (FQ), otherwise known as *generalized processing sharing* (GPS). In this lab, we implement a simplified weighted fair queueing (WFQ) scheduler. The main simplification we make is to assume that once a flow (or job) has started, the flow never goes idle. If a flow can go idle, when it resumes service, we would need to compute the scheduling round at which service is resumed. This computation is complicated by the arrival and departure of other flows during this flow's idle time. Everytime a flow arrives or departs, it stretches or shrinks the duration of a scheduling round by changing the total allocated resources. Hence to compute the round at which a flow resumes service we would have to keep track of the changes in scheduling round during its idle time. This is known as "round catchup" in the literature. By assuming that a flow doesn't go idle, we avoid doing "round catchup" in our implementation.

Another assumption we make is that once a packet starts transmission we do not pre-empt it to serve another newly arriving packet with a smaller finish time. This assumption implies that we account for flow arrivals and departures only between packet transmissions.

We continue to build on our UDP-based client-server code. This code base does not implement flow control, reliable delivery, nor rate control. As usual, you're provided with a skeleton code and a reference Linux binary executable of the server called `refwfqdb`. The client is the same one used in Lab7. The implementation of WFQ requires you to modify only the server code. The provided `Makefile` builds `netimg` and `imgdb`. You can download the support code from the Course Folder and copy the files to your working folder for Lab7 (you may want to save a copy of your Lab7 `Makefile`, `imgdb.h`, and `imgdb.cpp` before overwriting them with the ones for this lab). As usual, you can search for the string "YOUR CODE HERE" in the code to find places where your code must go. You may also want to consult the lecture notes on WFQ and PA4 walk-through. Your implementation of the server **must** interoperate with the provided client.

Recall that the client's `-r` command-line option is used to specify the flow's rate, in Kbps. The server in this lab takes two command-line options `-l` and `-g`. The `-l` option allows you to specify the server's link rate, in Mbps, ranging from 1 to 10. The `-g` option allows you to specify the minimum number of flows that must be initiated before transmission begins. Transmission will not start unless either this many number of flows have been initiated or if total reserved rate has reached link capacity. We call this "gated start". Without gated start, a flow could easily finish transmission before we could start a second one. The default values of the command line options can be found in `netimg.h`

Weighted Fair Queueing (WFQ)

Your first task is to complete the code in `imgdb::handleqry()` to add a new flow. When a new flow is to be added, first look for an empty slot in the flow table to hold the flow. Also check that the link still have enough capacity to serve the flow's reserved rate. If the new flow cannot be accommodated, send back to the client an `imsg_t` packet with `im_type` set to `NETIMG_EFULL`. Otherwise, increment the flow count and total reserved rate of the scheduler and call `Flow::init()` to initialize the flow. This task should take

about 7 lines of code.

To send a packet, you first compute the finish time of each flow's next segment. Complete the 2-line computation for next finish time in `Flow::nextFi()` (Task 2). [To avoid unnecessary arithmetic, we can assume that the finish times computed are multiplied by 128 (1024/8), i.e., the segment size can be in bytes instead of Kbits.] Then determine, in `imgdb::sendpkt()`, the flow with the smallest next finish time and call `Flow::sendpkt()` on the flow to send the next packet (Task 3). This task should take about 10 lines of code.

Your last task (Task 4) is to call `Flow::done()` when the flow has finished sending, as indicated by the return value of `Flow::sendpkt()`. When a flow has finished sending, decrement the scheduler's flow count and deduct the flow's reserved rate from the scheduler's total reserved rate. This is a 2-line task for a total of about 20 lines of code for this whole lab.

Testing Your Code

Unfortunately you may not always be able to see the effect of the scheduling order on the displayed images. You would have to rely on the transmission log instead. The code for logging packet transmission is part of the support code that is provided to you. Do not modify this code.

Run `imgdb` without any command line argument, which starts it with the default link rate of 10 Mbps and minimum flow of 2. Then run two instances of `netimg` on two different shells/windows:

```
% netimg -s localhost:<port> -q ShipatSea.tga -r 256
% netimg -s localhost:<port> -q BlueMarble2004-08.tga -m 8192 -r 1024
```

The two transmissions should complete about the same time (a few hundred microseconds apart) with the BlueMarble flow (`flow 1`) completing before the ShipatSea flow (`flow 0`). Observing the transmission log, you'll see that multiple BlueMarble flow's packets are sent for each ShipatSea flow's. (The display of the two instances of `netimg` may be on top of each other; drag the top image display window aside to view the other display window.)

After quitting the two instances of `netimg` (you can leave the `imgdb` running), start two new instances of `netimg`:

```
% netimg -s localhost:<port> -q ShipatSea.tga -r 128
% netimg -s localhost:<port> -q BlueMarble2004-08.tga -m 8192 -r 128
```

You should see the BlueMarble flow completing much later (well, by a millisecond or so) than the ShipatSea flow and that multiple ShipatSea flow's packets are sent for each BlueMarble flow's.

Submission Instructions

Do NOT use any libraries or compiler options not already used in the provided `Makefile`, to ensure that we will be able to build your code for grading. If we can't compile your code, you will be heavily penalized.

Test your compilation on CAEN eecs489 hosts! Your submission must compile and run **without** errors on CAEN eecs489 hosts using the provided `Makefile`, unmodified. Your server must interoperate with the provided client code.

Your "*Lab8 files*" should comprise only your `imgdb.cpp`.

To turn in your Lab8:

1. Submit the SHA1's of your *Lab8 files* on the CTools Assignments page. (If the URL doesn't work for you, just click the "Assignments" item on the left menu of the CTools page for EECS 489.) Once you've submitted your SHA1's, don't make any more changes to the files, or your SHA1's will become invalid.
2. Upload your *Lab7 files* by pointing your web browser to Course folder and navigate to your `lab8` folder under your uniqname. Or you can `scp` the files to your `lab8` folder on IFS: `/afs/umich.edu/class/eecs487/w15/FOLDERS/<uniqname>/lab8/`.
   This path is accessible from any machine you've logged into using your ITCS (`umich.edu`) password. Please report any problems to ITCS.
3. Keep your own backup copy! Don't make any more changes to the files once you've submitted your final SHA1's.

The timestamp on your SHA1 submission on CTools' Assignments page will be your time of submission. If this is past the deadline, your submission will be considered late. You are allowed multiple "submissions" without late-policy implications as long as you respect the deadline.

Do NOT turn in an archival (`.zip` or `.tgz`) file, instead please turn in your solution files individually. Turn in ONLY the files you have modified. Do not turn in support code we provided that you haven't modified. Do not turn in any binary files (object files, executables, or images) with your assignment.

Do remove all `printf()`'s or `cout`'s and `cerr`'s and any other logging statements you've added for debugging purposes. You should debug using a debugger, not with `printf()`'s. If we can't understand the output of your code, you will get zero point.