Operating Systems: Assignment Two.

By Thomas Adam

17th January 2005

Contents

Question 1

DMA is an acronym for *Direct Memory Access*. Traditionally, when a device wanted to transfer data to and from memory (typically a hard-drive) it was the job of the CPU to do it. But this was extremely inefficient since it meant that the CPU was occupied entirely with handling that, and thus was unable to do anything else.

Bus mastering DMA is a means by which devices can transfer information directly to and from memory, by-passing the CPU completely[1]. This helped speed up the transfer from peripheral devices such as hard drives and graphics cards, by allowing the CPU to be doing other things, rather than handling the actual transfer[2]. The concept works like this: the *master* is the device which drives the bus. This also implies that the device perform the I/O (input and output of data) is capable of performing much more complex operations. This makes sense, when you consider that it is operating without communicating with the CPU. As a result, often these device have a built-in microcontroller, or even their own CPU[3].

But as with anything in the technology industry some standards were more popular than others. *Bus mastering DMA* never really got used properly as it was hoped it might. The operating system support was either flaky or non-existent, with more often than not, buggy drivers[4]. Also, the support for it was dependant on certain types of motherboards, although this was not much of a problem as they were the de facto used in the PCs at the time. Yet dispite that, it was widely used, and with success. Especially when *UDMA* (Ultra Direct Memory Access) was used.

UDMA is an extension of DMA. Aside from general improvements, the main advantage of UDMA is the fact that it can handle large transfer rates along the bus. It was always a well known contention of DMA that the data transfer rate was slow[5]. UDMA came into its own when working with hard drives along the IDE interface. As technology improved, there was a demand that the data rate increased along with it.

There were a lot of additional features behind the scenes with the UDMA implementation. One of which was the use of a CRC (cyclic redundancy check). The concept of adding this in to the data transfer was an innovative idea. Because of the addition of CRC, this meant the data could be checked for errors. This was done for two reasons. One was to ensure the integrity of the data, and the other was to compensate for redundant data that is usually sent with each block of data. Of course where any errors were indicated, the block of data is resent.

UDMA defines relative modes of operation. As mentioned, the increase gain in newer hardware meant that the data rate offered by the hard drive had to be supported. Table1[6] shows typically the modes and the amount of data that can be formed in a single I/O operation at any one time.

---

[1] The term *completely* is slightly erroneous. It only appears as such – but the CPU is still used to negotiate the flow of the information to and from memory. Its input is removed from acting as the middle man. Also, the CPU still has to be notified when the operation has been successful.

[2] http://en.wikipedia.org/wiki/Bus_mastering

[3] ibid

[4] http://www.pcguide.com/ref/hdd/if/ide/modes_DMA.htm

[5] That's relative to the technology at the time. Back then, it was considered to be fast.

[6] http://kerneltrap.org/node/422?PHPSESSID=55ddf431bcdf5e299d2b7aa5893d381b

UDMA Mode	Maximum Transfer Rate (MB/s)
Mode 0	16.7
Mode 1	25.0
Mode 2	33.3
Mode 3	44.4
Mode 4	66.7
Mode 5	100.0

Table 1: UDMA and maximum transfer rates.

Security with UDMA is only a theoretical concern. The data, when it is travelling down the bus, is in binary form. There is always a possibility that if the bus became overloaded, that the data could somehow "leak" and become available via the device transferring the data. But this presupposes that there was a means to do that. UDMA has techniques in place to prevent that – if the data is corrupted it is resent, and if the data is being sent too quickly, then the bus is told to reduce its speed accordingly[7].

Of course, that's just down the bus. If the I/O leak occured when the data was being mapped into memory, then there could be a problem, even more so if it were in the virtual memory area. Data is only ever recognisable if it is stored in a format that can be translated. In the VM, the paged data is stored in the swapfile which is in a structured order. If this data were not swapped back to the I/O device (for some reason), it would remain there indefinitely for someone to use (or until the page was reaped by whatever means the OS happens to use).

So whether security is a concern with regards to UDMA is questionable. I'd say at best that as UDMA performance increases, it might become necessary to consider it if the amount of swapping between the device and memory is great enough.

[7]Depending on the UDMA mode, this could effectively mean to reduce back to DMA.

Question 2

As with (U)DMA earlier, I/O operations are performed on hard drives. When data is sent to the drive there is a means by which the data is written to it. How that is determined depends on the policy used to schedule the way the disk writes its data. There are a number of ways of doing this, but mostly there are six main types employed on IDE drives.

First Come, First Served (FCFS)

In this scheduling policy, the disk processes the data it receives (I/O requests) in the order that it receives it in. With each request, the disk head is having to move backwards and forwards across the disk with each read and write. This has the advantage that the data is written to the disk immediately as it is received, and further defines the order of how data is to be written to disk. If the amount of data the disk is scheduled to commit is too large, then queues are used to buffer the data. Of course, main advantage of using FCFS is that there is no starvation. Because each request is handled as and when it is received, there is never (or highly unlikely) the probability that the data will not be written to the disk[8].

The disadvantage in using this method is that there is no reordering of the work queue. This means that if something goes wrong with the buffering of the data, then the whole lot is lost. But also, there is performance to consider. If the disk is receiving I/O requests without any ordering, then this is inefficient, as it could mean that there is short and long gaps in the way the data is processed (meaning the disk head is continually having to move over the disk).

Shortest Seek Time First (SSTF)

This technique improves upon FCFS, and in many respects is similar to it. Unlike FCFS, SSTF works by servicing requests to the disk which are adjacent to its current position. This has the advantage that data can be serviced quickly. There is data reordering of the buffer so that data to the disk can be serviced adjacent to its postition. This reduces seek time on the disk.

The two main disadvantages of this is that starvation can occur. It's perfectly possible that when the disk is serving several requests that it stays in one particular area. If this happens over a period of time, a request to another area is ignored.

SCAN

Using this method, the drive head continually moves backwards and forwards over the disk. The scan starts from the outside working in. As a request is received, the data is written to it, only when the head is moving in the correct direction. This has the advantage that there is no starvation as the whole of the disk is accessed, regardless. The disadvantage of it, is that the perfomance to the drive is reduced.

[8]"Advanced Unix: A programmer's Guide", Prata, S., SAMS, 1971

LOOK

This is very similar to SCAN but the head of the disk stops in the direction it is travelling when there is no more data in that given direction. This has the added advantage that disk seeking is reduced.

Circular Scan (C-SCAN)

This is very similar in operation to SCAN, but the head moves back and forth across a cylinder[9]. The request is only satisfied when the head of the drive is sweeping outwards to the other-end of the cylinder. When the head moves back inwards, no requests of I/O are performed, even if some are waiting. This reduces starvation, but the disadvantage is that it is slow.

Circular Look (C-LOOK)

Similar to C-SCAN, this also uses a method like LOOK whereby the head of the drive only satifies requests in a given direction. But unlike C-SCAN, the boundaries are not defined, and so it is possible for a C-LOOK to beyond the cylinder limit is is processing if the request means it can come back across to its origin.

As to which method I would employ, I would not use a method which is prone to starvation. Since operating systems are becoming more advanced, and the need to store data quickly increases, using a method prone to starvation is not something I'd recommend. This rules out using SSTF since out of all of the methods described, this is prone to starvation the most.

It could be argued that this is the perfect disk scheduling method to use though in situations such as embedded applications where the emphasis is not so much on the number of processes running[10] but on efficiency. Since the ordering of the buffer takes place relative to where the disk head is at that moment, for an embedded system with limited processes this would certainly be a consideration.

However, for an actual operating system in use daily, other considerations have to be used. Under Unix-like operating systems the drives are split up into logical partitions. These are effectively sections of a disk that are reserved for an area of operation. In such instances, a C-LOOK scan would be ideal, where the disk is usually concentrated in one area, but not always. And unlike the boundary limitations that C-SCAN has, this not applicable to C-LOOK. Moreover, of course, is the fact that starvation has been addressed by using this disk scheduling policy – and although it might appear to be the case, on ever multi-user operating system, I/O to a drive is essential to almost any process that is running. Therefore, starvation cannot be allowed to happen.

[9] A cylinder is a set area of a disk.

[10] In embedded systems, this is not a concern as such, since they concentrate on smaller tasks.

Question 3

With the progression of the lastest stable release of the Linux kernel going from version 2.4.X to 2.6.X, there have been many improvements, both in terms of hardware support, and internal architectural support. One such feature has been the addition (and first implementation) of NPTL (Native Posix Threading Library).

A thread, in computer terms is very similar to a process, except that a thread often is stateful and does one task in paralell with another to achieve a set goal. In Linux systems[11], this is achieved through the use of threads because of the way the kernel operates in relation to the operating systems. The kernel is monolithic in nature, which means that it is esssentially one big program[12] which runs in its own protected environment. The way it communicates with other processes is in an area known as *user-land.* This area is what the user is able to control in terms of starting and stopping programs, etc.

In the 2.4 kernel, the threads that applications created (as a result of the clone() function) were managed by the threading manager. It was the job of the threading manager to context switch betweeen each threads, given an appropriate timeslice. Indeed, while there is nothing wrong with that, the problem that the 2.4 kernels had was that in addition to the context switch that took place, regardless of whether the thread had been looked at before or not, any data structures associated with that thread had to be reloaded into memory[13]. This is an extremely slow and ineffecient means to deal with threads. Of course, the knock-on effect this had was that there was an effective saturation limit between the number of threads that could start, and the amount of memory allocated to handle them.

There were also a number of other issues associated with this implementation. Under Linux, there is a virtual directory called */proc* which details the state of the kernel, the hardware, and processes which are running at any one time. Because of the way the threads were handled these threads were being reported as processes, which is incorrect. Information obtainable in this way (to the user of the system) is both pointless and unnecessary[14].

To combat this, and to try and standardise the failing problems inherent in the 2.4 kernel, the 2.6 kernel saw the introduction of NPTL. This specification rewrote how the threads were managed, specifically with regards to the concerns already mentioned. The biggest change was the use of pthread() which effectively made the context-switching of threads a lot faster by caching data structures[15].

In terms of how this has helped developers, it has meant that the use of threads can really be a viable alternative to having to shift some application logic of the kernel to userland. This reduces a number of security concerns, since by not having some of the code in userland, they're not susceptible to SIGKILL[16] or SIGTERM[17]. Furthermore, threads in the 2.6 kernel have allowed developers to not have to worry about dynamic

[11] Note that we cannot say "Unix" here (or BSD for that matter) since NPTL is an addition to the Linux kernel only.

[12] While the technique of using modules dispells this, it is only relative to use the term *monolithic* when comparing it to other kernel types such as a *microkernel.*

[13] http://people.redhat.com/drepper/nptl-design.pdf

[14] Ibid.

[15] http://kerneltrap.org/node/422?PHPSESSID=55ddf431bcdf5e299d2b7aa5893d381b

[16] "man 7 signal" – UNIX man page.

[17] Ibid.

structures. The fact that the data structure is available at each context switch, means that more threads can be spawned where necessary without the overhead of having to slow the memory usage down.

The use of modules (in terms of hardware accessible code) has meant that now these are on-demand loaded, unlike 2.4 kernels that required the modules to be loaded whether the device in question was in use or not. This operation is controllable via threading.

In a general sense, the impact this has had on developers is no more so than what was present in 2.4. Most of the changes have generally been behind the scenes standardisations, without which, the threading scheme in 2.6 would never have worked. Although a test[18] has shown that in 2.6, the creation of 100,000 threads took a little over two seconds.

[18] http://kerneltrap.org/node/422?PHPSESSID=55ddf431bcdf5e299d2b7aa5893d381b

Question 4

Fast User Switching[19] allows multiple users on a machine to exist so that different users can leave the running state of their applications open, safe in the knowledge that they will be safe when the user returns. Fast user switching is considered an extension of session management, where the application's state is never saved on exit, because the programs are left in situ, running.

Mac OS X uses the same concept, although variations of it exist for Windows 2000 upwards. For fast user switching to work, there must have to be a lot of spare system resources necessary to support a number of users that would have open applications running.

Applications, even in their idle state, use up memory where pages have been allocated. Since the application is still running, these pages are never freed, and so a considerable amount of resources in terms of memory is used. As a developer (and where it were a possibility that fast user switching was being used) there are a number of things that could be done to try and avoid doing that. Certain swap partitions allow for state to be saved to disk in a manner the application can then use when an an operation is performed on it[20]. If this could be utilised when the applications are used in fast user switching, this would allow for an application to suspend its operation until such time it needed to be resumed.

Memory management of the application is a concern. Since fast user switching is subject to multiple users, the only memory that can be used is global to all of them. If a poorly written program were to leak memory while in this idle state, this could affect not only the running application (and the user environment it is contained in), but also affect the state of the other programs. Although this is only a concern of the operating system, the means by which a memory-leaked program is dealt with should be considered. Since Mac OS X uses a Unix-like model it is likely that some form of the OOM-Killer[21] might be used. In that case, as a developer, it would be necessary that the application in question had some means of saving recovery data for fear of the program being killed.

Security is another issue. In any multi-user environment there should be a mechanism in place to prevent others users from accessing other people's work. Mac OS X uses a Unix-like model of security. By that, it is clear that the operating system will predominately define this, and so it is not really an application-specific feature to worry about. However, it might be useful to password protect running applications, based on a timeout delay.

Shared resources (and files) is something else to consider. In situations where access needs to be granted to more than one user to a file, then there should be as a means to do this. This is linked closely with security though, as this presupposes that the set permissions have already been set by the administrator. There are dangers in doing so, and as a developer, it would be wise to address the possibility of buffer overloads, where data is shared – since if data could be leaked in a state when the user was not at the computer, this would be a potential risk.

[19] http://www.apple.com/macosx/features/fastuserswitching/

[20] This is synonymous with page referencing, but for this to work the application must support such a feature.

[21] http://www.rossfell.co.uk/~rickp/oom/

Bibliography

The following references were used throught this text:

Websites

http://en.wikipedia.org/wiki/Bus_mastering
http://www.pcguide.com/ref/hdd/if/ide/modes_DMA.htm
http://www.apple.com/macosx/features/fastuserswitching/
http://rossfell.co.uk/~rickp/oom/
http://people.redhat.com/drepper/nptl-design.pdf
http://kerneltrap.org/node/422?PHPSESSID=55ddf431bcdf5e299d2b7aa5893d381b

Books

"Advanced Unix: A programmer's Guide", Prata, S., SAMS, 1971
"man 7 signals" – a Unix man page.