Data Structures Added in the Berkeley Virtual Memory Extensions to the UNIX† Operating System‡

Özalp Babaoğlu

William Joy

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

ABSTRACT

This document describes the major new data structures that have been introduced to the Version 7 UNIX system to support demand paging on the VAX*-11/780. The reader should be basically familiar with the VAX architecture, as described in the *VAX-11/780 Hardware Handbook.*

When relevant, along with the data structures, we present related system constants and macro definitions, and some indications of how the data is used by the system algorithms. We also describe the extensions/deletions that have been made to some of the existing data structures. Full description of the paging system algorithms, however, is not given here.

† UNIX is a Trademark of Bell Laboratories

‡ Work supported by the National Science Foundation under grants MCS 7807291, MCS 7824618, MCS 7407644-A03 and by an IBM Graduate Fellowship to the second author.

* VAX is a trademark of Digital Equipment Corporation.

Data Structures Added in the Berkeley Virtual Memory Extensions to the UNIX† Operating System‡

Özalp Babaoğlu

William Joy

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

Introduction

The paging subsystem of the virtual memory extension to the system maintains four new, basic data structures: the *(system and process) page tables,* the *kernel map,* the *core map* and the *disk map.* This document consists of a description of each of these data structures in turn.

PAGE TABLES

The format of the process page tables are defined in the system header file **pte.h.†** The basic form of the Page Table Entry (PTE) is dictated by the VAX-11/780 architecture. Both the first level page table, known as *Sysmap,* and the second level per-process page tables consist of arrays of this structure. The paging system makes use of several bit fields which have no meaning to the hardware. The individual fields are:

pg_prot	The *Protection* Bits. Define the access mode of the corresponding page. Modes used by the paging system include PG_NOACC for invalid entries, PG_KR for the text of the kernel, PG_KW for the kernel data space, PG_URKR for text portions of user processes, PG_URKW for text portions of user processes during modification (old form *exec,* and *ptrace*) of text images, and PG_UW for normal user data pages.
pg_m	The *Modify* Bit. Set by hardware as a result of a write access to the page. Examined and altered by the paging subsystem to find out if a page is *dirty* and has to be written back to disk before the page frame can be released.
pg_swapm	Indicates whether the page has been initialized, but never written to the swapping area. This bit is necessary because **pg_m** is normally or'ed with **pg_vreadm** to see if the page has ever been modified, and thus **pg_m** is unavailable to force a swap in this case.
pg_vreadm	Indicates whether the page has been modified since it was last initialized. (Initialization occurs during stack growth, growth due to *break* system calls, and as a result of *exec* and *vread* system calls.) For use by the *vwrite* system call, which looks at the inclusive-or of this and the **pg_m** bit. A *vwrite* also clears this bit.

† UNIX is a Trademark of Bell Laboratories

‡ Work supported by the National Science Foundation under grants MCS 7807291, MCS 7824618, MCS 7407644-A03 and by an IBM Graduate Fellowship to the second author.

* VAX is a trademark of Digital Equipment Corporation.

† Copies of the system definitions (header) files related to the paging subsystem appear at the end of this document.

pg_fod	The *Fill on Demand* Bit. Set only when the valid bit (described below) is reset, indicating that the page has not yet been initialized. When referenced, the page will either be zero filled, or filled with a block from the file system based on other fields described next.
pg_fileno	Meaningful only when the *pg_fod* bit is set. If the *pg_fileno* field has value PG_FZERO, then a reference to such a page results in the allocation and clearing of a page frame rather than a disk transfer. When stack or data segment growth occurs, the page table entries are initialized to fill on demand pages with PG_FZERO in their *pg_fileno* fields. The page is otherwise filled in from the file system, either from the inode associated with the currently executing process (if *pg_fileno* is PG_FTEXT), or from a file.
pg_blkno	Gives the block number, in a file system determined by the value of the *pg_fileno* field, from which the page is to be initialized. Note that this is the logical block number in the file system, not in the mapped object. Thus no mapping is required at page fault time to locate the actual disk block; the system simply uses the *pg_fileno* field to locate an inode and uses the *i_dev* field of that inode in a call to a device strategy routine. The size of this field (20 bits) limits the maximum size of a filesystem to 2↑20 blocks (1M block).
pg_v	The *Valid* bit. Set only when the *pg_fod* bit is reset. Indicates that the mapping established in the PTE is valid and can be used by the hardware for address translation. Access to the PTE when this bit is reset causes an *Address Translation Not Valid* fault and triggers the paging mechanism. If both the valid and fill on demand bits are reset, but the *pg_pfnum* field is non-zero, then the page is still in memory and can be reclaimed either from the loop or the free list without an I/O operation by simply revalidating the page, after possibly removing it from the free list. If the *pg_fod* bit is not set, and the *pg_pfnum* field is zero, then the page has to be retrieved from disk. Note that resetting the valid bit for pages which are still resident allows for software detection and recording of references to pages, simulating a *reference* bit, which the VAX hardware does not provide.
pg_pfnum	The *Page Frame Number.* Meaningful only when the *pg_fod* bit is reset. If the page frame is valid, then this gives the physical page frame number that the virtual page is currently mapped to. If no physical page frame is currently allocated this field is zero (except in page table entries in *Sysmap,* where unused entries are not always cleared.)

System Page Tables

The first level page table *Sysmap* consists of a physically contiguous array of PTEs defined by the processor registers SBR (System Base Register), and SLR (System Length Register). SLR is loaded with the constant *Syssize* at system initialization and remains fixed thereafter.

The first four pages of the *Sysmap* map the kernel virtual memory from addresses 0x80000000 to the end of kernel data space onto the first pages of physical memory. Four pages is enough to map a kernel supporting a full load of memory on a VAX-11/780. Immediately after the pages which map the kernel text and data are the entries which map the user structure of the current running process (the *u.* area.) The *u.* area is currently six pages long, with the first two of these pages mapping the *user* structure itself, and the other four pages mapping the per-process kernel stack.† The position of the *u.* in *Sysmap* determines that it will live, in this system, at 0x80040000.

After the map entries reserved for the *u.* area are 16 words of system map reserved for utility usage. The *copyseg* routine uses one of these (CMAP2) while making a copy of one data page to map the destination page frame. This is necessary because at the point of copying (during the *fork* system call) the parent

† Currently all six pages are allocated physical memory; it is planned that in the future, the third of these six pages will be made a read-only copy of *zeropage.* Since the stack is observed rarely to enter the third page this will leave a full page for unanticipated worst-case stack growth, and give a clean termination condition should the stack ever accidentally grow beyond three pages.

process is running, while the destination page is not in the parents address space, but rather destined for the childs address space. Since the parent may fault on the source page during the copy, the contents of this map are saved in the software extension to the *pcb* during context switch. Other utilities are used by *clearseg* to map pages to be cleared in implementing zero fill on demand pages, and by the **mem.c** driver to map pages in **/dev/mem** when accessing physical memory.

The *Sysmap* continues with sets of entries for the UBA control and map registers, the physical device memory of a UNIBUS adaptor, and the control and map registers of upto three MASSBUSS adaptors. Each of these consists of 16 page table entries, mapping 8K bytes.

Next, there are a set of map entries for manipulating *u.* structures other than the one of the current running process. For instance, the page out demon, which runs as process 2, needs access to the diskmap information of a process whose page is being written to the disk. To get access to this, it uses six entries in the *Sysmap,* known as *Pushmap,* to map this *u.* into kernel virtual memory at a virtual address given by *pushutl.* There are several other map/utl pairs: *Swapmap* and *swaputl, Xswapmap* and *xswaputl, Xswap2map* and *xswap2utl, Forkmap* and *forkutl, Vfmap* and *vfutl.* These are used in swapping and forking new processes.

The final portion of the *Sysmap* consists of a map/utl like pair *Usrptmap/usrpt* which is a resource allocated to hold the first level page tables of all currently core-resident processes. This is a very important structure and will be described after we describe the basic structure of the page tables of a process.

Per-process page tables

Each process possesses three logical page tables: one to map each of the text, data and stack segments. Large portions of the system refer to page table entries in each of these segments by an index, with the first page of each segment being numbered 0.

For the VAX-11/780 version of the system, these page tables are implemented by two physically distinct page tables, the *P0 Page Table,* mapping the text and data segments, and the *P1 Page Table,* mapping the stack segment. Within the P0 region, the text segment is mapped starting at virtual address 0 with the data segment following on the first page boundary after it.* The stack segment, on the other hand, starts at the bottom of the P1 region and grows toward smaller virtual addresses. The constant USRSTACK corresponds to the address of the byte one beyond the user stack. The process page tables are contiguous in kernel virtual memory (KVM) and are situated with the P0 table followed by the P1 table such that the top of the first and the bottom of the second are aligned at page boundaries. Note that this results in a *gap* between the two page tables whose size does not normally exceed one page.

The size of the process' page tables (P0 + gap + P1) in pages is contained in the software extension to the pcb located at the top of the process' *u.* area (in *u_pcb.pcb_szpt).* This number is also duplicated in the *proc* structure field *p_szpt.*

Given *x* as the virtual size of a process, **ctopt(x)** results in the minimum number of pages of page tables required to map it. A process accesses its page tables through the descriptors P0BR, P0LR, P1BR, and P1LR. The per-process copies of these processor registers are contained in the pcb and are loaded and saved at process context switch time. A copy of the P0 region base register is contained in the *proc* structure field *p_p0br.*

Given the above description of the process layout in virtual memory, a pointer to a process and a page table entry, the **isa?pte** macros result in *true* if the PTE is within the respective segment of process *p:*

isaspte(p, pte)	stack segment?
isatpte(p, pte)	text segment?
isadpte(p, pte)	data segment?

Conversion between segment page numbers and pointers to page table entries can be achieved by the following macros, where *p* is the process of interest, and *i* is the virtual page number within the segment (a

*Later versions of the system for the VAX-11/780 may align the data starting at a 64K byte boundary, i.e. each of the text, data and stack segments will use an integral number of first level (*Sysmap*) entries. There would then be a minimum of one page of page tables for each segment, and sharing of text page table pages will be made simple using the ability of the first level entries to point to common page table pages.

non-negative integer). These are used in dealing with the *core map* where the page numbers are kept in this form for compactness.

tptopte(p, i)	text page number to pte
dptopte(p, i)	data page number to pte
sptopte(p, i)	stack page number to pte

The VAX hardware also supports a virtual page frame number. These begin at 0 for the first page of the P0 region and increase through the text and data regions. For the stack region they begin at the frame before *btop(USRSTACK)* and decrease. Note that the first stack page has a large (but positive) virtual page frame number.

Page frame numbers in the system are very machine dependent, and are referred to as ‘‘v’’s. The function **vtopte(p, v)** will take a *v* for a given process *p* and give back a pointer to the corresponding page table entry. The function **ptetov(p, pte)** performs the inverse operation. To decide which segment a pte is in, and to thereafter convert from pte’s to segment indices and back, the following macros can be used:

isatsv(p, v)	is v in the text segment of process p?
isadsv(p, v)	is v in the data segment of process p?
isassv(p, v)	is v in the stack segment of process p?
vtotp(p, v)	segment page number of page v, which is in text
vtodp(p, v)	segment page number of page v, which is in data
vtosp(p, v)	segment page number of page v, which is in stack
tptov(p, i)	v of i’th text page
dptov(p, i)	v of i’th data page
sptov(p, i)	v of i’th stack page
ptetotp(p, pte)	pte to a text segment page number
ptetodp(p, pte)	pte to a data segment page number
ptetosp(p, pte)	pte to a stack segment page number
tptopte(p, i)	pte pointer for i’th text page
dptopte(p, i)	pte pointer for i’th data page
sptopte(p, i)	pte pointer for i’th stack page

The functions *vtopte* and *ptetov* have trivial definitions in terms of these macros.

Page table entries as integers

In a few places in the kernel, it is convenient to deal with page table entry fields *en masse.* In this case we cast pointers to page table entries to be pointers to integers and deal with the bits of the page table entry in parallel. Thus

```
struct pte *pte;

*(int *)pte = PG_UW;
```

clears a page table entry to have only an access field allowing user writes, by referencing it as an integer. When accessing the page table entry in this way, we use the manifest constant declarations in the *pte.h* file which give us the appropriate bits.

THE KERNEL MAP

Defined in *map.h.* The kernel map is used to manage the portion of kernel virtual memory (KVM) allocated to mapping page tables of those processes that are currently loaded. On the VAX-11/780 this involves managing page table entries in the first level page table, in the *Usrptmap/usrpt* portion of the *Sysmap.* The size of the KVM devoted to mapping resident process page tables is set by USRPTSIZE in number of Sysmap entries. Note that this allows the mapping of a maximum of 64K * USRPTSIZE bytes of resident user virtual address space. The maximum can be achieved only if there is no fragmentation in the allocation.

KVM required to map the page tables of a process that is being swapped in is allocated according to a *first fit* policy through a call to the standard system resource allocator *malloc.* Once a process is swapped in, its page tables remain stationary in KVM unless the process grows such that it requires additional pages of page tables. At that time, the process' page tables are moved to a new region of KVM that is large enough to contain them. Upon swap out, the process deallocates KVM required to map its page tables through a call to the standard resource release routine *mfree.*†

There are two macros which can be used for conversion between *Usrptmap* indices and kernel virtual addresses.

kmxtob(a)	converts *Usrptmap* index to virtual address
btokmx(b)	converts virtual address to *Usrptmap* index

CORE MAP

The core map structure is defined in **cmap.h.** Each entry of core map contains eight bytes of information consisting of the following fields:

c_next	Index to the next entry in the free list. The size of this field (14 bits) limits the number of page frames that can exist in the main memory to 16K (8M bytes).
c_prev	Index to the previous entry in the free list.
c_page	Virtual page number within the respective segment (text, data or stack). The size of this field (17 bits) limits the virtual size of a process segment to 128K pages (i.e., 64M bytes).
c_ndx	Index of the proc structure that owns this page frame. In the case of shared text, the index is that of the corresponding text structure. The size of this field (10 bits) limits the number of slots in the *proc* and *text* structures to 1024.
c_intrans	The intransit bit. Important only for shared text segment pages, but set for private data pages for purposes of post-mortem analysis. Indicates that a page-in operation for the corresponding page has already been initiated by another process. Causes the faulting process to enter a wait state until awakened by the process that initiated the transfer. (This is logically part of the **c_flag** field, and is separate because of alignment considerations in the coremap.)
c_flag	8 bits of flags.

The meanings of the flags are:

MWANT	The page frame has a process sleeping on it. The process to free it should perform a wakeup on the page.
MLOCK	Lock bit. The page frame is involved in raw I/O or page I/O and consequently unavailable for replacement.
MFREE	Free list bit. The page frame is currently in the free list.
MGONE	Indicates that the virtual page corresponding to this page frame has vanished due to either having been deallocated (contraction of the data segment) or swapped out. The page will eventually be freed by the process which is holding it, usually the page-out demon.
MSYS	System page bit. The page frame has been allocated to a user process' *u.* area or page tables and therefore unavailable for replacement.
MSTACK	Page frame belongs to a stack segment.

† Due to the way in which *malloc* works, the KVM mapped by the first entry in *Usrptmap* (index 0) is not used.

MDATA	Page frame belongs to a data segment.
MTEXT	Page frame belongs to a shared text segment.

The core map is the central data base for the paging subsystem. It consists of an array of these structures, one entry for each page frame in the main memory excluding those allocated to kernel text and data.

The memory free list, managed by **memall** and **memfree** is created by doubly linking entries in core map. The reverse link is provided to speed up page reclaims from the free list which have to perform an unlink operation.

There are a pair of macros for converting between core map indices and page frame numbers, since no core map entries exist for the system.

cmtopg(x)	converts core map index x to a page frame number
pgtocm(x)	converts a page frame number to a core map index

The macros for manipulating segment page numbers, which we described in the section on page tables above, are very useful when dealing with the core map.

DISK MAP

Defined in *dmap.h.* The disk map is a per-process data structure that is kept in the process *u.* area. The fields are:

dm_size	The amount of disk space allocated that is actually used by the segment.
dm_alloc	The amount of physical disk space that is allocated to the segment.
dm_dmap	An array of disk block numbers marking the beginning of disk areas that constitute the segment disk image.

The four instances of the disk map allow the mapping of process virtual addresses to disk addresses for the parent data, parent stack, child data, and child stack segments. The two child maps are used during the *fork* system call serving to make both the parent and the child disk images accessible simultaneously.†

Each entry in the disk map array contains a disk block number (relative to the beginning of the swap area) that marks the beginning of a disk area mapping the corresponding segment of virtual space. The initial creation of the segment results in DMMIN blocks (512 bytes each) pointed to by the first disk map entry to be allocated. These disk blocks map precisely to the first DMMIN virtual pages of the corresponding segment. Subsequent growth of the segment beyond this size results in the allocation of 2*DMMIN blocks mapping segment virtual page numbers DMMIN through 3*DMMIN-1. This doubling process continues until the segment reaches a size such that the next disk area allocated has size *DMMAX* blocks. Beyond this point, the segment receives DMMAX additional blocks should it require them. Limiting the exponential growth at this size is in an effort to reduce severe disk fragmentation that would otherwise result for very large segments.

Note that increasing entries in the array map increasing segment virtual page numbers. However, in the case of the stack segment, this actually means mapping *decreasing* process virtual page numbers. Also note that since a shared text segment is static in size, its disk image is allocated in one contiguous block that is described by the text structure fields *x_daddr* and *x_size.*

The maximum size (in pages) that a segment can grow is determined by MAXTSIZ, MAXDSIZ, or MAXSSIZ for text, data, or stack segment respectively. Since the procedures that deal with the disk map panic on segment length overrun, setting the maximum size of a segment to a value greater than that can be mapped by it's disk map can lead to a system failure. To avoid such a situation, the disk map parameters should be set so that possible segment overgrowth will be detected at an earlier time in life of a process by **chksize.** Note that the maximum segment size that can be mapped by disk map can be increased through raising any one or more of the constants NDMIN, DMMAX, and NDMAP.

† These could actually be located on the kernel stack, rather than in the *u.* area.

INSTRUMENTATION RELATED STRUCTURES

Currently, the system maintains counters for various paging related events that are accumulated and averaged at discrete points in time. The basic structure as defined in *vm.h* has the following fields:

v_swpin	Process swap ins.
v_swpout	Process swap outs.
v_pswpin	Pages swapped in.
v_pswpout	Pages swapped out.
v_pgin	Page faults requiring disk I/O.
v_pgout	Dirty page writes.
v_intrans	Page faults on shared text segment pages that were found to be intransit.
v_pgrec	Page faults that were serviced by reclaiming the page from memory.
v_exfod	Fill on demand from file system of executable pages (text or data from demand initialized executables.)
v_zfod	Fill on demand type page faults which filled zeros.
v_vrfod	Fill on demand from file systems of pages mapped by *vread.*
v_nexfod	Number of pages set up for fill on demand from executed files.
v_nzfod	Number of pages set up for zero fill on demand.
v_vrfod	Number of pages set up for fill on demand with *vread.*
v_pgfrec	Pages reclaimed from the free list.
v_faults	Address translation faults, any one of the above categories.
v_scan	Page frames examined by the page demon.
v_rev	Revolutions around the loop by the page demon.
v_dfree	Pages freed by the page demon.
v_swtch	Cpu context switches.

The three instances of this structure under the names of *cnt, rate,* and *sum* serve the following purposes:

cnt	Incremental counters for the above events.
rate	The moving averages for the above events that are updated at various integral clock tick periods. The relevant macro for this operation is **ave(smooth, cnt, time)** which averages the incremental count *cnt* into *smooth* with aging factor *time.*
sum	Accumulated totals for the above events since reboot.

EXISTING DATA STRUCTURES

Here we describe fields within existing data structures that have been newly introduced or have taken a new meaning.

The Process Structure

p_slptime	Clock ticks since last sleep.
p_szpt	Number of pages of page table. This field is a copy of the *pcb_szpt* field in the pcb structure.
p_tsize	Text segment size in pages. This is a copy of the *x_size* field in the text structure.
p_dsize	Data segment size in pages.

p_ssize	Stack segment size in pages.
p_rssize	The current private segment (data + stack) *resident set size* for the process. The resident set is defined as the set of pages owned by the process that are either valid or reclaimable but not in the free list.
p_swrss	The size of the resident set at time of last swap out.
p_p0br	Pointer to the base of the P0 region page table. This is a copy of the *pcb_p0br* field in the pcb structure.
p_xlink	Pointer to another proc structure that is currently loaded and linked to the same text segment. The head of this linked list of such processes is contained in the text structure field *x_caddr.* Since the shared text portion of the process page tables are duplicated for each resident process attached to the same text segment, modifications to any one are reflected in all of them by sequentially updating the page table of each process that is on this linked list.†
p_poip	Count of number of page outs in progress on this process. If non-zero, prevents the process from being swapped in.
p_faults	Incremental number of page faults taken by the process that resulted in disk I/O.
p_aveflt	Moving average of above field.
p_ndx	Index of the process slot on behalf of which memory is to be allocated. During *vfork,* the memory of a process will be given to a child, but the reverse entries in *cmap* must still point to the original process so that the reverse links will point there when the *vfork* completes. This field thus indicates the original owner of the current process' virtual memory.

The new bits in the *p_flag* field are:

SSYS	The swapper or the page demon process.
SLOCK	Process being swapped out.
SSWAP	Context to be restored from *u_ssave* upon resume.
SPAGE	Process in page wait state.
SKEEP	Prevents process from being swapped out. Set during the reading of the text segment from the inode during exec and process duplication in fork.
SDLYU	Delayed unlock of pages. Causes the pages of the process that are faulted in to remain locked, thus ineligible for replacement, until explicitly unlocked by the process.
SWEXIT	Process working on *exit.*
SVFORK	Indicates that this process is the child in a *vfork* context; i.e. that the virtual memory being used by this process actually belongs to another process.
SVFDONE	A handshaking flag for *vfork.*
SNOVM	The parent of a *vfork.* The process has no virtual memory during this time. While this bit is set, the *p_xlink* field points to the process to which the memory was given.

The Text Structure

The new fields in the text structure are:

x_caddr	Points to the head of the linked list of proc structures of processes that are currently loaded and attached to this text segment.
x_rssize	The resident set size for this text segment.
x_swrss	The resident set size for this text segment at the time of last swap out.
x_poip	Count of number of page outs in progress on this text segment. If non-zero, prevents the process from being swapped in.

† Used slightly differently when otherwise unused during *vfork*, see **SNOVM** below.

The User Area Structure

The per-process user area contains the *u.* structure as well as the kernel stack. It is mapped to a fixed kernel virtual address (starting at 0x80040000) at process context switch. The user area is swapped in and out of disk as a separate entity and is pointed to by the proc structure field *p_swaddr* when not resident. The number of pages allocated for the process' user area and kernel stack is six pages (UPAGES), thus the base of the kernel stack for a process is 0x80040c00.

The new fields that have been added to the *u.* structure are the following:

u_pcb.pcb_cmap2	Contains the copy of Sysmap entry CMAP2 at context switch time. This kernel virtual address space mapping is made part of the process context due to the operation of the process duplication code that implements fork. Briefly, the process duplication is accomplished by copying from parent process' virtual address space to the child's virtual address space by mapping it to kernel virtual memory through CMAP2. Since this can result in faulting in the parent's address space, thus causing a block and context switch, the mapping of the child memory in the kernel must be saved and restored before the process can resume.
u_nswap	Number of times the process has been swapped. Not yet maintained.
u_majorflt	Number of faults taken by the process that resulted in disk I/O.
u_cnswap	Number of times the children of this process have been swapped. Not yet maintained.
u_cmajorflt	Number of faults taken by the children of this process that resulted in disk I/O. Not yet maintained.
u_minorflt	Number of faults taken by the process that were reclaims.
u_dmap	The disk map for the data segment.
u_smap	The disk map for the stack segment.
u_cdmap	The disk map for the child's data segment to be used during fork.
u_csmap	The disk map for the child's stack segment to be used during fork.
u_stklim	Limit of maximum stack growth. To be varied through system calls. Currently not implemented.
u_wantcore	Flag to cause core dump even if the process is very large. Set by a system call. Currently not implemented.
u_vrpages	An array with an element for each file descriptor. Gives the number of fill on demand page table entries which have this file as their **pg_fileno.** If the count is non-zero, then the file cannot be closed, either by *close* or implicitly by *dup2.*

The Inode Structure.

One field was added to the inode structure to support the *vread* system call:

i_vfdcnt	This counts the number of file descriptors (fd's) that have pages mapping this file with *vread.* If the count is non-zero, then the file cannot be truncated.