Virtual Memory, Disk Devices

1 Virtual Memory

- Virtual memory describes methods that give processes more memory than is physically available, or that makes the computer appear to have more memory than is physically available. There are several VM techniques; **paging** is the one most commonly used these days.

1.1 Why Use Virtual Memory?

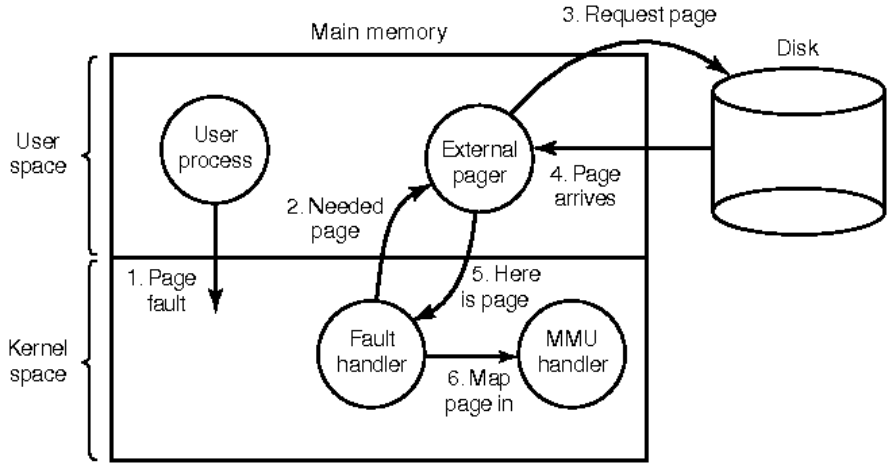
- The memory management methods shown in the last two lectures arise because of one basic requirement: the entire logical address space of a process must be in physical memory before the process can execute and while it is executing. In many cases, the entire process need not be in memory:
  - Parts of the process (e.g error handling) get used rarely.
  - Often statically declared objects (e.g arrays) are only partially used (e.g a list of records in an array).
  - If we could get the operating system to have in memory only the needed bits of a process:
  - Users could write programs bigger than the size of available memory.
  - As each process would use less physical memory while running, the operating system could fit more processes into physical memory.
  - Less I/O would be needed to load a process into memory.
- The sections of memory currently in use by a process are known as its *working set*, or as its *locality of reference*. The whole point here is to keep the working set in memory (which is fast), and to leave the rest of a process's memory out on disk (which is slow).

1.2 Paging

- A paged architecture has a memory granularity less than the process size. Thus, if we can determine what pages are not needed immediately, we can write them out to disk, and release these pages for other pages to use. In other words, we copy to disk the contents of pages which are not part of any process' working set.

1.3 Paging – How It Works

- The operating system either has an abundancy of free pages, or is running short of free pages. As new processes start, or existing processes grow, they request pages from the operating system. If the operating system has spare pages, it allocates them to the requesting process.
- However, when the operating system is running short of free pages, the operating system looks through the list of used pages and selects pages that it believes won't be used for a while. It writes these pages to the disk, **scrubs** the pages, and gives them to the process that requested them. If the selection mechanism is good, then eventually only the *working set* of all processes is left in memory.
- What happens when a process tries to access a page of memory that has been written to disk? The missing page is marked as **invalid** (it's no longer in main memory), and so a **page fault** occurs when the process tries to access it. The operating system realises that the page is on disk, so it finds a free page frame, and reads the disk copy back into the new page frame.
- Note that the operating system must know *why* a page is invalid, and if invalid because it is on the disk, exactly *where* it is on the disk.



(from Tanenbaum)

- After copying out a page to free it, the operating system **scrubs** the page for security reasons – if it didn't do this, a process requesting memory would receive pages with information from another process.
- A paging optimisation: if the operating system can tell that the page is **clean**, i.e unchanged since it was last read from disk, there is no need to copy the page out – it is still there.

1.4 Hardware Requirements

- To provide paging, the operating system needs a method to pick suitable pages to page out. How can it tell if a page is dirty (i.e modified), or if it has been accessed recently? Even better, which pages have been accessed recently?
- The MMU on paged systems keeps this information. For each page, it keeps an accessed (A) bit and a modified (M) bit, which are turned on when there is a read or write memory access, respectively. The operating system can look at these bits to help make a decision. It can also turn the bits off as required, e.g when a page frame becomes free.

1.5 Optimal Page Replacement

- The best paging algorithm is to choose the page to copy out which is the one that will be used the **furthest** in the future, especially if this period of time is infinity. Obviously, the operating system cannot see into the future, so we can't implement this method.

1.6 Not Recently Used Algorithm

- We use the A and M bits in the following manner. When a process starts up, all of its pages have A=0, M=0. Every Nth clock tick, the A bits are set back to 0, to distinguish those pages not recently used. The M bits are left untouched to indicate the **dirty** pages. The hardware sets the A and M bits as described before.
- When a page is needed, the operating system categorises the pages into four categories:
  - Class 1: not accessed, not modified (clean)
  - Class 2: not accessed, modified
  - Class 3: accessed, not modified (clean)
  - Class 4: accessed, modified
- Class 2 indicates that a page was changed a while ago, but not accessed recently (for N clock ticks).
- NRU picks a page from the **lowest** non–empty class, i.e. Class 0, or Class 1 if 0 is empty etc. The justification here is that it is better to page out a modified page that hasn't been accessed recently, as against a clean by heavily used page.
- The main attraction of NRU is:
  - It's easy to understand
  - It's fast to implement
  - It provides reasonable paging performance, but not optimal

1.7 FIFO Algorithm – First In, First Out

- FIFO assumes the oldest page is the one least likely to be needed. The operating system keeps a list of pages: the one at the head is the **oldest**, the tail holds the page most recently allocated by the operating system. When a page must be copied out, the **oldest** is chosen.

- This is simple to implement, but it doesn't check page use, so a heavily used page may be paged out.

1.8 Second Chance

- Second chance is a variation on FIFO. When a page is needed, look at the oldest page. If it A bit is 0 (i.e it hasn't been accessed recently), use it. Otherwise, set A to 0 and put it on the tail (as if it had just been allocated).
- What this is doing is looking for a page that has not been accessed in a long time. If *all* the pages have been referenced, second chance degenerates to FIFO. Both FIFO and second chance are simple, but usually worse than NRU, as an old page is not the same as an unused page.

1.9 Least Recently Used (LRU)

- LRU is a good approximation to the optimal algorithm. It assumes that a page heavily used in the recent past will also be heavily used in the future. When a page is needed, pick the page *unused* for the longest time.
- This isn't easy to calculate. The operating system needs to keep a list of all used pages, with the most recently used page at the front and the least recently used at the back. The difficulty here is that the page list has to updated after **every** process memory access. This is too expensive.
- Instead, the operating system periodically checks for page use (A bit), and reorders the page list. This can still be expensive because the operating system must check every page for recent use. There is a tradeoff here between the frequency of list updates and the accuracy of the result. Fewer list updates is inaccurate but more efficient, more list updates is accurate but expensive CPU–wise.
- Although LRU is more expensive to implement than other page choice algorithms, it performs better, and thus is the algorithm most used.

1.10 VM Problems

- Obviously, disk I/O is costly. If we page in/out too often, we waste time waiting for the disk.
- Optimisation: when a running process needs a **page in**, block it and select another process. Then when the page arrives, make the process ready to run.
- Virtual memory makes the machine appear to have more memory than it actually has. However, it's better to have more **real** memory, thus eliminating the need for paging.

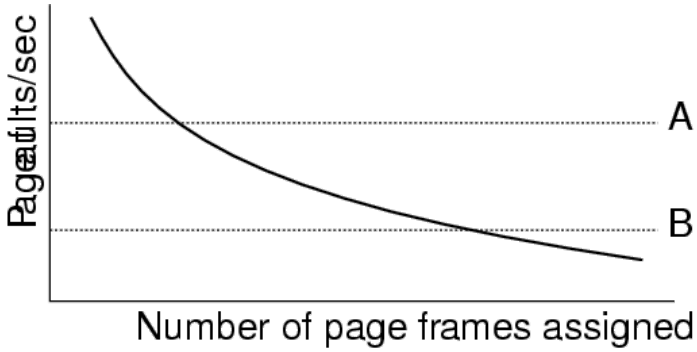
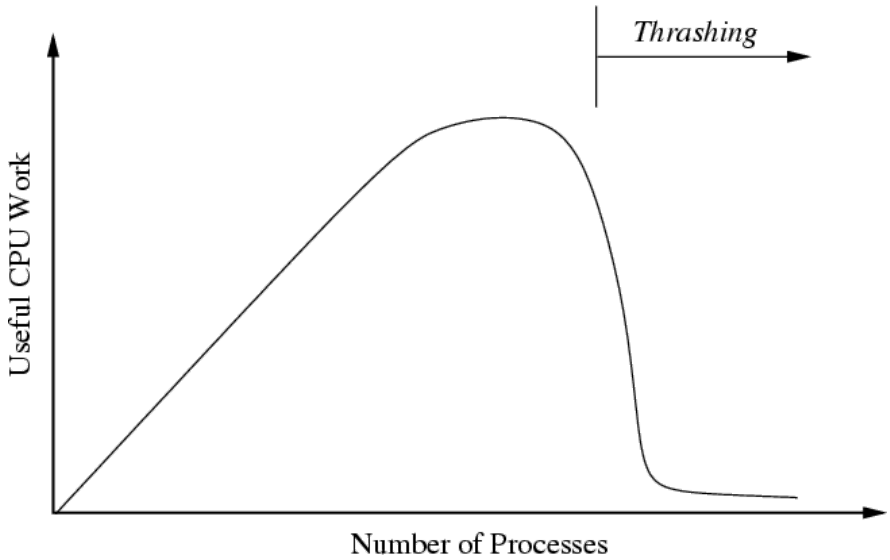


Figure 4-18. Page fault rate as a function of the number of page frames assigned.

(from Tanenbaum)

- Virtual memory increases the CPU utilisation, especially when most processes are I/O bound, because there can be more I/O bound processes in memory, increasing the use of the CPU. But, if all pages are in use, and paging becomes continuous, the machine spends all of its time paging, and no work is done. This is known as **thrashing**.



- With an interactive operating system, there is no easy way to limit the number of processes, so you have to live with the possibility of thrashing.
- Sharing code pages and other pages *copy-on-write* can help minimise paging.
- Instead of waiting for *no* pages to be free, most systems start paging out pages when a *low-water mark* is reached, e.g 5% of total memory. Paging cuts in earlier, but can cope with peaks in memory demands better than `desperation' paging.

1.11 Initial Process Memory Allocation

- This falls somewhere within the following methods:

  Lazy Page Allocation:
  Mark every page as **invalid**, including all the code pages. Every time a page is **needed**, one is allocated and possibly pages in from disk. This maximises free memory, but can be slow to start up a process. This is also known as **Demand Paging**.

  Prepaging:
  Allocate **every** code and global data page. Only allocate new pages when the global data of the stack grows. This minimises paging but wastes memory.

2 Device Drivers and Interrupt Handlers

- We now switch our attention to how disks work and how the operating system accesses them.
- As we saw back in lecture 5, peripheral devices are typically accessed through a set of memory-mapped **device registers**.
- Some of the registers are control or status registers: they accept commands from the OS, and provide status bits that detail the current state of the device.
- Other registers are data registers: they accept data which needs to be sent, or hold data which has arrived.
- While we can poll a device's registers from kernel mode, this wastes a lot of CPU time.
- Instead, devices usually send in interrupt to indicate that they need attention, and set values in the status registers to indicate what sort of attention.
- Controlling a device, from the operating system's point of view, is done with two separate pieces of code:
  - a **device driver**, in the top-half of the kernel, sends device-dependent I/O commands to a device by placing values in the control registers as well as output data in the data registers.
  - an **interrupt handler**, in the bottom-half of the kernel, reacts to interrupts from the device. The interrupt can indicate that new data has arrived, or a previous device command was successful, or even that a previous command has failed.
- A device driver and associated interrupt handler need to communicate with each other. This is usually performed through memory shared by the various parts of the operating system, or via messages in a microkernel system.
- There may be several instantiations of a single device driver in progress, because several processes have made I/O requests. The device driver must:
  - Convert the abstract request from the process into a matching device command.
  - Send the command to the device through its controller's registers.
  - Move the current process to the blocked list.

- All of the above is done by the top-half of the operating system at the behest of the calling process. All of this is done at interrupt priority level zero. This implies that the operating system could pre-empt the process (and the I/O operation in progress) anywhere in the above three lines. This is how several instantiations of the driver can exist at any one time.
- The device driver may also be interrupted by its interrupt handler. Therefore, both the device driver and the interrupt handler must be **re-entrant**: the code can be in use by several threads of execution, and all non-private data structures need to be properly shared. Note that on a single CPU machine, only one device driver instantiation or the interrupt handler is actually in execution; the others are temporarily suspended.
- Because of the multiple driver instantiations, any access to shared memory by the driver or the interrupt handler must by *synchronised*, to prevent the corruption of shared data structures. Under many systems, this is achieved by *masking out* the interrupts that may pre-empt the device driver or start the interrupt handler. This may disable all interrupts below a certain level, and is known as *raising the interrupt level*. Here is an example from the FreeBSD IDE disk device driver:

```
                                  /* queue transfer on drive, activate drive and controller if idle */
s = splbio();                     /* Raise int. level to stop block I/O */
disksort(dp, bp);
if (dp->bactive == 0) wdustart(du); /* start drive */
splx(s);                          /* Return int. level to previous level */
```

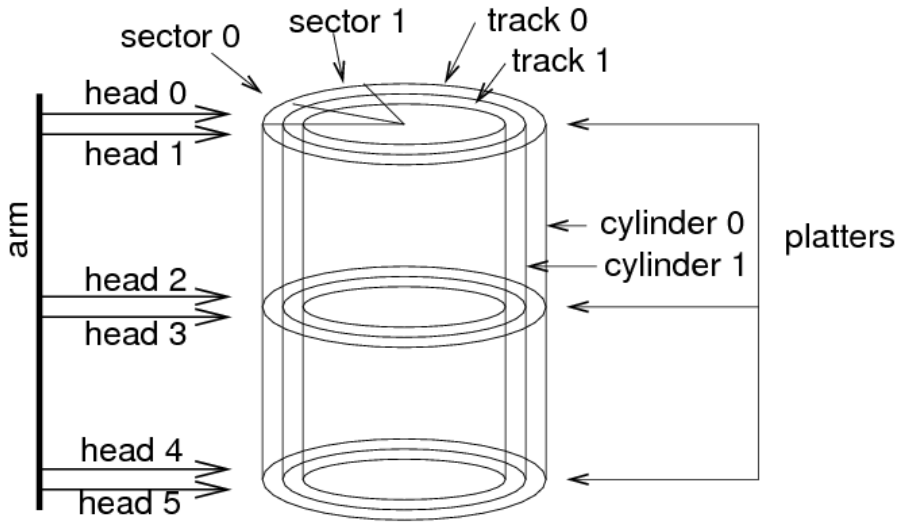
- Device drivers provide a set of operations to the device-independent I/O layer. Here are some of the device driver operations provided by Unix device drivers. In Unix, a `character' device is a `non-block' device.

Function	Device Type	Description
d_open	Block and character	Initialise device when first used
d_close	Block and character	Used when device is released. May
		shutdown device or take it off-line.
d_strategy	Block	Read/write interface, allows event re-ordering.
d_read	Character	Reads data from device.
d_write	Character	Writes data to device.
d_ioctl	Block and character	Generic control operations on device.

- Writing and debugging devices drivers and interrupt handlers is a real pain. There is no protected process environment, and many device drivers or interrupt handlers cannot be single-stepped or stopped at a breakpoint. We will look at disks as an example of peripheral devices.

3 The Disk Device

3.1 Disk Hardware



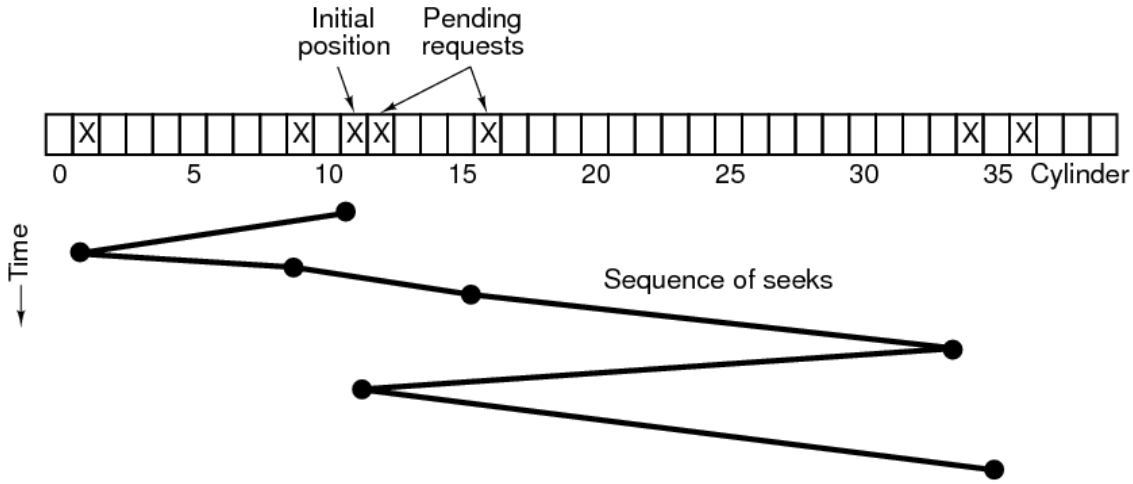
- Hard disks consist of a number of **platters,** each of which is flat and circular. Each platter **platter** has 2 surfaces, and both are covered with magnetic material which is used to record information. Disks spin at high speeds, 7200 rpm or higher.
- For each available platter surface, there is a read/write head, which records a **track** on the surface. There are usually hundreds or thousands of tracks on each platter. The head is mounted on an **arm**, which moves or **seeks** from track to track. The tracks form concentric circles on the platter's surface, and are invisible to the naked eye.
- Each track can hold a lot of information (1M to 100M), so tracks are usually broken into **sectors**, each holding a portion of the data. Sectors can store a fixed amount of data, generally 512 bytes or sometimes multiples like 1K, 2K or 4K. A vertical group

of tracks is known as a **cylinder**. Cylinders are important, because all heads move at the same time. Once the heads arrive at a particular track position, all the sectors on the tracks that form a cylinder can be read without further arm motion.

- To access a track, the arm must seek to it. The average **seek time** on drives is 10–50 milliseconds. Then, the disk must rotate to bring the data to the head: the **latency time**. Finally, the data is read: the **transfer time**.
- Generally, a disk's seek time >> latency time >> transfer time.
- On a multiprogramming operating system, an operating system may have a queue of requests, as several processes may have passed through the running state and issued a disk I/O operation. Disk operations take a long time, when compared with the speed of the CPU and main memory. If software can minimise the overall I/O time, then disk I/O performance can be improved. Let us look at some algorithms that can schedule the motion of the arm.

3.2 Arm Scheduling – FCFS

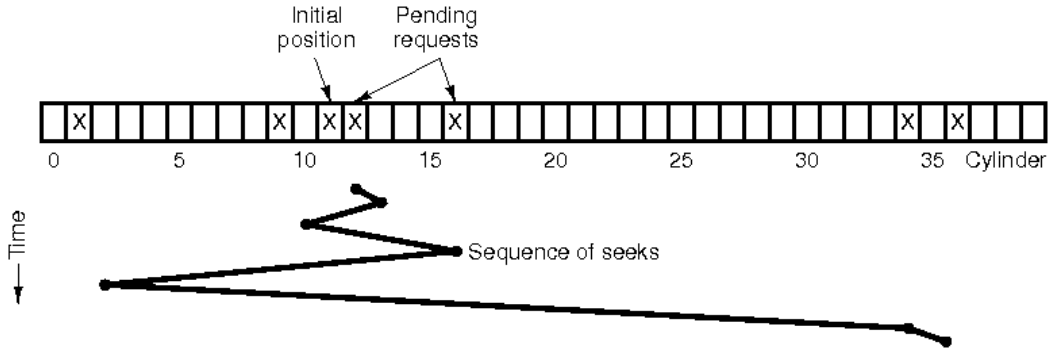
- With this disk scheduling algorithm, disk requests are serviced as they arrive. Because we can't predict what proceses will make requests when and for what, the set of disk requests on a normal system usually causes a lot of head movement. However, the algorithm does treat all requests equally, so the method is fair.
- Consider a set of pending requests to cylinders 11, 1, 9, 16, 34, 12 and 37.



(from Tanenbaum)

3.3 Shortest Seek Time First

- If the queue if pending disk requests is bigger than one, we could improve the disk's performance by reordering the requests so that the request with the smallest seek is chosen to be next.



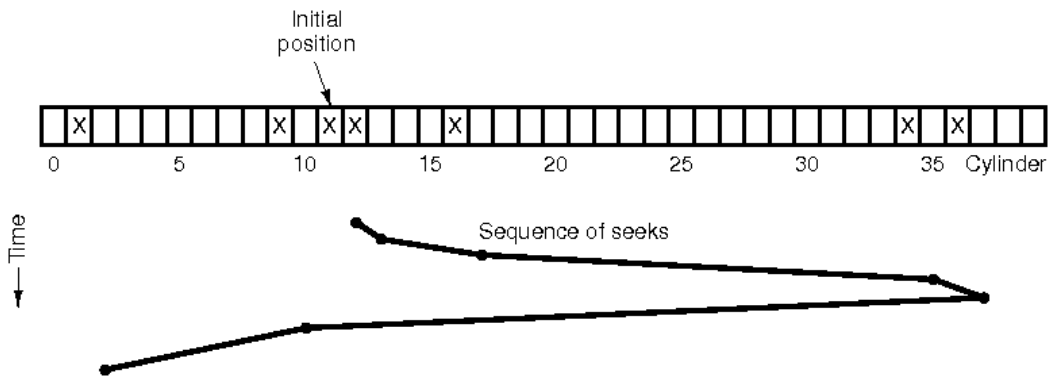
(from Tanenbaum)

- By reordering the queue in this way, the movement of the arm (and hence the seek time) minimised. Thus, on average, a pending disk request is serviced faster than with FCFS. However, this reordering or requests can lead to starvation on big seek requests, especially if new requests are continually arriving. This most affects disk requests for tracks on the extreme edge of the disk, whereas the middle tracks are preferentially selected by the algorithm. In other words, the algorithm is unfair.

3.4 SCAN Algorithm

- This queue reordering algorithm is also known as the **elevator** algorithm. Here, the head starts at one end of the disk, and moves towards the other end, servicing requests in that order, until there are no more requests in that direction. The arm then

reverses direction, and services requests the other way.

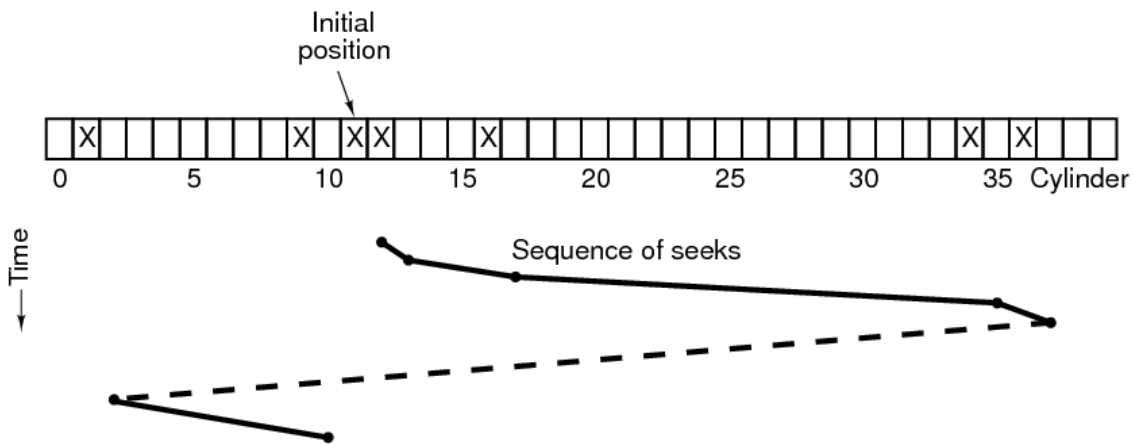


(from Tanenbaum)

- Because requests are reordered so that they are performed in ascending or descending order, this again helps to minimise seek time, and thus improve overall disk I/O performance. One nice property of SCAN is that, given any collection of requests, the upper bound on the motion of the arms is fixed at exactly 2 * the number of tracks.

3.5 C-SCAN Algorithm

- This is a modified SCAN algorithm which lowers the average response time. Unlike SCAN, C-SCAN always services requests in the **same** direction, for example, ascending. This helps to make coding of the algorithm easier. When there are no more requests above the last request, the head is moved to the lowest request, and requests are again serviced in an ascending motion.



(from Tanenbaum)

- Note the following: if the queue is usually of size 1 or less, then no request reordering scheme is going to be useful, and you end up with FCFS by default. But on most multiprogramming systems, disk request reordering is useful, and generally C-SCAN is the preferred algorithm.

4 Direct Memory Access

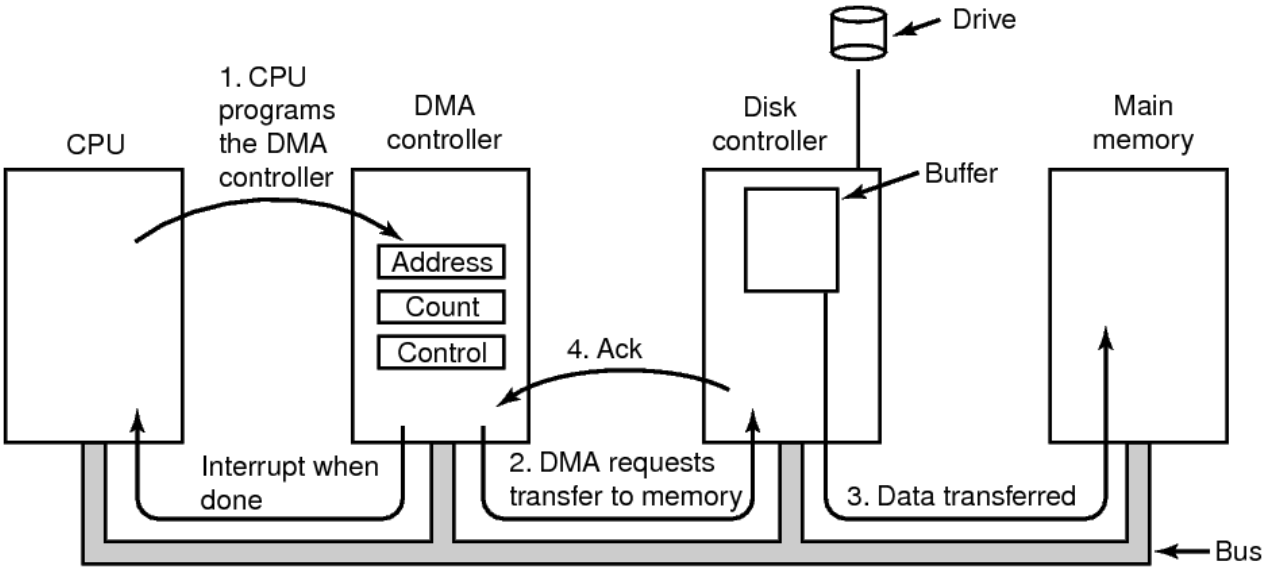
- Using polling or interrupts to transfer data can be quite CPU intensive. Consider a UART that interrupts each time a character arrives from a serial port. Now imagine that the serial port is connected to a 56Kbps modem downloading a large JPEG image from the Internet. 56Kbps is roughly 5,600 characters per second. Therefore, while the image is being downloaded, the UART is interrupting the current running process 5,600 times a second.
- On each interrupt, the CPU is diverted to the UART interrupt handler. The handler must read the incoming character, find the process waiting for it, copy the character to the process, change its state to ready before returning to the running process. You can imagine that this takes a lot of the CPU away from the running process.
- A better method is to use direct memory access (**DMA**). Here, the device controller delivers the data to the appropriate process location in main memory, and sends an interrupt when the transfer is complete. And if the device can buffer and deliver a bundle of data (e.g 16 characters with the 16550AN UART), then this can again reduce the interrupt handling load on the CPU.
- Disks usually use DMA. The software must tell the controller: what operation to perform (read, write, format), which cylinder to move to, which sector is required and which head, and which DMA address to put or get the sector. The controller then

performs the operation, and sends in an interrupt on success or error.

- DMA requires some intelligence to be built into the device controller. For example, consider a disk drive controller which has the following registers:

Location	Meaning of Register
10,000	Disk address: cylinder, head, sector
10,004	Operation to perform: read or write
10,008	Start of DMA buffer in main memory

- Imagine the operating system wants to write one disk block (say 1,000 bytes) from address **34,500** to cylinder 159, head 7, sector 5. The operating system also knows that the disk device can perform DMA. To make the write to disk occur, the operating system writes the value 34,500 into address **10,008**, the disk address `cyl 159, hd 7, sc 5' into address **10,000**, and finally the command `write' into address **10,004**.
- The disk device controller reads these values, and understands that it must write the buffer starting at location 34,500 to disk location `cyl 159, hd 7, sc 5'. It then uses the address/data/status buses to read the 1,000 bytes from main memory into a buffer on the controller. The device **asserts** each address in turn (just like the CPU does), and can thus temporarily stop the CPU (or any other device) from accessing the bus.
- Once the data is copied into the disk controller, it can command the disk hardware to write the data at the requested disk location. Once the operation is complete, the disk controller sends an interrupt into the CPU to indicate completion. Only now is the operating system's disk interrupt handler brought into action, and it has much less work to do now.



(from Tanenbaum)

- When the device uses the data bus, this stops the CPU from accessing main memory. You might imagine that this is a problem. Fortunately it isn't for two reasons:
  - Modern CPUs have large numbers of registers, and a lot of memory cache. CPUs can continue to run the current program by using the data in registers and cache, and minimise the use of the main memory. The CPU can usually continue to work temporarily without requiring main memory access.
  - Because I/O devices don't operate as fast as the CPU, they won't lock the CPU out from using the data bus that often. Instead, DMA will prevent the CPU from using the data bus on average about 1 access out of every N clock cycles; this is known as "stealing a cycle" from the CPU.
- As with the DMA write operation described above, reads from the disk can be performed without supervision from the operating system/CPU. Many peripheral device controllers (disks, UARTs, network cards, printer ports etc.) support DMA data transfers. Good operating systems make use of this support to improve their performance.

File translated from TEX by TTH, version 3.85.
On 8 Dec 2010, 08:43.