CS 6210
Project 2 Report

Simulating the Xen Split Driver Model

Group Members

Sam Britt (901-75-9808)
- Architecture design
- Ring buffer implementation
- Client registration and file service implementation
- Architecture design writeup

Yohanes Suliantoro (902-76-1560)
- Architecture design
- Client synchronization and statistics calculations
- Experimental design and analysis

Ring Buffers in Xen

Ring buffers in the Xen hypervisor [1] are shared memory, asynchronous I/O rings designed to provide a consistent interface to devices for all domains ported to Xen. These ring buffers are allocated by the domains and contain pointers to out-of-band I/O requests, rather than the requests (and responses) themselves. All domains' device access requests pass via the ring buffer to the privileged *Domain0*, which interfaces directly to the hardware. Communication via the ring buffer is really two asynchronous producer/consumer problems occurring simultaneously in the same shared memory space: In one, the domain produces requests and Xen consumes them, and in the other, Xen produces responses and the domain consumes them. Thus there are four pointers into the shared ring, one for each of these processes. Domains employ a hypercall to instruct Xen to process new requests, and Xen notifies domains of complete responses using an asynchronous event notification system similar to signals; that is, Xen calls a handler in the domain that was registered previously for this particular type of I/O access. There is no guarantee of the order in which requests will be processed. Since Xen has more complete knowledge of the hardware than do the domains, it can re-order requests if doing so will improve performance. Each request has a unique identifier that couples it to its response. Also, request and response production are decoupled from notification (in either direction). That is, a domain can generate many requests at once before notifying Xen. Similarly, Xen can process batches of request for a domain before notifying the domain of their completion, and the domain can specify that it wishes to be notified only when Xen completes a threshold number of requests. This allows for tuning the trade-off between throughput and latency.

The bulk of the ring buffer implementation in Xen is done by macros in the file /xen/include/public/io/ring.h in the Xen source. The use of macros allows for rings to be created for arbitrary request and data types, employing the C preprocessor token pasting techniques to generate the proper data types and the

code to manipulate them at compile time. As will be described in more detail below, our implementation uses similar techniques to allow for code reuse in both the client registration process and the file read service. Another interesting feature of the implementation is that all ring buffer sizes are required to be a power of two. This way, the current index of any of the four pointers into the ring never has to be "wrapped" around to zero when it increases past the size of the ring via a modulo operation. Instead, the indices can increase indefinitely, and are masked with a bitmask of size-1 to leave only the relevant (least significant) bits of the index.

In contrast to Xen, our design uses *synchronous* I/O rings in shared memory. All disk access request passes through server's ring buffer. These ring buffers are allocated by the server, and contain space for the actual request and response data. Our design is two synchronous producer/consumer problems in the same shared memory. First, clients put requests on the I/O ring, which is picked up by server's thread. The server processes the request, and puts the response on the ring. Then the same client thread will pick up the response. Since the process is synchronous, we only need 1 pointer for the server and a pointer for each client thread. Client requests will be done in the order it is received. The server does not perform any batch processing. All notification between client and server is performed using shared memory semaphores, mutexes, and condition variables.

Architecture Design

The server consists of one registration thread to handle client registration and one file server thread to manage file read. In addition, each client process registered with the server communicates over its own ring buffer; there is a server worker thread to manage each of these.

Client Registration Service

Before clients can make file read requests to the server, they must register with the server and establish a line of communication. Our design uses the same ring buffer technology used in the file read service communication for client registration. The process is shown in Figure 1. At startup, the server creates a registration ring buffer (shown in green in Figure 1), which is a block of shared memory with a published name (this is part of the server interface). The server also spawns a registration thread to manage this ring buffer (each ring buffer in our design has its own manging thread; see also the section below on providing the file read service). While the registration ring buffer is empty, the registration thread blocks, waiting for clients to request registration with the server.

When a client process wants to register with the server, it first must map the registration ring buffer into its own memory address space. It then places its request, in the form of its process identifier (PID), in the next available slot in the registration ring buffer (there can be many clients with pending registration requests), and then blocks until the registration process is complete. This is step 1 of Figure 1. In step 2, the registration server thread must do several things. First, it allocates a new shared memory ring buffer to be used for all of that client's file read requests (shown in blue in Figure 1), and it spawns a server worker thread to manage it. The shared memory name for the new buffer is in a published format which incorporates the client PID; in this way, the client can determine the name to use when mapping it into its own address space, and also be assured that no other process will be using it. The registration thread then places a pointer to the new ring buffer in the global list of registered clients. This list is used by the main

file server thread to respond to file read requests; see below in the section on the file read service. Finally, the registration thread places, in its own ring buffer, the minimum and maximum sector number for the file to be serviced (this is the "response" to the client's registration request), and notifies the client that registration is complete. At this point, the client thread can unmap the registration ring buffer, and map the new file service ring buffer into its own memory address space. Any new file read requests will be put into this ring buffer (step 3 of Figure 1). The registration thread can then move to the next pending request in its ring buffer and repeat the registration process for the next client.

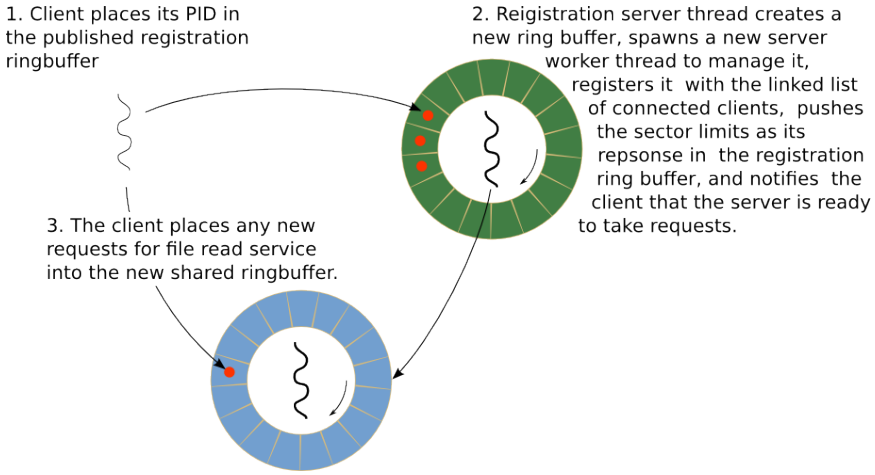

Figure 1. The client registration process: (1) The client registers its process ID with the published registration ring buffer (in green). (2) The server registration thread then allocates a new ring buffer for the clients file read requests (in blue), spawns a server worker thread to manage it, adds it to the file read work queue, and notifies that the client that the registration process is complete. (3) The client can now place its file read requests in the newly allocated read buffer.

File Read Service

Each ring buffer created by the registration thread is assigned to a worker thread that acts as a thin interface between the client and the file service thread. The file service thread performs the actual work of reading the file from disk. At a high level: Each client places its requests into its own shared ring buffer, the server worker thread for that buffer places the next pending request into the work queue of the file read service, and the file read service thread loops through each the connected clients in a round robin fashion, reads the next pending request from each client in turn, and puts the response data directly into the shared ring buffer for that client. The entire process can be seen in Figure 2. Details of the implementation follow.

In step 1 of Figure 2, a client places its request in its ring buffer. Requests are simply in the form of a sector number that the client would like read, which must be between the limits established beforehand. The client places its request in the first empty slot after all previous requests still pending; if the buffer is full, the client will block until a slot becomes available.

In step 2, the server worker thread places the next pending request from its client in the work queue of the file service thread, and will block until the request has been serviced. The work queue of the file service thread is in the form of a circular linked list. In this list, each node corresponds to an entire client ring buffer/worker thread pair, one node per pair. Each node includes a pointer to the element in that ring buffer with the next pending file read request for that particular client; for *n* registered clients, there can

be at most n pending jobs for the file service thread at any one time.

In step 3, the file service thread picks the next pending job in its work queue. The file service thread moves through this list in a round-robin fashion. That is, once the file server processes the request from the client i, the next request to be processed will be that from client $i + 1$ (if there is such a request pending), regardless of how many requests are pending in the ring buffer for client i. In this manner fairness is enforced: every client can be guaranteed that its request will be serviced in $O(n)$ time if there are n registered clients, and no one client can dominate access to the server. This layer of indirection between the request the client makes and the actual work performed provides also provides for stability — a malicious (or buggy) client can only affect the ring buffer and worker thread assigned to it, without the ability to bring the entire service down.

In step 4, the file service thread reads the data from disk corresponding to the requested sector; the data is read directly into the ring buffer for the client. The file service thread can then wake the blocking worker thread and notify it that its request has been completed. At this point, the file service thread can move to the next pending request in its work queue, or block if there are none.

In step 5, the worker thread for the ring buffer notifies the blocking client thread that its request has been completed and its data is ready. It must then block until the client has completely copied or processed its data. In step 6, the client copies the data to its own memory address space, and notifies the blocking worker thread that the ring buffer slot is now available for use for another request. The worker thread finally move on to the next pending request in its ring buffer (or block if there is none), and the entire process repeats.

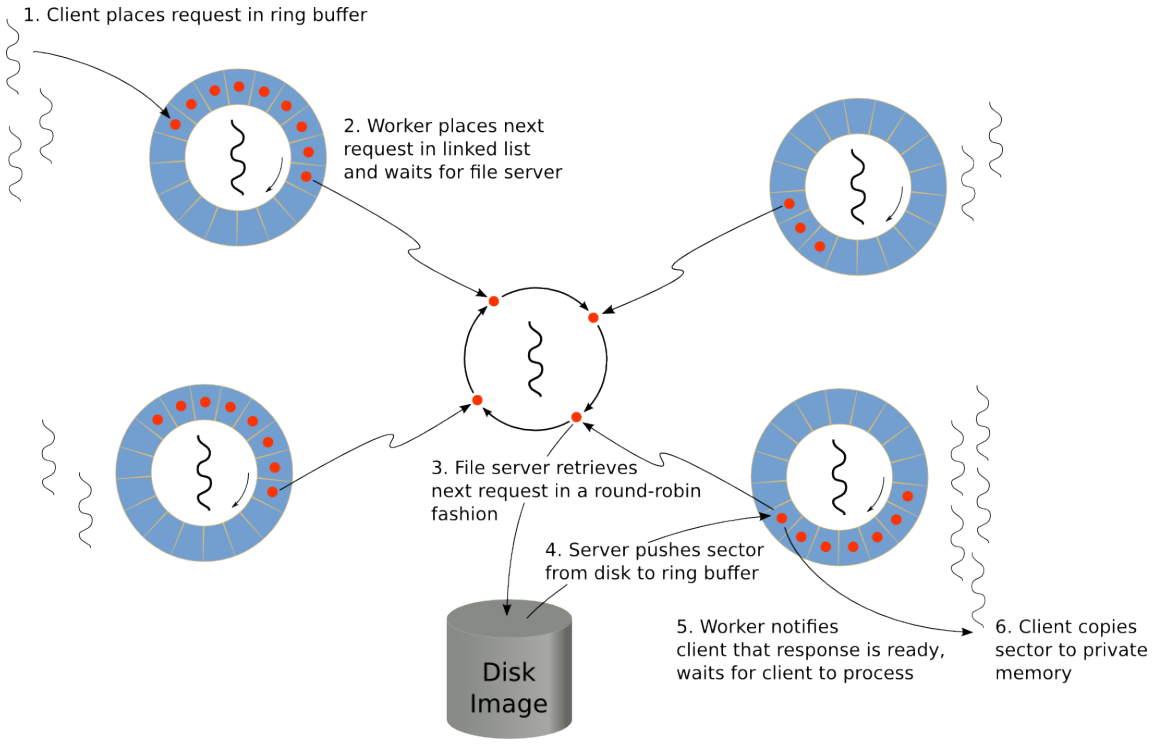

Figure 2. The file read process: (1) The client places its request (file sector number) into its ring buffer. (2) The worker thread for that ring buffer places its next request in the file service work queue, a circular linked list. (3) The file service thread gets its next request and (4) pushes the sector data directly to the clients ring buffer. (5) The worker thread notifies the client that its request has been completed, and (6) the client copies the response to its private memory space.

Results and Optimization

We do not implement cache because with the random nature of the client requests, a cache would not make much improvement to performance. Since we do not implement cache, the variable that we can modify to optimize performance is the ring buffer size; that is, the maximum number of slots available for client requests in any ring buffer. Based on our experiment, which can be seen below on Table 1, using 100 ring buffer slot count gives a good performance. Thus, we use 100 slot count for the rest of the experiments.

The second experiment on Table 2, we varied the number of client threads and number of total requests on a 100-slot ring buffer and 1MB file. We use 10000 and 100000 as number of total requests to simulate lighter and heavier load, respectively.

Table 1. File Server Performance Using 100 Client threads, 100000 requests, and 1MB file

Client slot count	Max Latency (μs)	Avg. Latency (μs)	Min Latency (μs)	Std. Dev. Latency (μs)	Throughput (requests/s)
10	239752	118516.931	2316	5520.517	843.07
100	8832	113468.016	2554	4397.961	880.86
1000	129138	112950.950	2527	4146.525	884.87

Table 2. File Server Performance Using Client ring slot count of 100 and a 1MB file

Client Threads	Total Requests	Max Latency (μs)	Avg. Latency (μs)	Min Latency (μs)	Std. Dev. Latency (μs)	Throughput (requests/s)
10	10000	15024	11384.955	2170	831.513	877.94
10	100000	29533	11516.229	1312	807.8	868.24
100	10000	126097	113049.869	1584	6892.599	879.86
100	100000	135497	114308.035	1340	4450.233	874.36
1000	10000	3121053	975754.025	1765	476539.107	1028.23
1000	100000	3158899	1120931.946	1325	119346.902	887.45

The last experiment that we did is to vary the file size under heavy load to see how it affects performance. Thus we use 100 ring buffer slot, 100 client threads, and 100000 total requests on file sizes from 1 to 500 MB.

Table 3. File Server Performance Using 100 client threads, 100000 total requests, and 100 ring slot count

File size (MB)	Max Latency (μs)	Avg. Latency (μs)	Min Latency (μs)	Std. Dev. Latency (μs)	Throughput (requests/s)
1	135497	114308.035	1340	4450.233	874.36
10	218628	115952.489	2314	5163.315	861.96
100	129600	116955.023	2282	3706.755	854.59
500	138287	119372.210	2487	4596.009	837.30

References

1. Barham, et al. (2003), "Xen and the Art of Virtualization," *Symposium on Operating Systems Principles*, Oct. 19-22, 2003, Boston Landing, New York.