Table of Contents, Show Frames, No Frames

Chapter 3
Memory Management

The memory management subsystem is one of the most important parts of the operating system. Since the early days of computing, there has been a need for more memory than exists physically in a system. Strategies have been developed to overcome this limitation and the most successful of these is virtual memory. Virtual memory makes the system appear to have more memory than it actually has by sharing it between competing processes as they need it.

Virtual memory does more than just make your computer's memory go further. The memory management subsystem provides:

Large Address Spaces
The operating system makes the system appear as if it has a larger amount of memory than it actually has. The virtual memory can be many times larger than the physical memory in the system,

Protection
Each process in the system has its own virtual address space. These virtual address spaces are completely separate from each other and so a process running one application cannot affect another. Also, the hardware virtual memory mechanisms allow areas of memory to be protected against writing. This protects code and data from being overwritten by rogue applications.

Memory Mapping
Memory mapping is used to map image and data files into a processes address space. In memory mapping, the contents of a file are linked directly into the virtual address space of a process.

Fair Physical Memory Allocation
The memory management subsystem allows each running process in the system a fair share of the physical memory of the system,

Shared Virtual Memory
Although virtual memory allows processes to have separate (virtual) address spaces, there are times when you need processes to share memory. For example there could be several processes in the system running the bash command shell. Rather than have several copies of bash, one in each processes virtual address space, it is better to have only one copy in physical memory and all of the processes running bash share it. Dynamic libraries are another common example of executing code shared between several processes.

Shared memory can also be used as an Inter Process Communication (IPC) mechanism, with two or more processes exchanging information via memory common to all of them. Linux supports the Unix ™ System V shared memory IPC.

3.1 An Abstract Model of Virtual Memory

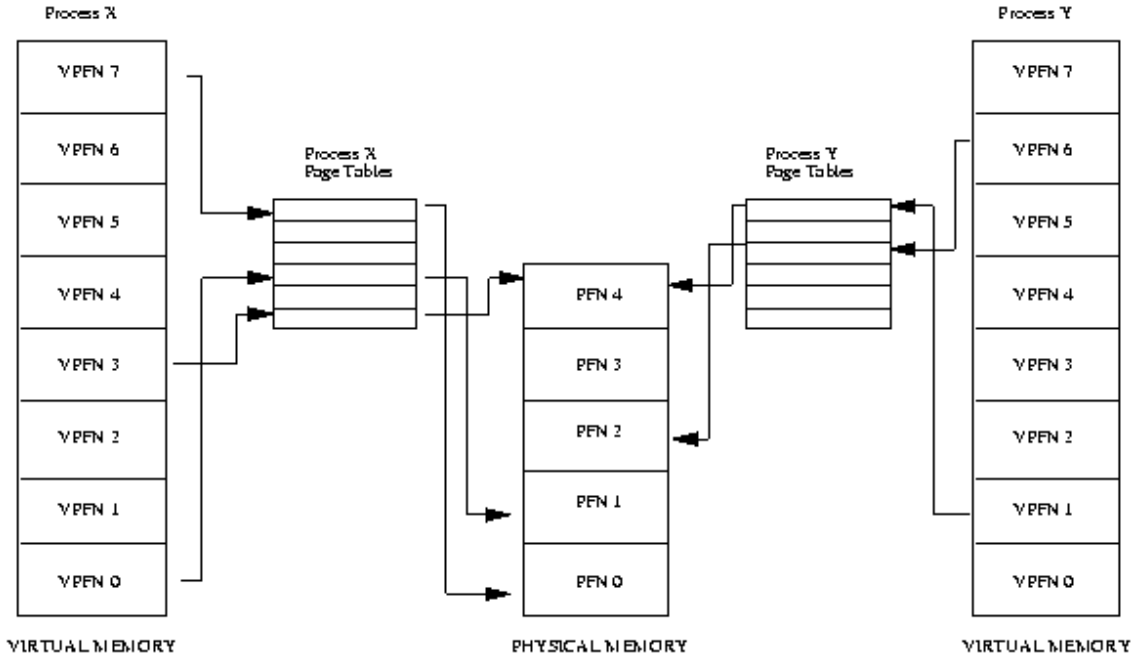

Figure 3.1: Abstract model of Virtual to Physical address mapping

Before considering the methods that Linux uses to support virtual memory it is useful to consider an abstract model that is not cluttered by too much detail.

As the processor executes a program it reads an instruction from memory and decodes it. In decoding the instruction it may need to fetch or store the contents of a location in memory. The processor then executes the instruction and moves onto the next instruction in the program. In this way the processor is always accessing memory either to fetch instructions or to fetch and store data.

In a virtual memory system all of these addresses are virtual addresses and not physical addresses. These virtual addresses are converted into physical addresses by the processor based on information held in a set of tables maintained by the operating system.

To make this translation easier, virtual and physical memory are divided into handy sized chunks called *pages*. These pages are all the same size, they need not be but if they were not, the system would be very hard to administer. Linux on Alpha AXP systems uses 8 Kbyte pages and on Intel x86 systems it uses 4 Kbyte pages. Each of these pages is given a unique number; the page frame number (PFN).

In this paged model, a virtual address is composed of two parts; an offset and a virtual page frame number. If the page size is 4 Kbytes, bits 11:0 of the virtual address contain the offset and bits 12 and above are the virtual page frame number. Each time the processor encounters a virtual address it must extract the offset and the virtual page frame number. The processor must translate the virtual page frame number into a physical one and then access the location at the correct offset into that physical page. To do this the processor uses *page tables*.

Figure 3.1 shows the virtual address spaces of two processes, process *X* and process *Y*, each with their own page tables. These page tables map each processes virtual pages into physical pages in memory. This shows that process *X's* virtual page frame number 0 is mapped into memory in physical page frame number 1 and that process *Y's* virtual page frame number 1 is mapped into physical page frame number 4. Each entry in the theoretical page table contains the following information:

- Valid flag. This indicates if this page table entry is valid,
- The physical page frame number that this entry is describing,
- Access control information. This describes how the page may be used. Can it be written to? Does it contain executable code?

The page table is accessed using the virtual page frame number as an offset. Virtual page frame 5 would be the 6th element of the table (0 is the first element).

To translate a virtual address into a physical one, the processor must first work out the virtual addresses page frame number and the offset within that virtual page. By making the page size a power of 2 this can be easily done by masking and shifting. Looking again at Figures 3.1 and assuming a page size of *0x2000* bytes (which is decimal 8192) and an address of *0x2194* in process *Y's* virtual address space then the processor would translate that address into offset *0x194* into virtual page frame number 1.

The processor uses the virtual page frame number as an index into the processes page table to retrieve its page table entry. If the page table entry at that offset is valid, the processor takes the physical page frame number from this entry. If the entry is invalid, the process has accessed a non-existent area of its virtual memory. In this case, the processor cannot resolve the address and must pass control to the operating system so that it can fix things up.

Just how the processor notifies the operating system that the correct process has attempted to access a virtual address for which there is no valid translation is specific to the processor. However the processor delivers it, this is known as a *page fault* and the operating system is notified of the faulting virtual address and the reason for the page fault.

Assuming that this is a valid page table entry, the processor takes that physical page frame number and multiplies it by the page size to get the address of the base of the page in physical memory. Finally, the processor adds in the offset to the instruction or data that it needs.

Using the above example again, process *Y's* virtual page frame number 1 is mapped to physical page frame number 4 which starts at *0x8000* (4 x *0x2000*). Adding in the *0x194* byte offset gives us a final physical address of *0x8194*.

By mapping virtual to physical addresses this way, the virtual memory can be mapped into the system's physical pages in any order. For example, in Figure 3.1 process *X's* virtual page frame number 0 is mapped to physical page frame number 1 whereas virtual page frame number 7 is mapped to physical page frame number 0 even though it is higher in virtual memory than virtual page frame number 0. This demonstrates an interesting byproduct of virtual memory; the pages of virtual memory do not have to be present in physical memory in any particular order.

3.1.1 Demand Paging

As there is much less physical memory than virtual memory the operating system must be careful that it does not use the physical memory inefficiently. One way to save physical memory is to only load virtual pages that are currently being used by the executing program. For example, a database program may be run to query a database. In this case not all of the database needs to be loaded into memory, just those data records that are being examined. If the database query is a search query then it does not make sense to load the code from the database program that deals with adding new records. This technique of only loading virtual pages into memory as they are accessed is known as demand paging.

When a process attempts to access a virtual address that is not currently in memory the processor cannot find a page table entry for the virtual page referenced. For example, in Figure 3.1 there is no entry in process *X's* page table for virtual page frame number 2 and so if process *X* attempts to read from an address within virtual page frame number 2 the processor cannot translate the address into a physical one. At this point the processor notifies the operating system that a page fault has occurred.

If the faulting virtual address is invalid this means that the process has attempted to access a virtual address that it should not have. Maybe the application has gone wrong in some way, for example writing to random addresses in memory. In this case the operating system will terminate it, protecting the other processes in the system from this rogue process.

If the faulting virtual address was valid but the page that it refers to is not currently in memory, the operating system must bring the appropriate page into memory from the image on disk. Disk access takes a long time, relatively speaking, and so the process must wait quite a while until the page has been fetched. If there are other processes that could run then the operating system will select one of them to run. The fetched page is written into a free physical page frame and an entry for the virtual page frame number is added to the processes page table. The process is then restarted at the

machine instruction where the memory fault occurred. This time the virtual memory access is made, the processor can make the virtual to physical address translation and so the process continues to run.

Linux uses demand paging to load executable images into a processes virtual memory. Whenever a command is executed, the file containing it is opened and its contents are mapped into the processes virtual memory. This is done by modifying the data structures describing this processes memory map and is known as *memory mapping*. However, only the first part of the image is actually brought into physical memory. The rest of the image is left on disk. As the image executes, it generates page faults and Linux uses the processes memory map in order to determine which parts of the image to bring into memory for execution.

3.1.2 Swapping

If a process needs to bring a virtual page into physical memory and there are no free physical pages available, the operating system must make room for this page by discarding another page from physical memory.

If the page to be discarded from physical memory came from an image or data file and has not been written to then the page does not need to be saved. Instead it can be discarded and if the process needs that page again it can be brought back into memory from the image or data file.

However, if the page has been modified, the operating system must preserve the contents of that page so that it can be accessed at a later time. This type of page is known as a *dirty* page and when it is removed from memory it is saved in a special sort of file called the swap file. Accesses to the swap file are very long relative to the speed of the processor and physical memory and the operating system must juggle the need to write pages to disk with the need to retain them in memory to be used again.

If the algorithm used to decide which pages to discard or swap (the *swap algorithm* is not efficient then a condition known as *thrashing* occurs. In this case, pages are constantly being written to disk and then being read back and the operating system is too busy to allow much real work to be performed. If, for example, physical page frame number 1 in Figure 3.1 is being regularly accessed then it is not a good candidate for swapping to hard disk. The set of pages that a process is currently using is called the *working set*. An efficient swap scheme would make sure that all processes have their working set in physical memory.

Linux uses a Least Recently Used (LRU) page aging technique to fairly choose pages which might be removed from the system. This scheme involves every page in the system having an age which changes as the page is accessed. The more that a page is accessed, the younger it is; the less that it is accessed the older and more stale it becomes. Old pages are good candidates for swapping.

3.1.3 Shared Virtual Memory

Virtual memory makes it easy for several processes to share memory. All memory access are made via page tables and each process has its own separate page table. For two processes sharing a physical page of memory, its physical page frame number must appear in a page table entry in both of their page tables.

Figure 3.1 shows two processes that each share physical page frame number 4. For process *X* this is virtual page frame number 4 whereas for process *Y* this is virtual page frame number 6. This illustrates an interesting point about sharing pages: the shared physical page does not have to exist at the same place in virtual memory for any or all of the processes sharing it.

3.1.4 Physical and Virtual Addressing Modes

It does not make much sense for the operating system itself to run in virtual memory. This would be a nightmare situation where the operating system must maintain page tables for itself. Most multi-purpose processors support the notion of a physical address mode as well as a virtual address mode. Physical addressing mode requires no page tables and the

processor does not attempt to perform any address translations in this mode. The Linux kernel is linked to run in physical address space.

The Alpha AXP processor does not have a special physical addressing mode. Instead, it divides up the memory space into several areas and designates two of them as physically mapped addresses. This kernel address space is known as KSEG address space and it encompasses all addresses upwards from *0xfffffc0000000000*. In order to execute from code linked in KSEG (by definition, kernel code) or access data there, the code must be executing in kernel mode. The Linux kernel on Alpha is linked to execute from address *0xfffffc0000310000*.

3.1.5 Access Control

The page table entries also contain access control information. As the processor is already using the page table entry to map a processes virtual address to a physical one, it can easily use the access control information to check that the process is not accessing memory in a way that it should not.

There are many reasons why you would want to restrict access to areas of memory. Some memory, such as that containing executable code, is naturally read only memory; the operating system should not allow a process to write data over its executable code. By contrast, pages containing data can be written to but attempts to execute that memory as instructions should fail. Most processors have at least two modes of execution: *kernel* and *user*. You would not want kernel code executing by a user or kernel data structures to be accessible except when the processor is running in kernel mode.

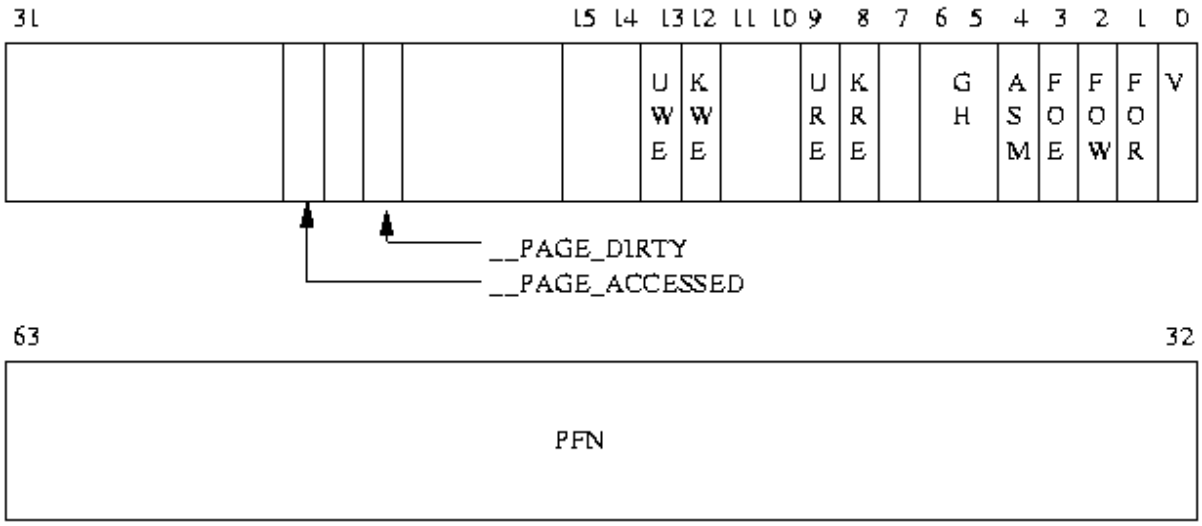

Figure 3.2: Alpha AXP Page Table Entry

The access control information is held in the PTE and is processor specific; figure 3.2 shows the PTE for Alpha AXP. The bit fields have the following meanings:

V
: Valid, if set this PTE is valid,

FOE
: ``Fault on Execute", Whenever an attempt to execute instructions in this page occurs, the processor reports a page fault and passes control to the operating system,

FOW
: ``Fault on Write", as above but page fault on an attempt to write to this page,

FOR
: ``Fault on Read", as above but page fault on an attempt to read from this page,

ASM

Address Space Match. This is used when the operating system wishes to clear only some of the entries from the Translation Buffer,

KRE
Code running in kernel mode can read this page,

URE
Code running in user mode can read this page,

GH
Granularity hint used when mapping an entire block with a single Translation Buffer entry rather than many,

KWE
Code running in kernel mode can write to this page,

UWE
Code running in user mode can write to this page,

page frame number
For PTEs with the **V** bit set, this field contains the physical Page Frame Number (page frame number) for this PTE. For invalid PTEs, if this field is not zero, it contains information about where the page is in the swap file.

The following two bits are defined and used by Linux:

_PAGE_DIRTY
if set, the page needs to be written out to the swap file,

_PAGE_ACCESSED
Used by Linux to mark a page as having been accessed.

3.2 Caches

If you were to implement a system using the above theoretical model then it would work, but not particularly efficiently. Both operating system and processor designers try hard to extract more performance from the system. Apart from making the processors, memory and so on faster the best approach is to maintain caches of useful information and data that make some operations faster. Linux uses a number of memory management related caches:

Buffer Cache

The buffer cache contains data buffers that are used by the block device drivers.

These buffers are of fixed sizes (for example 512 bytes) and contain blocks of information that have either been read from a block device or are being written to it. A block device is one that can only be accessed by reading and writing fixed sized blocks of data. All hard disks are block devices.

The buffer cache is indexed via the device identifier and the desired block number and is used to quickly find a block of data. Block devices are only ever accessed via the buffer cache. If data can be found in the buffer cache then it does not need to be read from the physical block device, for example a hard disk, and access to it is much faster.

Page Cache

This is used to speed up access to images and data on disk.

It is used to cache the logical contents of a file a page at a time and is accessed via the file and offset within the file. As pages are read into memory from disk, they are cached in the page cache.

Swap Cache

Only modified (or *dirty*) pages are saved in the swap file.

So long as these pages are not modified after they have been written to the swap file then the next time the page is swapped out there is no need to write it to the swap file as the page is already in the swap file. Instead the page can simply be discarded. In a heavily swapping system this saves many unnecessary and costly disk operations.

Hardware Caches

One commonly implemented hardware cache is in the processor; a cache of Page Table Entries. In this case, the processor does not always read the page table directly but instead caches translations for pages as it needs them. These are the Translation Look-aside Buffers and contain cached copies of the page table entries from one or more processes in the system.

When the reference to the virtual address is made, the processor will attempt to find a matching TLB entry. If it finds one, it can directly translate the virtual address into a physical one and perform the correct operation on the data. If the processor cannot find a matching TLB entry then it must get the operating system to help. It does this by signalling the operating system that a TLB miss has occurred. A system specific mechanism is used to deliver that exception to the operating system code that can fix things up. The operating system generates a new TLB entry for the address mapping. When the exception has been cleared, the processor will make another attempt to translate the virtual address. This time it will work because there is now a valid entry in the TLB for that address.

The drawback of using caches, hardware or otherwise, is that in order to save effort Linux must use more time and space maintaining these caches and, if the caches become corrupted, the system will crash.

3.3 Linux Page Tables

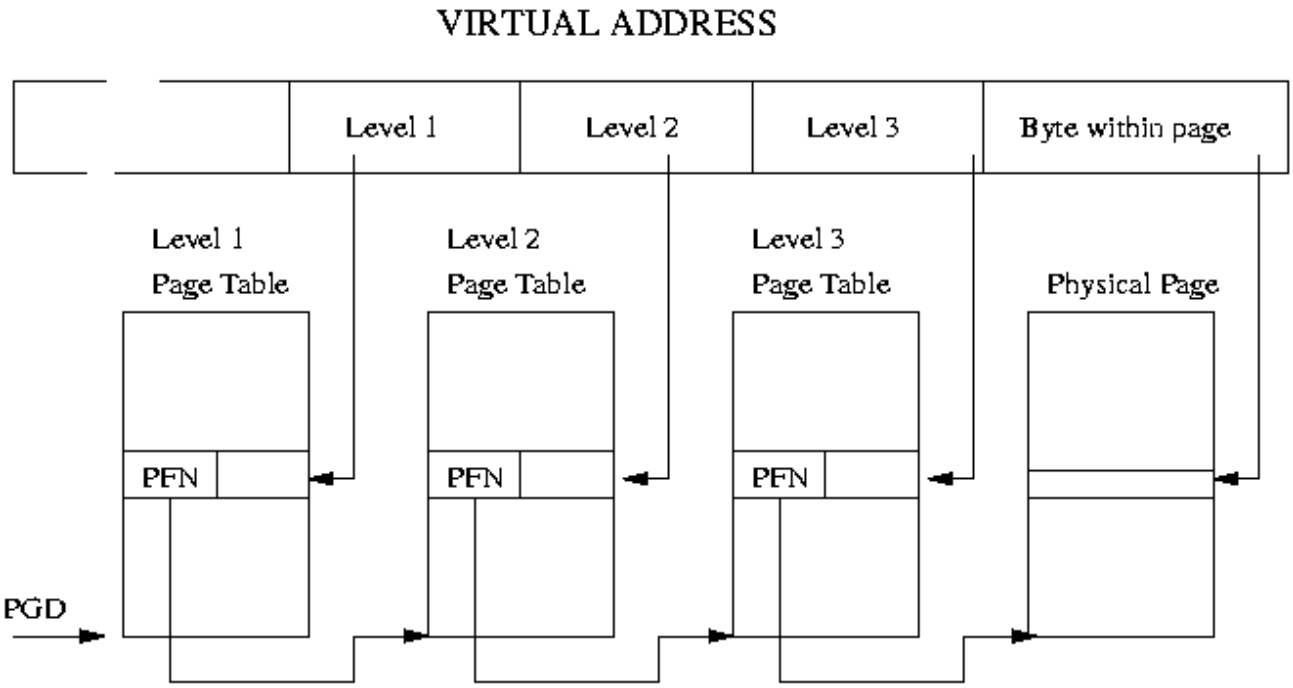

Figure 3.3: Three Level Page Tables

Linux assumes that there are three levels of page tables. Each Page Table accessed contains the page frame number of the next level of Page Table. Figure 3.3 shows how a virtual address can be broken into a number of fields; each field providing an offset into a particular Page Table. To translate a virtual address into a physical one, the processor must take the contents of each level field, convert it into an offset into the physical page containing the Page Table and read the page frame number of the next level of Page Table. This is repeated three times until the page frame number of the physical page containing the virtual address is found. Now the final field in the virtual address, the byte offset, is used to find the data inside the page.

Each platform that Linux runs on must provide translation macros that allow the kernel to traverse the page tables for a particular process. This way, the kernel does not need to know the format of the page table entries or how they are arranged.

This is so successful that Linux uses the same page table manipulation code for the Alpha processor, which has three levels of page tables, and for Intel x86 processors, which have two levels of page tables.

3.4 Page Allocation and Deallocation

There are many demands on the physical pages in the system. For example, when an image is loaded into memory the operating system needs to allocate pages. These will be freed when the image has finished executing and is unloaded. Another use for physical pages is to hold kernel specific data structures such as the page tables themselves. The mechanisms and data structures used for page allocation and deallocation are perhaps the most critical in maintaining the efficiency of the virtual memory subsystem.

All of the physical pages in the system are described by the mem_map data structure which is a list of mem_map_t

[1] structures which is initialized at boot time. Each mem_map_t describes a single physical page in the system. Important fields (so far as memory management is concerned) are:

count
: This is a count of the number of users of this page. The count is greater than one when the page is shared between many processes,

age
: This field describes the age of the page and is used to decide if the page is a good candidate for discarding or swapping,

map_nr
: This is the physical page frame number that this mem_map_t describes.

The free_area vector is used by the page allocation code to find and free pages. The whole buffer management scheme is supported by this mechanism and so far as the code is concerned, the size of the page and physical paging mechanisms used by the processor are irrelevant.

Each element of free_area contains information about blocks of pages. The first element in the array describes single pages, the next blocks of 2 pages, the next blocks of 4 pages and so on upwards in powers of two. The list element is used as a queue head and has pointers to the page data structures in the mem_map array. Free blocks of pages are queued here. map is a pointer to a bitmap which keeps track of allocated groups of pages of this size. Bit N of the bitmap is set if the Nth block of pages is free.

Figure free-area-figure shows the free_area structure. Element 0 has one free page (page frame number 0) and element 2 has 2 free blocks of 4 pages, the first starting at page frame number 4 and the second at page frame number 56.

3.4.1 Page Allocation

Linux uses the Buddy algorithm [2] to effectively allocate and deallocate blocks of pages. The page allocation code

attempts to allocate a block of one or more physical pages. Pages are allocated in blocks which are powers of 2 in size. That means that it can allocate a block 1 page, 2 pages, 4 pages and so on. So long as there are enough free pages in the system to grant this request (nr_free_pages > min_free_pages) the allocation code will search the free_area for a block of pages of the size requested. Each element of the free_area has a map of the allocated and free blocks of pages for that sized block. For example, element 2 of the array has a memory map that describes free and allocated blocks each of 4 pages long.

The allocation algorithm first searches for blocks of pages of the size requested. It follows the chain of free pages that is queued on the *list* element of the free_area data structure. If no blocks of pages of the requested size are free, blocks of the next size (which is twice that of the size requested) are looked for. This process continues until all of the free_area has been searched or until a block of pages has been found. If the block of pages found is larger than that requested it must be broken down until there is a block of the right size. Because the blocks are each a power of 2 pages big then this breaking down process is easy as you simply break the blocks in half. The free blocks are queued on the appropriate queue and the allocated block of pages is returned to the caller.

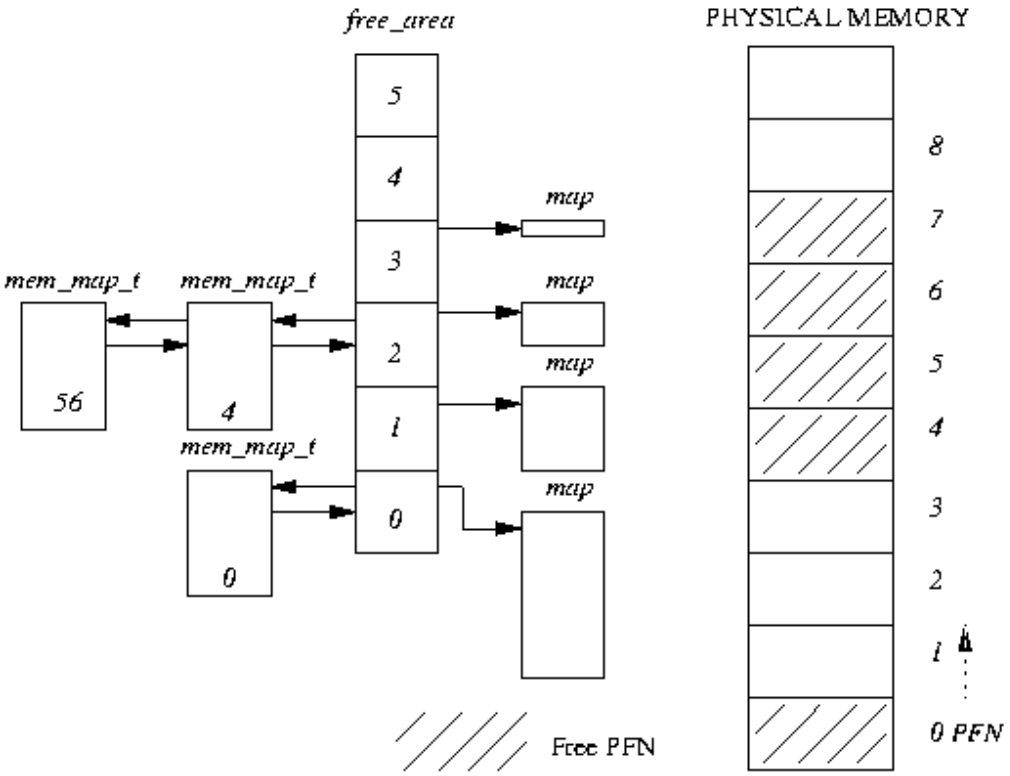

Figure 3.4: The free_area data structure

For example, in Figure 3.4 if a block of 2 pages was requested, the first block of 4 pages (starting at page frame number 4) would be broken into two 2 page blocks. The first, starting at page frame number 4 would be returned to the caller as the allocated pages and the second block, starting at page frame number 6 would be queued as a free block of 2 pages onto element 1 of the free_area array.

3.4.2 Page Deallocation

Allocating blocks of pages tends to fragment memory with larger blocks of free pages being broken down into smaller ones. The page deallocation code

recombines pages into larger blocks of free pages whenever it can. In fact the page block size is important as it allows for easy combination of blocks into larger blocks.

Whenever a block of pages is freed, the adjacent or buddy block of the same size is checked to see if it is free. If it is, then it is combined with the newly freed block of pages to form a new free block of pages for the next size block of pages. Each time two blocks of pages are recombined into a bigger block of free pages the page deallocation code attempts to recombine that block into a yet larger one. In this way the blocks of free pages are as large as memory usage will allow.

For example, in Figure 3.4, if page frame number 1 were to be freed, then that would be combined with the already free page frame number 0 and queued onto element 1 of the free_area as a free block of size 2 pages.

3.5 Memory Mapping

When an image is executed, the contents of the executable image must be brought into the processes virtual address space. The same is also true of any shared libraries that the executable image has been linked to use. The executable file is not actually brought into physical memory, instead it is merely linked into the processes virtual memory. Then, as the parts of the program are referenced by the running application, the image is brought into memory from the executable image. This linking of an image into a processes virtual address space is known as memory mapping.

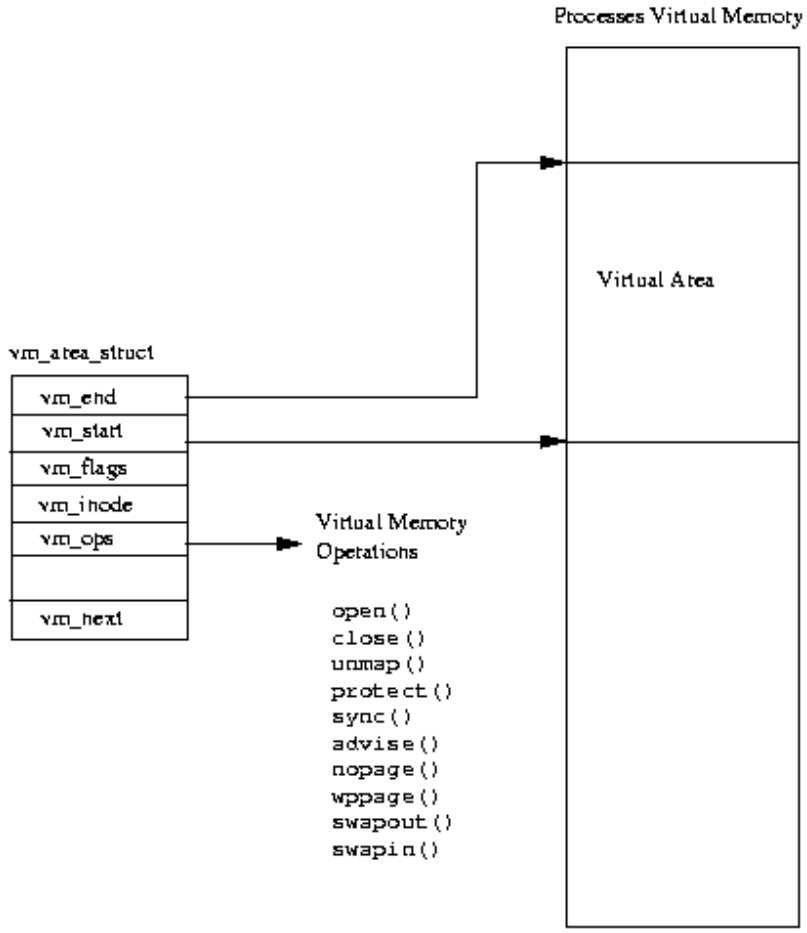

Figure 3.5: Areas of Virtual Memory

Every processes virtual memory is represented by an mm_struct data structure. This contains information about the image that it is currently executing (for example bash) and also has pointers to a number of vm_area_struct data structures. Each vm_area_struct data structure describes the start and end of the area of virtual memory, the processes access rights to that memory and a set of operations for that memory. These operations are a set of routines that Linux must use when manipulating this area of virtual memory. For example, one of the virtual memory operations performs the correct actions when the process has attempted to access this virtual memory but finds (via a page fault) that the memory is not actually in physical memory. This operation is the *nopage* operation. The *nopage* operation is used when Linux demand pages the pages of an executable image into memory.

When an executable image is mapped into a processes virtual address a set of vm_area_struct data structures is generated. Each vm_area_struct data structure represents a part of the executable image; the executable code, initialized data (variables), unitialized data and so on. Linux supports a number of standard virtual memory operations and as the vm_area_struct data structures are created, the correct set of virtual memory operations are associated with them.

3.6 Demand Paging

Once an executable image has been memory mapped into a processes virtual memory it can start to execute. As only the very start of the image is physically pulled into memory it will soon access an area of virtual memory that is not yet in physical memory. When a process accesses a virtual address that does not have a valid page table entry, the processor will report a page fault to Linux.

The page fault describes the virtual address where the page fault occurred and the type of memory access that caused.

Linux must find the vm_area_struct that represents the area of memory that the page fault occurred in. As searching through the vm_area_struct data structures is critical to the efficient handling of page faults, these are linked together in an AVL (Adelson-Velskii and Landis) tree structure. If there is no vm_area_struct data structure for this faulting virtual address, this process has accessed an illegal virtual address. Linux will signal the process, sending a SIGSEGV signal, and if the process does not have a handler for that signal it will be terminated.

Linux next checks the type of page fault that occurred against the types of accesses allowed for this area of virtual memory. If the process is accessing the memory in an illegal way, say writing to an area that it is only allowed to read from, it is also signalled with a memory error.

Now that Linux has determined that the page fault is legal, it must deal with it.

Linux must differentiate between pages that are in the swap file and those that are part of an executable image on a disk somewhere. It does this by using the page table entry for this faulting virtual address.

If the page's page table entry is invalid but not empty, the page fault is for a page currently being held in the swap file. For Alpha AXP page table entries, these are entries which do not have their valid bit set but which have a non-zero value in their PFN field. In this case the PFN field holds information about where in the swap (and which swap file) the page is being held. How pages in the swap file are handled is described later in this chapter.

Not all vm_area_struct data structures have a set of virtual memory operations and even those that do may not have a *nopage* operation. This is because by default Linux will fix up the access by allocating a new physical page and creating a valid page table entry for it. If there is a *nopage* operation for this area of virtual memory, Linux will use it.

The generic Linux *nopage* operation is used for memory mapped executable images and it uses the page cache to bring the required image page into physical memory.

However the required page is brought into physical memory, the processes page tables are updated. It may be necessary for hardware specific actions to update those entries, particularly if the processor uses translation look aside buffers. Now that the page fault has been handled it can be dismissed and the process is restarted at the instruction that made the faulting virtual memory access.

3.7 The Linux Page Cache

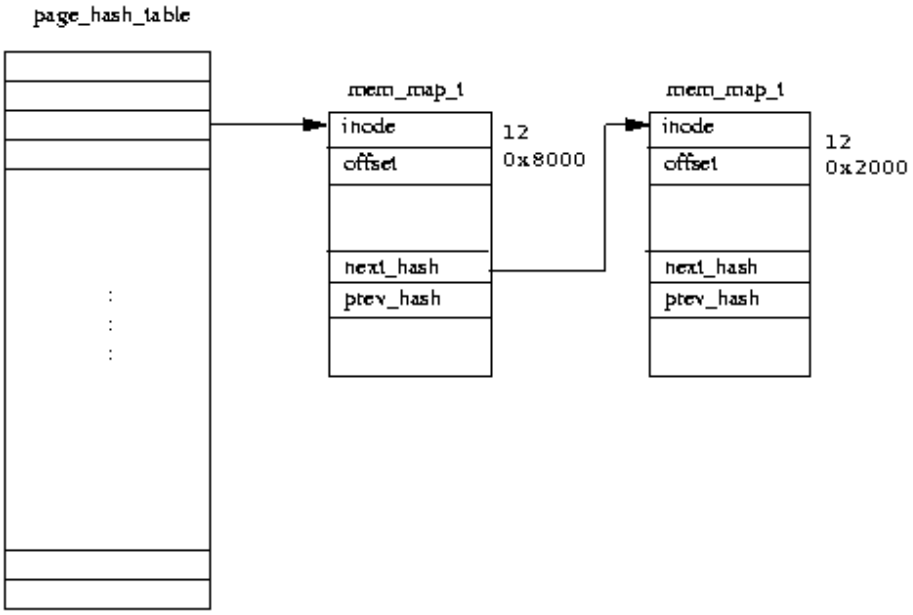

Figure 3.6: The Linux Page Cache

The role of the Linux page cache is to speed up access to files on disk. Memory mapped files are read a page at a time and these pages are stored in the page cache. Figure 3.6 shows that the page cache consists of the page_hash_table, a vector of pointers to mem_map_t data structures.

Each file in Linux is identified by a VFS inode data structure (described in Chapter filesystem-chapter) and each VFS inode is unique and fully describes one and only one file. The index into the page table is derived from the file's VFS inode and the offset into the file.

Whenever a page is read from a memory mapped file, for example when it needs to be brought back into memory during demand paging, the page is read through the page cache. If the page is present in the cache, a pointer to the mem_map_t data structure representing it is returned to the page fault handling code. Otherwise the page must be brought into memory from the file system that holds the image. Linux allocates a physical page and reads the page from the file on disk.

If it is possible, Linux will initiate a read of the next page in the file. This single page read ahead means that if the process is accessing the pages in the file serially, the next page will be waiting in memory for the process.

Over time the page cache grows as images are read and executed. Pages will be removed from the cache as they are no longer needed, say as an image is no longer being used by any process. As Linux uses memory it can start to run low on physical pages. In this case Linux will reduce the size of the page cache.

3.8 Swapping Out and Discarding Pages

When physical memory becomes scarce the Linux memory management subsystem must attempt to free physical pages. This task falls to the kernel swap daemon (*kswapd*).

The kernel swap daemon is a special type of process, a kernel thread. Kernel threads are processes have no virtual memory, instead they run in kernel mode in the physical address space. The kernel swap daemon is slightly misnamed in that it does more than merely swap pages out to the system's swap files. Its role is make sure that there are enough free pages in the system to keep the memory management system operating efficiently.

The Kernel swap daemon (*kswapd*) is started by the kernel init process at startup time and sits waiting for the kernel swap timer to periodically expire.

Every time the timer expires, the swap daemon looks to see if the number of free pages in the system is getting too low. It uses two variables, *free_pages_high* and *free_pages_low* to decide if it should free some pages. So long as the number of free pages in the system remains above *free_pages_high*, the kernel swap daemon does nothing; it sleeps again until its timer next expires. For the purposes of this check the kernel swap daemon takes into account the number of pages currently being written out to the swap file. It keeps a count of these in *nr_async_pages*; this is incremented each time a page is queued waiting to be written out to the swap file and decremented when the write to the swap device has completed. *free_pages_low* and *free_pages_high* are set at system startup time and are related to the number of physical pages in the system. If the number of free pages in the system has fallen below *free_pages_high* or worse still *free_pages_low*, the kernel swap daemon will try three ways to reduce the number of physical pages being used by the system:

Reducing the size of the buffer and page caches,
Swapping out System V shared memory pages,
Swapping out and discarding pages.

If the number of free pages in the system has fallen below *free_pages_low*, the kernel swap daemon will try to free 6 pages before it next runs. Otherwise it will try to free 3 pages. Each of the above methods are tried in turn until enough pages have been freed. The kernel swap daemon remembers which method it was using the last time that it attempted to free physical pages. Each time it runs it will start trying to free pages using this last successful method.

After it has free sufficient pages, the swap daemon sleeps again until its timer expires. If the reason that the kernel swap daemon freed pages was that the number of free pages in the system had fallen below *free_pages_low*, it only sleeps for half its usual time. Once the number of free pages is more than *free_pages_low* the kernel swap daemon goes back to sleeping longer between checks.

3.8.1 Reducing the Size of the Page and Buffer Caches

The pages held in the page and buffer caches are good candidates for being freed into the free_area vector. The Page Cache, which contains pages of memory mapped files, may contain unneccessary pages that are filling up the system's memory. Likewise the Buffer Cache, which contains buffers read from or being written to physical devices, may also contain unneeded buffers. When the physical pages in the system start to run out, discarding pages from these caches is relatively easy as it requires no writing to physical devices (unlike swapping pages out of memory). Discarding these pages does not have too many harmful side effects other than making access to physical devices and memory mapped files slower. However, if the discarding of pages from these caches is done fairly, all processes will suffer equally.

Every time the Kernel swap daemon tries to shrink these caches

it examines a block of pages in the mem_map page vector to see if any can be discarded from physical memory. The size of the block of pages examined is higher if the kernel swap daemon is intensively swapping; that is if the number of free pages in the system has fallen dangerously low. The blocks of pages are examined in a cyclical manner; a different block of pages is examined each time an attempt is made to shrink the memory map. This is known as the *clock* algorithm as, rather like the minute hand of a clock, the whole mem_map page vector is examined a few pages at a time.

Each page being examined is checked to see if it is cached in either the page cache or the buffer cache. You should note that shared pages are not considered for discarding at this time and that a page cannot be in both caches at the same time. If the page is not in either cache then the next page in the mem_map page vector is examined.

Pages are cached in the buffer cache (or rather the buffers within the pages are cached) to make buffer allocation and deallocation more efficient. The memory map shrinking code tries to free the buffers that are contained within the page being examined.

If all the buffers are freed, then the pages that contain them are also be freed. If the examined page is in the Linux page cache, it is removed from the page cache and freed.

When enough pages have been freed on this attempt then the kernel swap daemon will wait until the next time it is periodically woken. As none of the freed pages were part of any process's virtual memory (they were cached pages), then no page tables need updating. If there were not enough cached pages discarded then the swap daemon will try to swap out some shared pages.

3.8.2 Swapping Out System V Shared Memory Pages

System V shared memory is an inter-process communication mechanism which allows two or more processes to share virtual memory in order to pass information amongst themselves. How processes share memory in this way is described in more detail in Chapter IPC-chapter. For now it is enough to say that each area of System V shared memory is described by a shmid_ds data structure. This contains a pointer to a list of vm_area_struct data structures, one for each process sharing this area of virtual memory. The vm_area_struct data structures describe where in each processes virtual memory this area of System V shared memory goes. Each vm_area_struct data structure for this System V shared memory is linked together using the vm_next_shared and vm_prev_shared pointers. Each shmid_ds data structure also contains a list of page table entries each of which describes the physical page that a shared virtual page maps to.

The kernel swap daemon also uses a *clock* algorithm when swapping out System V shared memory pages.

. Each time it runs it remembers which page of which shared virtual memory area it last swapped out. It does this by keeping two indices, the first is an index into the set of shmid_ds data structures, the second into the list of page table entries for this area of System V shared memory. This makes sure that it fairly victimizes the areas of System V shared memory.

As the physical page frame number for a given virtual page of System V shared memory is contained in the page tables of all of the processes sharing this area of virtual memory, the kernel swap daemon must modify all of these page tables to show that the page is no longer in memory but is now held in the swap file. For each shared page it is swapping out, the kernel swap daemon finds the page table entry in each of the sharing processes page tables (by following a pointer from each vm_area_struct data structure). If this processes page table entry for this page of System V shared memory is valid, it converts it into an invalid but swapped out page table entry and reduces this (shared) page's count of users by one. The format of a swapped out System V shared page table entry contains an index into the set of shmid_ds data structures and an index into the page table entries for this area of System V shared memory.

If the page's count is zero after the page tables of the sharing processes have all been modified, the shared page can be written out to the swap file. The page table entry in the list pointed at by the shmid_ds data structure for this area of System V shared memory is replaced by a swapped out page table entry. A swapped out page table entry is invalid but contains an index into the set of open swap files and the offset in that file where the swapped out page can be found. This information will be used when the page has to be brought back into physical memory.

3.8.3 Swapping Out and Discarding Pages

The swap daemon looks at each process in the system in turn to see if it is a good candidate for swapping.

Good candidates are processes that can be swapped (some cannot) and that have one or more pages which can be swapped or discarded from memory. Pages are swapped out of physical memory into the system's swap files only if the data in them cannot be retrieved another way.

A lot of the contents of an executable image come from the image's file and can easily be re-read from that file. For example, the executable instructions of an image will never be modified by the image and so will never be written to the swap file. These pages can simply be discarded; when they are again referenced by the process, they will be brought back into memory from the executable image.

Once the process to swap has been located, the swap daemon looks through all of its virtual memory regions looking for areas which are not shared or locked.

Linux does not swap out all of the swappable pages of the process that it has selected; instead it removes only a small number of pages.

Pages cannot be swapped or discarded if they are locked in memory.

The Linux swap algorithm uses page aging. Each page has a counter (held in the mem_map_t data structure) that gives the Kernel swap daemon some idea whether or not a page is worth swapping. Pages age when they are unused and rejuvinate on access; the swap daemon only swaps out old pages. The default action when a page is first allocated, is to give it an initial age of 3. Each time it is touched, it's age is increased by 3 to a maximum of 20. Every time the Kernel swap daemon runs it ages pages, decrementing their age by 1. These default actions can be changed and for this reason they (and other swap related information) are stored in the swap_control data structure.

If the page is old (age = 0), the swap daemon will process it further. *Dirty* pages are pages which can be swapped out. Linux uses an architecture specific bit in the PTE to describe pages this way (see Figure 3.2). However, not all *dirty* pages are necessarily written to the swap file. Every virtual memory region of a process may have its own swap operation (pointed at by the vm_ops pointer in the vm_area_struct) and that method is used. Otherwise, the swap daemon will allocate a page in the swap file and write the page out to that device.

The page's page table entry is replaced by one which is marked as invalid but which contains information about where the page is in the swap file. This is an offset into the swap file where the page is held and an indication of which swap file is being used. Whatever the swap method used, the original physical page is made free by putting it back into the free_area. Clean (or rather not *dirty*) pages can be discarded and put back into the free_area for re-use.

If enough of the swappable processes pages have been swapped out or discarded, the swap daemon will again sleep. The next time it wakes it will consider the next process in the system. In this way, the swap daemon nibbles away at each processes physical pages until the system is again in balance. This is much fairer than swapping out whole processes.

3.9 The Swap Cache

When swapping pages out to the swap files, Linux avoids writing pages if it does not have to. There are times when a page is both in a swap file and in physical memory. This happens when a page that was swapped out of memory was then brought back into memory when it was again accessed by a process. So long as the page in memory is not written to, the copy in the swap file remains valid.

Linux uses the swap cache to track these pages. The swap cache is a list of page table entries, one per physical page in the system. This is a page table entry for a swapped out page and describes which swap file the page is being held in together with its location in the swap file. If a swap cache entry is non-zero, it represents a page which is being held in a swap file that has not been modified. If the page is subsequently modified (by being written to), its entry is removed from the swap cache.

When Linux needs to swap a physical page out to a swap file it consults the swap cache and, if there is a valid entry for this page, it does not need to write the page out to the swap file. This is because the page in memory has not been modified since it was last read from the swap file.

The entries in the swap cache are page table entries for swapped out pages. They are marked as invalid but contain information which allow Linux to find the right swap file and the right page within that swap file.

3.10 Swapping Pages In

The dirty pages saved in the swap files may be needed again, for example when an application writes to an area of virtual memory whose contents are held in a swapped out physical page. Accessing a page of virtual memory that is not held in physical memory causes a page fault to occur. The page fault is the processor signalling the operating system that it cannot translate a virtual address into a physical one. In this case this is because the page table entry describing this

page of virtual memory was marked as invalid when the page was swapped out. The processor cannot handle the virtual to physical address translation and so hands control back to the operating system describing as it does so the virtual address that faulted and the reason for the fault. The format of this information and how the processor passes control to the operating system is processor specific.

The processor specific page fault handling code must locate the vm_area_struct data structure that describes the area of virtual memory that contains the faulting virtual address. It does this by searching the vm_area_struct data structures for this process until it finds the one containing the faulting virtual address. This is very time critical code and a processes vm_area_struct data structures are so arranged as to make this search take as little time as possible.

Having carried out the appropriate processor specific actions and found that the faulting virtual address is for a valid area of virtual memory, the page fault processing becomes generic and applicable to all processors that Linux runs on.

The generic page fault handling code looks for the page table entry for the faulting virtual address. If the page table entry it finds is for a swapped out page, Linux must swap the page back into physical memory. The format of the page table entry for a swapped out page is processor specific but all processors mark these pages as invalid and put the information neccessary to locate the page within the swap file into the page table entry. Linux needs this information in order to bring the page back into physical memory.

At this point, Linux knows the faulting virtual address and has a page table entry containing information about where this page has been swapped to. The vm_area_struct data structure may contain a pointer to a routine which will swap any page of the area of virtual memory that it describes back into physical memory. This is its *swapin* operation. If there is a *swapin* operation for this area of virtual memory then Linux will use it. This is, in fact, how swapped out System V shared memory pages are handled as it requires special handling because the format of a swapped out System V shared page is a little different from that of an ordinairy swapped out page. There may not be a *swapin* operation, in which case Linux will assume that this is an ordinairy page that does not need to be specially handled.

It allocates a free physical page and reads the swapped out page back from the swap file. Information telling it where in the swap file (and which swap file) is taken from the the invalid page table entry.

If the access that caused the page fault was not a write access then the page is left in the swap cache and its page table entry is not marked as writable. If the page is subsequently written to, another page fault will occur and, at that point, the page is marked as dirty and its entry is removed from the swap cache. If the page is not written to and it needs to be swapped out again, Linux can avoid the write of the page to its swap file because the page is already in the swap file.

If the access that caused the page to be brought in from the swap file was a write operation, this page is removed from the swap cache and its page table entry is marked as both dirty and writable.

Footnotes:

[1] Confusingly the structure is also known as the *page* structure.

[2] Bibliography reference here
