Vinay Bharadwaj – ID – Primary code implementation
Kelsey Hawkins – 902177434 – Code inspection, cleanup, and report

Xen Ring Buffer Design

To allow for different types of ring buffers based on the data being passed, an abstract interface is implemented in Xen using macros. The implementor of the ring buffer provides two types representing the requests and responses in the buffer. Calling *DEFINE_RING_TYPES* creates the necessary definitions for the ring. Both of these types are unified into a single entry and in the **_RING_INIT* macros, a shared memory object provided by the implementor is populated with a number of these entries based on the size of the page *PAGE_SIZE.*

The client begins by putting a request in the ring. The macro *RING_GET_REQUEST* returns the address of the request to the client to be filled. After writing the request, it is pushed into the ring using *RING_PUSH_REQUESTS_AND_CHECK_NOTIFY*. At this point, the server, when available, can receive the request, process it, and queue up a response. The request is again read using RING_GET_REQUEST, processed by whatever task is handling the memory sharing, and added back into the ring using *RING_GET_RESPONSE* which returns a response object to be written by the implementor. Finally, the *RING_PUSH_RESPONSES_AND_CHECK_NOTIFY* macro is called and the written response is reinserted back into the ring.

For optimization purposes, the ring is of an optimal base-2 length. This makes indexing the ring easier by using a simple bit mask for modular arithmetic.

System Architecture

The server begins (in *server.c)* by opening the disk image and creating a shared memory file to store the ring buffer. The ring buffer object consists of *buffer*, an array of *struct buf* objects which can contain either a request or a response, along with values holding queue positions and other clerical objects. We chose the length of our buffer to be around 10000 because it is not incredibly large but can still support the magnitude of requests in the examples. We should expect that many of those requests will be serviced before more will be added. The ring buffer is written to disk and memory mapped to a pointer. The server enters the service loop where *process_requests* is called repeatedly. When the ring buffer has available requests, the server processes them with *read_request.*

The *read_request* function grabs the oldest request *next* from the ring buffer. Since we are using the O_DIRECT flag, we first allocate a memory aligned pointer and read the desired sector from the disk image into *result.* The result is copied into the ring buffer and the indexes are incremented. At this point, the client threads are awoken through a signal to check for the new response.

The client begins by opening the shared memory ring buffer and launching all of the threads. Each thread begins by writing a request in the buffer for a random sector (*write_request)*. The client begins waiting for responses to be ready by spinning on *pthread_cond_wait.* Once *response_ready* is signaled and flagged to *true,* the result is looked up in the buffer. The result's data is written to the *read.** file. Finally, time statistics are accumulated. Once all of the client thread have joined, the final statistics are printed.

Optimization Efforts

We have found that disabling the O_DIRECT flag and using the kernel's built in buffers is probably one of the best ways to increase performance. However, since that is not allowed, we kept the flag enabled. Although the assignment mentions implementing a cache, we note that this will probably not increase performance under the client testing we were asked to implement. Since each request is purely random, the performance gains from sequential reads of the same sectors will never be seen. You should only expect that at best 1% of reads will hit the cache. For this reason, we did not implement a cache. We did find, however, that adjusting the size of the memory aligned pointer helped. By sizing the pointer to multiples of the page size, we seemed to get better read times.