ADVANCED OPERATING SYSTEMS — PROJECT 2
Filling in the gaps **OR** A first evaluation of allocating to elastic applications

1 Overview

Due to the limited time left in the semester, you must choose what your second and final project is going to be. Specifically, your group will need to choose one of these experiments:

1. **An extension of one of the papers we have read:** As originally proposed for this assignment, your group may select some experiment that directly extends or modifies experiments in one of the papers we have discussed. The goal is to select an experiment that was not done in one of these papers that you wish the researchers had performed and presented. More on this option in Section 2.

2. **An analytic attempt to allocate resources to elastic applications:** Given the papers we read on *elastic applications* (e.g., CRAMM, compressed caching), we sketched, in class, an approach to determine the efficacy of allocating resources to such applications such that overhead is minimized. For this choice, you would carry out that analysis, as more fully described in Section 3.

3. **Something else:** Since there is already a great deal of latitude on these projects, and because we are continuing to read about a number of compelling ideas, you may create your own project that falls into neither of the two categories above. If you have a different idea of this kind, you must have it approved, but you should certainly pursue that option if you have sufficient interest.

2 Extending existing experiments

For this option, you must select a paper that we have read and discussed and then extend or modify one of its experiments. You are **not** supposed to design a wholly new experiment, since doing so is beyond the bounds of our limited time in this course. Rather, by changing relatively small aspects of existing experiments, and (ideally) obtaining the code and inputs used to perform those experiments, you can produce some meaningful result within a reasonable time.

As an example, consider one paper that we read, The Memory Fragmentation Problem: Solved?. This paper presents four ways of measuring fragmentation, and present some results based on those metrics. You may conceive of another metric that would change the analysis or reveal meaningful patterns about the allocation behavior of various applications. Here, you could re-create the experiments from that paper, adding your own fragmentation metric to compare to the four presented in the paper.

The details of how you would go about such an experiment depend on your choice of paper, the experiment you want to recreate and extend, and the tools and inputs that can be downloaded or obtained from the original authors. If you choose this option, you must select a paper and be able to describe the experimental extension or modification that you intend. From there, the details of what you must implement and how you will perform the experiments must be worked out in consultation with me. The final result will be a short paper describing the new experiment, what it revealed, and how it would affect the conclusions of the original paper.

3 Approximating the efficacy of elastic applications

We discussed, in class, the problem of allocating resources to programs that employ memory management strategies such as *compressed caching* and *dynamically resized garbage collected heaps*—strategies that cause processes to change their memory referencing behavior in response to the main memory allocated to them. These *elastic applications* can trade CPU time for main memory space, and therefore they make the kernel's task of allocating those resources more complex.

We sketched a brute-force strategy for using information about the behavior of each process to perform these allocation in a way that minimizes *overhead*, which is defined as the fraction of running time that the process spends waiting for *disk accesses* and for *CPU-based memory management tasks* (such as page compression and decompression, or garbage collection). What is not clear is that the use of these memory management strategies, and the application of this more complex resource allocation policy, would improve system performance and provide the *gradual degredation* of performance when memory becomes increasingly scarce.

If you select this option, you would create code that would analytically determine, based on simulation results and approximations, the type of benefit that elastic applications and their allocation would gain for a system. Moreover, since compressed caching yields a simpler model with which to work, you would assume that we are apply it to a group of "standard" processes. For inputs, you would need, for each of a group of applications:

- Reference or miss histograms
- Mean compressibility per page
- Mean compression and decompression time per page
- Mean disk access time
- The non-paging running time (that is, the time to run the process when sufficient uncompressed memory is available to store the entire footprint of the process)

With this information, you should be able to calculate, for each possible allocation size, the CPU and disk overheads for the given process. Given groupings of these processes—pretending that they were being executed concurrently—you could then calculate all possible allocations of main memory space to each process, determinining which allocation would yield the least overhead. This result could be compared to calculations in which the compressed cache size is fixed at zero, which is identical to using non-elastic, traditional processes.

A more elaborate version of this experiment could simulate a number of processes at once (by using their reference traces), maintaining histograms that evolve (and decay) over time, and periodically allocating main memory to them. Doing so would reveal how much additional gain is possible by periodic reallocation of resources, rather than the static, single-allocation that the analysis described above would represent.

4 Tools and details

This section contains a grab-bag of information about the tools, inputs, outputs, formats, and other implementation details that will be useful to those of you carrying out particular portions of the project. This section will be expanded as needed throughout the project's progression. 0G

4.1 The database and basic per-application data

In order to calculate the overhead curves for each process, you will be the reference or miss histograms for that application, as well as basic data such as the non-paging running time. These results have already been computed and recorded in our database, and so you need only know how to access that information.

Specifically, on the CS departmental servers and workstations, you should connect to the database named `cs39` on `rigel`, like so:

```
$ psql -h rigel cs39
```

This database is owned by the *cs39* role within the roles-based user/group system employed by PostgreSQL. Each of your subgroups (e.g., *cs39c*) is a role that belongs to the *cs39* role; each of your user accounts is then a role that belongs to your subgroup role. Thus, all of you have full access to this database and its tables.

Once you have connected to the `cs39` database, use the command to list all of the tables within it:

```
cs39=# \dt
```

You will see two tables of particular interest: `attributes` and `lru`. To see the field names for each, use the `psql` command to describe a specific table. For example:

```
cs39=# \d attributes
```

Once you see the field names, you can see how to form a query that will obtain the information you need. Again, for example, to obtain the non-paging running time of the trace *cc1*, you can enter the query:

```
cs39=# select non_paging_runtime from attributes
       where trace_name = 'cc1';
```

These two tables should provide the basic inputs needed to calculate the overhead curves. Moreover, **any new tables that you produce for this project should be created within the `cs39` database** so that all other members of the class can access such tables.

This assignment is due on Friday, 2009-December-04, 9:00 am