Thursday, February 3. 2011

Synchronicity in asynchronicity

Sometimes you just think ... "Hell ... this shouts for being misunderstood". Asynchronous filesystem semantics and Asynchronous I/O are such concepts. Sounds the same, but isn't. And often people get it wrong, even when you not talking about the application, but just about the filesystem.

As you know there are two important concepts of executing writes. Synchronous and asynchronous or to say it differently blocking or non-blocking. When you trigger an asynchronous write, the write call comes back, right after you issued the call. An synchronous write call (by definition) is only allowed to come back, when the system has assured (as far it's technically possible) that the data is on some kind of nonvolatile storage.

Synchronous writes are absolutely essential for example for mail servers: The MTA can't send OK to the MUA or to another MTA the "OK, got the mail", as long it's sure the mail is nonvolatile on the other server, because the sending MTA deletes it from its queue after such a OK. A mail could be lost, when the power fails after the OK, but before the non-volatile storing of the mail. So you send the OK afterwards. How to you ensure, that the OK is after the nonvolatile storing? Yes ... it's done by a sync write.

It's the same for databases: One of the important foundations of ACID is the synchronous write. Without synchronous writes, forget about the D ...

However a synchronous write has a problem and it is because of its greatest advantage. It doesn't come back until the write has been completed. The problem is that you can't do anything else in this time in this thread.

Translating this to an example in the real world, you can imagine this like being the dispatcher of large group of employees. When you give your employees their tasks synchronously, you give your employees the job and wait for the completion before you dispatch another job. You can be sure, that the task has been completed before you throw the the piece of paper with the todo, however it's not really efficient.

Switching back to asynchronous writes isn't an option as well, as the only way to check, if the write call was really processed through the complete code path and the code paths inside the HBA and the controller adjacent to the disk, is to read the location (okay, they are other reasons to do so, however that a completely different discussion). The asynchronous write call gives you no feedback of that the write call has been completed in terms of writing stuff to nonvolatile storage. Basically you trust the operating system that it will do everything right and aside of this there is absolutely no guarantee that asynchronously written data is on disk after a system crash.

Translated to the real world example, this would be like giving a task to an employee, throwing away the piece of paper where you wrote down the task ... and forget about it. Obviously you can work very fast this way, and often this works well, however when an employee quits or is absent due to illness, the job may not completed.

By the way: It's the same for reads. A reads blocks the calling process until the data has been delivered, and this time there is no way around with it with normal means. Reading is synchronous by necessity. The process can't say "Well ... when i have to wait for it ... i will take some other data available"

So, waiting processes until the write comes call returns, missing guarantees and no feedback on the other side. And reads synchronous anyway. And even more devastating: When a write or read call blocks a thread, because it waits until all the stuff between the OS and the disk has been done, no matter how many parallel disks or filesystem you have. How do you get out of this challenge?

There is a way around that and this is Asynchronous I/O. The read and writes are still synchronous or asynchronous in the sense of blocking or non-blocking. So you can have basically an synchronous write in a asynchronous I/O model. This is important: As a colleague said it years ago: You can configure Oracle to use a asynchronous I/O model, but you can't tell Oracle to use non-synchronous write semantics. Remember the D.

To use this model, there are a number of new calls in the OS that support this asynchronous I/O modell.

The write call aiowrite or read call call aioread returns right after you issue the call. Sounds like asynchronous write

semantics. However there is an important difference: At first the read or write is now run in a concurrent thread and the aioread or aiowrite returns immediately after they are issued. But contrary to the asynchronous write semantic, you get a feedback about the result of the write.

There are essentially two important ways to yield this information, at first you could call the aiowait function. This returns as soon as a outstanding read or write triggered by aiowrite or aioread returns. It waits either forever (until an outstanding aio request completes), not at all ("Is something completed available ... no ... okay ... let's go further". You use this mode of operating to implement a polling mechanism) or wait for a certain time.

The other way is to implement a signal handler. Whenever a I/O request made by those calls, they will send the signal SIGPOLL, the signal handler can then dequeue the notification of a completed I/O request with aiowait (in fact, they have to because it's the only way to get those notifications out of the queue)

However: Both mechanisms notify (SIGPOLL fires, aiowait returns), when one of outstanding asynchronous requests is completed. They doesn't report the I/O call that has been completed. You have to find which one. Most often this is done by scanning the return code buffers of the aiowrite/aioread requests after you have initialized them with a value expressing "in progress". When a i/o request has been completed, it's set to something else, so you just have to scan for return code buffers, that are not on the state "in progress).

Translated into the real world, it's like dispatching a steady stream of tasks to your employees. However you don't monitor their progress yourself, but you have some colleagues for that. Those colleagues just send you a notification "Task xyz has been completed with that result" and could throw the piece of paper with the task away, as soon as this notification was send to you. You could do it at a jour-fixe, you could wait in front of your colleagues or you could wait for 5 minutes for completion notifications and then do something else for the next hours.

The advantage of asynchronous I/O is obvious: The application can issue I/O requests without waiting for the completion of others, you can even issue asynchronous reads. By doing so a single application thread can have multiple I/O requests in flight. However you addition you get still a feedback, that the I/O request has been successfully completed and you can use it in areas where you would have opted to use synchronous aka blocking write semantics in the past. Important to know: Your application must be enabled by the developers to use an asynchronous I/O pattern.

Posted by Joerg Moellenkamp in English, Solaris, Technology at 08:21

Good article!

In what way is a synchronous write really synchronous in a VM in VMware? Is a disk commit really a commit to volatile write cache in the underlying FS of the ESX server?

Some benchmarks I have done comparing a VM in WMware to a bare metal server suggest it is so. Faster (small) writes in the VM suggest that the application is fooled into thinking that the write is commited to non volatile storage when it in reality is commited to memory.
Anonymous on Feb 3 2011, 12:01

this is the first time someone explained this whole synchronous/asychronous thingy in an understandeable way. Great!

Just one question: would one have one writer thread in an application and one or more "monitoring" threads to implement aio* effectively? Or the other way round? Or "depends on"... ?
Anonymous on Feb 3 2011, 15:52

I've recently done some testing on VMware (4.1) for our oracle db migration.

Using OEL 5.5 as the OS and iozone as the test app, I was specifically looking at io latencies for sync, async and sync to an array that then did sync replication to a remote array. The FS was standard ext3 with no real tuning.

The latencies reported were all as expected: the async returned immediately, the sync took 0.5ms to the array and the sync w/ sync copy took 1.0ms to the array. This was the same as our existing non-virtualized boxes.

In this case the array was an HDS 2500 w/ 32G of cache, so the write was done as soon as it hit the mirrored array cache, not when it hit the disk.
Anonymous on Feb 3 2011, 19:08

Similar setup here with USP-V and a large cache. Could our different outcomes be caused by me using VMFS and you RDM? I guess RDM will make the commit happen in the disksystem(the cache) but not so for VMFS?

Anyway it seems I have to read up on this thing called VMWare. It is everywhere.

Anonymous on Feb 3 2011, 19:42

I wasnt using RDM, just good old files on a VMFS that were then presented up to the host via the paravirt sas driver and then formatted w/ ext3.

In theory, RDM would be faster, but only by a percent or two while significantly increasing the administrative overhead.

The main goal of my test was to see what impact TrueCopy (sync) would have vs our existing host level mirroring. Sure enough, the latency doubled. If VMFS was caching the write then we would not have seen an increase in latency between the replicated and non-replicated lun.

Anonymous on Feb 3 2011, 21:51