Operating Systems II
— Additional Information —

Steven Hand
Lent Term 2004

Contents

1 A Comparison of Page Tables

1.1 Multi-Level Page Tables

Multi-level page tables (MPTs) are perhaps the simplest practical form of mapping structure. A virtual address is treated as a structured word containing a small number of fields, with the width of each field being fixed. Translation of a virtual address proceeds by treating each field (starting with the most significant) as an index into an array. The result of a single step of translation is either a pointer to another array, a translation fault or, in the ultimate step, a page table entry (PTE). A page table entry contains enough information to compute the physical address. This is illustrated in Figure 1.

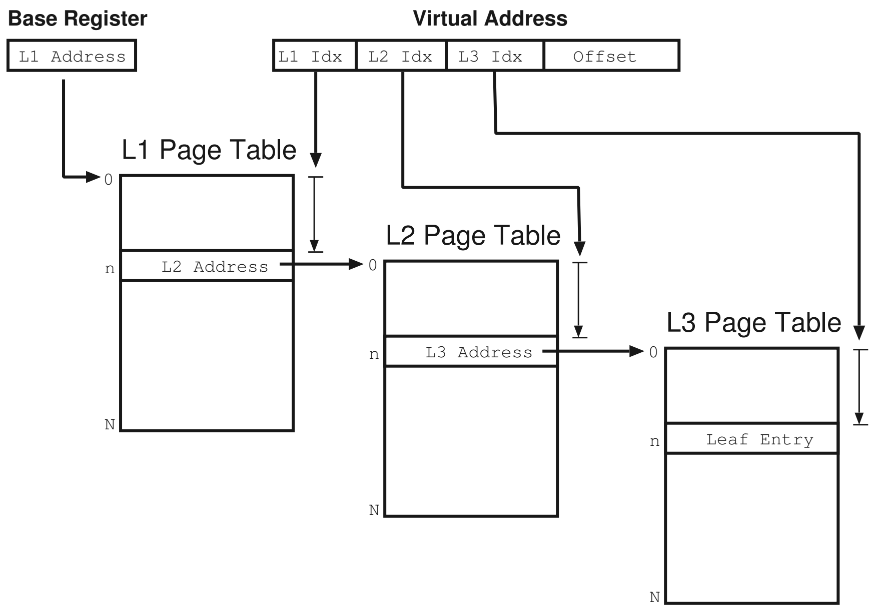


Figure 1: Multi-Level Page Tables

The location of the first array (the "root page table") is typically stored in an internal processor register, or some well defined location. Due to the simplicity of the structure, it is relatively easy to implement in hardware: this frees the programmer from concerns about address translation, and also may allow efficient interaction with the processor pipeline structure. Contemporary CPUs which provide hardware support for MPTs include the ARM series and the x86 family.

However MPTs also have a number of disadvantages:

- Potentially poor space overhead.
  While best case overhead is superb (when all mapped virtual address are contiguous), in the average case, half of every bottom level page table will be empty, almost doubling the required space for mapping, while if memory is allocated in a 'random' fashion, overhead can push towards

75%.

- Poor superpage support [a *superpage* is a naturally aligned region of 2^n pages, $n \geq 0$].
  The use of superpages can considerably improve TLB usage in certain cases, in particular if one may select an appropriate superpage size for a given application. However, MPTs effectively constrain superpage size to one of those sizes mapped by a particular level of the page table.
- Memory reference behaviour.
  The page table is always walked from the top down. This means that n memory references are typically required for a successful lookup in an n-level MPT, although superpage mappings will be successfully resolved in fewer steps.
- Inflexibility.
  As MPTs are typically implemented in hardware, they require that all levels of the page table be in physical memory at all times. Additionally, by hardwiring the choices for level-size and page table format, operating system designers have little flexibility in modifying the structures to better support alternative address space models.

In conclusion, although MPTs tend to be the mapping structure of choice for hardware implementations due to their simplicity, they have several undesirable properties.

1.2 Linear Page Tables

For machines in which the TLBs are filled by software (e.g. MIPS, Alpha), one may use a slight variation on the MPT scheme: *linear page tables* (LPTs). A LPT is a large array of PTEs in the *virtual* address space, where pages of the page table are mapped on demand using a secondary lookup scheme.

Consider for example a virtual load in an application program which causes a TLB miss. The processor stalls the pipeline and vectors to a software TLB miss handler. This handler simply uses the page number of the faulting virtual address as an index into the LPT and performs a virtual load. If this load succeeds, then the TLB may be filled using the retrieved PTE, and the faulting instruction may be restated

However it is possible that this load may itself cause a TLB miss which must be handled. Some processors (e.g. the DEC alpha series) provide a separate vector for double misses; in others the writer of the TLB miss handler must somehow recognise that a nested TLB miss has occurred. Regardless of this, the faulting virtual address must somehow be translated and loaded into the TLB. This secondary page table lookup can usually be simpler than the general purpose one; the protection bits may be synthesised, for example, since one knows the mode of the faulting access. In addition, since the LPT spans a contiguous region, a standard MPT may be used with less fear of poor memory utilisation.

Figure 2 shows how a (software) implementation of LPTs works on the Alpha 21164 processor:

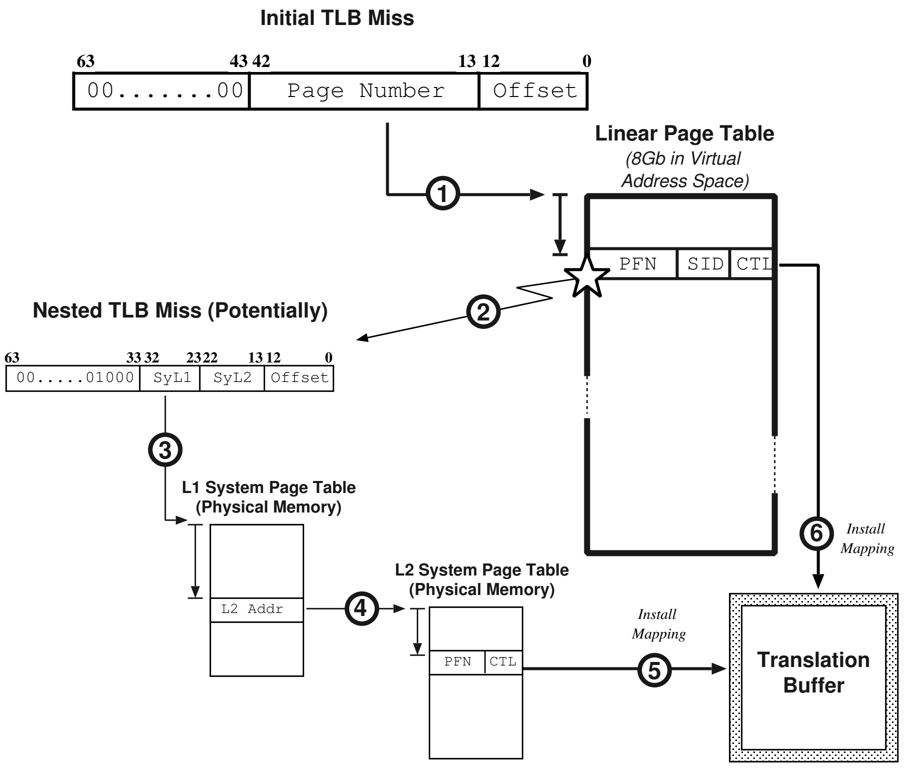


Figure 2: Translating Virtual Addresses with Linear Page Tables

Notice that the nested lookup is only required if the relevant page of the linear page table has no translation cached in the TLB. Hence if a second user-level TLB miss occurs on a page mapped by the same part of the LPT, only stages ① and ⑥ are required. This means the TLB fill routines are very efficient if the code being executed exhibits spatial locality of reference.

LPTs have several advantages:

- Good memory reference behaviour.
  In many cases, a TLB miss may be resolved with only a single memory reference. In particular if a number of consecutive TLB misses occur within a "page squared" (i.e. the range of virtual addresses spanned by a page in the linear page table), the cost of the secondary page table lookup is amortised across all of these.

- Simple and fast (first level) TLB miss handler.
  On most architectures, the PTE lookup can be implemented in very few instructions — determine page number, scale and add to page table base, and load PTE.

- Pageable page tables.
  On operating systems which provide an address space per process, the

space overhead of maintaining a large number of resident page tables can be large. Using a linear page table allows per-process regions of the address space to be mapped by pages which themselves may be *paged*, thus saving space.

- Potentially good sharing.
  On operating systems with multiple virtual address spaces, it is possible to share the entire LPT; only the secondary TLB fill mechanism need be context switched. In addition, certain "page squared" regions of the virtual address space can be efficiently shared between processes by use of *process id* (PID) tags, if supported by the TLB.

Despite these features, LPTs also share several of the poor properties of MPTs. In particular, the space overhead in the case of sparse mappings grows rapidly, and once again there is only limited support for variable sized pages. An LPT also requires a statically allocated region of the virtual address space, although this is seldom a problem.

Added to these is the complexity of handling nested misses; this essentially precludes hardware implementations (although the VAX did use a similar scheme), and requires the instruction set to support a "physical load" instruction (one can conceivably implement LPTs without this, but it is more difficult). Nonetheless, LPTs are an attractive solution to the mapping problem for machines with the appropriate hardware support.

1.3 Inverted Page Tables

Inverted page tables (IPTs) are based on the observation that the virtual address space is typically several orders of magnitude larger than the physical address space. Hence one should focus upon maintaining one PTE per *mapped frame* rather than per allocated virtual page. They were first used in the IBM System/38 in 1981, and more recently in the IBM 801 minicomputer.

An IPT is a fixed size hash table containing one entry for each physical frame of memory. In its simplest form, each of these IPT entries (IPTEs) holds either a virtual page number, or some distinguished value ω which identifies an invalid mapping.

When a TLB miss occurs, the page number of the faulting virtual address, vpn, is hashed to obtain a table index, k. If the kth entry contains the matching page number, then a TLB entry is constructed which maps the faulting virtual address onto the kth physical frame of memory. Hence the choice of hash function determines the mappings from virtual to physical addresses.

An inverted page table is illustrated in Figure 3.

There are several nice features of this scheme:

- If a TLB miss is taken on a page currently in memory, the successful lookup requires only a hash (hopefully not too expensive) and a single

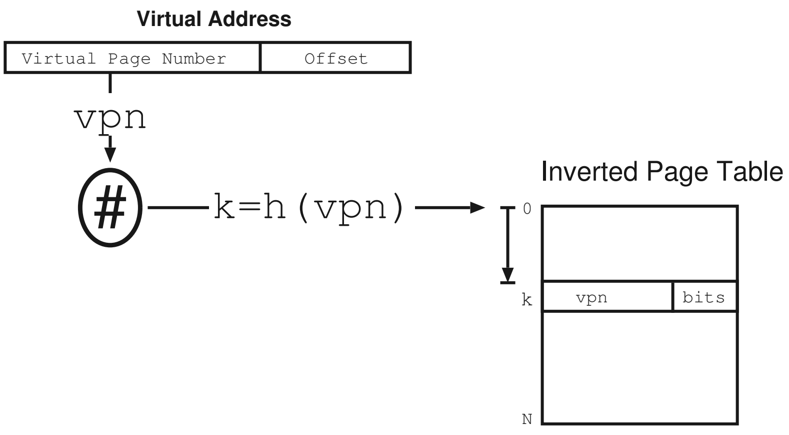


Figure 3: Inverted Page Tables

memory reference.

- The space requirements are bounded by the amount of physical memory installed in the machine. Furthermore, in the simple form described above, IPTEs tend to be small.
- Since a page number is generally shorter than a word by $\log_2 P$ bits, where P is the page size, one may easily augment the entry with protection bits.
- If one has room, it may also be possible to arrange for a process id (or similar) to be present in the IPT entries. This allows a single IPT to be used for multiple address spaces. Alternatively, since IPTs are (relatively speaking) small, it is quite practical to have one per address space.

Unfortunately, there are a number of problems also.

- In case of hash collision, one must decide either to use open hashing (thus adding a pointer to every entry) or closed hashing. Use of either solution means that TLB misses on resident pages are no longer guaranteed to be resolved in one memory reference.
- As the frame to which a page maps is implicit in the hash function (unless using closed hashing), one cannot easily benefit from techniques such as page colouring, or hardware features such as superpages.
- Aliasing is not possible except by virtual addresses which hash to the same value; and in this case, one has the collision problem described above, but may only resolve it by open hashing.
- If one has a system IPT with process ids as postulated above, then *sharing* pages between address spaces may also require that IPTEs have space for multiple process ids, or that the are chained as in the aliasing case.
- Any virtual addresses which do not map to a physical frame are not covered by the IPT; information about these must be stored in some other page table. If this secondary page table is in core, then the memory

overhead will degrade once some pages are non-resident. If, on the other hand, this secondary page table is on disk, then distinguishing between page faults and invalid page references becomes expensive.

Solutions have been proposed to a number of these problems. Hashed paged tables (HPTs) are a variant of IPTs which, in addition to the virtual page number, store the physical frame number in the IPTE, and are the form of page-table suggested by the PA-Risc architecture.

Figure 4 shows how a hashed page table might work.

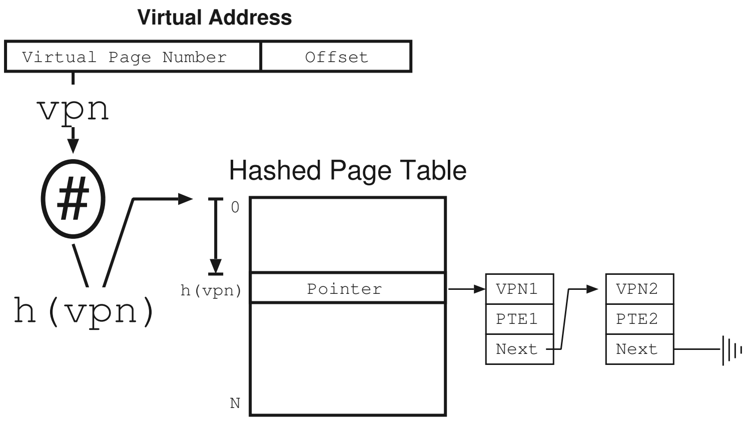


Figure 4: Hashed Page Tables

This allows the frame to which a page maps to be chosen (allowing potential benefit from page colouring and superpage mappings) and enables aliasing of arbitrary virtual addresses. Furthermore the size (in terms of number of entries) of a HPT is not required to be the same as the number of physical frames.

These improvements do not come for free: HPTEs are larger than IPTEs both since they hold the frame number. They also tend to contain a pointer for chaining entries since this is necessary once aliasing is to be supported.

In conclusion, while IPTs appear to offer $O(1)$ in both space and time, in general they achieve neither due to problems with with aliasing, sharing and non-resident pages. HPTs solve the first two of these problems (through increasing the memory overhead to approximately three words per mapped resident page), but still do not address the issue of non-resident pages.

1.4 Guarded Page Tables

As seen earlier, multi-level page tables have certain problems: in particular they have poor space overhead when the virtual address space is used sparsely. J. Liedtke of GMD recently proposed *guarded page tables* as a modification of n-level page tables which hope to solve this problem.

The basic idea is the same: a virtual address is translated by successively decoding a subset of its bits, starting from the most significant, obtaining a series of intermediate page tables and culminating (in the successful case) with a PTE. But, rather than hardwire the number of bits translated at each stage, a GPT is constructed in such a way that variable length prefix of a given virtual address is decoded in each stage.

This is achieved by the use of the two following items:

1. A *page table pointer* (PTP) is logically a tuple (p, s) where p is the address of the base of a guarded page table and s is its size (in terms of number of entries).
   The use of PTPs rather than simple pointers allows the use of variable size page tables in the multi-level tree.
2. A *guarded page table entry* (GPTE) is a tuple (g, s, u) where g is a bit string called a *guard*, s is the length (in bits) of the guard, and u is a discriminated union holding either a PTP or a PTE.
   The use of GPTEs rather than PTEs means that a certain amount of the translation process can occur "in place", thus providing an alternative to additional levels for sparse mappings.

These two concepts complement each other nicely, although either could be used on its own with certain restrictions. Figure 5 illustrates their potential use in comparison with MPTs.

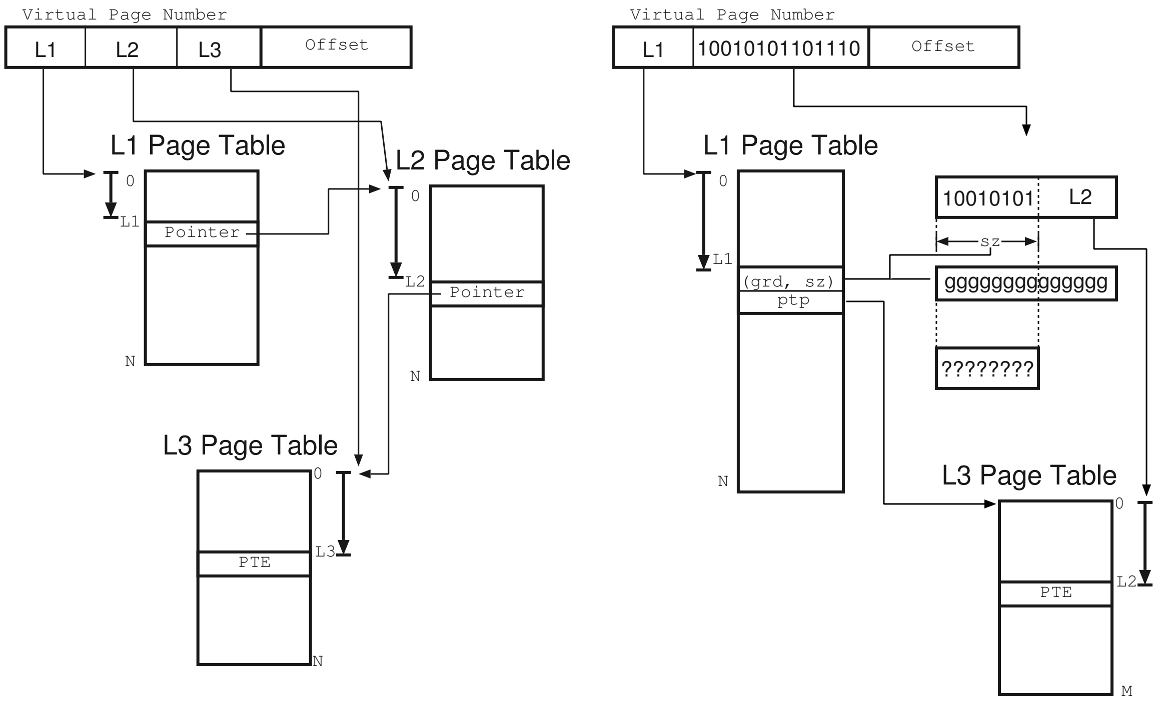


Figure 5: Guarded Page Tables

On the left hand side is the standard multi-level lookup: the virtual address is divided into three fixed bit strings labelled "L1", "L2" and "L3", and these are used as successive indices into fixed size page tables. The size of the each page table is $2^{|Ln|}$ entries, where $|Ln|$ is the length of the bit string "Ln".

On the right hand side, we see the guarded page table scheme. Once again the first lookup takes place using a fixed number of the most-significant bits of the virtual address — but here the size of the level one page table (and hence the length of the bit string "L1") is determined by the root page table pointer.

The first level lookup gives us a GPTE containing a guard "grd" of "sz" bits. This is compared against the remaining prefix of the virtual address and, if it matches, the process continues with the second level page table whose address and size in entries, M, are given by the "ptp" field of the GPTE. In the illustration there are exactly $\log_2 M$ bits left to decode and so after indexing into the level two page table, our translation is complete. In the more general case, there could be zero or more guard bits at any stage in the translation.

Hence GPTs allow variable page table sizes and a variable number of translation steps. This makes them more complicated than the previous schemes — their very flexibility means that when adding a virtual address to the page table, there are a number of valid possibilities as to where the new PTE should go.

Liedtke has done some theoretical work on the space overhead and depth of guarded paged tables. Some of his key results deal with *binary guarded page tables* (BGPTs): these are trees in which every page table contains exactly two entries. Using these he manages to prove:

1. For any size of address space, and any size of page, there exists a BGPT G which can map k pages such that $|G| \leq 2k|GPTE|$, where $|G|$ is the total size of the page table, and $|GPTE|$ is the size of an individual guarded page table entry.
2. Given a virtual address space n bits wide, every BGPT G which maps k pages, for any size of page, satisfies $log_2 k \leq d(G) \leq \max(k-1, n)$, where $d(G)$ is the *depth* of G — the maximum number of translation steps required for any page mapped by G.
3. Given a virtual address space n bits wide with pages of size $s = 2^i$ there exists a BGPT G which can map k pages such that: $d(G) \leq \lceil (n-i)/2 \rceil$.

The first result gives an upper bound on the space overhead of GPTs, but only by existence. There is no guarantee that a given GPT will conform to the condition.

The second result, while looking impressive on the surface, actually only bounds the depth from above by one less than the number of pages — hence a GPT mapping 4Gb of virtual memory composed of 8K pages sizes has a depth "bounded" only by 511; a large number of memory references in order to translate an address. Even the lower bound is not superb: 8K pages require at least 13 translation steps!

The third result is only a slight improvement: it reduces the upper bound on depth to $\lceil (n-i)/2 \rceil$, which for a 64-bit address space with 8K pages works out as 26 steps. And, as in the case of the first result, it is once again only an existence proof. One cannot even be sure that the depth of a given BGPT will be bounded by this (rather large) number.

Clearly one could improve the bounds on depth by allowing larger page tables, but it seems intuitively clear that doing this will increase the bound on overall space requirements. Indeed, Liedtke arrives at:

> Given an address space n bits wide with pages of size $s = 2^i$, there exists a GPT G which can map k pages such that: $d(G) \leq \lceil (n - i)/m \rceil$ **and** $|G| \leq 2^{m-1}k|GPTE|$, for $m > 1$.

This may be considered a generalisation of results 1 and 3 for the binary GPTs, and shows quite clearly that there is a trade-off between maximum size and maximum depth. This is illustrated in Figure 6.

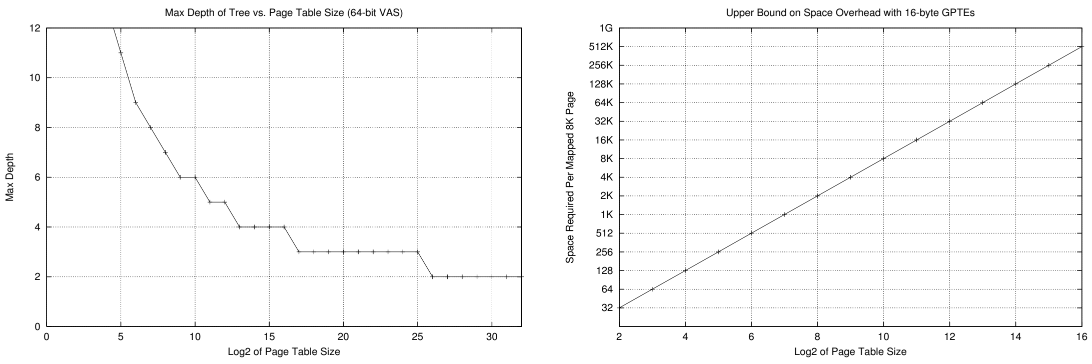


Figure 6: Upper Bounds on GPT Depth and Size

The difficulty with this trade-off is that there seems to be no particularly good solution to the dual problem. In addition, even these upper bounds are based, once again, upon an existence proof — actually constructing (and maintaining) such trees may be difficult.

Liedtke does suggest some improvements to his scheme, most notably *restricted guarded page tables* (RGPTs) which cut down on space overhead by combining the guard and the PTE into a single word. Other, rather less practical proposals include performing translations in parallel, a set of local transformations which may be applied to a tree to "improve" it's properties, and a side-entry cache to allow short cuts in the translation process.

In summary, guarded page tables look uniformly good in available publications, but tend have subtle flaws when one looks at them in detail. The depth versus space trade off of GPTs is an interesting issue which is difficult to resolve.

1.5 Clustered Page Tables

Another recent solution to the mapping problem are *clustered page tables* (CPTs), proposed by Talluri *et al.* These are a modification of hashed page tables in which each entry in the table contains a *cluster* of PTES — a set of PTEs for contiguous pages in the virtual address space. CPTs are illustrated in Figure 7.

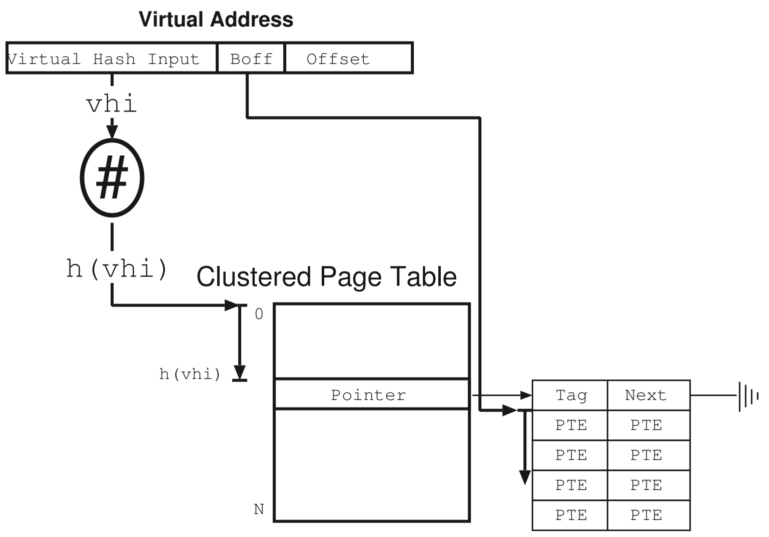


Figure 7: Clustered Page Tables

As can be seen, the virtual page number is split into two parts: the virtual hash input (VHI), and the block offset (BOFF). The first part is hashed and used as an index into the CPT. Chained out from this are a number of clustered page table entries (CPTEs), each of which holds a single tag (which should match the VHI) and a set of n PTEs, where $n = |BOFF|$. CPTEs also contain pointer fields to allow the chaining of entries whose VHI hashes to the same value.

There are some benefits when using such a scheme:

- The cost of the tag and the pointer are amortised over a number of PTEs. In the diagram, for example, eight PTEs share a single tag and pointer. Assuming locality of reference, lookups will also benefit more from caching than in the HPT case.
- In the case where the TLB supports various intermediate superpage sizes, it is possible for the CPTE to be reduced in size. For example, the Digital 21164 processor supports superpages of 64K — these could be represented in the CPT by a CPTE which contains only one PTE (plus some sort of marker that this is a superpage). Support for *large* superpages, however, it poor.

Talluri *et al* go on to illustrate that CPTs are also suitable for supporting *partial subblock* TLBs. A partial subblock TLB entry is rather like a superpage entry, but where the validity of each subpage entry is indicated by a valid bit vector. Hence the benefit of using a single TLB entry may be gained even if a small number of subpages are not currently resident,

The space efficiency of CPTs is good: space overheads depend mainly on the number of PTEs within the CPTE which are actually in use. CPTEs improve

	Time		**Space**		**Superpages**	
Type	*#Refs*	*Cache*	*|PTE|*	*Sparsity*	*Small*	*Large*
MPTs	∼5	✔	1	✘	✘	✔
LPTs	1.x	✔	1	✘	✘	✔
GPTs	2–11	✔	2	✔	✔	✔
IPTs	1	✘	1	✔	✘	✘
HPTs	∼2	✘	2	✔	✘	✘
CPTs	∼3	✘	1.x	✔	✔	✘

Table 1: Summary of Page-Table Properties

upon space overhead (compared to HPTs) only if one can ensure that on average $\geq$ 50% of the PTEs within a CPTE are full. This appears to be the case, however, for a number of workloads — in a recent paper, a number of applications which spend a considerable amount of time in TLB miss handlers were simulated using a variety of page table schemes. In all cases the space overhead for CPTs was comparable or better to MPTs or LPTs.

The main disadvantage with using CPTs is, as with HPTs, the way in which the average length of a chain increases as the amount of memory mapped increases. Clearly it would be more sensible to use a large enough hash table, but in the case that non-resident pages are being mapped, the optimum size can be difficult to precompute. Rehashing techniques could be used to overcome this problem.

Hence CPTs are somewhat like GPTs in that they present a trade-off between space overhead and access time (in terms of memory references required). However the trade-off in this case is easier to understand:

- Increasing the size of the hash table increases the value of the "critical point" at which access time becomes a problem, but adds to the fixed space overhead of the page tables.
- Increasing the size of the CPTEs likewise increases the value of the critical point, but *may* significantly increase the space overhead, depending on the average density of virtual address space usage per cluster.

Both of these seem eminently tunable for a given architecture, operating system, and application load. Furthermore, choosing a sensible size of CPTE can also enable the exploitation of underlying hardware features such as superpage or subblock TLBs.

1.6 Discussion

Each of the page table structures examined has a number of advantages and a number of shortcomings. These are summarised in Table 1.

The first two columns look at the performance of the page tables in terms of *time*, first considering the average number of memory references required to perform a translation assuming a 64-bit virtual address and 8K pages. Clearly these are only a rough guide, since actual figures would depend on implementation.

The second of the pair looks at the level of "cache friendliness" the exhibit — that is, whether locality of reference in the virtual address maps onto locality of reference within the page table. The first three schemes do well in this regard, while the hashing solutions do not (although CPTs will generally do better than either IPTs or HPTs).

The next pair of columns focus upon *space* overhead, both in terms of the number of words required for a PTE, and the behaviour in the face of sparse mappings. Again these are estimated figures and would depend on implementation — for example the *restricted* form of GPTs reduce the PTE to a single word, albeit by losing true 64-bit support.

The final two columns illustrate whether support is present for small (e.g. 64K) or large (e.g. 4Mb) superpages, where 'support' means that replication of PTEs is not required. GPTs are by far the most flexible in this regard, supporting an arbitrary range of page sizes, while IPTs and HPTs are poor.

It is not easy to choose a clear winner. However given that *small* superpages are difficult to use in operating systems due to physical memory fragmentation, it appears that either GPTs or LPTs would be a sensible choice, depending on the estimated amount of support required for sparsity. CPTs would be a good solution in the presence of subblock TLBs.

2 Virtual Memory Case Studies

2.1 Unix

UNIX was developed in Bell Labs by a number of members of the team that had been involved in the costly, complex and incomplete Multics project. Starting with Ken Thompson's development of a simple time sharing system for the PDP-7 in 1969, the system quickly gained adherents and by 1971 a working version on the PDP-11 was in daily use. The worldwide popularity of the PDP-11 series in the early 70's, and the availability of UNIX (including source code) for a nominal fee ensured its success.

Unlike Multics, UNIX did not make use of segments as the fundamental operating system objects. Rather, it explored the use of *files* as a unifying abstraction: in addition to "standard" files — named containers for non-volatile data — UNIX represented I/O devices by special files. For example, the file `/dev/mnt` might represent a block-oriented magnetic tape device. Reading bytes from this file causes the operating system to read blocks from the tape; similarly writing bytes to the file caused blocks to be written onto the tape. The same kind of scheme was used for character mode devices such as teletypes. In contemporary systems, the file abstraction is also used for various forms of inter-process communication, and for networking.

This emphasis on files did mean, however, that there was no inherent solution for dealing with a large number of processes (or, indeed, for individually large processes). A process's text, data, `bss` and stack were no longer implicitly backed on disk by virtue of being contained within segments. Partly as a result of this, and partly due to the limitations of the initial hardware on which UNIX had been implemented, early versions of UNIX provided no support for paging, or even for partial segment swapping — instead, an entire process was either resident in core, or swapped out to disk.

In order to minimise the amount of information which must be kept resident, the UNIX kernel divided the information it kept about a process into two kinds. Essential items such as the `proc` structure (which held the process id, priority, state, etc.) and the `text` structure (which recorded information about the process's text segment) were always kept resident. Other information, including the per-process page tables, the process's kernel stack, and the `user` structure could be swapped out.

The control of the swapping mechanism was intimately tied in with the scheduling process: a special process called the *swapper* would periodically awaken and inspect the queue of processes currently swapped out to disk. Assuming that at least one process had become runnable, the swapper would select the one which had been on disk for the longest time and prepare to swap it in.

Swapping one process in usually required swapping another out; the victim was chosen by inspection of the scheduler queues in the hope of locating a process blocked on I/O. If multiple such processes existed, a decision was made based on the process priorities, and the amount of time each had been resident. In order to provide some stability, a process recently[1] swapped in was not an eligible victim to be swapped out.

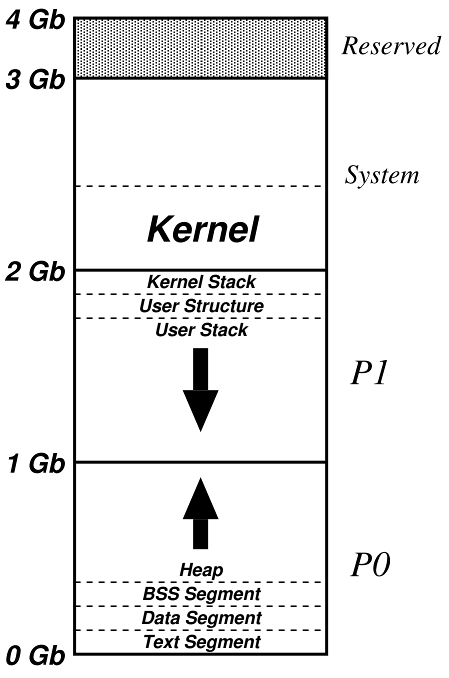
As has been covered in lectures, swapping has a number of problems. Hence later versions of UNIX, starting with the 3rd Berkeley release (3BSD), implemented demand paging in addition to swapping. All subsequent versions of BSD UNIX have had support for demand paging, as have System V variants from Release 2, Version 4 onwards. In these systems, swapping only occurs when there is a dire shortage of physical memory.

For simplicity, the remainder of this discussion will focus on the virtual memory support in 4.3BSD; the scheme used in System V is only slightly different, while the 4.4BSD virtual memory system is largely based on Mach (see Section 2.3).

The virtual address space and translation mechanisms in 4.3BSD are borrowed from the VAX architecture. The 32-bit address space is divided into four 1Gb

[1]Various quanta of time were used in different implementations, typically of the order of a few seconds.

regions, with the bottom two, $P0$ and $P1$, being used for processes, and the third, called the *system* region, being reserved for use by the kernel[2]



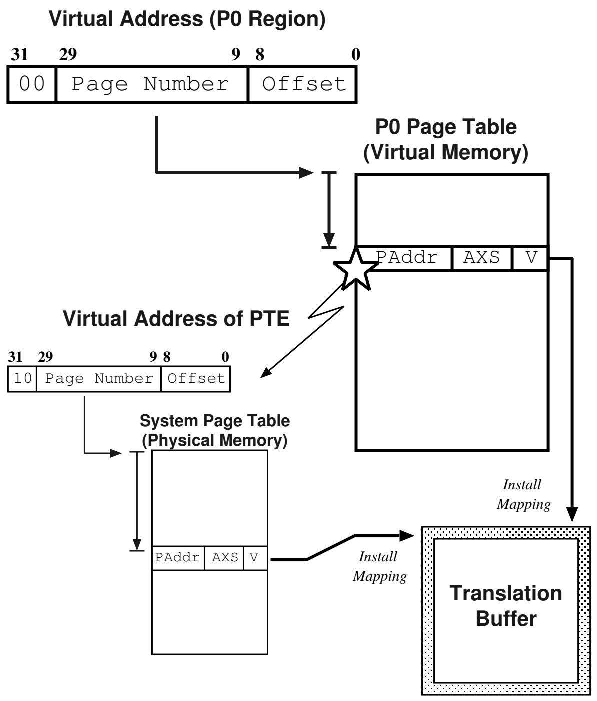


Figure 8: 4.3BSD Address Space Figure 9: Address Translation on the VAX

On the VAX, address translation is implemented in hardware and uses three page tables, one each for $P0$, $P1$ and the system region. Each of these page tables is specified by a pair of internal processor registers which determine the base and length of the table. A virtual address is translated by using its page number (i.e. bits [29:9] of the address) as an index into the relevant page table (as determined by bits [31:30]). Hence if a page table has n page table entries (PTEs), it will map a contiguous region of n pages within the relevant region. In 4.3BSD, the $P1$ region is used to hold the process stack, and hence grows downward; the other regions grow upward. This is illustrated in Figure 8.

During address translation, the system page table was accessed using physical addresses, and hence was always entirely resident. The $P0$ and $P1$ page tables for each process, however, were located in the system region, and hence were accessed using system virtual addresses. This means that the lookup of a virtual address in one of the lower regions could require a recursive lookup in the system page table, as illustrated in Figure 9. This allowed process page tables themselves to be paged, which reduced the minimum cost of a resident process.

For valid pages, the format of the PTE is determined by the hardware and includes the 21-bit frame number, protection bits, and a modified bit. If the valid bit is not set, any access will cause a page fault and the handler will

[2]The top 1Gb region was not supported by the original VAX hardware.

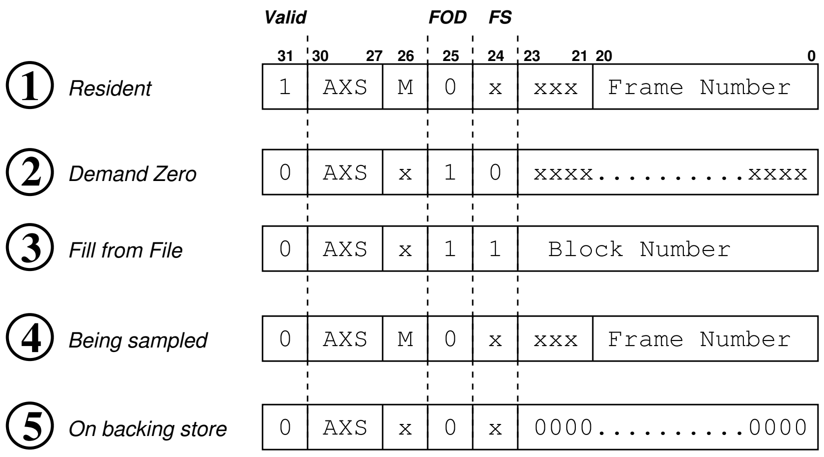


Figure 10: Use of PTEs in 4.3BSD

examine the faulting page's PTE to determine what state the page is in. This is achieved by looking at the *fill on demand* (FOD) bit, the *filesystem* (FS) bit and the 24-bit block number, as shown in Figure 10.

The demand zero (DZ) PTE is used for pages within the uninitialised data area (viz. `bss`) and for dynamically growing the stack — each process will have a fixed number of DZ PTEs installed in $P1$ when it is created[3]. The fill-from-file (FFF) PTE is used for the demand paging *in* of text and initialised data from an executable file. For such pages, 4.3BSD implements a simple pre-paging scheme by using page *klusters*. A kluster is a set of FFF pages which are adjacent both in memory and within the file-system. If a page fault occurs on an FFF page, an entire kluster may be fetched rather than just the faulting page. If the process exhibits locality of reference, this scheme improves performance since the cost of the disk transaction setup is amortised over a number of pages.

The fourth type of PTE is used on the VAX in order to simulate a reference bit, since none is supported by the hardware. Pages are periodically marked invalid despite having a frame in core. Then, if they are accessed, a page fault is taken, the software reference bit is updated, and the valid bit is set. This type of PTE can also occur when the page's frame has been freed, but not yet reallocated. In this case it is possible to *reclaim* the frame and continue.

The final form of PTE represents the case when a page has been swapped out to the backing store. This includes pages which were originally pre-loaded, filled on demand from the file-system, or filled with zeroes. Since the block number in the PTE is zero, it is not possible to use this field to locate the data (as is done with fill-from-file pages). Instead, each process has two *swap maps* which store the on-disk location of pages in $P0$ and $P1$ respectively. The location of text pages is determined via a separate swap map since text may be shared by multiple processes.

[3]On most UNIX systems, this amount can be specified by the `ulimit` command.

Physical memory in 4.3BSD is managed by means of an array of structures called the *core map*. This maintains information about all physical frames save those reserved for the kernel. As the page size on the VAX is only 512-bytes, the core map actually keeps track of physical memory in terms of *clusters*, where a cluster is group of adjacent and naturally aligned frames. Typically a cluster is configured to be 1K in size, and so represents a pair of frames.

For each cluster the core map records:

- if it is free or in use,
- the page number it is mapped under,
- its associated disk block, and
- miscellaneous status bits.

Obviously, not all of these will be applicable to all clusters.

All clusters which are not currently in use are members of the *free list*, which threads through the core map. When a page fault occurs, the system tries to find a cluster to map under the page from this list; if is is empty, then resolution of the page fault must wait. In order to reduce the probability of this occurring, 4.3BSD uses a page dæmon.

All page replacement is carried out by the page dæmon. The algorithm used is second-chance FIFO, implemented using a variant of CLOCK which uses two 'hands' instead of one. Each hand points to an entry in the core map, a fixed distance apart. Every 250ms, the system checks if there is enough physical memory free (as determined by a system parameter, *lotsfree*). If there is, nothing need be done. Otherwise the page dæmon is woken and instructed to perform a number of iterations based on the scarcity of physical memory.

On each iteration, the front cluster is first examined to see if it is a candidate for replacement. If it is not, then the dæmon clears the the valid bits of the cluster's PTEs. The PTEs of the cluster pointed to by the back hand are then sampled — if the page has been referenced since the front hand passed, it is given a "second chance", and left alone. Otherwise, the cluster is written to disk if it was dirty, and then placed on the end of the free list. This "freeing" is speculative: the cluster retains its contents, and should a page fault occur before it is reused, then it may be reclaimed, as described above.

Once both front and back clusters have been processed, both hands are incremented. The process continues until sufficient iterations have occurred, or until *lotsfree* memory is available again. At this stage, the page dæmon will sleep again.

The System V virtual memory system is almost identical to that described above. Two significant differences are:

1. it uses a single-handed CLOCK, but requires that a page be unreferenced for multiple revolutions before it is considered for replacement.

2. it uses a pair of system parameters *min* and *max* to control the page dæmon — it is run only if free memory falls below *min*, but stops only if at least *max* clusters are free. This adds some stability over the BSD scheme.

In summary, contemporary Unix implements a demand-paged scheme which aims to maximise the level of multi-programming achievable for a given mix of processes. From the beginning, the fundamental abstraction in Unix was the *file*, rather than the segment. This led to novel features such as the simple redirection of input and output streams, and the use of special files to represent I/O devices. However it does mean that certain features (most particularly memory mapped files) have had to be "tacked on" to the virtual memory system.

2.2 VMS

VMS was released by DEC in 1978 to run on their then recently introduced VAX-11/780 minicomputer. From the beginning, the intention was that VMS should operate efficiently on a wide variety of hardware configurations, and should support a mix of real-time, timeshared and batch jobs. These goals directly affected the design of the virtual memory system: supporting real-time tasks required minimising the effect of heavily paging programs on other parts of the system. In addition, the design could not rely on special hardware (e.g. fixed-head paging disks) since the planned range of VAX/VMS systems would include low-cost workstations.

These constraints led to a design with the following characteristics:

- A *local* page replacement scheme,
- A *quota* scheme for physical memory, and
- An aggressive *page clustering* policy.

The original implementation on the VAX-11 is described in "Virtual Memory Management in the VAX/VMS Operating System", by Levy and Lipman, Computer, 1982. The layout of virtual address space for system and process use is almost identical to that used by 4.3BSD, with the exception that the image for the command line interpreter was held within the $P1$ region.

The use of memory by processes in VMS is based on the idea of a *resident set* — the resident set of a process is simply the set of its pages which are currently in memory. At process creation time, the *resident-set limit* is specified (or set to a default value). This determines the maximum number of pages a process may have resident at any time.

When the process begins execution, its resident set is empty. As it progresses, each non-resident page it references is faulted in by the pager until such time as the process's resident set limit is reached. If the process subsequently faults on a page, the pager will select a victim from the faulting process's resident set to be replaced. This reduces the impact that a heavily faulting process may have on other processes since it will not receive a disproportionally large share of physical memory.

Since the VAX did not support page referenced bits, the designers of VMS chose to use a simple FIFO replacement scheme. As pages are faulted into a process's resident set, they are added to the tail of a list; when choosing a replacement, the head of the list is chosen. While this has the advantage of simplicity, a pure FIFO replacement policy tends to perform poorly.

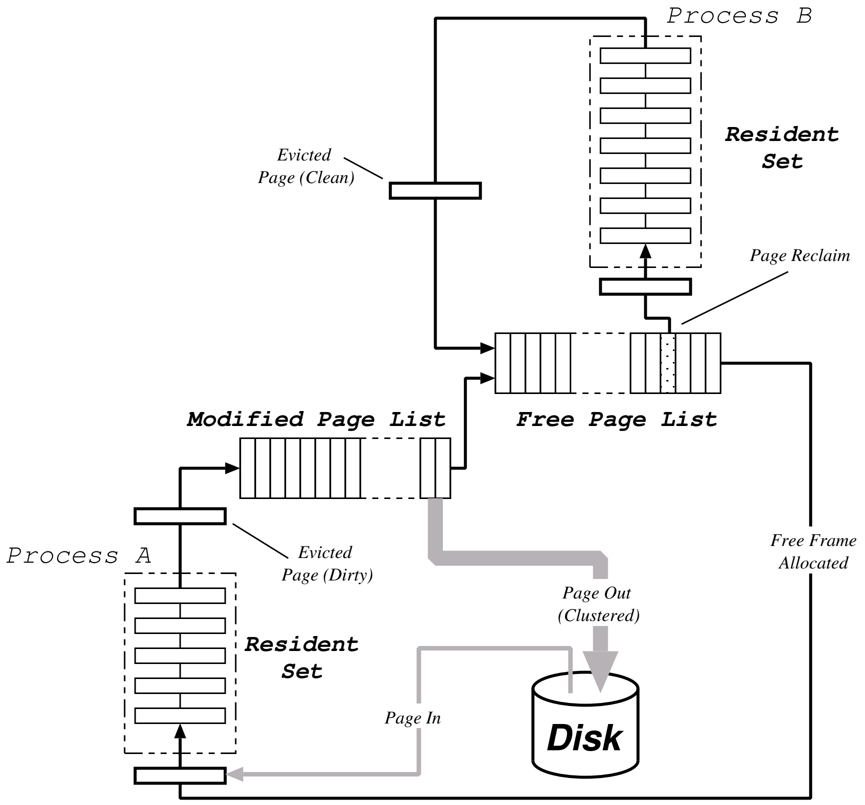


Figure 11: Page Lists in VMS

VMS handles this shortcoming by means of a software "victim cache" into which evicted pages are placed. This cache is in the form of two lists — the *free page list* and the *modified page list*. The free page list is used as the source of available frames for the system (such as when the pager requires a frame into which to load paged out data), and is processed in strict FIFO order. The modified page list holds pages which need to be written to disk; once this has occurred, they may be placed onto the tail of the free page list. The interaction between these global lists and the per-process resident set lists is shown in Figure 11.

Whenever a page is selected for replacement, it is placed onto the tail of one of these lists depending on whether it is dirty or not. If a process faults on a page, these lists are searched, and if the page is found it may be re-mapped[4]. In

[4]This is similar to the way in which 4.3BSD reclaims pages which have been freed but not

addition to being relatively inexpensive (since no disk I/O is needed), this has the effect of putting the page back on the end of the resident set FIFO. Hence pages which are heavily used will remain in the resident set. Levy and Lipman argue that as the size of the software cache increases, this scheme approaches LRU replacement.

The modified list also plays a role in the clustering of pages for write-back. With moving-head disks, the time taken to position the head correctly can take considerably longer than the time required to transfer a page-worth of data to or from the disk. Since VMS wished to support the use of such disks as a backing store, it uses an aggressive page clustering scheme. Two system parameters `lo` and `hi` define low and high limits for the size of the modified page list; if the current length of the modified page list, n, reaches the high limit, then the swapper is invoked and writes ($n - \texttt{lo}$) pages to disk. The swapper is also used for the swapping in and out of entire processes under situations of resource scarcity.

Since the same paging file is shared by many processes, per-process clustering is not used in this case. Instead the swapper determines the set of pages to be written out, and then flushes the entire set to disk in one transaction. It is interesting to note that this is almost entirely opposite to the use of a page dæmon, since the write-back of pages is delayed past page replacement, rather than predicted. The benefit, however, appears to be considerable: in the Levy paper, experimental results show that a total of 11,501 pages are written back in just 117 I/O operations, an average of 98.3 pages per disk write.

Clustering is also used on read: when paging in the system checks if pages adjacent to the faulting one in the virtual address space are contiguously located on disk. If so, multiple pages may be read in with a single transaction. The maximum read-clustering size is further modified by a per-segment value, specified at link time. Read clustering performs better when paging in executables or memory mapped files: this is since the paging file is shared by multiple processes (via the modified page list) and its layout is optimised for write-back. Nonetheless, the experiments show that on average 5.7 pages are paged in per disk read.

More recent versions of VMS, while based on the same core functionality, have provided a number of enhancements. These include:

- Modified page replacement.
  Rather than perform strict FIFO replacement, the design allows for two optimisations. The first is the introduction of a *callback* which may be used by privileged processes. This routine is invoked by the page replacement code with the selected victim. Should the process consider this a poor choice, it may indicate this to the replacement algorithm.

  The second modification has to do with frequency of use: if a page which would normally be selected has a translation cached in the TLB, then it is

yet reallocated.

skipped. An upper bound is set on the number of candidates which can be skipped in this way in order to ensure that the algorithm terminates. Note that this requires some hardware support to determine whether a given entry is present in the TLB. This feature was provided as an enhancement to the VAX architecture in the early 80's.

- Automatic working set limit adjustment.
  Introduced in VAX/VMS Version 2, automatic working set adjustment attempts to grow the working set size of heavily faulting processes in order to reduce their page fault rates. In addition to the explicit `$ADJWSL` system service, a routine is invoked at quantum end which examines the process's page fault rate; real-time processes are exempt, as are any processes which do not wish the service.

  If the fault rate is high enough (as determined by a system parameter, `PFRATH`) and at most 25% of the working set is unused, the working set limit is increased by the system parameter `WSINC`. In some cases (when the free list contains a lot of pages), the limit may even be adjusted past the process's quota. If, on the other hand, the fault rate is low enough (as determined by a system parameter, `PFRATL`), the process's working set may be reduced down to a minimum size. Normally `PFRATL` is zero, and automatic working set limit reduction is disabled.

  Instead, a technique called *swapper trimming* is used. When the swapper is invoked it first tries to reclaim memory by reducing the working set sizes of those processes which previously benefited from automatic working set limit augmentation. Only if this fails to produce enough free physical memory need an entire process be swapped to disk.

VMS also provides a number of system services by which a process may modify the way in which it is treated for paging and swapping. These include `$SETSWM`, which can be used to disable process swapping, `$LCKPAG`, which allows the locking of pages into memory, and `$LKWSET`, which enables the locking of pages into the working set. The first two of these require special privilege since, used arbitrarily, they could severely impact the overall system performance. The last, however, is unprivileged as a process cannot unduly impact others — if too many pages are locked into the working set, the process can be swapped out.

Development of VMS continues — the most recent release at the time of writing is OpenVMS 7.1 which supports a variety of VAX machines and the newer Alpha based computers. The Alpha's 64-bit architecture has been fully integrated; an additional per-process region $P2$ is present above $P1$. Similarly an additional system region $S2$ is defined below $S0/S1$. This allows OpenVMS to exploit very large memory sizes.

2.3 Mach

Mach is a μ-kernel operating system which was developed at Carnegie-Mellon University (CMU) starting in 1984. It was the successor of *Accent*, an earlier CMU operating system which pioneered the idea of modular decomposition, and the use of virtual memory techniques for efficient inter-process communication (IPC). The goals of Mach were to create a high-performance object-oriented operating system with compatibility with Berkeley UNIX and support for multi-processors.

One of the key design decisions in Mach was to treat virtual memory and IPC as part of the same subsystem. This led to the development of three fundamental abstractions:

- *Ports*: a port in Mach is a process-local identifier for a kernel supported *mailbox*. The process which creates a port may grant another process the ability to access it. Ports are used for the transmission and reception of messages.
- *Messages*: a message in Mach is a collection of data objects which may be sent or received via a port. In general messages are variable length and comprise a header followed by a number of typed data records. The type can be used when communicating across a network in order to perform machine-specific conversions. In addition there is a special type of message in which the data is *out-of-line* — in this case, the transmission of the message is performed using virtual memory techniques rather than by copying.
- *Memory Objects*: a memory object is one of the basic virtual memory abstractions. Mach processes allocate *regions*, which are contiguous pieces of their virtual address space. Any initial attempt to use the address within a region, however, will cause a page fault. Instead a memory object must be associated with the region; the memory object conceptually holds the contents of the region. Several well defined operations (e.g. read, write) are defined on memory objects.

The allocation and deletion of ports, and the format and transmission of messages are all functions of the kernel. However, memory objects are not necessarily kernel provided. Instead, Mach introduces the concept of *external memory management* via user-level *data managers* (sometimes called *external pagers*). The basic idea is that the data manager — an unprivileged task — may cooperate with the Mach kernel in order to provide the contents of a memory object. The kernel maintains a cache of those pages within the memory object it knows to be resident. If a page fault occurs on non-resident page, the kernel performs IPC to the data manager requesting that it produce the contents of the page.

Page replacement takes place at the discretion of the kernel. If the page selected for replacement was dirty, the kernel will notify the relevant data manager via IPC. The data manager is then responsible for flushing that data to the backing

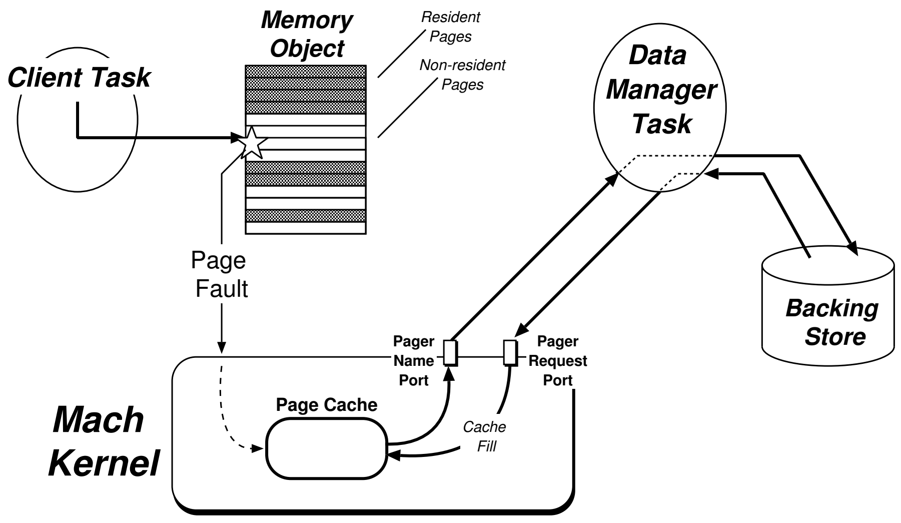


Figure 12: Mach Virtual Memory Architecture

store, if appropriate. Additional IPC services are provided by the kernel for the use of data managers including the flushing of a range of pages from the cache, and the cleaning of a page. These are necessary to support cache consistency in the case of distributed shared memory. The Mach virtual memory architecture is shown in Figure 12.

A client task has mapped a memory object into its address space using the `vm_allocate_with_pager` system service, and now accesses a non-resident page. The page fault is trapped by the kernel which checks if it has a cached copy of the page's contents. As it does not, it invokes `pager_data_request` on the data manager for that memory object. The data manager retrieves the information from the backing store and returns it to the kernel using `pager_data_provide`. In certain cases, additional information may be provided (i.e. the data manager may implement page read clustering). The kernel updates its resident page cache and installs the mapping. Once this is done, the client task may resume.

This separation of functionality between kernel- and user-space greatly simplifies the implementation of the virtual memory system. In one paper, for example, a design is sketched for a consistent distributed shared memory system which requires no changes to the kernel at all. It also means that portability considerations are restricted to the kernel; all user-space functions are forced to be portable by the use of clearly defined interfaces.

The use of external pagers is not without problems, however. Since they are not trusted parts of the system, very little can be guaranteed about their behaviour. For example, a buggy (or malicious) data manager might never reply to a `pager_data_request` invocation from the kernel. In this case, the user task waiting for its page fault to be resolved will never be resumed. It will not cause the kernel to block (since Mach IPC is asynchronous), but may tie up data structures or parts of the page cache.

There is also a possibility that the page cache could be flooded by a data manager which consistently returned far more information than was requested by the kernel. Finally, there is the problem of potential *starvation*: when modified pages are evicted, the kernel invokes `pager_data_write` on the relevant data manager, allowing it to flush the changes to disk. Until the data manager indicates completion, the physical memory underlying the modified pages cannot be reused. However, if an errant data manager fails to release the memory, the kernel could eventually run out of physical frames.

Due at least in part to this, Mach included a *default pager.* Although this was constrained to make use of the same interfaces as any other external pager, it was a trusted system component. Mach used this to prevent starvation: if a (untrusted) data manager failed to release memory within a certain time, the default pager would be invoked to flush the changes to disk. Once this occurred, the memory could be reclaimed.

In summary, the Mach external memory management architecture introduced a radical new way of looking at virtual memory. No longer need the implementation be complex and kernel-bound; instead much of the functionality could be pushed into user-space, thereby improving modularity, portability and flexibility.

2.4 Windows NT

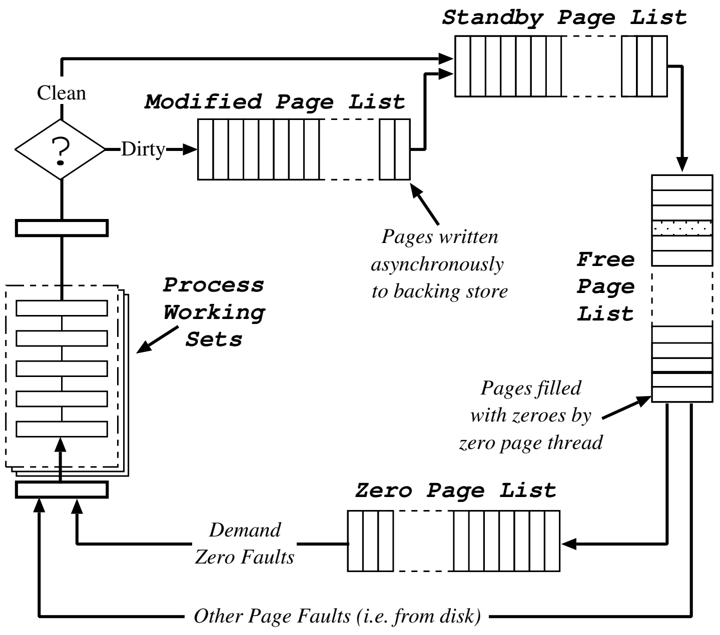
Memory management in Windows NT borrows heavily from VMS. It uses per-process working sets (aka. resident sets), local page replacement, modified and free page lists, automatic working set trimming (aka. swapper trimming), etc. The internal structure, however, is completely different, being based on *objects.* In this regard, some resemblance to Mach may be noticed.

Memory allocation is logically performed in a two stage manner. First a region of the virtual address space is *reserved*: this merely updates some internal structures called virtual address space descriptors (VADs). Any attempt at accessing the reserved memory will cause an access violation. Secondly, the region may be *committed*, which associates some "disk storage" with the region. This is either some space in the paging file (for a standard region of virtual memory), or a "real" file on disk (for memory mapped files). For convenience, the Win32 API also provides a means to both reserve and commit memory in one go.

In general, committed regions of memory are private to a process. This is useful for thread stacks, or internal heaps. If sharing is required, one must use *section objects.* A section object is the NT analog of a segment in Unix, or a memory object in Mach. It is a contiguous region of the virtual address space, and is protected (as are all NT objects) by ACLs. Sections are used by NT to map executable images, DLLs, and device drivers into memory.

Rather than allow processes to access sections directly, NT introduces another level of indirection: a *view* of a section object. This is essentially a window onto the section, allowing access to a subregion (which may be the entire section). Using this technique enables processes to conserve virtual address space. In particular it means that processes can map large files[5] into a section, but only view a smaller amount at any particular time.

As mentioned previously, the demand paging scheme bears a considerable resemblance to VMS. Figure 13 illustrates the various page lists used in NT.



Standby Page List
Clean
Modified Page List
Dirty
?
Pages written asynchronously to backing store
Free Page List
Process Working Sets
Pages filled with zeroes by zero page thread
Zero Page List
Demand Zero Faults
Other Page Faults (i.e. from disk)

Figure 13: Page Lists on NT

Note that the figure is somewhat simplified. In particular, it is possible for page faults to be satisfied from entries on the modified or standby page lists. This is similar to the *reclaim* faults seen in VMS.

Pages move from the modified page list to the standby list once they have been written to disk. This is performed by a thread running at priority 17 (a "real-time" priority) which is called the *modified page writer.* In addition, there is a *mapped page writer* thread which runs at the same priority. This is responsible for writing back dirty pages in memory mapped files. It is necessary to have both of these since the latter thread can cause page faults.

Finally, a thread called the *zero page thread* which runs at priority 0 is used to fill pages on the free page list with zeroes, and move them onto the zero page list. This is so that demand zero page faults may be satisfied more quickly. Since no other thread ever runs at priority zero, the zero page thread will only run when no other thread is runnable.

[5]The standard version of NT4 for x86 & Alpha limits the process virtual space to 2Gb; NT5 will allow up to 28Gb on Alpha.

Additional information about the Windows NT virtual memory system (including details of VAD format, and of the working set adjustment policies) can be found in “Inside Windows NT (2nd Ed)”, D. A. Solomon, Microsoft Press.

3 File System Case Studies

3.1 FAT16 & FAT32

The FAT16 filesystem (originally just called “FAT”) was developed by Microsoft in 1977 as a way to store data on floppy disks, but continued to be used for hard-disks up to about 1996. FAT16 is quite a simple file system which basically uses the “chaining in a map” technique described in lectures to manage files.

The basic idea is to treat files as a linked list of *clusters*: a cluster is a set of 2^n contiguous disk blocks, where $n \geq 0$. Other systems use the term “logical block” or “filesystem block” for this concept.

The file allocation table (FAT) consists of a 2 byte entry (= 16 bits, hence the name) for every cluster on the disk. Each entry in the table contains either:

- The index of another entry within the FAT or
- A special value EOF[6] meaning “end of file”, or
- A special value FREE meaning “free”.

A file is thus represented by the index in the FAT of its first block; the remaining blocks are found by chaining through the FAT until EOF is found. This illustrated in Figure 14.

In the example, two files are held in the FAT. The first file, $\mathcal{A}$, starts at cluster 2 and continues through clusters 8, 7, and 4. The second file, $\mathcal{B}$, starts at cluster 3 and is only 3 clusters in length: 5, 3, and $n - 2$. The starting cluster for a given file is found from that files *directory entry.* The original directory entry is shown in Figure 15.

Since FAT16 only uses 16-bits to specify clusters (in both the FAT itself, and in directory entries), it can only handle partitions of up to $(2^{16} \times c)$ bytes, where c is the number of bytes in a cluster. Hence different cluster sizes are used depending on the size of the partition, as shown in Table 2. Notice that, for standard systems, FAT16 did not allow partitions greater than 2Gb. Given that vendors in ’96 were starting to ship larger drives which they wanted to treat a single large partition (viz. `C:`), this presented a problem.

A simple solution was found by introducing FAT32, which simply extends relevant fields to use 4 bytes rather than 2. This gives considerably more freedom in handling larger partitions. In particular, the cluster size can remain relatively

[6] In practice, there are a set of these: the range 0xFF8–0xFFF in FAT16.

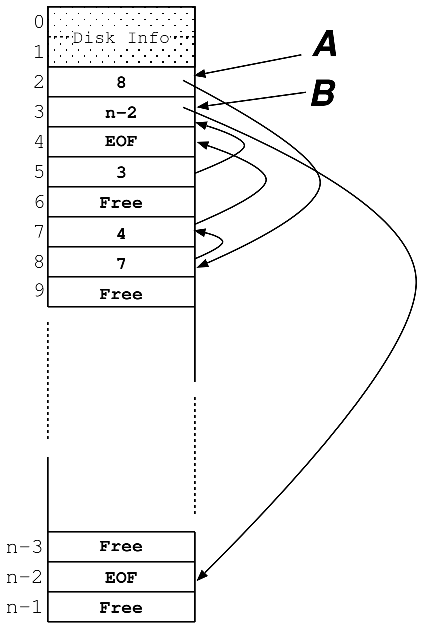


Figure 14: File Allocation Table

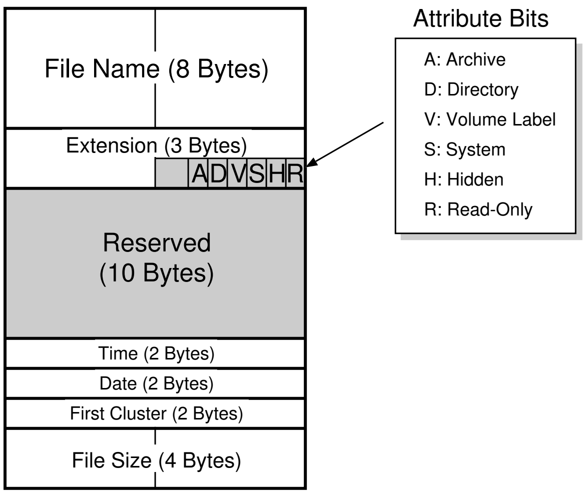


Figure 15: A FAT16 Directory Entry

small. This is a benefit since, on average, the last cluster in each file will be only 50% free[8].

Further enhancements with FAT32 include:

- Locate the root directory anywhere on the partition (in FAT16, the root directory had to immediately follow the FAT(s)).
- Use the backup copy of the FAT instead of the default.
- Improved support for demand paged executables (consider the 4K default cluster size . . .).

A final word is justified concerning VFAT (very FAT?), which is built on top of FAT32. This is the part of the system which provides support for "long" names: unicode strings of up to 256 character. This allows internationalisation, and the (convenient?) use of spaces and other special characters to name files.

Rather than redefine the directory entry structure, VFAT works with the standard entries. Since these contain only 11 bytes for the file name, additional directory entries are pressed into service to hold the remaining parts of the name. In order that these additional entries not be listed as files in their own right, the `V` attribute bit is abused: these suggest a volume label is being stored, and as such are ignored by standard file system tools. Further gory details available on the web (e.g. `http://magic.hurrah.com/~sabre/os/SFileSystems/VFATInfo.txt`)

[8]There are other concerns, since Unix traces have shown that very many small files (< 2K) exist. It is not clear that these results are valid for M$ OSes ("bloatware"?)

Partition Size	FAT16 Cluster Size	FAT32 Cluster Size
< 32Mb	512 bytes	N/A
< 64Mb	1K	N/A
< 128Mb	2K	N/A
< 256Mb	4K	N/A
< 512Mb	8K	4K
< 1Gb	16K	4K
< 2Gb	32K	4K
< 4Gb	64K[7]	4K
< 8Gb	N/A	4K
< 16Gb	N/A	8K
< 32Gb	N/A	16K
> 32Gb	N/A	32K

Table 2: Cluster Size in FAT16 and FAT32

3.2 BSD FFS

The fast file-system (FFS) was developed for 4.3BSD in an attempt to overcome some of the shortcomings of the original UNIX scheme (the "old file-system). The two most critical problems were:

1. Data/Metadata layout.
   The old file-system placed the metadata (superblock, the free list, inodes, etc) at the start of the partition, and the actual data following this. Separating things in this way meant that long seeks were almost guaranteed when accessing files. Furthermore, a head crash near the start of the disk can be disastrous.

2. Data block allocation.
   As mentioned before, files tend to be small: a recent study ('93) showed that the average file size was 22K, but that the *median* file size was $\sim$ 2K. The old file-system used a small (512 byte) allocation size for (data) blocks, which reduces the internal fragmentation seen with small files. But: smaller blocks increase the number of seeks required to read a given amount of data.

The FFS set out to address these two issues. The former was handled by the introduction of a new way to layout metadata on disk. Recall that seeks between two blocks in the same cylinder are relatively cheap: in the best case (consecutive sectors on a track), the "seek" is free. In the worse case, one has a head switch and a rotation time. Although this latter seek time may be considerable, the additional traversal time (when the arm moves to another cylinder) is avoided.

Hence FFS introduced the idea of *cylinder groups*. A cylinder group is a set of contiguous cylinders with its own metadata, including a block allocation bitmap and some inodes. Ideally, the FFS will try to allocate a file an inode and its data blocks from the same cylinder group. Doing this reduces the average seek time when doing file reads (even when inodes are cached).

In addition, FFS strives to be more resilient. Each cylinder group contains a copy of the file-system superblock at a (computed) offset from the start of the first cylinder in the group. Hence superblocks are replicated evenly between different cylinders and platters, allowing partial recovery of the file-system even in the event of mechanical failure.

With regard to block allocation, a trade-off is made. Using larger blocks reduces the book-keeping overhead, and improves performance, but smaller blocks give better space utilisation. FFS choose to have both: blocks are relatively large (e.g. 2K, 4K, 8K), but the last block in a file *may* be split into fragments of e.g. 512 bytes.

Using this scheme complicates the implementation somewhat, mainly for any parts of the file system involved in allocating new blocks to a file. This only occurs when a file is created, extended, or truncated, and hence read-performance (which is the main sort of file-system traffic) is not affected. Nevertheless, some modern FFS-based file-systems use cylinder groups but *not* block fragments one the principle that storage space is cheap and slow: and so one should attempt to optimise time rather than space.

More details on FFS, including the scheme used to handle cylinder group overflow and the design of the buffer cache, are available in chapter 7 of "The Design and Implementation of the 4.3BSD UNIX Operating System".

3.3 OS/2 HPFS

This section presents some information about the OS/2 high-performance file system (HPFS) developed by M$ in the late 1980's. The below text is *not* mine, but is edited down from an article in Microsoft Systems Journal, "Design goals and implementation of the new High Performance File System", R. Duncan, 1989. The full article is available on `http://www.lionsgate.com/home/baden/public_html_index/hpfs.txt`.

Note also that the current NTFS implementation is strongly influenced by that of the HPFS; one major change is that NTFS uses *clusters* (variable size) like FAT rather than the fixed sector size used in HPFS. Up-to-date information on the NTFS file system may be found in "Inside Windows NT (2nd Ed)", D. Solomon, Microsoft Press, 1998.

3.3.1 HPFS Volume Structure

IBM-compatible HPFS volumes use a sector size of 512 bytes and have a maximum size of 2199Gb (2^{32} sectors). An HPFS volume has very few fixed structures. Sectors 0-15 of a volume (8Kb) are the *BootBlock* and contain a volume name, 32-bit volume ID, and a disk bootstrap program. The bootstrap is relatively sophisticated (by MS-DOS standards) and can use the HPFS in a restricted mode to locate and read the operating system files wherever they might be found.

Sectors 16 and 17 are known as the *SuperBlock* and the *SpareBlock* respectively. The SuperBlock is only modified by disk maintenance utilities. It contains pointers to the free space bitmaps, the bad block list, the directory block band, and the root directory. It also contains the date that the volume was last checked out and repaired with CHKDSK/F. The SpareBlock contains various flags and pointers that will be discussed later; it is modified, although infrequently, as the system executes.

The remainder of the disk is divided into 8Mb bands. Each band has its own free space bitmap in which a bit represents each sector. A bit is 0 if the sector is in use and I if the sector is available. The bitmaps are located at the head or tail of a band so that two bitmaps are adjacent between alternate bands. This allows the maximum contiguous free space that can be allocated to a file to be 16Mb. One band, located at or toward the seek center of the disk, is called the directory block band and receives special treatment (more about this later). Note that the band size is a characteristic of the current implementation and may be changed in later versions of the file system.

3.3.2 Files and Fnodes

Every file or directory on an HPFS volume is anchored on a fundamental file system object called an *Fnode* (pronounced "eff node"). Each Fnode occupies a single sector and contains control and access history information used internally by the file system, extended attributes and access control lists (more about this later), the length and the first 15 characters of the name of the associated file or directory, and an allocation structure. An Fnode is always stored near the file or directory that it represents.

The allocation structure in the Fnode can take several forms, depending on the size and degree of contiguity of the file or directory. The HPFS views a file as a collection of one or more runs or extents of one or more contiguous sectors. Each run is symbolized by a pair of doublewords-a 32-bit starting sector number and a 32-bit length in sectors (this is referred to as runlength encoding). From an application program's point of view, the extents are invisible; the file appears as a seamless stream of bytes.

The space reserved for allocation information in an Fnode can hold pointers to as many as eight runs of sectors of up to 16Mb each. (This maximum run size is a result of the band size and free space bitmap placement only; it is not an inherent limitation of the file system.) Reasonably small files or highly contiguous files can therefore be described completely within the Fnode.

HPFS uses a new method to represent the location of files that are too large or too fragmented for the Fnode and consist of more than eight runs. The Fnode's allocation structure becomes the root for a B+ Tree of allocation sectors, which in tum contain the actual pointers to the file's sector runs. The Fnode's root has room for 12 elements. Each allocation sector can contain, in addition to various control information, as many as 40 pointers to sector runs. Therefore, a two-level allocation B+ Tree can describe a file of 480 (12 × 40) sector runs with a theoretical maximum size of 7.68Gb (12 × 40 × 16Mb) in the current implementation (although the 32-bit signed offset parameter for DosChgFilePtr effectively limits file sizes to 2Gb).

In the unlikely event that a two-level B+ Tree is not sufficient to describe a highly fragmented file, the file system will introduce additional levels in the tree as needed. Allocation sectors in the intermediate levels can hold as many as 60 internal (nonterminal) B+ Tree nodes, which means that the descriptive ability of this structure rapidly grows to numbers that are nearly beyond comprehension. For example, a three-level allocation B+ Tree can describe a file with as many as 28,800 (12 × 60 × 40) sector runs.

Run-length encoding and B+ Trees of allocation sectors are a memory-efficient way to specify a file's size and location, but they have other significant advantages. Translating a logical file offset into a sector number is extremely fast: the file system just needs to traverse the list (or B+ Tree of lists) of run pointers until it finds the correct range. It can then identify the sector within the run with a simple calculation. Run-length encoding also makes it trivial to extend the file logically if the newly assigned sector is contiguous with the file's previous last sector; the file system merely needs to increment the size doubleword of the file's last run pointer and clear the sector's bit in the appropriate freespace bitmap.

3.3.3 Directories

Directories, like files, are anchored on Fnodes. A pointe to the Fnode for the root directory is found in the SuperBlock. The Fnodes for directories other than the root are reache through subdirectory entries in their parent directories.

Directories can grow to any size and are built up from 2Kb directory blocks, which are allocated as four consecutive sectors on the disk. The file system attempts to allocate directory blocks in the directory band, which is located at or near the seek center of the disk. Once the directory band is full, the directory blocks are allocated wherever space is available.

Each 2Kb directory block contains from one to many directory entries. A directory entry contains several fields, including time and date stamps, an Fnode pointer, a usage count for use by disk maintenance programs, the length of the file or directory name, the name itself, and a B-Tree pointer. Each entry begins with a word that contains the length of the entry. This provides for a variable amount of flex space at the end of each entry, which can be used by special versions of the file system and allows the directory block to be traversed very quickly.

The nuniber of entries in a directory block varies with the length of names. If the average filename length is 13 characters, an average directory block will hold about 40 entries. The entries in a directory block are sorted by the binary lexical order of their name fields (this happens to put them in alphabetical order for the U.S. alphabet). The last entry in a directory block is a dummy record that marks the end of the block.

When a directory gets too large to be stored in one block, it increases in size by the addition of 2Kb blocks that are organized as a B-Tree. When searching for a specific name, the file system traverses a directory block until it either finds a match or finds a name that is lexically greater than the target. In the latter case, the file system extracts the BTree pointer from the entry. If there is no pointer, the search failed; otherwise the file system follows the pointer to the next directory block in the tree and continues the search.

A little back-of-the-envelope arithmetic yields some impressive statistics. Assuming 40 entries per block, a two-level tree of directory blocks can hold 1640 directory entries and a three-level tree can hold an astonishing 65,640 entries. In other words, a particular file can be found (or shown not to exist) in a typical directory of 65,640 files with a maximum of three disk hits-the actual number of disk accesses depending on cache contents and the location of the file's name in the directory block B-Tree. That's quite a contrast to the FAT file system, where in the worst case more than 4000 sectors would have to be read to establish that a file was or was not present in a directory containing the same number of files.

The B-Tree directory structure has interesting implications beyond its effect on open and find operations. A file creation, renaming, or deletion may result in a cascade of complex operations, as directory blocks are added or freed or names are moved from one block to the other to keep the tree balanced. In fact, a rename operation could theoretically fail for lack of disk space even though the file itself is not growing. In order to avoid this sort of disaster, the HPFS maintains a small pool of free blocks that can be drawn from in a directory emergency; a pointer to this pool of free blocks is stored in the SpareBlock.

3.3.4 Extended Attributes

File attributes are information about a file that is maintained by the operating system outside the file's overt storage area. The FAT file system supports only a few simple attributes (read only, system, hidden, and archive) that are actually stored as bit flags in the file's directory entry; these attributes are inspected or modified by special function calls and are not accessible through the normal file open, read, and write calls.

The HPFS supports the same attributes as the FAT file system for historical reasons, but it also supports a new form of fileassociated, highly generalized information called *Extended Attributes* (EAs). Each EA is conceptually similar to an environment variable, taking the form $(name, value)$

except that the value portion can be either a null-terminated (ASCIIZ) string or binary data. In OS/2 1.2, each file or directory can have a maximum of 64Kb of EAs attached to it. This limit may be lifted in a later release of OS/2.

The storage method for EAs can vary. If the EAs associated with a given file or directory are small enough, they will be stored right in the Fnode. If the total size of the EAs is too large, they are stored outside the Fnode in sector runs, and a B+ Tree of allocation sectors can be created to describe the runs. If a single EA gets too large, it can be pushed outside the Fnode into a B+ Tree of its own.

The kernel API functions DosQFileInfo and DosSetFileInfo have been expanded with new information levels that allow application programs to manipulate extended attributes for files. The new functions DosQPathInfo and DosSetPathInfo are used to read or write the EAs associated with arbitrary pathnames. An application program can either ask for the value of a specific EA (supplying a name to be matched) or can obtain all of the EAs for the file or directory at once.

Although application programs can begin to take advantage of EAs as soon as the HPFS is released, support for EAs is an essential component in Microsoft's long-range plans for object-oriented file systems, Information of almost any type can be stored in EAs, ranging from the name of the application that owns the file to names of dependent files to icons to executable code. As the HPFS evolves, its facilities for manipulating EAs are likely to become much more sophisticated. It's easy to imagine, for example, that in future versions the API might be extended with EA functions that are analogous to DosFindFirst and DosFindNext and EA data might get organized into B-Trees.

I should note here that in addition to EAs, the LAN Manager version of HPFS will support another class of file-associated information called Access Control Lists (ACLs). ACLs have the same general appearance as EAs and are manipulated in a similar manner, but they are used to store access rights, passwords, and other information of interest in a networking multiuser environment.

3.3.5 Performance Issues

The HPFS attacks potential bottlenecks in disk throughput at multiple levels. It uses advanced data structures, contiguous sector allocation, intelligent caching, read-ahead, and deferred writes in order to boost performance.

First, the HPFS matches its data structures to the task at hand: sophisticated data structures (B-Trees and B+ Trees) for fast random access to filenames, directory names, and lists of sectors allocated to files or directories, and simple compact data structures (bitmaps) for locating chunks of free space of the appropriate size. The routines that manipulate these data structures are written in assembly language and have been painstakingly tuned, with special focus on the routines that search the freespace bitmaps for patterns of set bits (unused sectors).

Next, the HPFS's main goal — its prime directive, if you will — is to assign consecutive sectors to files whenever possible. The time required to move the disk's read/write head from one track to another far outweighs the other possible delays, so the HPFS works hard to avoid or minimize such head movements by allocating file space contiguously and by keeping control structures such as Fnodes and freespace bitmaps near the things they control. Highly contiguous files also help the file system make fewer requests of the disk driver for more sectors at a time, allow the disk driver to exploit the multisector transfer capabilities of the disk controller, and reduce the number of disk completion interrupts that must be serviced.

Of course, trying to keep files from becoming fragmented in a multitasking system in which many files are being updated concurrently is no easy chore. One strategy the HPFS uses is to scatter newly created files across the disk-in separate bands, if possible-so that the sectors allocated to the files as they are extended will not be interleaved. Another strategy is to preallocate approximately 4Kb of contiguous space to the file each time it must be extended and give back any excess when the file is closed.

If an application knows the ultimate size of a new file in advance, it can assist the file system by specifying an initial file allocation when it creates the file. The system will then search all the free space bitmaps to find a run of consecutive sec tors large enough to hold the file. That failing, it will search for two runs that are half the size of the file, and so on.

The HPFS relies on several different kinds of caching to minimize the number of physical disk transfers it must request. Naturally, it caches sectors, as did the FAT file system. But unlike the FAT file system, the HPFS can manage very large caches efficiently and adjusts sector caching on a perhandle basis to the manner in which a file is used. The HPFS also caches pathnames and directories, transforming disk directory entries into an even more compact and efficient inmemory representation.

Another technique that the HPFS uses to improve performance is to preread data it believes the program is likely to need. For example, when a file is opened, the file system will preread and cache the Fnode and the first few sectors of the file's contents. If the file is an executable program or the history information in the file's Fnode shows that an open operation has typically been followed by an immediate sequential read of the entire file, the file system will preread and cache much more of the file's contents. When a program issues relatively small read requests, the file system always fetches data from the file in 2Kb chunks and caches the excess, allowing most read operations to be satisfied from the cache.

Finally, the OS/2 operating system's support for multitasking makes it possible for the HPFS to rely heavity on lazy writes (sometimes called deferred writes or write behind) to improve performance. When a program requests a disk write, the data is placed in the cache and the cache buffer is flagged as dirty (that is, inconsistent with the state of the data on disk). When the disk becomes idle or the cache becomes saturated with dirty buffers, the file system uses a captive thread from a daemon process to write the buffers to disk, starting with the oldest data.

In general, lazy writes mean that programs run faster because their read requests will almost never be stalled waiting for a write request to complete. For programs that repeatedly read, modify, and write a small working set of records, it also means that many unnecessary or redundant physical disk writes may be avoided. Lazy writes have their dangers, of course, so a program can defeat them on a per-handle basis by setting the writethrough flag in the OpenMode parameter for DosOpen, or it can commit data to disk on a perhandle basis with the DosBufReset function.

3.3.6 Fault Tolerance

The HPFS's extensive use of lazy writes makes it imperative for the HPFS to be able to recover gracefully from write errors under any but the most dire circumstances. After all, by the time a write is known to have failed, the application has long since gone on its way under the illusion that it has safely shipped the data into disk storage. The errors may be detected by the hardware (such as a "sector not found" error returned by the disk adapter), or they may be detected by the disk driver in spite of the hardware during a read-after-write verification of the data.

The primary mechanism for handling write errors is called a hotfix. When an error is detected, the file system takes a free block out of a reserved hotfix pool, writes the data to that block, and updates the hotfix map. (The hotfix map is simply a series of pairs of doublewords, with each pair containing the number of a bad sector associated with the number of its hotfix replacement. A pointer to the hotfix map is maintained in the SpareBlock.) A copy of the hotfix map is t written to disk, and a waming message is displayed to let the user know that all is not well with the disk device.

Each time the file system requests a sector read or write from the disk driver, it scans the hotfix map and replaces any bad sector numbers with the corresponding good sector holding the actual data. This lookaside translation of sector numbers is not as expensive as it sounds, since the hotfix list need only be scanned when a sector is physically read or written, not each time it is accessed in the cache.

One of CHKDSK's duties is to empty the hotfix map. For each replacement block on the hotfix map, it allocates a new sector that is in a favorable location for the file that owns the data, moves the data from the hotfix block to the newly allocated sector, and updates the file's allocation information (which may involve rebalancing allocation trees and other elaborate operations). It then adds the bad sector to the bad block list, releases the replacement sector back to the hotfix pool, deletes the hotfix entry from the hotfix map, and writes the updated hotfix map to disk.

Of course, write errors that can be detected and fixed on the fly are not the only calamity that can befall a file system. The HPFS designers also had to consider the inevitable damage to be wreaked by power failures, program crashes, malicious viruses and Trojan horses, and those users who tum off the machine without selecting Shutdown in the Presentation Manager Shell. (Shutdown notifies the file system to flush the disk cache, update directories, and do whatever else is necessary to bring the disk to a consistent state.)

The HPFS defends itself against the user who is too abrupt with the Big Red Switch by maintaining a Dirty FS flag in the SpareBlock of each HPFS volume. The flag is only cleared when all files on the volume have been closed and all dirty buffers in the cache have been written out or, in the case of the boot volume (since OS2.INI and the swap file are never closed), when Shutdown has been selected and has completed its work.

During the OS/2 boot sequence, the file system inspects the DirtyFS flag on each HPFS volume and, if the flag is set, will not allow further access to that volume until CHKDSK has been run. If the DirtyFS flag is set on the boot volume, the system will refuse to boot; the user must boot OS/2 in maintenance mode from a diskette and run CHKDSK to check and possibly repair the boot volume.

In the event of a truly major catastrophe, such as loss of the SuperBlock or the root directory, the HPFS is designed to give data recovery the best possible chance of success. Every type of crucial file object-including Fnodes, allocation sectors, and directory blocks-is doubly linked to both its parent and its children and contains a unique 32-bit signature. Fnodes also contain the initial portion of the name of their file or directory. Consequently, CHKDSK can rebuild an entire volume by methodically scanning the disk for Fnodes, allocation sectors, and directory blocks, using them to reconstruct the files and directories and finally regenerating the freespace bitmaps.

3.3.7 Summary

The HPFS solves all of the historical problems of the FAT file system. It achieves excellent throughput even in extreme cases — many very small files or a few very large files — by means of advanced data structures and techniques such as intelligent caching, read-ahead, and write-behind. Disk space is used economically because it is managed on a sector basis. This article is based on a prerelease version of the HPFS that was still undergoing modification and tuning. therefore, the final release of the HPFS may differ in some details from the description given here.