CS 6210: Advanced Operating Systems **February 8, 2012**

Project 2: Simulating the Xen Split Driver Model

Instructor: Professor Karsten Schwan(schwan@cc) TA: Minsung Jang (minsung@gatech)

1. Outline

The goal of the project is to simulate the device sharing model among virtual machines used in the *Xen* hypervisor. This project can be done by teams of 2 students each, and as usual, you are allowed to discuss your ideas with other teams, but sharing or copying from others is **not** allowed. Please refer to the Georgia Tech honor code available at http://www.honor.gatech.edu/.

Turn in your final deliverables for the project on **February 25, 2012 by 11:59pm on the T-square** website. Emails or other methods for submission will not be accepted.

2. Goal

The goal of this project is to simulate the split driver model of the *Xen* hypervisor. The model uses a *ring buffer* located in a memory area that both Dom0 and DomU can share. However, this project asks you to implement this model through *Shared Memory* and at user-level in Unix, since programming codes for an actual hypervisor requires super-user access privileges and is more difficult to debug. Specifically, you will be creating a server and client(s) running at user-level, where the server and clients simulate Dom0 and DomUs, respectively. Your implementation must allow blocking (synchronous) calls and show good performance. The server has to treat all clients fairly. When implementing this system, ***you may not download code or use any other external libraries*** without consulting the TA first. To get full credit for this project you must implement the following:

- A ***ring buffer*** that provides a way to communicate between a server and each client
- A ***service for*** simulating a blocking, synchronous "***read***" operation , which uses the ring buffer
- A ***server*** (providing the service) as a daemon
- A ***client*** application as described below.
- A ***write up*** discussing your design and the performance of your inter-process communication facility
- Adherence to the submission practices
- Proper output schematic as mentioned below
- All codes you submit should work on the Factor cluster machines

3. Details

In *Xen*, there is a domain called Dom0, which, among other things, handles requests from operating systems in DomUs for performing I/O. The operating systems place their data for the operation (e.g., a file read) on a ring buffer, and Dom0 handles the requests in turn. You will be creating an analog of Dom0, which will handle requests for a service from a client, an analog of DomU. However, to make the project simple, the project asks you to let each client have its own ring buffer instead of one global buffer across clinets.

3-1. Service Requirements

You are required to implement a simple service, which takes *a sector number as parameter and returns the data read from the sector*. Other details for this service as follows:

- The service will need to support synchronous calls, which means that the caller has to block until the call is complete and the result is delivered
- The service must run in the background as a daemon
- It must be started as "`./bin/service start`" and stopped as "`./bin/service stop`"
- The service must be able to communicate with multiple client processes through producer-consumer ring buffers, similar to that used in *Xen*, where the client is generating requests and the service is generating results (think of using an ID to match the requests and responses)
- The size of the shared memory ring buffer is up to you. Make sure it is mentioned in your report. It should not be anything absurd and must be able to accommodate a large amount of requests/replies
- All of the data carried in this service must be done in shared memory
- This service must implement fairness among the multiple clients. You may need to implement some round robin method for processing pending requests from clients
- Note that you need a notification mechanism to inform the service about new clients. One of the ways to do this is to use a simple structured file (with locking, of course). You can also encapsulate the client id
- The service should support multiple client connections

3-2. Server Requirements

- A server as a daemon process provides the service to all clients
- Before starting the server, a virtual disk image should be established as below:

```
$ dd if=/dev/urandom of=disk1.img bs=X count=Y
```

where X=1M, Y≤ 500,
A sector number starts with zero at the beginning of the file, and sector size 512 bytes

- It reads a sector on the virtual disk image and then returns it to the client that requests it
- It should terminate if continuing an idle state (i.e., no requests from a client) over five minutes
- The virtual disk image must be opened with "`O_DIRECT`" flag in the server
- It may have cache for offering enhanced performance, but `the size of cache ≤ 0.01 × (the size of the virtual disk image)` (e.g., if the image is 100 Mbytes, the maximum size of cache is 1 Mbytes) You can use any cache algorithm if you mention it your report
    - If cache is available, the hit ratio should be printed out when the server stops

3-3. Client Requirements

Your sample client must exercise all of the functionality of your service, issuing synchronous requests. You are expected to implement a multi-threaded client-process that generates N requests and the thread on which the request is pending simply blocks till the result arrives. That way you can demonstrate the queue management in your service process. Note that all of the threads in a given client-process use the same shared ring buffer with the service and also that there could be multiple client-processes.

- The client must be run as

```
$ ./bin/client <num_of_threads> <num_of_requests for a client>
```

- Before requesting a "*read*" service, the client should ask the server about the first and last sector number of the virtual disk image
- When requesting a sector read to the server, it specifies the amount of "num_of_requests"
- A sector number for each request should be randomly generated within the range of sectors on the virtual disk image
- It terminates with showing the maximum, average, minimum time taken to process each request, standard deviation, and processed read requests per second on the screen
- Once the server returns a request, it should store the request sector number and the return value on individual files. File names are "*sectors.PID_of_client*" and "*read.PID_of_client*," respectively. Your TA will check the integrity of client reads via those files and the virtual disk image

4. Directions

A demo to the TA for this project is not required, but may be requested when necessary. For full credit, therefore, please use the following guidelines for your project:

- We will execute your project automatically in the following sequence:

```
$make clean
$make
$dd if=/dev/urandom of=disk1.img bs=X count=Y
$./bin/service start
$./bin/client <num_of_threads> <num_of_requests>
/*Maybe multiple of these*/

$./bin/service stop
/*Called after all clients terminate cleanly*/

$ ./bin/tool_for_integrity  arg1 arg2 arg3
/* TA tool for checking the integrity of reads*/
```

- Any deviation from this will result in deduction of points
- Non-working code or failing the integrity check will be given consideration if this pattern is followed
- The "stop" command must terminate any/all of your background service processes. We will deduct points if they do not
- Each client should produce output on the screen in the following format as well as two aforementioned files:

max_time_per_request|average|minimum|std. dev.|requests_per_second (in "nanoseconds")
(example)

```
$client 4 100000 [Enter]
212200|120000|100000|234.1|8333.3 /* terminated */
$ls [Enter]
….. read.1234 sectors.1234…. /* two result files */
```

- Your code will be evaluated on a dedicated and isolated workstation, which means that there is nothing that will interfere with running your application. **Each submission will be partially graded based on the performance shown by a client**. Your submission will be sorted out by performance.

The final document should include items as follows:

- Members' name, GT id and role (i.e., who does what)
- Detail description of a ring buffer in the *Xen* hypervisor (You must mention and point *Xen* source codes)
- Your architecture design (if you decide to introduce a cache in the server, please explain that)
- Your efforts for optimizing performance
- Results

5. Submission

Include all reports, all source codes including Makefile, etc., in the top (and same) directory, compress them into a ZIP file, and then upload that on T-Square.

6. Hints and Resources

You should also become familiar with shared memory functions such as the following (look at the Linux man pages for the description):

- `shmget`, which creates a file for shared memory use OR gives a handle (called a key) to access it;
- `ftok`, which maps a filename to a key for use in `shmget`;
- `shmat`, which allows ("at"taches) a piece of shared memory to the calling process's address space;
- `shmdt`, which removes ("de"taches) a piece of shared memory from the calling process's address space.

You can also consider the following memory mapping functions, which can be a replacement for `shmget`:

- `shm_open`, which creates a file for shared memory use OR gives a handle (called a file descriptor) for accessing it;
- `mmap`, which maps a file descriptor taken from shm_open to a piece of memory;
- `munmap`, which `unmaps` the same file from the process's memory;

The figure below explains how each component in the project interacts with others.