15-749/15-449: Engineering Distributed Systems

Project 1: Implementing LRU and Adaptive Cache Replacement in VMNetX

Key Dates

Assigned	Wednesday, January 22, 2014
Checkpoint 1	Monday, January 27, 2014
Checkpoint 2	Monday, February 3, 2014
Due	Monday, February 10, 2014

Instructions

- All projects should be implemented individually, not in groups.
- You are free to give and receive help from anyone in class on high-level design issues, clarification on how something works, tracking down hard-to-find bugs, use of Linux tools, structure of the code base, and so on.
- It is *not* acceptable to copy code from/to someone else or tell/ask someone where lines of code have to be added, deleted or modified. If in doubt, ask one of the instructors to help you, or check whether what you propose to do is acceptable.

Goal of the Project

The goal of this project is to add to VMNetX a cache replacement algorithm for on-disk caching. As currently implemented, VMNetX assumes a local disk of infinite size for use as a cache. Your task is to eliminate this assumption.

This project is divided into three parts:

- For checkpoint 1, you will build and familiarize yourself with the existing VMNetX codebase, create workloads and explore the stream that shows chunk-level trace data.
- For checkpoint 2 you will add LRU cache replacement to VMNetX and a mechanism to report cache evictions. You will then analyze and find workloads that show the operation and effectiveness of the cache replacement algorithm as well as workloads that show where LRU gives suboptimal results.
- Finally, you will replace the LRU cache replacement algorithm with the Adaptive Replacement Cache algorithm and show how it affects system performance and/or usability.

Over the course of the project you are expected to track any changes you make, and submit the code used during the final demonstration by pushing to a provided Git repository.

After completing this project, you will understand in more detail the complexities of caching: how simple high-level concepts may get obscured by tricky implementation details. You will also learn that when caching occurs at many layers there is typically a tradeoff between simplicity and effectiveness.

By the time you successfully complete this project, you will have done the following:

- Explored the code base of a non-trivial system to understand its internals.
- Used the VMNetX system and executed a virtual machine.
- Explored different workloads in the virtual machine and how they can lead to suboptimal cache behavior.
- Implemented two cache replacement algorithms in the VMNetX daemon.
- Evaluated and demonstrated the effectiveness of the new cache replacement algorithms.
- Used the Git version control system to manage your changes and to submit the version of the code you will show during the final demonstration.

Using the VMNetX system

The laptop you have received already has VMNetX build dependencies pre-installed, but the source still needs to be checked out from version control and built.

- Create an account on http://gitlab.cmusatyalab.org/.
- Once you are logged in, navigate to the list of public projects. (a small globe icon at the top)
- Select the project named "15749/vmnetx" and choose 'Fork repository'.
- You now have a private copy of the repository which you can check out and commit to during development.

```
git clone http://gitlab.cmusatyalab.org/<username>/vmnetx.git
cd vmnetx
autoreconf -f -i
./configure
make
sudo make install
```

Once the code is built, you can start a session manually by downloading a vmnetx descriptor from https://vmnetx-images.cmusatyalab.org/. For instance,

```
wget https://vmnetx-images.cmusatyalab.org/fedora-20.netx
vmnetx fedora-20.netx
```

Once the virtual machine has been started it may grab the mouse and/or keyboard focus which can be released by pressing the **ctrl-alt** key combination.

Some VMNetX internals

VMNetX consists of a frontend that handles command line parsing, startup of the virtual machine, and the graphical user interface; and a backend that provides a virtual disk through FUSE and handles I/O requests from the virtual machine. The frontend is written in Python and located in the **vmnetx/** subfolder of the source tree, and should not need any modifications for this project. The backend is implemented in C and is located in the **vmnetfs/** subfolder.

Within the VMNetX implementation there are two main caches. To avoid (or add to) any confusion as you are reading the source, here is an attempt to describe these caches.

One of the caches is the **pristine cache**. This is a persistent cache that stores *chunks* of the virtual disk image for a virtual machine. The main purpose of this cache is to ensure that a particular chunk of data is not retrieved from the server more than once. This cache does not have an active replacement policy.

The second cache is the **modified cache**, which holds any dirty state for running virtual machine images. This cache is a sparse file that holds any modified chunks.

The intent of this project is to add cache replacement to the **pristine cache** part of VMNetX.

When VMNetX mounts a FUSE filesystem it will create a subdirectory to expose virtual disk and memory images, as well as several *streams* which can be read to obtain reference traces of which chunks are accessed, cached, or modified. These streams are, for instance, used to visualize I/O activity on the virtual disk image.

Reads and writes from the FUSE kernel module arrive in **fuse.c** and are dispatched to their appropriate handlers in **fuse-image.c** (for virtual disk image accesses), **fuse-stats.c** (for statistics), or **fuse-stream.c** (for logging data). The virtual disk requests are broken down into individual chunk read/write requests and passed on to **vmnetfs/io.c**, where bitmaps are used to identify which chunks have been accessed during the current session (**accessed_map**), have been previously written to (**modified_map**), or are present in the pristine cache (**present_map**). The present and modified bitmaps are used to decide if the data has to be fetched from the server, or if it can be read from the pristine or modified caches.

Monitoring the existing chunk cache access patterns

When VMNetX is running, the second-to-last toolbar icon opens up a System Activity Viewer widget that shows the number of chunks read and written, as well as information about cache hits. You can mouse over the statistics to get an explanation.

To get actual numbers for offline analysis, you can directly read the stream files under the vmnetfs FUSE mountpoint. A cat of one of the streams shows the index of the accessed chunk.

```
# cat /var/tmp/vmnetfs-XXXXXX/disk/streams/chunks_accessed
101768
101769
101770
82979
82980
82718
```

The vmnetfs FUSE mountpoint can be identified by running **mount | grep vmnetfs**.

Using this we can create an ordered reference trace of chunks which were accessed during the execution of the virtual machine.

Checkpoint 1: Get your own build of VMNetX working on your system [10 points]

For the first checkpoint you must build, install and execute VMNetX from your own source tree. Start exploring the system and capture reference traces for several different workloads in the virtual machine: building a program from source, playing audio or video, running **dd if=/dev/sda of=/dev/null**, etc.

We have also provided a virtual machine image with random data. This image will fail to boot, at which point you can safely interact with the */var/tmp/vmnetfs-XXXXX/disk/image* virtual disk image file from the host.

Checkpoint 2: Implement LRU cache replacement for the VMNetX pristine cache [50 points]

Allow your cache size to be configured through an environment variable **VMNETX_CACHE_SIZE**, which should be an integer value in binary megabytes. Add the necessary hooks so that when the decision is made to fetch a new chunk into the pristine cache, and this would exceed the configured cache size, the oldest discardable chunk is evicted from the cache first. If **VMNETX_CACHE_SIZE** is not a positive integer value, cache replacement should be disabled, giving effectively an unlimited pristine cache.

Take care with tracking used cache space, as multiple read operations may run in parallel, but it is not necessary to worry about multiple VMNetX instances running concurrently. Also, at startup, check if the already used space exceeds VMNETX_CACHE_SIZE and discard enough data to drop below the limit.

Also make sure to add a new vmnetfs *stream* named **chunks_evicted** that outputs the chunk IDs as chunks are evicted.

Finally, explore different workloads and, with your understanding of how certain actions within the virtual machine result in particular I/O patterns in the host, find one or more workloads that show poor cache utilization.

Final: Implement ARC cache replacement for the VMNetX pristine cache [40 points]

Implement the more-advanced Adaptive Replacement Cache algorithm for cache replacement. Introduce a new VMNETX_CACHE_ALGO enviroment variable which can be set to either **LRU** or **ARC** to change which cache replacement algorithm is used by VMNetX.

The ARC algorithm and paper will be discussed in class before this part of the project is started.

You will be asked to give a live demo where you show workloads where ARC outperforms LRU. You must submit the version of the code used during this final demo by creating a merge request for the upstream http://gitlab.cmusatyalab.org/15749/vmnetx project.

Version control using Git

For the final submission of this project we will be using the Git distributed version control system. Although you may develop using any tools you want, it is highly recommended to make yourself familiar with Git and use it to your benefit.

By committing early and often it becomes easier to recover from mistakes, or look back on how you got where you are. It also allows you to work more quickly and more aggressively. Recovering from accidentally removed or broken functionality is much easier when such changes are tracked. Try to create multiple branches as you explore different approaches. Best of all, you can push and pull your work between multiple machines.

For a guided walk-through for Git, see http://gitimmersion.com/.

But a very minimal description follows:

Every checked-out git repository has a full copy of the project history. You can add new files and commit any local changes into your local repository:

```
    git status      # get summary of what has changed
    git diff        # see detailed changes
    git add <file>  # add <file> to the repository
    git commit -a   # create a new commit of your current changes
```

You can browse the commit history with **git log**, or graphically with **gitk**.

To back up your work, or if you want to resume working on the code from another machine, push your commits to your Gitlab repository. Be aware that this will only save committed changes.

```
    git push
```

For the final submission, create a merge request for the upstream vmnetx repository from which your original project was forked. To create the pull request:

- Push your changes to your private repository on http://gitlab.cmusatyalab.org/.
- Navigate to your private vmnetx repository in the Gitlab interface.
- Select “Merge Requests”, and then “New Merge Request”.
- Select your source branch, target 15479/vmnetx repository and master branch.
- Add a title and submit your merge request.