4.3BSD Virtual Memory Management

Felix Stahlberg

14.07.2010

Contents

1 Introduction

This paper introduces you to the virtual memory management system used in 4.3BSD. However, since the first 4.3BSD was released 1986 the question begs to be asked to what extend this is relevant for us today. On the one hand there is a historical concern about that. 4.3BSD is a milestone of the evolution of operating systems over the years. This can be affirmed by a quote from an article in the Information Week in the year 2006.

> The single Greatest Piece of Software Ever, with the broadest impact on the world, was BSD 4.3. Other Unixes were bigger commercial successes. But as the cumulative accomplishment of the BSD systems, 4.3 represented an unmatched peak of innovation. BSD 4.3 represents the single biggest theoretical undergirder of the Internet. [Bab06]

Additionally not only the 4.3BSD system but the underlying hardware is historically significant as well. The VAX was one of the first adopter of virtual memory techniques. Nowadays almost every architecture (except for special purpose embedded systems) provides virtual memory capabilities.

On the other hand some aspects of the virtual memory management in 4.3BSD are still present in current operating systems. For instance the basic idea of the page replacement algorithm of 4.3BSD is used in all Solaris releases published so far. Furthermore the concept of a *pagedaemon*, a process which is permanently resident in the system and manages paging related tasks, was already implemented in 4BSD and is still very widely-used. Even certain tuning parameters of the paging process can be found today. The influence of the 4.3BSD virtual memory management to current systems is explored in detail in the last chapter.

This paper assumes that the reader is at least slightly familiar with the C programming language and has already encountered terminology such as *segmentation*, *paging*, *virtual and physical address space* and *process*. Knowledge about the history of UNIX could also be helpful.

This work is a part of my coursework on the seminar *OS Internals* at the Karlsruhe Institute of Technology in summer term 2010. The other part is the talk I gave on 21th June 2010 in Karlsruhe. The slides and this paper are available in PDF format on my homepage.

`http://www.xilef-software.de/en/`

Please do not hesitate to send me feedback and corrections to following address.

`mailto:fstahlberg@gmail.com`

Felix Stahlberg, 14th July 2010

2 History and Goals

2.1 The 4.3BSD Operating System

4.3BSD was released in 1986. Back then there were three major UNIX development groups. Among commercial organizations like AT&T and Bell Laboratories, the University of California at Berkeley contributed UNIX software in so called *Berkeley Software Distributions* (BSD). The first notable BSD release in our context was 3BSD in the year 1979 which ports 32V UNIX to the VAX hardware and provided, for instance, demand paging and page replacement by using its virtual memory management hardware [SJLQ89]. Further developments made 4BSD the operating system of choice for VAX computers, at least for many research and networked installations. 4.3BSD came with major performance improvements all over so that its influences can be found in many current general purpose systems.

2.2 Evolution of 4.3BSD Memory Management

Since the motivation for exploring 4.3BSD virtual memory management could partly be based on its historical significance, it should not be forgotten to determine how its predecessors' approach to memory management looks like [SJLQ89].

Bell Labs released its last version of UNIX (Version 7 UNIX) 1979 for PDP-11. Although Version 7 was an important UNIX release in general, the memory management was kept simple using segmented memory architecture and a virtual address space consisting of eight segments that span 8 KiB each. Therefore the virtual address space was only as large as KiB. Programs had to be fully resident in contiguous memory to be executed, so this upper bound was quite low even in those days. UNIX 32V made this limit tunable but did not touch the memory architecture in-depth. Only subsequent revisions to the memory management provided enhancements like *scatter-loading* which broke the upper bound for the program size because code was then loaded on a page-by-page basis.

With 3BSD virtual memory management techniques like paging finally found their way into UNIX so that very large processes could be executed. The 3BSD paging system used "a demand-paging fetch policy [...], a page-by-page placement policy [and] a replacement algorithm that approximated global least recently used (LRU)" [SJLQ89]. Thus 3BSD laid the foundation of the virtual memory management system in 4.3BSD. The virtual memory management system was reviewed again in 4.1BSD, mainly focused on performance-tuning. These changes encompass, among other things, using *page clusters* and *prepaging* techniques. *Page clusters* are multiples of the underlying hardware page size. The system takes advantage of using this clusters instead of hardware pages by abstracting from the actual hardware page size and reducing the size of certain kernel data structures. *Prepaging* means that the system does not always fetch a single page from the disk but as many pages as reasonable and possible (for example a whole cluster). Since prepaging only makes sense if assumed that the prefetched pages are probably referenced shortly after, processes in 4.1BSD can disable prepaging when they expect themselves to name a random memory reference pattern instead of a sequential one.

2.3 Memory Management Design Decisions

Although it "will never be a best seller" [Ope02], the *Berkeley Software Architecture Manual (4.3 Edition)* [WNJM86] gives a detailed description of the user-level process' point of view to the virtual memory management system and design decisions concerning it.

The address space of user-level processes is divided into three logical areas called *text*, *data* and *stack*. The *text* area contains the machine instructions of a program. It is marked as read-

only and thus can be shared among different processes. The other two areas are both private to the process and readable as well as writeable. Furthermore, the size of data and stack areas may be changed by the system calls *sbrk()* and *sstk()* (*sstk()* is not implemented). The stack area, which holds the stack of the process, is automatically extended as needed.

4.3BSD defines a rich and highly-developed (at least in the time of 4.3BSD's release) interface for user processes to the virtual memory management system. The most interesting component of this interface is the *mmap()* system call which enables the processes to map certain pages either to files or to the virtual address space of other processes. Thus, implementing the *mmap()* interface enables the system to support advanced features like *memory mapped files* or *shared memory* for inter process communication. Consequently, there are synchronization primitives (such as *msleep()* and *mwakeup()*) for memory sharing processes and protection mechanisms (*mprotect()*) for single pages. Unfortunately, basically time pressure prevented developers to provide a complete implementation of the interface [1]. Even the last Berkeley Software Distribution release (4.4BSD) implemented the interface completely.

Inside the kernel, memory allocations often have certain characteristics – for instance, fixed-size, short-term allocations – which allows system developers to write well-performing and specialized memory allocators for certain allocation purposes. Thus 4.3BSD uses at least 10 different memory allocators for memory management inside the kernel. In order to reduce complexity of writing kernel code and the development time of new kernel services *4.3BSD Tahoe* introduced a general-purpose memory allocator [MKM88]. This allocator applies a power-of-two list strategy for small allocations (up to 2 KB). Larger allocations are done using paging and a simple first-fit strategy. The interface for this memory allocator is defined similar to C's *malloc()* and *free()*, that is, the allocation method only takes the size of the requested allocation and returns a pointer which is sufficient for the free routine.

[1] except for the system calls *getpagesize()*, which gets the size of the underlying hardware pages, and *sbrk()*

3 VAX Memory Management Hardware

VAX is an acronym for *Virtual Address eXtension* and names a 32-bit CISC architecture which was an early adopter of virtual memory [SJLQ89]. Since 3BSDs major design goals included using virtual memory capabilities of VAXes and providing virtual memory techniques, VAX is the reference architecture for all BSD releases at least up to 4.4BSD. Especially the code of the virtual memory management system is highly dependent on VAX-specific characteristics, so portings of 4BSD releases to other architectures than VAX generally simulated the VAX architecture on the target systems. Hence the underlying hardware should be known before the implementation in 4.3BSD can be explored.

3.1 VAX Virtual Address Space

Figure 1 shows the 32-bit virtual address space of the VAX. Since it is divided into four equal-sized *regions*, the first two bits of a virtual address are sufficient to define the current region. The first two regions, called *P0* and *P1*, contain the address space of the process. The *system* region is shared among all processes and contains the kernel virtual memory. The address translation of a virtual address in the system region into a physical address is performed differently than one in the P0 and P1 regions (see Section 3.3). The last region *reserved* was initially not used in 4.3BSD until 4.3BSD Tahoe has moved the system region to the top and used all four regions.

Note that the virtual address space of user processes is directly addressable by the kernel. Therefore 4.3BSD is able to provide very efficient implementations of *copyin()* and *copyout()* routines that move data between kernel and user space.

3.2 Format of a VAX Page Table Entry

A VAX *page table entry* (*PTE*) has a length of four bytes with common information about the page, as shown in Figure 2. We make two observations about it that are relevant to the 4.BSD memory management.

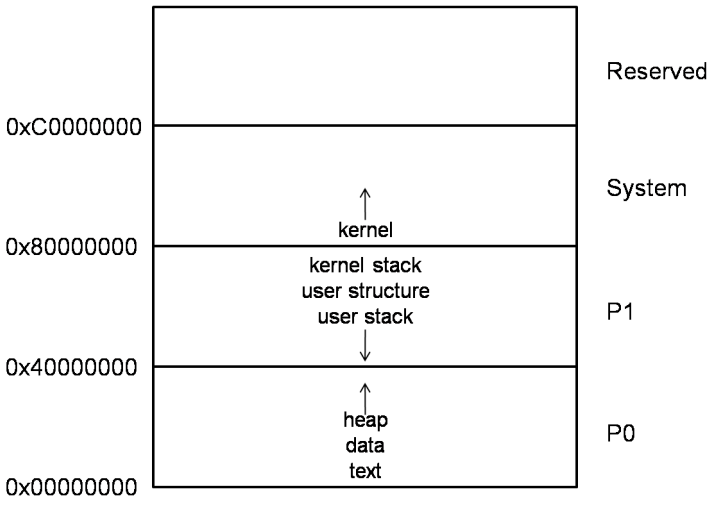

Figure 1: VAX Virtual Address Space [SJLQ89]

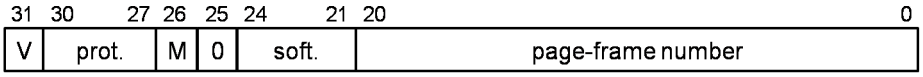

Figure 2: Scheme of a page table entry. V: valid bit, prot: protection mode, M: modified bit, soft: software defined field. [SJLQ89]

First, the *page-frame number* consists of 21 bits. That implies that we have 2^{21} page-frame numbers. Since the common page size on VAX is fixed on $512 = 2^9$ bytes, the limit of physical memory which could be supported is $2^{21} * 2^9 = 2^{30} bytes = 1GiB$.

Second, there is no *reference bit* which would be set automatically by hardware when the page is referenced. This caused major performance issues for the page replacement algorithm which is discussed in Section 5.2.

3.3 Address Translation

3.3.1 System Address Translation

System virtual addresses always start with the bits 10 to identify the third region (*system*) of the virtual address space. As shown in Figure 3, translation of kernel addresses is very straightforward. The page index, which can be extracted from bits 9 through 29, is added to the *system base register* (*SBR*) to find a page table entry. The memory management hardware checks the valid bit and the protection mode field and (if the checks are successful) combines the page-frame number found in the PTE with the byte offset which can be directly taken from bits 0 through 8 of the virtual address to finally generate a physical memory address. Therefore system virtual addresses are translated by a simple one-level translation mechanism.

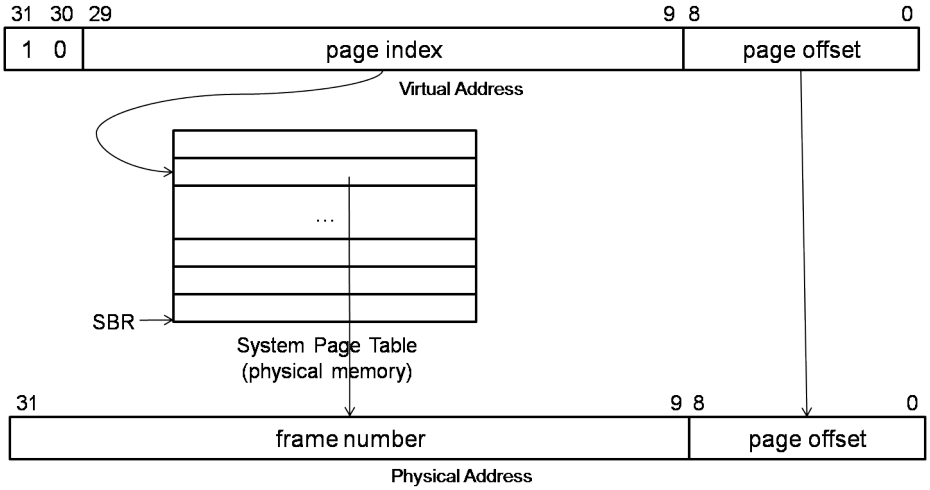

Figure 3: System virtual address translation [SJLQ89]

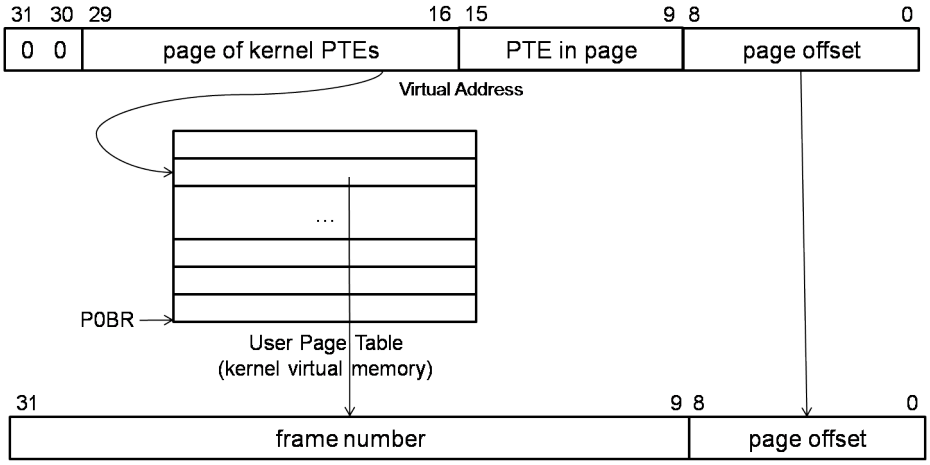

Figure 4: User virtual address translation

3.3.2 User Address Translation

Although translation of user addresses follows the same idea, there is one important difference. The page tables for the *P0* and the *P1* regions are not in physical memory but in kernel virtual address space. Instead of the *SBR*, another register (*P0BR* or *P1BR*) is used to locate the appropriate page table. This is necessary because user page tables are too large to stay resident in contiguous physical memory. The length of the *page index* allows 2^{21} pages to be referenced. Since a PTE has a size of four bytes, the total memory consumption of a page table of each of the P0 and P1 regions would be $2^{21} * 4bytes = 8MiB$. In total, each process would require 16 MiB main memory plus a little portion of the `userptmap` for page tables alone.

Figure 4 illustrates the translation of user addresses. The *page index* is now split up into two parts. Since the page table is in kernel virtual memory, the first 14 bits of the page index select a kernel page which is full of PTEs. The next 7 bits select a certain PTE in this page. From here on, the workflow is similar to the system address translation mechanism.

This solution (paging page tables) is not adaptable for kernel virtual address space since the system page table cannot be paged by some other table. Therefore the developers minimized the size of sections which had to stay resident in memory (for instance the `userptmap` has a size of only 32 pages) and implemented the other sections in a way specific to their purpose (see Section 4.1).

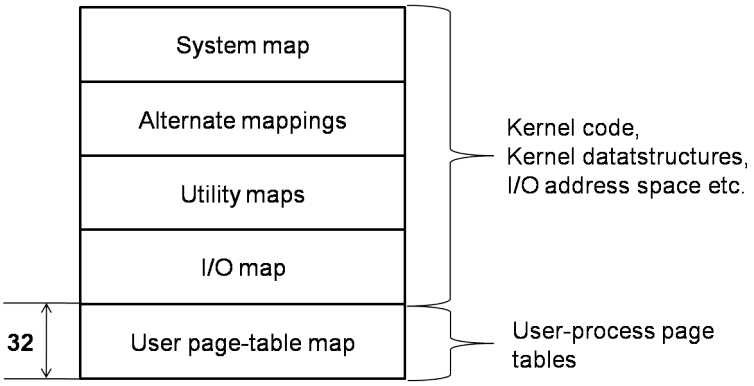

Figure 5: Layout of the system page table

4 Page Tables and PTEs

4.1 System Page Table

Figure 5 shows the structure of the system page table, which basically maps the kernel virtual address space [SJLQ89]. An entry in the system page table may be static (filled in at boot time) or dynamic (filled in on use). Entries in the first section (*system map*) are static and hold data structures allocated at boot time as well as the kernel code. The second (*alternate mappings*) and the fourth section (*I/O map*) are not relevant to our case. The *utility maps* are used to handle dynamic or temporary allocation needs of the kernel. The last section, the user page table map (also called `userptmap`), has a fixed size of commonly 32 pages and is used to map the user page tables.

The system page table is implemented as a concatenation of small page tables for each section. Thus not the whole system page table always has to be resident in main memory just because its certain parts (for instance, the system map and the userptmap) have to be.

With all the facts mentioned until now it is possible to think about the overall capabilities of the paging system of 4.3BSD in standard configuration. The *userptmap* with a size of $32 = 2^5$ pages at $512 = 2^9$ bytes filled up with $4 = 2^2$ byte PTEs can map $2^5 * 2^9 * 2^{-2} = 2^{12} = 4096$ pages of page tables. This number of pages is enough for $2^{12} * 2^9 * 2^{-2} = 2^{19} = 524,288$ PTEs. Therefore the total virtual memory size of all resident processes has to be less than $2^{19} * 2^9 = 2^{28}$ bytes (256 MiB) or less.

4.2 PTE Types

When processes reference memory, the system has to access a certain page table entry to find out what to do. How 4.3BSD analyses the content of this entry is discussed next.

The scheme of a VAX page table entry in Figure 2 defines bit 25 to be set to 0. The purpose of bit 25 is to distinguish between the two major PTE types in 4.3BSD – *Normal Page Table Entries* and *Fill-on-Demand Page Table Entries*.

The 21 least significant bits of a normal Page Table Entry (Figure 6) are occupied by the page frame number according to the VAX PTE scheme from Figure 2. The software defined field is not relevant in this context. *pg_fod* is set to 0 to make sure the PTE is interpreted in this manner. The six most significant bits are med by VAX memory management hardware requirements.

```
struct pte {
  unsigned int pg_pfnum:21, /* core page frame number or 0 */
               :2,
               pg_vreadm:1, /* modified since vread (or with _m) */
               pg_swapm:1,  /* have to write back to swap */
               pg_fod:1,    /* is fill on demand (=0) */
               pg_m:1,      /* hardware maintained modified bit */
               pg_prot:4,   /* access control */
               pg_v:1;      /* valid bit */
};
```

Figure 6: Normal Page Table Entry (`machine/pte.h`)

```
struct fpte {
  unsigned int pg_blkno:24, /* file system block number */
               pg_fileno:1, /* file mapped from or TEXT or ZERO */
               pg_fod:1,    /* is fill on demand (=1) */
               :1, pg_prot:4, pg_v:1;
};
```

Figure 7: Fill-On-Demand Page Table Entry (`machine/pte.h`)

When the system finds a normal page table entry, at first it checks the *valid bit*. If it is set, there will be a nonzero page number which will refer to a valid and resident page in physical memory. If it is cleared and the page number is nonzero, either a swapping operation (such as page-in or page-out) is in progress or a reference bit is being simulated. More about reference bit simulation for the page replacement algorithm in Section 5.2.

When the content of a page is not loaded so far, the associated page table entry is called *fill-on-demand page table entry*. Such pages are filled when they are referenced the first time, either with zeros or from file. The structure definition in Figure 7 shows two noticeable aspects. First, the modified bit is not used since it would have no purpose. Second, the bits originally reserved for the page frame number are combined with the software defined field. They are used to store `pg_blkno`, which is a 24 bit long file system block number, and `pg_fileno`. The valid bit has to be cleared for fill-on-demand PTEs to make sure that as soon as the associated page has been referenced a *fill-on-demand page fault* is generated and the page fault routine is executed. This routine then determines whether `pg_fileno` has the value `PG_ZERO`, and the page should be filled up with zeroes, or `PG_TEXT`, and the desired content of the page is in secondary storage. In order to load only useful parts of machine code instead of loading the whole source code of a process, text segments are typically stored in fill-on-demand pages with `pg_fileno` set to `PG_TEXT` in their page table entry.

Fill-on-demand PTEs store precomputed block numbers. This precomputation is done because computation of block numbers requires many secondary storage accesses and thus is quite slow. Precomputation speeds up this process since necessary data structures probably are available in the cache after their first access and many PTEs are handled at a single blow in a loop.

All other combinations of `pg_fod`, `pg_v` and the state of the page frame number (zero or nonzero) – like `pg_fod` and `pg_v` are both set or both cleared and there is a zero page frame number – are either unused or irrelevant in this context.

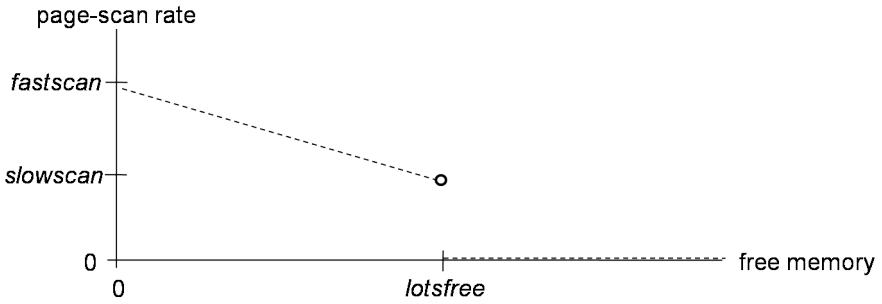

Figure 8: Page-scan rate [SJLQ89]

5 Page Replacement

The design decisions and implementations of page replacement capabilities are the parts of 4.3BSDs virtual memory management systems with the most influence on today's systems. The paging parameters and the two-handed clock algorithm in particular can be found in modern operating systems nowadays (further details in Section 6).

5.1 Paging Parameters

The most important parameters related to page replacement are shown in Table 1. Figure 8 illustrates their effect on the page-scan rate. Page replacement is done by the *pagedaemon*. This pagedaemon is a permanently present process in the system, which is woken up four times per second and scans a certain number of page-clusters (depending on the current page-scan rate) in order to free them. The page-scan rate depends on several parameters and how much memory is currently needed. If available memory is greater than *lotsfree*, the page-scan rate is set to 0 and the pagedaemon stops to scan pages. Otherwise it ranges between *slowscan* and *fastscan* straight proportional to the free memory. *Desfree* is the desired amount of free memory. We will examine this parameter in detail in Section 5.4. *Minfree* defines a threshold for involuntary swapping of complete processes. Since it causes a lot of overhead and slow secondary storage accesses, swapping is avoided in 4.3BSD if possible. Nevertheless it could be reasonable to swap out complete processes to prevent the CPU from spending too much time paging.

5.2 Reference Bit Simulation on VAX

Two page-replacement policies are discussed in this context. Both the *CLOCK algorithm* and the *two-handed CLOCK algorithm* need a reference bit in PTEs which is set by hardware whenever

Name	Common Value	Upper Bound	Description
fastscan	200 pages / sec	$\frac{1}{5}$ of memory	Fastest page-scan rate
slowscan	100 pages / sec	*fastscan*	Page-scan rate on *lotsfree*
lotsfree	512 KByte	$\frac{1}{4}$ of memory	Threshold for stopping scanning
desfree	200 KByte	$\frac{1}{8}$ of memory	Desired amount of free memory
minfree	64 KByte	$\frac{1}{16}$ of memory	Involuntary swapping begins

Table 1: Paging Parameters

a page is referenced. Since the VAX architecture does not provide such a bit it has to be simulated. This simulation is realized using the valid bit (until 4.3BSD Tahoe which actually uses the reference bit when it is supported by the hardware). To this end, the valid bit is cleared when the reference bit should be cleared. The first time the page is referenced afterwards, a page fault occurs. The page-fault routine is optimized to handle this kind of page faults. Thus, whenever a PTE has both `pg_v` and `pg_fod` cleared (see Section 4.2 for further information about PTEs) and contains a nonzero page number, the page-fault handler assumes a reference bit is simulated and simply restores the valid bit, marking this page as both valid and *referenced*.

The initial cost of this simulation was about three percent of CPU time (assuming a system with serious memory shortfall), which was not acceptable. Rewriting parts of the page-fault handler in assembly language and further optimizations reduced this cost to about one percent. Nonetheless, simulating the reference bit on VAX breakes the overall performance of the system not insignificantly.

In the following sections the term *reference bit* is used for both simmulated and real reference bits.

5.3 Global CLOCK Algorithm

Until the release of 4.3BSD, 4BSD used a variant of the *CLOCK algorithm*. This *global* algorithm keeps all the available page clusters in a circular array and traverses it with a pointer called *clock hand*. When pages have to be freed because of memory shortfalls, the clock hand rotates with a certain speed (depending on the page-scan rate) and checks the PTEs of the page cluster it points to. If the reference bit is set, the algorithm clears it and proceed with the next page cluster. If the reference bit is already cleared, the page was not referenced for a long time and can be freed.

The CLOCK algorithm forces all processes to compete for memory on an equal basis since they are all in the same system-wide loop and no process is preferred. This also implies that 4.3BSD uses no *working set model*. This model was well known when 4.3BSD had been released, but it was a conscious design descisions not to implement it because of the large administrative overhead of tracking many parameters for each process.

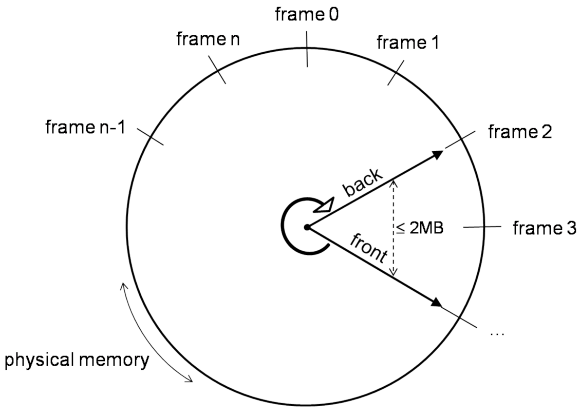

Figure 9: Two-handed clock [SJLQ89]

5.4 Two-Handed Clock

The global CLOCK algorithm has major performance issues on systems with large physical memory and sudden memory shortfalls after a period above *lotsfree* (without scanning). For instance the clock hand takes at least 80 seconds for one revolution on a machine with 16 MiB of memory (assuming a page cluster size of 1KiB and a page-scan rate of 200 pages per second). Thus 4.3BSD uses a generalization of the CLOCK algorithm called *two-handed clock* which is depicted in Figure 9. 4.3BSDs implementation is shown in Figure 10. This algorithm traverses the page clusters with the *backhand* and the *fronthand* pointers The fronthand clears the reference bits and the backhand checks if they are still cleared and pages can be freed.

The main loop of the process with id 2 – the *pagedaemon* – is in the `pageout()` function, which is called in 4.3BSD's `main()` method. After that, `fronthand` and `backhand` are set to proper initial values and the process enters a infinite loop and stays there the rest of systems life time. After `PSWP`+1 milliseconds (or four times per second), the while-loop is executed until there is more than `lotsfree` free memory or the number of pages scanned so far (`nscan`) exceeds `desscan`. `Desscan` is set by `schedpaging()` (`sys/vm_sched.c`) and realizes the linear interpolation between `slowscan` and `fastscan` (Figure 8). `checkpage()` (`sys/vm_page.c`) implements the tasks of the two clock hands (clearing or checking reference bits and freeing page clusters) and returns 1 if a page cluster could be freed. In certain cases even the front hand can free pages by stealing them from other processes. Lines 14-20 manage implementation details of the circular array except for one line 17. This line deals with the unlikely case that the fronthand rotates twice and no pages could be freed. In that case the while loop is leaved and the pagedaemon waits $\frac{1}{4}$ seconds.

```
1  pageout() {
2    register int count, maxhand = pgtocm(maxfree);
3    register int fronthand = HANDSPREAD/CLBYTES, backhand = 0/CLBYTES;
4    if (fronthand >= maxhand) fronthand = maxhand - 1;
5  loop:  /* (...) */
6    sleep((caddr_t)&proc[2], PSWP+1);
7    (void) spl0();
8    count = 0; pushes = 0;
9    while (nscan < desscan && freemem < lotsfree) {
10     if (checkpage(fronthand, FRONT)) count = 0;
11     if (checkpage(backhand, BACK)) count = 0;
12     cnt.v_scan++;
13     nscan++;
14     if (++fronthand >= maxhand) {
15       fronthand = 0;
16       cnt.v_rev++;
17       if (count > 2) goto loop;
18       count++;
19     }
20     if (++backhand >= maxhand) backhand = 0;
21   }
22   goto loop;
23 }
```

Figure 10: Implementation of Two-handed clock (shortened) (`sys/vm_page.c`)

6 Conclusion

The quote in the introduction from [Bab06] is definitely reasonable, although other parts than its virtual memory management system were more important for finally making 4.3BSD to the "Greatest Piece of Software Ever". However, the memory management system has great influences on today's systems. Solaris 10 still knows the parameters `lotsfree`, `desfree`, `minfree`, `slowscan` and `fastscan` [Sun09] and uses the two-handed clock replacement policy. Even certain functions like `checkpage()` or `schedpaging()` still exist in its source code. Many other systems have equivalents of the paging parameters (such as `pages_low` on linux or `freemin` on FreeBSD for `lotsfree`) [Gor04]. On almost all Unix-like systems there is a pagedaemon with process ID 2. Summing up, the 4.3BSD memory management system was one of the first efficient and sophisticated implementations of virtual memory. Particularly the two-handed clock algorithm was a crucial enhancement and a milestone in the history of virtual memory techniques.

References

[Bab06] Charles Babcock. What's the greatest software ever written? *InformationWeek*, 2006.

[Gor04] Mel Gorman. *Understanding The Linux Virtual Memory Manager*. 2004.

[MKM88] Michael J. Karels Marshall Kirk McKusick. Usenix conference. In *Design of a General Purpose Memory Allocator for the 4.3BSD UNIX Kernel*, San Francisco, 1988.

[Ope02] Unix programmer's supplementary documents, volume 1. http://www.openbsd.org/4.3-ps1.html, April 2002.

[SJLQ89] Michael J. Karels Samuel J. Leffler, Marshall Kirk McKusick and John S. Quarterman. *The Design and Implementation fo the 4.3BSD UNIX Operating System*. Addison-Wesley Publishing Company, 1989.

[Sun09] Sun Microsystems, Inc. *Solaris Tunable Parameters Reference Manual*, 2009.

[WNJM86] S. J. Leffler W. N. Joy, R. S. Fabry and M. K. McKusick. *Berkeley Software Architecture Manual*. University of California, Berkeley, 4.3bsd edition edition, 1986.