HP-UX Memory Management

White Paper

Version 1.3

5965-4641

Last modified April 7, 1997

Legal Notices

The information contained within this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Hewlett-Packard shall not be liable for errors contained herein nor for incidental consequential damages in connection with the furnishing, performance, or use of this material.

Warranty. A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

Restricted Rights Legend. Use, duplication, or disclosure by the U.S. Government Department is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 for DOD agencies, and subparagraphs (c) (1) and (c) (2) of the Commercial Computer Software Restricted Rights clause at FAR 52.227-19 for other agencies.

First Edition: April 1997 (HP-UX Release 10.30)

1 MEMORY MANAGEMENT

Objectives of this chapter

- Give an overview of physical and virtual memory
- Describe the different structures associated with virtual memory and explain their purposes
- Explain how memory is mapped from physical to virtual and vice versa.
- Explain how pages of memory are made and kept available for process/thread execution.
- Describe how swap space is managed
- Describe how process structures are set up in memory
- Understand how and why memory pages are allocated, freed up, and recovered

OVERVIEW OF PHYSICAL AND VIRTUAL MEMORY

The memory management system is designed to make memory resources available safely and efficiently among threads and processes:

- It provides a complete address space for each process, protected from all other processes.
- It enables program size to be larger than physical memory.
- It decides which threads and processes reside in physical memory and manipulates threads and processes in and out of memory.
- It manages the parts of the virtual address space of a thread or process not in physical memory and determines what portions of the address space should reside in physical memory.
- It allows efficient sharing of memory between processes.

The data and instructions of any process (a program in execution) or thread of execution within a process must be available to the CPU by residing in physical memory at the time of execution.

To execute a process, the kernel creates a per-process virtual address space that is set up by the kernel; portions of the virtual space are mapped onto physical memory. Virtual memory allows the total size of user processes to exceed physical memory. Through “demand paging”, HP-UX enables you to execute threads and processes by bringing virtual pages into main memory only as needed (that is, “on demand”) and pushing out portions of a process’s address space that have not been recently used.

The term “memory management” refers to the rules that govern physical and virtual memory and allow for efficient sharing of the system’s resources by user and system processes.

The system uses a combination of pageout and deactivation to manage physical memory. Paging involves writing recently unreferenced pages from main memory to disk from time to time. A page is this smallest unit of physical memory that can be mapped to a virtual address with a given set of access attributes. On a loaded system, total unreferenced pages might be a large fraction of memory.

Deactivation takes place if the system is unable to maintain a large enough free pool of physical memory. When an entire process is deactivated, the pages associated with the process can be written out to secondary storage, since they are no longer referenced. A deactivated process cannot run, and therefore, cannot reference its data.

Secondary storage supplements physical memory. The memory management system monitors available memory and, when it is low, writes out pages of a process or thread to a secondary storage device called a swap device. The data is read from the swap device back into physical memory when it is needed for the process to execute.

Pages

Pages are the smallest contiguous block of physical memory that can be allocated for storing data and code. Pages are also the smallest unit of memory protection. The page size of all HP-UX systems is four kilobytes.

On a PA-RISC system, every page of physical memory is addressed by a physical page number (PPN), which is a software “reduction” of the physical page number from the physical address. Access to pages (and thus to the data they contain) are done through virtual addresses, except under specific circumstances.[1]

Virtual Addresses

When a program is compiled, the compiler generates virtual addresses for the code. Virtual addresses represent a location in memory. These virtual addresses must be mapped to physical addresses (locations of the physical pages in memory) for the compiled code to execute. User programs use virtual addresses only.

The kernel and the hardware coordinate a mapping of these virtual and physical addresses for the CPU, called “address translation,” to locate the process in memory.

A PA-RISC virtual address consists of a space identifier (SID) and an offset.

- Each space ID represents a 4 GB unit of virtual memory.

[1] When virtual translation must be turned off (the D and I bits are off), pages are accessed by their absolute addresses.

- The offset portion of a virtual address is the offset into this space.

Table 1-1 **Format of a 48-bit virtual address**

Space ID (16 bits)	Offset (32 bits)

Every process running on a PA-RISC processor shares a 48-bit (or larger, depending on HP-PA architecture version) global virtual address space with the kernel and with all other processes running on that machine. Although any process can create and attempt to read or write any virtual address, the kernel uses page granularity access control mechanisms to prevent unwanted interference between processes.

When a virtual page is “paged” into physical memory, free physical pages are allocated to it from the free list. These pages may be randomly scattered throughout the memory depending on their usage history. Translations are needed to tell the processor where the virtual pages are loaded. The process of translating the virtual into physical address is called virtual address translation.

Potentially the virtual address space can be much greater than the physical address space. The virtual memory system enables the CPU to execute programs much larger than the available physical memory and allows you run many more programs at a time than you could without a virtual memory system.

Demand Paging

For a process to execute, all the structures for data, text, and so on have to be set up. However, pages are not loaded in memory until they are “demanded” by a process -- hence the term, demand paging. Demand paging allows the various parts of a process to be brought into physical memory as the process needs them to execute. Only the working set of the process, not the entire process, need be in memory at one time. A translation is not established until the actual page is accessed.

THE ROLE OF PHYSICAL MEMORY

Memory is the “container” for data storage; the general repository for high-speed data storage is close to the CPU, and is termed random access memory (RAM) or “main memory.” For the CPU to execute a process, the code and data referenced by that process must reside in random access memory (RAM). RAM holds data during process execution in two even-faster implementations of memory, registers and cache, found on the processor. RAM is shared by all processes.

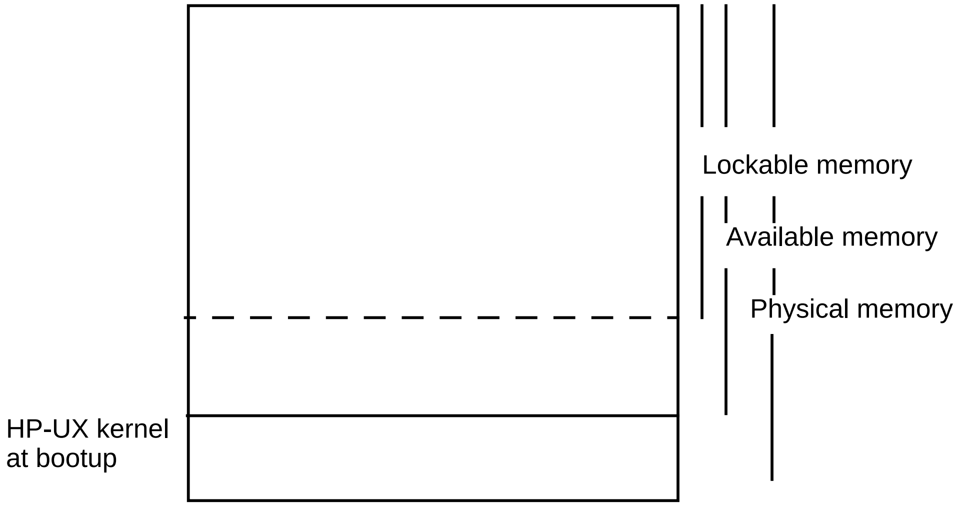
The more main memory in the system, the more data the system can access and the more (or larger) processes it can retain and execute without having to page or cause deactivation as frequently. Memory-resident resources (such as page tables) also take up space in main memory, reducing the space available to applications.

At boot time, the system loads HP-UX from disk into RAM, where it remains memory-resident until the system is shut down.

User programs and commands too are loaded from disk into RAM. When a program terminates, the operating system frees the memory used by the process.

Disk access is slow compared to RAM access. Excessive disk access can lead to increased latency or reduced throughput and can lead to the disk access becoming the bottleneck in the system. To avoid this, you need to do some sort of buffering. Buffering, paging, and deactivation algorithms optimize disk access and determine when data and code for currently running programs are returned from RAM to disk. When a user or system program writes data to disk, the data is either written directly to RAM (if raw data) or buffered in what is called buffer cache and written to disk in relatively big chunks. Programs also read files and database structures from disk into RAM. When you issue the `sync` command before shutting down a system, all modified buffers of the buffer cache are flushed (written) out to disk.

Figure 1-1 Physical memory available to processes



Available Memory

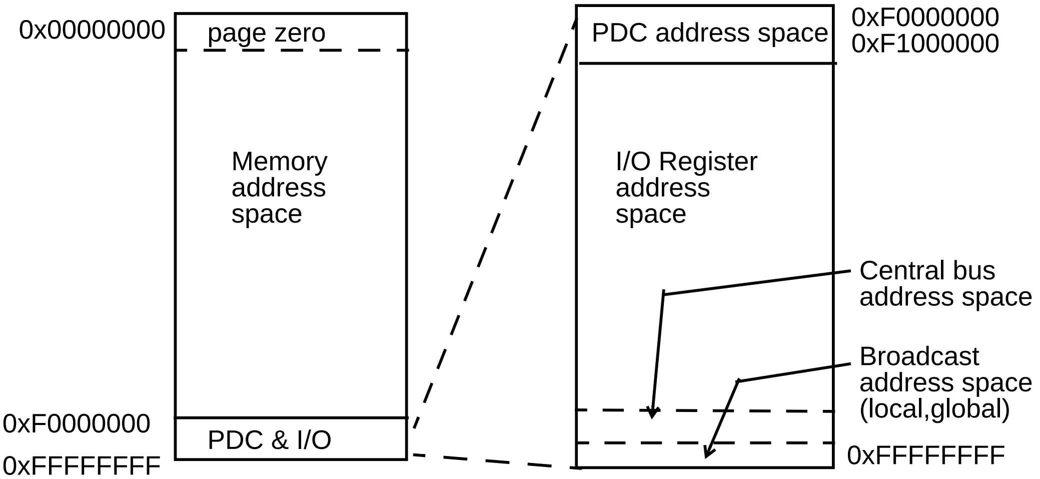
The amount of main memory not reserved for the kernel is termed available memory. Available memory is used by the system for executing user processes.

Not all physical memory is available to user processes. Kernel text and initialized data occupy about 8 MB of RAM.

Instead of allocating all its data structures at system initialization, the HP-UX kernel dynamically allocates and releases some kernel structures as needed by the system during normal operation. This allocation comes from the available memory pool; thus, at any given time, part of the available memory is used by the kernel and the remainder is available for user programs.

Physical address space is the entire range of addresses used by hardware (4GB), and is divided into memory address space, processor-dependent code (PDC) address space, and I/O address space. The next figure shows the expanse of memory available for computation. Memory address space takes up most of the system address space, while address space allotted to PDC and I/O consume a relatively small range of addresses.

Figure 1-2 **Major sections of system address space.**



Lockable Memory

Pages kept in memory for the lifetime of a process by means of a system call (such as `mlock`, `plock`, or `shmctl`) are termed lockable memory. Locked memory cannot be paged and processes with locked memory cannot be deactivated. Typically, locked memory holds frequently accessed programs or data structures, such as critical sections of application code. Keeping them memory-resident improves application performance.

The `lockable_mem` variable tracks how much memory can be locked.

Available memory is a portion of physical memory, minus the amount of space required for the kernel and its data structures. The initial value of `lockable_mem` is the available memory on the system after boot-up, minus the value of the system parameter, `unlockable_mem`.

The value of lockable memory depends on several factors:

- The size of the kernel varies, depending on the number of interface cards, users, and values of the tunable parameters.
- Available memory varies from system to system.

- The system parameter unlockable_mem is a kernel tunable parameter. Changing the value of unlockable_mem alters the default value of lockable_mem also.

HP-UX places no explicit limits on the amount of available memory you may lock down; instead, HP-UX restricts how much memory cannot be locked.

Other kernel resources that use memory (such as the dynamic buffer cache) can cause changes.

- As memory is used, the amount of memory that can be locked decreases.
- As memory is freed up, the amount of memory that can be locked increases.

As the amount of memory that has been locked down increases, existing processes compete for a smaller and smaller pool of usuable memory. If the number of pages in this remaining pool of memory falls below the paging threshold called lotsfree, the system will activates its paging mechanism, by scheduling vhand in an attempt to keep a reasonable amount of memory free for general system use.

Care must be taken to allow sufficient space for processes to make forward progress; otherwise, the system is forced into paging and deactivating processes constantly, to keep a reasonable amount of memory free.

Secondary Storage

Data is removed to secondary storage if the system is short of main memory. The data is typically stored on disks accessible either via system buses or network to make room for active processes.

Swap refers to a physical memory management strategy (predating UNIX) where entire processes are moved between main memoryand secondary storage. Modern virtual memory systems today no longer swap entire processes, but rather use a deactivation scheme that allows pages to be pushed out over time by a paging mechanism. While executing a program, pages of data and instructions can be paged out to or paged in from secondary storage if the system load warrants such behavior.

Device swap can take the form of an entire disk or LVM[1] logical volume of a disk. A file system can be configured to offer free space for swap; this is termed file-system swap. If more swap space is required, it can be added dynamically to a running system, as either device swap or file-system swap. The swapon command is used to allocate disk space or a directory in a file system for swap.

[1] Logical Volume Manager (LVM) is a set of commands and underlying software to handle disk storage resources with more flexibility than offered by traditional disk partitions.

THE ABSTRACTION OF VIRTUAL MEMORY

A computer has a finite amount of RAM available, but each HP-UX process has a 4GB virtual address space apportioned in four one-gigabyte quadrants, termed virtual memory.

Virtual memory is the software construct that allows each process sufficient computational space in which to execute. It is accomplished with hardware support.

Virtual Space in PA-RISC

As software is compiled and run, it generates virtual addresses that provide programmers with memory space many times larger than physical memory alone. The number of bits available for the space determines the ultimate size of the virtual address space. At PA-RISC 1.x, the operating system has 32-bit physical addressing and 48-bit virtual addressing (the latter consisting of 16-bit space and 32-bit offset to allow for 4 GB per space); the total virtual address range is

(2 ^ 16) * 4 GB = 262,144 GB

By comparison, Level 2 has a far greater total virtual address range of

(2 ^ 32) * 4 GB = 17,179,869,184 GB

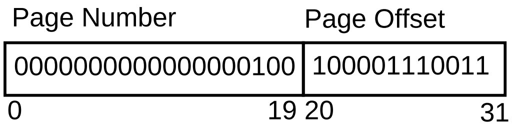
NOTE

Understand, however, that a single process has significant limitations on the virtual address space it is allowed to access. For example, a `SHARE_MAGIC` executable text is limited to 1 GB and data is limited to 1 GB. The total amount of shared virtual address space in the system is limited to 1.75 GB.

Physical Addresses

A physical address points to a page in memory that represents 4096 bytes of data. The physical address also contains an offset into this page. Thus, the complete physical address is composed of a physical page number(PPN) and page offset. The PPN is the 20 most significant bits of the physical address where the page is located. These bits are concatenated with an 12-bit page offset to form the 32-bit physical address.

Figure 1-3 Bit layout of physical address

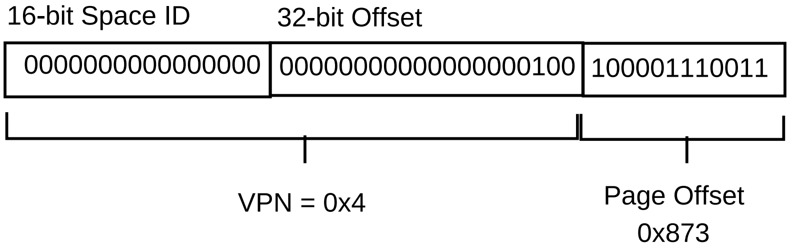


To handle the translation of the virtual address to a physical address the virtual address also needs to be looked at as a virtual page number(VPN) and page offset. Since the page size is 4096 bytes, the low order 12 bits of the offset are assumed to be the offset into the page. The space ID and the high order 20 bits of the offset are the VPN.

For any given address you can determine the page number by discarding the least significant 12 bits. What remains is the virtual page number for a virtual address or the physical page number for the physical address.

The next figure shows the bit layout of a virtual address of 0x0.4873.

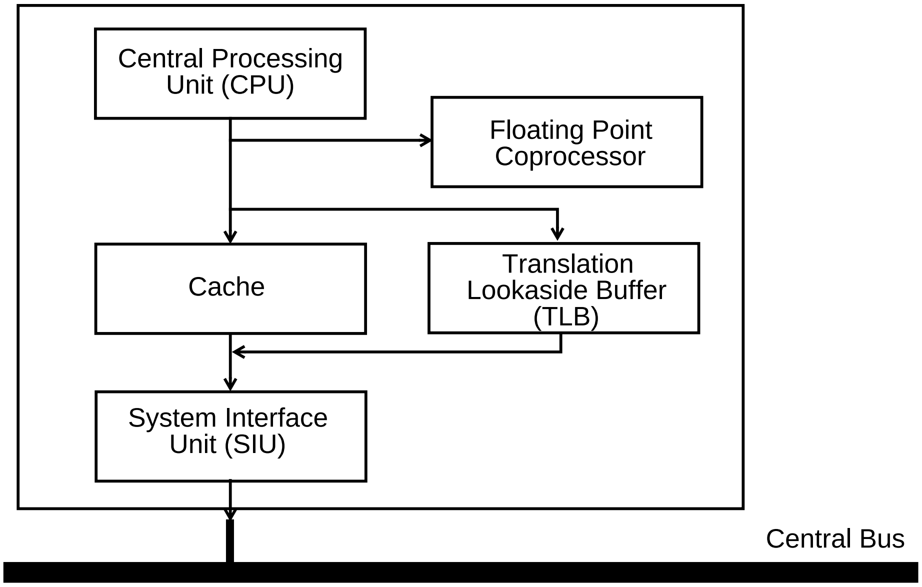
Figure 1-4 Bit layout of virtual page address



The virtual page number’s address must be translated to obtain the associated physical page number, with page offset 0x873.

MEMORY-RELEVANT PORTIONS OF THE PROCESSOR

Figure 1-5 Processor architecture, showing major components


Central Processing Unit (CPU)
Floating Point Coprocessor
Cache
Translation Lookaside Buffer (TLB)
System Interface Unit (SIU)
Central Bus

The figure above and the table that follows, name the principal processor components; of them, registers, translation lookaside buffer, and cache are crucial to memory management, and will be discussed in greater detail following the table.

Table 1-2 **Processor Architecture, components and purposes**

Component	Purpose
Central Processing Unit (CPU)	The main component responsible for reading program and data from memory, and executing the program instructions. Within the CPU are the following: • Registers, high-speed memory used to hold data while it is being manipulated by instructions, for computations, interruption processing, protection mechanisms, and virtual memory management. Registers are discussed shortly in greater detail. • Control Hardware (also called instruction or fetch unit) that coordinates and synchronizes the activity of the CPU by interpreting (decoding) instructions to generate control signals that activate the appropriate CPU hardware. • Execution Hardware to perform the actual arithmetic, logic, and shift operations. Execution Hardware can take on many specialized tasks but most common are the Arithmetic and Logic Unit (ALU) and the Shift Merge Unit (SMU).

Component	Purpose
Instruction and Data Cache	The cache is a portion of high-speed memory used by the CPU for quick access to data and instructions. The most recently accessed data is kept in the cache.
Translation Lookaside Buffer (TLB)	The processor component that enables the CPU to access data through virtual address space by : • Translating the virtual address to physical address. • Checking access rights, so that access is granted to instructions, data, or I/O only if the requesting process has proper authorization.
Floating Point Coprocessor	An assist processor that carries out specialized tasks for the CPU.
System Interface Unit (SIU)	Bus circuitry that allows the CPU to communicate with the central (native) bus.

The Page Table or PDIR

The operating system maintains a table in memory called the Page Directory (PDIR) which keeps track of all pages currently in memory. When a page is mapped in some virtual address space, it is allocated an entry in the PDIR. The PDIR is what links a physical page in memory to its virtual address.

The PDIR is implemented as a memory-resident table of software structures called page directory entries (PDEs), which contain virtual addresses. The PDIR maps the entire physical memory with one entry for every page in physical memory. Each entry contains a 48/64 bit virtual address. When the processor needs to find a physical page not indexed in the TLB, it can search the PDIR with a virtual address until it finds a matching address.

The PDIR table is a hash table with collision chains. The virtual address is used to hash into one of the buckets in the hash table and the corresponding chain is searched until a chain entry with a matching virtual address is found.

Page Fault

A trap occurs because translation is missing in the translation lookaside buffer (TLB, discussed shortly). If the processor can find the missing translation in the PDIR, it installs it in the TLB and allows execution to continue. If not, a page fault occurs.

A page fault is a trap taken when the address needed by a process is missing from the main memory. This occurrance is also known as a PDIR miss. A PDIR miss indicates that the page is either on the free list, in the page cache, or on disk; the memory management system must then find the requested page on the swap device or in the file system and bring it into main memory.

Conversely, a PDIR hit indicates that a translation exists for the virtual address in the TLB.

The Hashed Page Directory (hpde) structure

Each PDE contains information on the virtual-to-physical address translation, along with other information necessary for the management of each page of virtual memory.   The structural elements of the hashed page directory for PA-RISC 1.1 are shown in the following table.

Table 1-3 **`struct hpde`, the hashed page directory**

Element	Meaning
`pde_valid`	Flag set by the kernel to indicate a valid pde entry.
`pde_vpage`	Virtual page - high 15 bits of the virtual offset
`pde_space`	Contains the complete 16-bit virtual space
`pde_ref`	Reference bit set by the kernel when it receives certain interrupts; used by vhand() to tell if a page has been used recently

Element	Meaning
pde_accessed	Used by the stingy cache flush algorithm to indicate that the page may be in data cache[a]
pde_rtrap	Data reference trap enable bit; when set, any access to the page causes a page reference trap interruption
pde_dirty	Dirty bit; marked if the page differs in memory from what is on disk.
pde_dbrk	Data break; used by the TLB
pde_ar	Access rights; used by the TLB.[b]
pde_protid	Protection ID, used by the TLB.
pde_executed	Used by the stingy cache flush algorithm to indicate that page is referenced as text.
pde_uip	Lock flag used by trap-handling code.
pde_phys	Physical page number; the physical memory address divided by the page size.
pde_modified	Indicator to the high-level virtual memory routines as to whether the page has been modified since last written to a swap device.
pde_ref_trickle	Trickle-up bit for references. Used with pde_ref on systems whose hardware can search the htbl directly.
pde_block_mapped	Block mapping flag; indicates page is mapped by block TLB and cannot be aliased.
pde_alias	Virtual alias field. If set, the pde has been allocated from elsewhere in kernel memory, rather than as a member of the sparse PDIR.
pde_next	Pointer to next entry, or null if end of list.

[a] Stingy cache flush is a performance enhancement by which the kernel recognizes whether or not to flush the cache.

[b] For detailed information on access rights, see the *PA-RISC 2.0 Architecture reference*, chapter 3, “Addressing and Access Control.” For information about how programs can manipulate this field, see mmap(2) and mprotect(2) manpages.

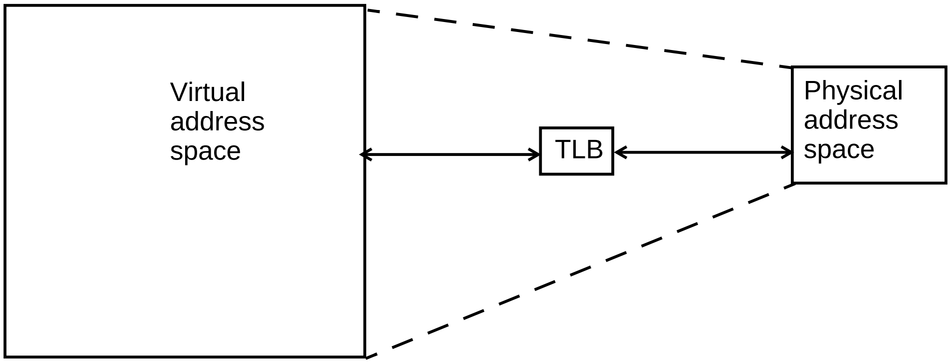
A word-oriented hpde structure (struct whpde) is implemented for faster manipulation and is documented in /usr/include/machine/pde.h. The pde.h header file also contains the definitions space for manipulation, maximum number of entries in the PDIR hashtable, constants related to field positions within the PDE structure, access rights (which are now given on a region basis), and another hashed page directory (struct hpde2_0) for PA-RISC 2.0.

NOTE The 2.0 version of the hpde structure has a field named var_page that can hold the page size information. This is used in implementing super-pages (>4K) on systems based on the PA 2.0 processor.

Translation Lookaside Buffer (TLB)

The translation lookaside buffer (TLB) translates virtual addresses to physical addresses.

Figure 1-6 Role of the TLB.



Address translation is handled from the top of the memory hierarchy hitting the fastest components first (such as the TLB on the processor) and then moving on to the page directory table (pdir in main memory) and lastly to secondary storage.

Organization and Types of TLB

Depending on model, the TLB may be organized on the processor in one of two ways:

- Unified TLB - A single TLB that holds translations for both data and instructions.

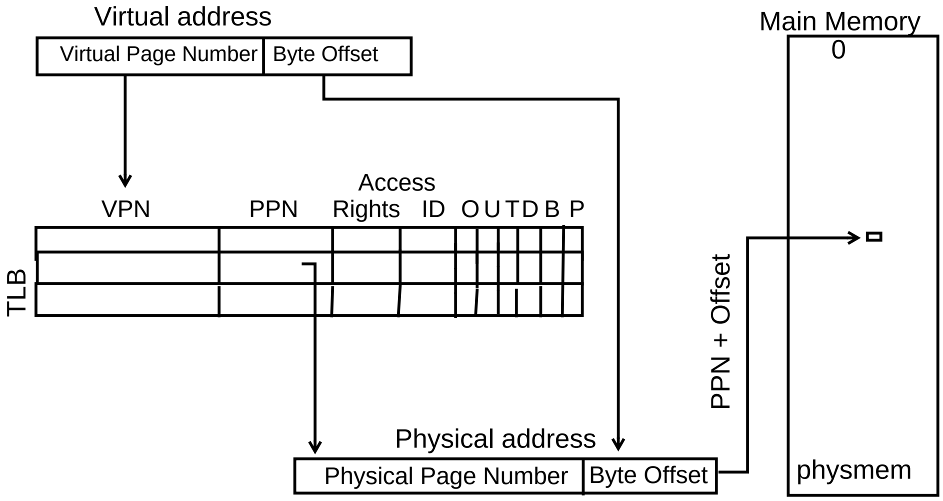
- Split Data and Instruction TLB - Dual TLB units in the processor each of which hold translations specifically for data or instructions.

At one time many systems were being designed with split Data TLB (DTLB) and Instruction TLB (ITLB), to account for the different characteristics of data and instruction locality and type of access (frequent random access of data versus relatively sequential single usage of instructions). Cost factors have allowed the inclusion of much larger TLBs on processors, which has lessened the disadvantages of a unified TLB. As a result many newer processors have unified TLBs.

Block TLB

In addition to the standard TLB that maps each entry to a single page of memory, many processors also have a block TLB. The block TLB is used to map entries to virtual address ranges larger than a single page, that is, multiple hpdes. Block TLB entries are used to reference kernel memory that remains resident. Since the operating system moves data in and out of memory by pages, a range of pages referenced by a block TLB entry is locked in memory and cannot be paged out. Addressing blocks of pages thus increases the overall address range of the TLB and the speed with which large transactions can be serviced, and thus may be thought of as a hardware implementation of large pages. The block TLB is typically used for graphics, because their data is accessed in huge chunks. It is also used for mapping other static areas such as kernel text and data.

Figure 1-7 The TLB is a cache for address translations



The TLB translates addresses

The TLB looks up the translation for the virtual page numbers (VPNs) and gets the physical page numbers (PPNs) used to reference physical memory.

Ideally the TLB would be large enough to hold translations for every page of physical memory; however this is prohibitively expensive; instead the TLB holds a subset of entries from the page directory table (PDIR) in memory. The TLB speeds up the process of examining the PDIR by caching copies of its most recently utilized translations.

Because the purpose of the TLB is to satisfy virtual to physical address translation, the TLB is only searched when memory is accessed while in virtual mode. This condition is indicated by the D-bit in the PSW (or the I-bit for instruction access).

TLB Entries

Since the TLB translates virtual to physical addresses, each entry contains both the Virtual Page Number (VPN) and the Physical Page Number (PPN). Entries also contain Access Rights, an Access Identifier, and five flags.

Table 1-4 **TLB flags (PA 2.x architecture)**

Flag	Meaning
O	Ordered. Accesses to data for load and store are ranked by strength -- strongly ordered, ordered, and weakly ordered. (See PA-RISC 2.0 specifications for model and definitions.)
U	Uncacheable. Determines whether data references to a page from memory address space may be moved into the cache. Typically set to 1 for data references to a page that maps to the I/O address space or for memory address space that must not be moved into cache.
T[a]	Page Reference bit. If set, any access to this page causes a reference trap to be handled either by hardware or software trap handlers
D	Dirty Bit. When set, this bit indicates that the associated page in memory differs from the same page on disk. The page must be flushed before being invalidated.
B	Break.This bit causes a trap on any instruction that is capable of writing to this page
P	Prediction method for branching; optional, used for performance tuning.

[a] The T,D, and B flags are only present in data or unified TLBs.

In PA 1.x architecture, an E bit (or “valid” bit) indicates that the TLB entry reflects the current attributes of the physical page in memory.

Instruction and Data Cache

Cache is fast, associative memory on the processor module that stores recently accessed instructions and data. From it, the processor learns whether it has immediate access to data or needs to go out to (slower) main memory for it.

Cacheable data going to the CPU from main memory passes through the cache. Conversely, the cache serves as the means by which the CPU passes data to and from main memory. Cache reduces the time required for the CPU to access data by maintaining a copy of the data and instructions most recently requested.

A cache improves system performance because most memory accesses are to addresses that are very close to or the same as previously accessed addresses. The cache takes advantage of this property by bringing into cache a block of data whenever the CPU requests an address. Though this depends on size of the cache, associativity, and workload, a vast majority of the time (according to performance measurements), the cache has what you're looking for the next time, enabling you to reference it.

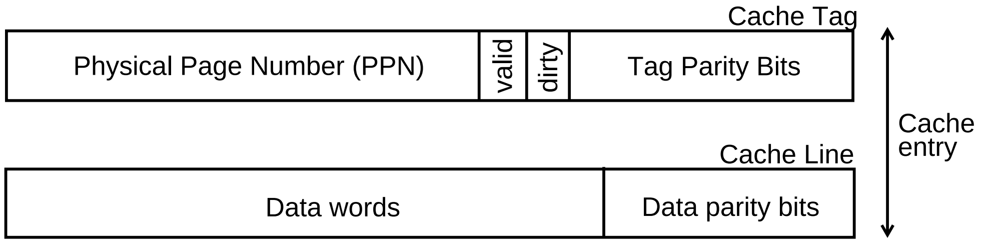
Cache Organization

Depending on model, PA-RISC processors are equipped with either a unified cache or separate caches for instructions and data (for better locality and faster performance). In multiprocessing systems, each processor has its own cache, and a cache controller maintains consistency.

Cache memory itself is organized as follows:

- A quantity of equal-sized blocks called cache lines, defined to be the same unit of size as data passed between cache and main memory. A cache line can be 16, 32, or 64 bytes long, aligned.
- One 15-bit long cache tag for every cache line, to describe its contents and determine if the desired data is present. The tag contains
    - Physical Page Number (PPN), identifying the page in main memory where the data resides.
    - Flag Bits When set, a valid flag indicates the cache line contains valid data. A dirty bit is set if the CPU has modified contents of the cache line; that is, the cache (not main memory) contains the most current data. If the dirty bit is not set, the flag is said to be “clean,” meaning that the cache line does not have modified contents. Other implementation-specific flags may be present.
- Both the cache tag and cache line have associated parity bits used for checksumming, to make sure the line is correct.

Figure 1-8 **Every cache entry consists of a cache tag and cache line.**



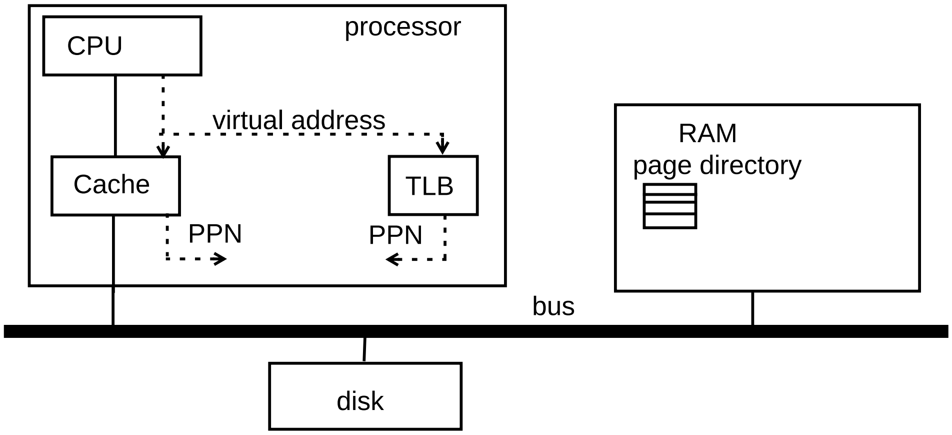
How the CPU Uses Cache And TLB

When a process executes, it stores its code (text) and data in processor registers for referencing. If the data or code is not present in the registers, the CPU supplies the virtual address of the desired data to the TLB and to the cache controller. Depending on implementation, caches can be direct mapped, set associative, or fully associative. Recent PA implementations use direct associative caches and fully associative TLBs. Virtual addresses can be sent in parallel to the TLB and cache because the cache is virtually indexed.

A physical page may not be referenced by more than one virtual page, and a virtual address cannot translate to two different physical addresses; that is, PA-RISC does not support hardware address aliasing, although HP-UX implements software address aliasing for text only in EXEC_MAGIC executables.

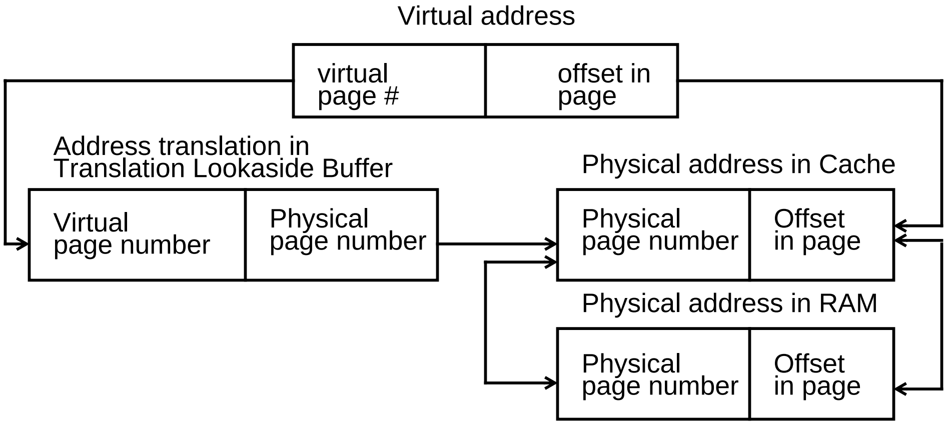
The cache controller uses the low-order bits of the virtual address to index into the direct-mapped cache. Each index in the cache finds a cache tag containing a physical page number (PPN) and a cache line of data. If the cache controller finds an entry at the cache location, the cache line is checked to see whether it is the right one by llooking at the PPN in the cache tag and the one returned by the TLB, because blocks from many different locations in main memory can be mapped legitimately to a given cache location. If the data is not in cache but the page is translated, the resultant data cache miss is handled completely by the hardware. A TLB miss occurs if the page is not translated in the TLB; if the translation is also not in the PDIR, HP-UX uses the page fault code to fault it in. If not in RAM, the data and code might have to be paged from disk, in which case the disk-to-memory transaction must be performed.

Figure 1-9 PPNs from Cache and TLB are compared



On a more detailed level, the next figure demonstrates the mapping of virtual and physical address components.

Figure 1-10 Virtual address translation



TLB Hits and Misses

The sequence followed by the processor as it validates addresses is one of “hit or miss.”

- The TLB is searched; that is, each virtual address and byte offset issued by the processor indexes an entry in the TLB.

- If the entry is valid, it is known as a TLB hit. The TLB contains a valid physical page number (PPN), which might be accessed in cache.
- If the entry is invalid or the TLB cannot provide a physical page number, a TLB miss occurs and must be handled. On certain systems, a hardware walker searches the PDIR and if it finds the page, updates the TLB. On systems not equipped with a hardware TLB handler or if the hardware walker does not find an entry in the PDIR, a software interrupt is generated. The software interrupt resolves the fault and updates the TLB, allowing the access to proceed.

There are five TLB miss handlers (instruction, data, non-access instruction, nonaccess data, and dirty) located in `locore.s`; the header file `pde.h` has the TLB/PDIR structure definition.

TLB Role in Access Control and Page Protection

In addition to assisting in virtual address translation, the translation lookaside buffer (TLB) serves a security function on behalf of the processor, by controlling access and ensuring that a user process sees only data for which it has privilege rights.

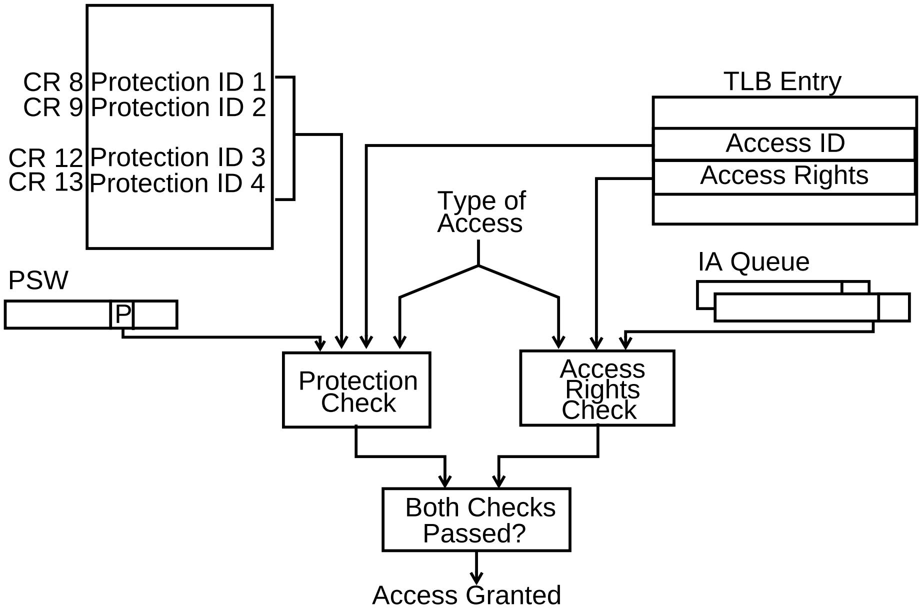
The TLB contains access rights and protection identifiers. PA-RISC allows up to four protection IDs to be associated with each process. These IDs are held in control registers CR-8, CR-9, CR-12, and CR-13.

Table 1-5 Security checks in the TLB

Security check	Purpose
Protection Checks	The P-bit (Protection ID Validation Enable bit) of the Processor Status Word (PSW) is checked: • If not set, protection checking on the page is waived, as though passed and checking proceeds to access rights validation. • If the protection ID validation bit is set, the access ID of the TLB entry is compared to the protection IDs in CR-8, CR-9, CR-12, and CR-13.
Access Rights Check	Access Rights are stored in a seven-bit field containing permissible access type and two privilege levels affecting the executing instruction: • Access types are read, write, execute. • Privilege levels checked for read access and write access, kernel and user execution.

Figure 1-11 shows the checkpoints for controlling access to a page of data through the TLB. Two checks are performed: protection check and access rights check. If both checks pass, access is granted to the page referenced by the TLB.

Figure 1-11 Access control to virtual pages



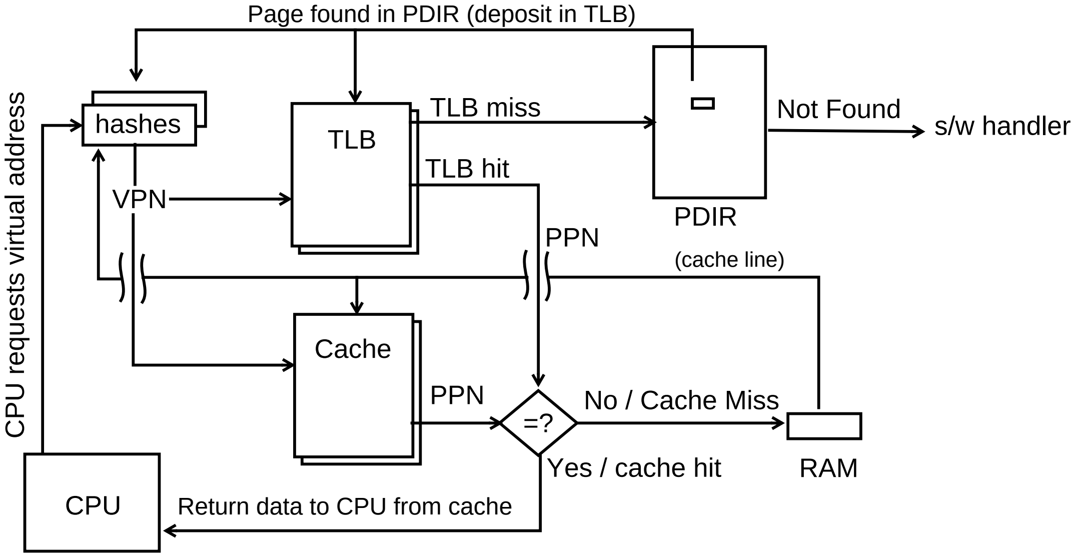
Cache Hits and Misses

- When the cache line was first copied into the cache, its Physical Page Number was stored in the corresponding cache tag. The cache controller compares the PPN from the tag to the PPN supplied by the TLB.
    - If the PPN in the cache tag matches the PPN from the TLB, a cache hit occurs. The data is present in the cache and is supplied to the CPU.
    - If the PPN in the cache tag does not match the PPN from the TLB, a cache miss occurs. In a cache miss, the cache line is loaded from memory, because the byted referenced on the virtual page are not yet in cache. (Typically, our implementations do not load an entire page at a time to the cache; they load a cache line at a time.) The data is absent from cache and the CPU must wait while the data is brought into the cache from main memory.

If the two PPNs do not match (assuming a TLB hit), the cache line is loaded because the bytes referenced on the virtual page are not yet in the cache. The time it takes to service a cache miss varies depending on if the data already present in the cache is clean or dirty. (When the cache is dirty, the old contents are written out to memory and the new contents are read in from memory.) If the cache line is "clean" (that is, not modified), it does not have to be written back to main memory, and the penalty is fewer instruction cycles than if the cache is dirty and must be written back to main memory.

All PA-RISC machines use a cache write-back policy, meaning that the main memory is updated only when the cache line is replaced.

Figure 1-12 Summary of page retrieval from TLB, Cache, PDIR



PA-RISC allows for privilege level promotion by using a GATEWAY instruction. This instuction performs an interspace branch to increase the privilege level. The most common example of this in HP-UX is a system call, which changes the privilege level from user to kernel.

Registers

Registers, high-speed memory in the processor's CPU, are used by the software as storage elements that hold data for instruction control flow, computations, interruption processing, protection mechanisms, and virtual memory management.

All computations are performed between registers or between a register and a constant (embedded in an instruction), which minimizes the need to access main memory or code. This register-intensive approach accelerates performance of a PA-RISC system. This memory is much faster than conventional main memory but it is also much more expensive, and therefore used for processor-specific purposes.

Registers are classified as privileged or non-privileged, depending on the privilege level of the instruction being executed.

Table 1-6 Types of Registers

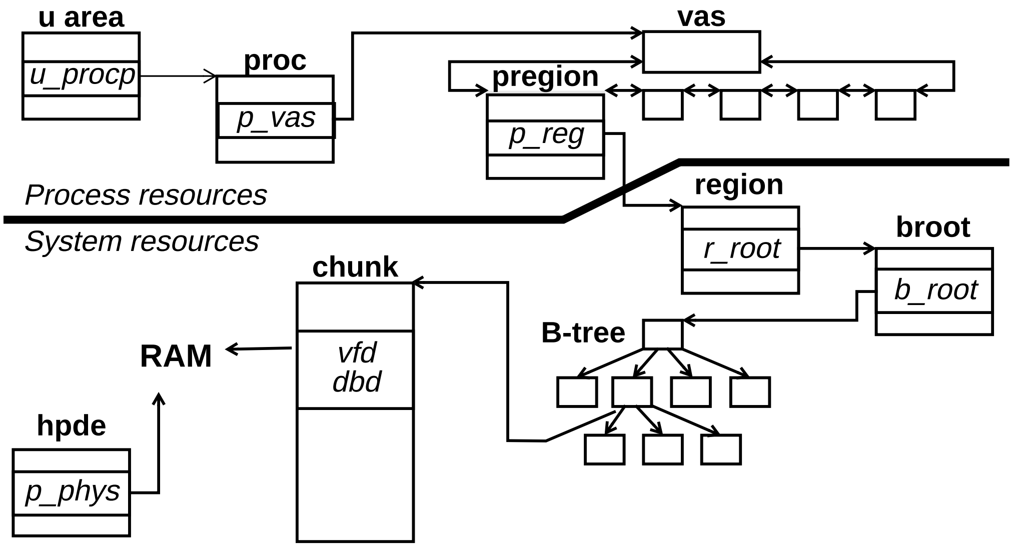
Type of Register	Purpose
32 General Registers, each 32 bits in size. (non-privileged)	Used to hold immediate results or data that is accessed frequently, such as the passing of parameters. Listed are those with uses specified by PA-RISC or HP-UX. • GR0 - Permanent Zero • GR1 - ADDIL target address • GR2 - Return pointer. Contains the instruction offset of the instruction to which to return • GR23 - Argument word 3 (arg3) • GR24 - Argument word 2 (arg2) • GR25 - Argument word 1 (arg1) • GR26 - Argument word 0 (arg0) • GR27 - Global data pointer (dp) • GR28 - Return value • GR29 - Return value (double) • GR30 - Stack pointer (sp)
7 Shadow Registers (privileged)	Store contents of GR1,8,9,16,17,24, and 25 on interrupt, so that they can be restored on return from interrupt. Numbered SHR0-SHR6.
8 Space Registers, holding 16, 24, or 32-bit space ID. (SR5-SR7 are privileged)	Hold the space IDs for the current running process. • SR0 - Instruction address space link register used for branch and link external instructions. • SR1-SR7 - Used to form virtual addresses for processes.

Type of Register	Purpose
32 Control Registers, each 32 bits. (most are privileged)	Used to reflect different states of the system, many related primarily to interrupt handling. • CR0 - Recovery Counter, used to provide software recovery of hardware faults in fault-tolerant systems and for debugging. • CR10 - Low-order bits are known as the Coprocessor Configuration Register (CCR), 8 bits that indicate presence and usability of coprocessors. Bits 0, 1 correspond to the floating point coprocessor; bit 2, the performance monitor coprocessor. • CR14 - Interruption Vector Address (IVA) • CR16 - Interval Timer. Two internal registers, one counting at a rate between twice and half the implementation-specific “peak instruction rate”, the other register containing a 32-bit comparison value. Each processor in a multi-processor system has its own Interval Timer, but they need not be synchronized nor clock at the same frequency. • CR17 - Stores the contents of the Instruction Address Space Queue at time of interruption. • CR19 - Used to pass an instruction to an interrupt handler. • CR20, CR21 - Used to pass a virtual address to an instruction handler. • CR26, CR27 - Temporary registers readable by code executing at any privilege level but writable only by privileged code.

Type of Register	**Purpose**
64 Floating Point Registers, 32-bits each, or 32, 64-bits each.	Data registers used to hold computations. • FP-0L - Status register. Controls arithmetic modes, enables traps, indicates exceptions, results of comparison, and identifies coprocessor implementation. • FP-0R through FP-3 - Exception registers, containing information on floating point operations whose execution has completed and caused a delayed trap.
2 Instruction Address Queues, each 64 bits	Two queues 2 elements deep. The front elements of the queues (IASQ_Front and IAOQ_Front) form the virtual address of the current instruction, while the back elements (IASQ_Back and IAOQ_Back) contain the address of the following instruction. • Instruction Address Space Queue holds the space ID of the current and following instruction. • Instruction Address Offset Queue holds the offset of the instruction for the given space High-order 62 bits contain the wrod offset of the instruction; the 2 low-order bits maintain the privilege level of the instruction.
1 Processor Status Word (PSW), 32 bits (privileged)	Contains the current processor state. When an interruption occurs, the PSW is saved into the Interrupt Processor Status Word (IPSW), to be restored later. Low-order five bits of the PSW are the system mask, and are defined as mask/unmask or enable/disable. Interrupts disabled by PSW bit are ignored by the processor; interrupts masked remain pending until unmasked.

VIRTUAL MEMORY STRUCTURES

Figure 1-13 Memory management structures



Process management uses kernel structures down to the `pregions` to execute a process. The `u_area`, `proc` structure, `vas`, and `pregion` are per-process resources, because each process has its own unique copies of these structures, which are not shared among multiple processes.

Below the `pregion` level are the systemwide resources. These structures can be shared among multiple processes (although they are not required to be shared).

Memory management kernel structures map pregions to physical memory and provide support for the processor's ability to translate virtual addresses to physical memory. The table that follows introduces the structures involved in memory management; these are discussed later in detail.

Table 1-7 **Principal Memory Management Kernel Structures**

Kernel structure	Purpose
vas	Keeps track of the structural elements associated with a process in memory. One vas maintained per process.
pregion	A per-process resource that describes the regions attached to the process.
region	A memory-resident system resource that can be shared among processes. Points to the process's B-tree, vnode, pregions.
B-tree	Balanced tree that stores pairs of page indices and chunk addresses. At the root of a B-tree of VFDs and DBDs is struct broot.
hpde	Contains information for virtual to physical translation (that is, from VFD to physical memory).

Virtual Address Space (vas)

The vas represents the virtual address space of a process and serves as the head of a doubly linked list of process region data structures called pregions. The vas data structure is always memory resident.

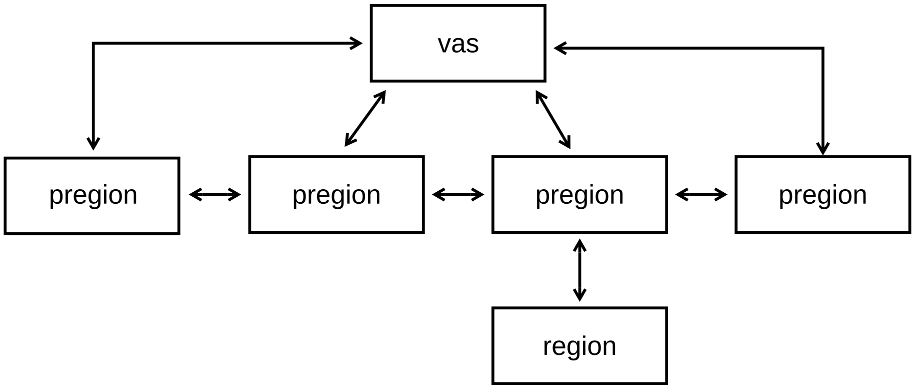
When aprocess is invoked, the system allocates a vas structure and puts its address in p_vas, a field in the proc structure.

The virtual address space of a process is broken down into logical chunks of virtually contiguous pages. (See the Process Management white paper for table of vas entries.)

Virtual memory elements of a pregion

Each pregion represents a process's view of a particular portion of pages and information on getting to those pages. The pregion points to the region data structure that describes the pages' physical locations in memory or in secondary storage. The pregion also contains the virtual addresses to which the process's pages are mapped, the page usage (text, data, stack, and so forth), and page protections (read, write, execute, and so on).

Figure 1-14 **Virtual memory elements of the pregion**



The following elements of a per-process pregion structure are important to the virtual memory subsystem.

Table 1-8 **Principal elements of struct pregion**

Element	Purpose
p_type	Type of pregion
*p_reg	Pointer to the region attached by the pregion.
p_space, p_vaddr	Virtual address of the pregion, based on virtual space and virtual offset.
p_off	Offset into the region, specified in pages.
p_count	Number of pages mapped by the pregion.
p_ageremain, p_agescan, p_stealscan, p_bestnice	Used in the vhand algorithm to age and steal pages of memory (discussed later).
*p_vas	Pointer to the vas to which the pregion is linked.
p_forw, p_back	The doubly-linked list, used by vhand to walk the active pregions.

Element	Purpose
p_deactsleep	The address at which a deactivated process is sleeping.
p_pagein	Size of an I/O, used for scheduling when moving data into memory.
p_strength, p_nextfault	Used to track the ratio between sequential and random faults; used to adjust p_pagein.

The Region, a system resource

The region is a system-wide kernel data structure that associates groups of pages with a given process. Regions can be one of two types, private (used by a single process) or shared (able to be used by more than one process). Space for a region data structure is allocated as needed. The region structure is never written to a swap device, although its B-tree may be.

Regions are pointed to by pregions, which are a per-process resource. Regions point to the vnode where the blocks of data reside when not in memory.

Table 1-9 **region (struct region)**

Element	Meaning
r_flags	Region flags (enumerated shortly).
r_type	• RT_PRIVATE: Multiple processes cannot share region. PT_DATA and PT_STACK pregions point to RT_PRIVATE regions. • RT_SHARED: Multiple processes can share region. PT_SHMEM and most PT_TEXT pregions point to RT_SHARED regions.
r_pgsz	Size of region in pages if all pages are in memory.
r_nvalid	Number of valid pages in region. This equals the number of valid vfds in the B-tree or b_chunk.

Element	Meaning
r_dnvalid	Number of pages in swapped region. If the system swaps the entire process, the value of r_nvalid is copied here to later calculate how many pages the process will need when it faults back in. This information is used to decide which process to reactivate.
r_swalloc	Total number of pages reserved and allocated for this region on the swap device. Does not account for swap space allocated for vfd/dbd pairs.
r_swapmem, r_vfd_swapmem	Memory reserved for pseudo-swap or vfd swap.
r_lockmem	Number of pages currently allocated to the region for lockable memory, including lockable memory allocated for vfd/dbd pairs.
r_pswapf, r_pswapb	Forward and backward pointers to lists of pseudo-swap pages.
r_refcnt	Number of pregions pointing at the region
r_zomb	Set to indicate modified text. If an executing a.out file on a remote system has changed, the pages are flushed from the processor's cache, causing the next attempted access to fault. The fault handler finds that r_zomb is non-zero, prints the message Pid %d killed due to text modification or page I/O error and sends the process a SIGKILL.
r_off	Offset into the page-aligned vnode, specified in pages; valid only if RF_UNALIGNED is not set. Page r_off of the vnode is referenced by the first entry of the first chunk of the region's B-tree.
r_incore	Number of pregions sharing the region whose associated processes have the SLOAD flag set.

Element	Meaning
r_mlockcnt	Number of processes that have locked this region in memory.
r_dbd	Disk block descriptor for B-tree pages written to a swap device Specifies the location of the first page in the linked list of pages.
r_fstore, r_bstore	Pointers to vnode of origin and destination of block. This data depends on the type of pregion above the region. In most cases, r_bstore is set to the paging system vnode, the global swapdev_vp that is initialized at system startup.
r_forw, r_back	Pointers to linked list of all active pregions.
r_hchain	Hash for region.
r_lock	Region lock structure used to get read or read/write locks to modify the region structure.
r_mlock	Wait for region to be locked in memory.
r_poip	Number of page I/Os in progress
r_root	Root of B-tree; if referencing more than one chunk, r_key is set to DONTUSE_IDX.
r_key, r_chunk	Used instead of B-tree search if referencing 32 or fewer pages.
r_excproc	Pointer to the proc table entry, if the process has RF_EXCLUSIVE set in r_flags.
r_hdl	Hardware-dependent layer structure
r_next, r_prev	Circularly linked list of all regions sharing pages/vnode.

Element	Meaning
r_pregs	List of pregions pointing to the region.
r_lchain	Linked list of memory lock ranges
r_mlockswap	swap reserved to cover locks

a.out Support for Unaligned Pages

Text and data of most executables start on a four-kilobyte page boundary. HP-UX can treat these as memory-mapped files, because a page in the file maps directly to a page in memory.

In addition to the fields shown, struct region has fields to support executables compiled on older versions of HP-UX whose text and data do not align on a (4 KB) page boundary. These executables are referenced by regions whose r_flag is set to RF_UNALIGNED.

Table 1-10 **a.out support by regions**

Element	Meaning
r_byte, r_bytelen	Offset into the a.out file and length of its text.
r_hchain	Hash list of unaligned regions.

Region flags

Various indicators of the state of the region are specified in r_flags.

Table 1-11 Region flags

Region flag	Meaning
RF_ALLOC	Always set because HP-UX regions are allocated and freed on demand; there is no free list.
RF_MLOCKING	Indicator of whether a region is locked; set before `r_mlock`, cleared after `r_mlock` is released.
RF_UNALIGNED	Set if text of an executable does not start on a page boundary. In this case, the text is read through the buffer cache to align it, and the `vfd`s are pointed at the buffer cache pages.
RF_WANTLOCK	Set if another stream wanted to lock this region, but found it already locked and went to sleep. After the region is unlocked, this flag ensures that `wakeup()` is called so the waiting stream(s) can proceed.
RF_HASHED	The text is unaligned (RF_UNALIGNED) and thus is on a hash chain. The region is hashed with `r_fstore` and `r_byte`; the head of each hash chain is in `texts[]`. The RF_UNALIGNED flag may be set without the RF_HASHED flag (if the system tries to get the hashed region but it is locked, the system will create a private one), but the RF_HASHED flag will never be set without the RF_UNALIGNED flag.
RF_EVERSWP, RE_NOWSWP	Set if the `B-tree` has ever been or is now written to a swap device. These flags are used for debugging.
RF_IOMAP	This region was created with an `iomap()` system call, and thus requires special handling when calling `exit()`.

Region flag	Meaning
RF_LOCAL	Region is swapped locally.
RF_EXCLUSIVE	The mapping process is allowed exclusive access to the region. This flag is set, and r_excproc is set to the proc table pointer.
RF_SWLAZYWRT	If an a.out is marked EXEC_MAGIC, a lazy swap algorithm is used, meaning swap is not reserved or allocated until needed. The text file is not likely to be modified, but if it is, a page of swap will be reserved for it at that time.
RF_STATIC_PREDICT	Text object uses static branch prediction for compiler optimization.
RF_ALL_MLOCKED	Entire region is memory locked, as a result of a plock having been performed on the pregion associated with the region.
RF_SWAPMEM	Region is using pseudo-swap; that is, a portion of memory is being held for swap use.
RF_LOCKED_LARGE	Region is using large pages; used with superpages.
RF_SUPERPAGE_TEXT	Text region using large pages.
RF_FLIPPER_DISABLE	Disable kernel assist prediction; a flag used for performance profiling.
RF_MPROTECTED	Some part of the region is subject to the system call mprotect, which is performed on an memory-mapped file.

pseudo-vas for Text and Shared Library pregions

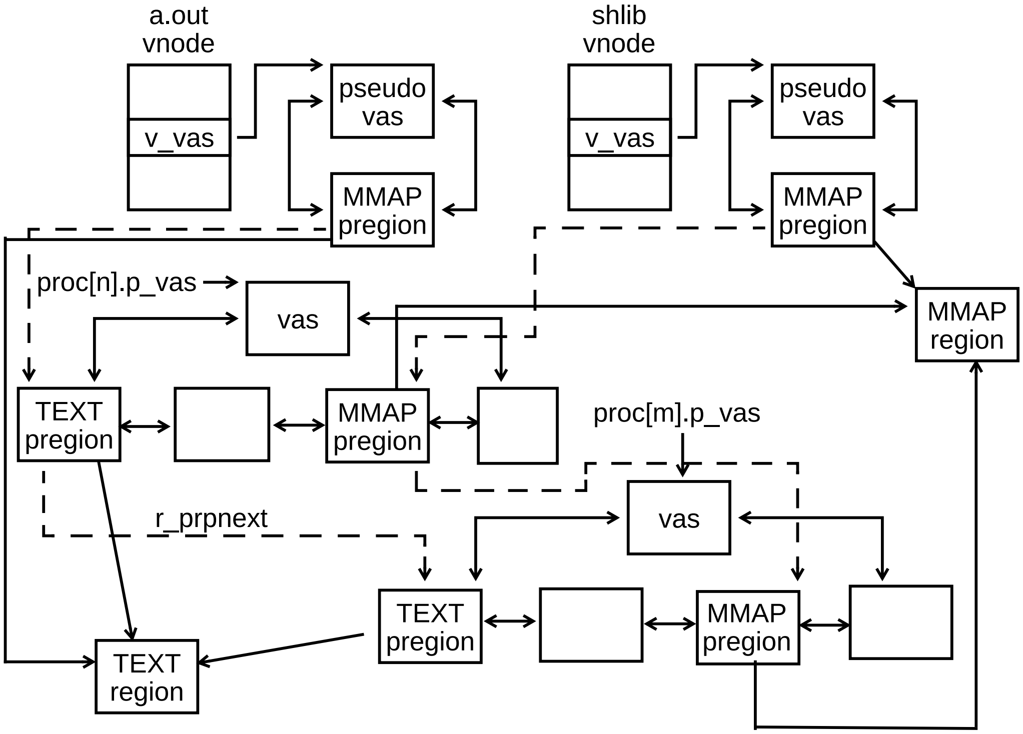
When a file is opened as an a.out or shared library, the easiest way to keep track of the region is to create a pseudo-vas the first time the file is opened as an executable. This is done by calling mapvnode() and storing the vas pointer in the vnode's v_vas element. On subsequent opens of the file as an executable, the non-NULL value in v_vas aids in finding the region to which the virtual address space is being attached.

The pseudo-vas is type PT_MMAP, and the associated pregion has PF_PSEUDO set in p_flags. This pregion is attached to the region for this vnode. All the processes that use this executable or shared library (non-pseudo pregions) then attach to the region with type PT_TEXT (a.out) or PT_MMAP (shared library). The number of processes using a particular vnode as an executable is kept in the pseudo-vas in va_refcnt.

All pregions associated with a region are connected with a doubly-linked list that begins with the region element r_pregs, and is defined in the pregions by p_prpnext and p_prpprev. The list is sorted by p_off, the pregion's offset into the region, and is NULL-terminated.

Even after all processes using the a.out or shared library exit, the handle to the region remains; its pages can be disposed of at that time.

Figure 1-15 Mapping the pseudo-vas structures



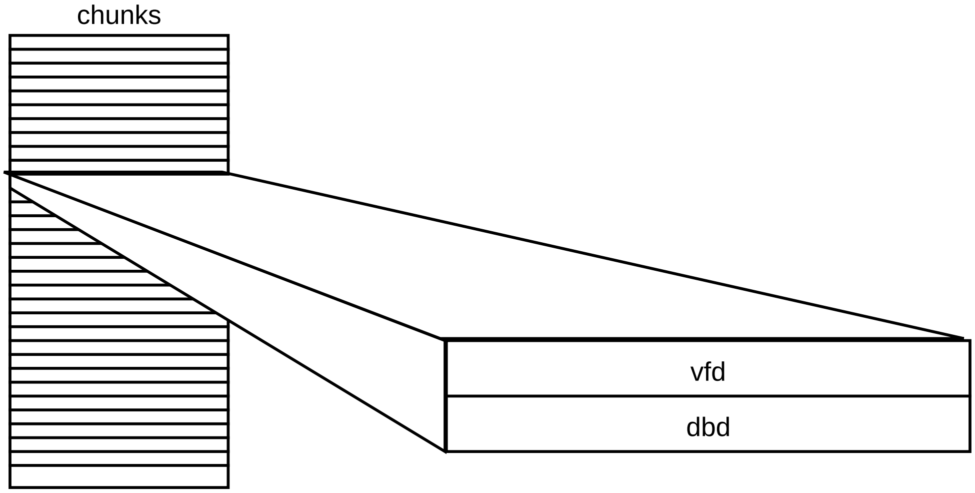
Chunks -- Keeping the vfds and dbds together in one place

Since information is typically needed about groups of (rather than individual) pages, pages are grouped into chunks. A chunk contains 32 pairs of virtual frame descriptors and disk block desciptors:

- The kernel looks for a page in memory by its virtual frame descriptor (vfd).
- The kernel looks for a page on disk by its disk block descriptor (dbd).
- By definition, if the vfd's pg_v bit is set, the vfd is used; if not, the dbd is used.

A one-to-one correspondence is maintained between vfd and dbd through the vfddbd structure, which simply contains one vfd (c_vfd) and one dbd (c_dbd).

Figure 1-16 A chunk contains 32 vfddbd (256 bytes)



HP-UX regions use chunks of vfds and dbds to keep track of page ownership:

- For assignment from virtual page to physical page if the page is valid. (This is required in addition to the PDIR. The term “assignment” is used (rather than mapping) because the page might not be translated but valid.
- Other virtual attributes of the page (such as whether the page is locked in memory, or whether it is valid).
- Location on disk for front-store and back-store pages.

Virtual Frame Descriptors (vfd)

A one-word structure called a virtual frame descriptor enables processes to reference pages of memory. The vfd is used when the process is in memory, and can be used to refer to the page of memory described in pfdat.

Figure 1-17 Virtual frame descriptor (vfd) contents

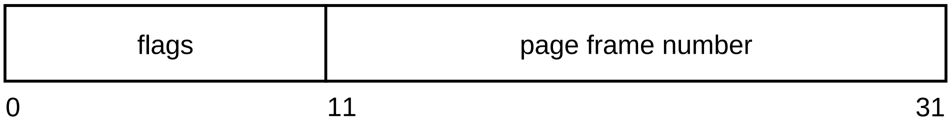


Table 1-12 Virtual Frame Descriptor (struct vfd)

Element	Meaning
pg_v	Valid flag. If set, this page of memory contains valid data and pg_pfnum is valid. If not set, the page's valid data is on a swap device.
pg_cw	Copy-on-write flag. If set, a write to the page causes a data protection fault, at which time the system copies the page.
pg_lock	Lock flag. If set, raw I/O is occurring on this page. Either the data is being transferred between the page and the disk, or data is being transferred between two memory pages. The kernel sleeps waiting for completion of I/O before launching further raw I/O to or from this page. Nothing can read the page while it is being written to disk.
pg_mlock	If set, the page is locked in memory and cannot be paged out.
pg_pfnum (aliased as pg_pfn)	Page frame number, from which can be accessed the correct pfdat entry for this page.

Disk Block Descriptor (dbd)

When the pg_v bit in a vfd is not set, the vfd is invalid and the page of data is not in memory but on disk. In this case, the disk block descriptor (dbd) gives valid reference to the data. Like the vfd structure, the dbd is one word long.

Figure 1-18 Contents of disk block descriptor (dbd)

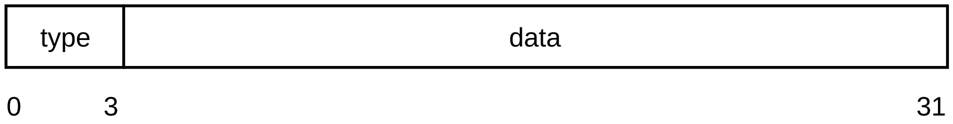


Table 1-13 Disk Block Descriptor (`struct dbd`)

Element	Meaning
`dbd_type`	One of six three-bit flags used to interpret `dbd_data`: • `DBD_NONE`: No copy of this data exists on disk. • `DBD_FSTORE, DBD_BSTORE`: Page can be found on a “front or back store” device, pointed to by a region’s vnode. [a] • `DBD_DFILL`: This is a demand-fill page. No space is allocated; when a fault occurs it is initialized by filling it with data from disk. • `DBD_DZERO`: This is a demand zero page; when requested, allocate a new page and initialize it with zeroes. • `DBD_HOLE`: Used for a sparse memory-mapped file; when read, the page gives zeros. When written to, a page is allocated, initialized to zero, data inserted, at which time the `dbd` type changes to `DBD_NONE`.
`dbd_data`	`vnode` type (`nfs, ufs`) specific data. A pointer points to data in a file pointed to by a `vnode`.

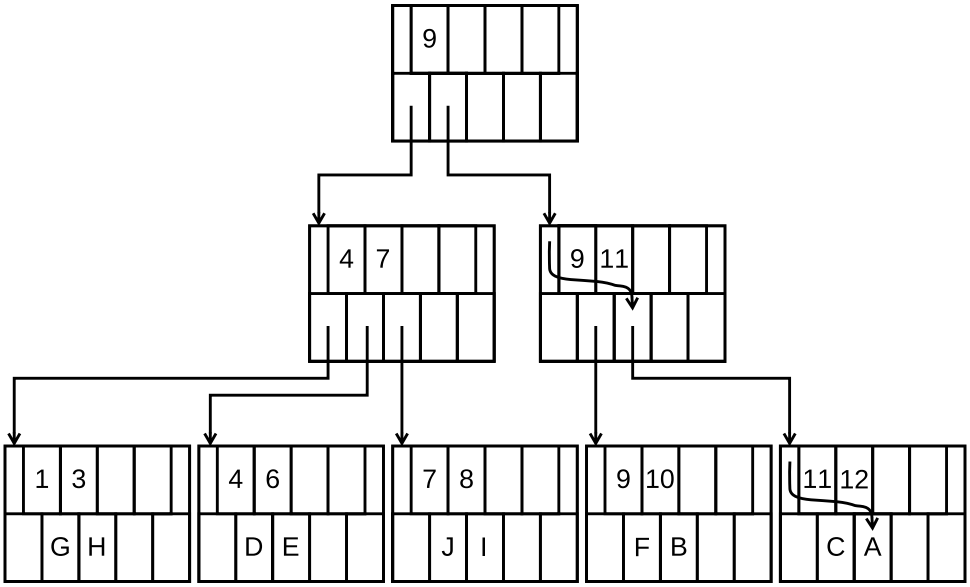
[a] When the `dbd_type` is `DBD_FSTORE`, it means that the page of data resides in the file pointed to by `v_fstore` (typically a file system). When the `dbd_type` is `DBD_BSTORE`, the page of data resides in the file or device file pointed to by `r_bstore` (typically a swap device).

Balanced Trees (B-Trees)

Each region contains either a single array of `vfd/dbd` (chunk) or a pointer to a `B-tree`. The structure called a `B-tree` allows for quick searches and efficient storage of sparse data. A `bnode` is the same size as a chunk; both can be gotten from the same source of memory. The region's B-tree stores pairs of page indices and chunk addresses. HP-UX uses an order 29 `B-tree`.

A `B-tree` is searched with a key and yields a value. In the region `B-tree`, the key is the page number in the region divided by 32, the number of `vfddbds` in a chunk.

Figure 1-19 A sample `B-tree` (order = 3, depth = 3)



Each node of a `B-tree` contains room for order+1 keys (or index numbers) and order+2 values. If a node grows to contain more than order keys, it is split into two nodes; half of the pairs are kept in the original node and the other half are copied to the new node. The `B-tree` node data also includes the number of valid elements contained in that node.

Table 1-14 **B-tree Node Description (struct bnode)**

Element	Meaning
b_key[B_SIZE]	The array of keys used for each page index of the bnode.
b_nelem	Number of valid keys/values in the bnode.
b_down[B_SIZE+1]	The array of values in the bnode, either pointers to another bnode (if this is an interior bnode) or pointers to chunks (if this is a leaf bnode).
b_scr1, b_scr2	bnode padding to the size of a chunk, to allow bnodes and chunks to be allocated from the same pool of memory.

Root of the B-tree

A structure called struct broot points to the start of the B-tree.

Table 1-15 **Struct broot**

Element	Meaning
b_root	Pointer to the initial point of the B-tree.
b_depth	Number of levels in the B-tree
b_npages	Pages used to construct the B-tree, counting both pages used for chunks and bnodes.
b_rpages	Number of real pages in the region; swap pages reserved for the B-tree by the kernel, using the routine vfdpgs(). Amount of swap allocated for the vfd/dbd pairs in the B-tree structure.
b_list	Pointer to a linked list of memory pages from which new bnodes or chunks can be added to the B-tree.
b_nfrag	Number of the next chunk available, derived from the unused 256-byte fragments in b_list.
b_rp	Pointer to the region using the B-tree.

Element	Meaning

`b_protoidx,`
`b_proto1,`
`b_proto2`
	Stores page index of default `dbd` and prototype to minimize time and memory costs to allocate chunk space.

`b_vproto`	List of page ranges whose bits are marked copy on write.

`b_key_cache`
`,`
`b_val_cache`
	Caches of most recently used keys and pointers to chunks associated with the keys; checked first when querying the virtual memory subsystem.

`vfd/dbd` prototypes

The `struct vfdcw` governs the `vfd` prototype.

Table 1-16 **`vfd` prototype (`struct vfdcw`)**

Element	Meaning
`v_start[MAXVPROTO]`	Page that indexes start of copy-on-write range; set to -1 if unused.
`v_end[MAXVPROTO]`	End of copy-on-write range

Hardware-Independent Page Information table (pfdat)

The hardware independent layer of the virtual memory subsystem manages pages in memory, pages written to swap devices, and the movement of pages from one to the other. The act of moving data from physical memory to a swap device, or moving data from a swap device to physical memory, is called paging.

Basic to hardware independence is the page frame data table (`pfdat`), a big array indexed directly through the page number. Each page of available memory is represented by one pfdat structure; one `pfdat` entry represents each page frame writable to a swap device. HP-UX never pages kernel memory (the pages containing kernel text, stack, and data); thus, `pfdat` manages only the subset representing freely available physical memory. When the `pfdat` is initialized, all free pages are linked in a list pointed to by `phead`.

Table 1-17 Principal entries in struct pfdat (page frame data)

Element	Meaning
pf_hchain	Hash chain link.
pf_flags	Page frame data flags (shown in the next table).
pf_pfn	Physical page frame number.
pf_use	Number of regions sharing the page; when pf_use drops to zero, the page can be placed on the free linked list.
pf_devvp[a]	vnode for swap device.
pf_data	Disk block number on swap device.
pf_next, pf_prev	Next and previous free pfdat entries.
pf_cache_waiting	If set, this element means that a thread is waiting to grab the pf_lock on that page. Required for synchronization.
pf_lock	Lock pfdat entry (beta semaphore), used to lock the page while modifying the pde (physical-to-virtual translation, access rights, or protection ID)
pf_hdl	Hardware dependent layer elements (see hdl_pfdat discussion, shortly).

[a] Hashing is done on the tuple (pf_devvp, pf_data).

Flags showing the Status of the Page

Table 1-18 Principal pf_flag values

Flag	Meaning
P_QUEUE	Page is on the free queue, headed by phead.
P_BAD	Page is marked as bad by the memory deallocation subsystem.
P_HASH	Page is on a hash queue; contains head of queue.

Flag	Meaning
P_ALLOCATING	Page is being allocated; prevents another process from taking the page while it is being remapped.
P_SYS	Page is being used by the kernel rather than by a user process. Pages marked with this flag include dynamic buffer cache pages, B-tree pages and the results of kernel memory allocation. They are used by the kernel for critical data structures in addition to the kernel static pages that were not included in pfdat.
P_DMEM	Page is locked by the memory diagnostics subsystem; set and cleared with an ioctl() call to the dmem driver.
P_LCOW	Page is being remapped by copy-on-write.
P_UAREA	Page is used by a pregion of type PT_UAREA.

Hardware-Dependent Layer page frame data entry

If pf_hdl is referenced in struct pfdat, the struct hdlpfdat (defined in hdl_pfdat.h) is used. pf_hdl is a type of struct hdlpfdat.

Table 1-19 **struct hdlpfdat**

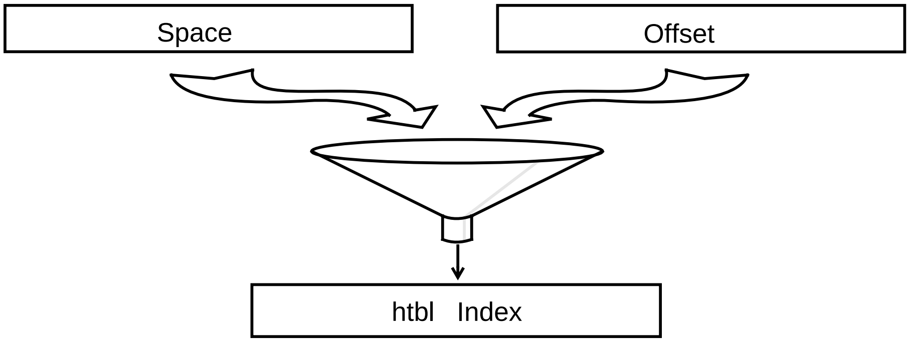
Element	Meaning
hdlpf_flags	Flags that show the HDL status of the page: • HDLPF_TRANS: A virtual address translation exists for this page. • HDLPF_PROTECT: Page is protected from user access. This flag indicates that the saved values are valid. • HDLPF_STEAL: Virtual translation should be removed when pending I/O is complete. • HDLPF_MOD: Analogous to changing the pde_modified flag in the pde. • HDLPF_REF: Analogous to changing the pde_ref flag in the pde. • HDLPF_READA: Read-ahead page in transit; used to indicate to the hdl_pfault() routine that it should start the next I/O request before waiting for the current I/O request to complete.
hdlpf_savear	Saved page access rights.
hdlpf_saveprot	Saved page protection ID.

MAPPING VIRTUAL TO PHYSICAL MEMORY

The PA-RISC hardware attempts to convert a virtual address to a physical address with the TLB or the block TLB. If it cannot resolve the address, it generates a page fault (interrupt type 6 for an instruction TLB miss fault; interrupt type 15 for a data TLB miss fault). The kernel must then handle this fault.

PA-RISC uses a hashed page table (HTBL) to pinpoint an address in the enormous virtual address space. A ratio is kept of PDIR to hash table entries, depending on specific PA-RISC implementation (see `cpu.h`). Control register 25 (CR25) contains the hash table address (see `reg.h`).

Figure 1-20 Contents of the `htbl` index



The HTBL

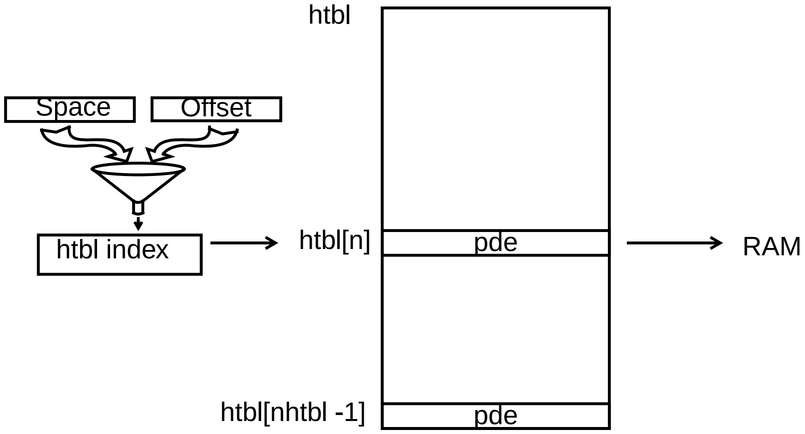
The algorithm for converting a virtual address to a physical address depends on the particular processor.

Likewise, the algorithm for choosing the size of HTBL has been developed empirically as the result of performance tests.

- Each graphics driver estimates how many entries it will need for I/O mapping. This number is stored in `niopdir`.
- The kernel first approximates `nhtbl` as the sum of `niopdir` and the number of pages of RAM (`physmem`).

- nhtbl is now adjusted to a power of two and rounded appropriately.

Figure 1-21 Mapping from the htbl entry to the page directory entry



The index generated by the hashing algorithm is now used as an index into HTBL. Each entry in the table is referred to as a pde (page directory entry), and is of type struct hpde.

The virtual space and offset are compared to information in the pde to verify the entry. The physical address is retrieved from the pde to complete the translation from virtual address to physical address.

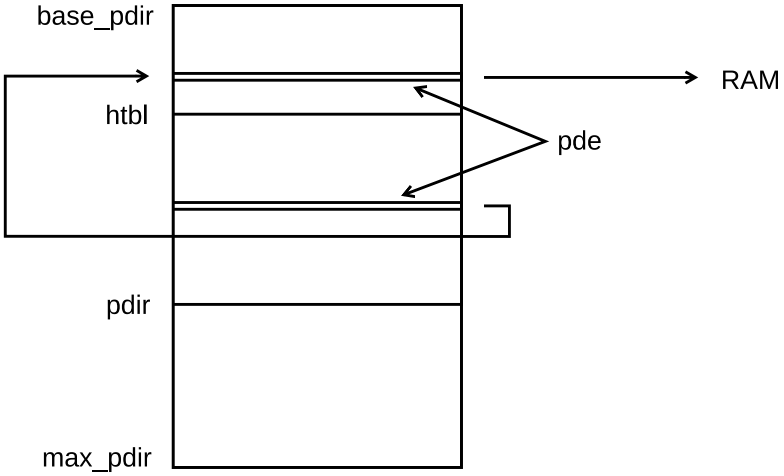
When multiple addresses hash to the same HTBL entry

As with any hash algorithm, multiple addresses can map to the same HTBL index. The entry in HTBL is actually the starting point for a linked list of pdes. Each entry has a pde_next pointer that points to another pde, or contains NULL if it is the last item of the linked list.

Each HTBL entry can point to two other collections of pdes, ranging from base_pdir to htbl and from pdir (which is also the end of HTBL) to max_pdir. The entirety of the HTBL and surrounding pdes is referred to collectively as the sparse PDIR. HTBL is always aligned to begin at an address that is a multiple of its size (that is, a multiple of nhtbl * sizeof(struct hpde)).

In practice, HTBL contains sufficient entries, as that the linked lists seldom grow beyond three links. pdir_free_list points to a linked list of sparse PDIR entries that are not being used and are available for use. pdir_free_list_tail points to the last pde on that linked list.

Figure 1-22 How multiple addresses hash to the same HTBL entry



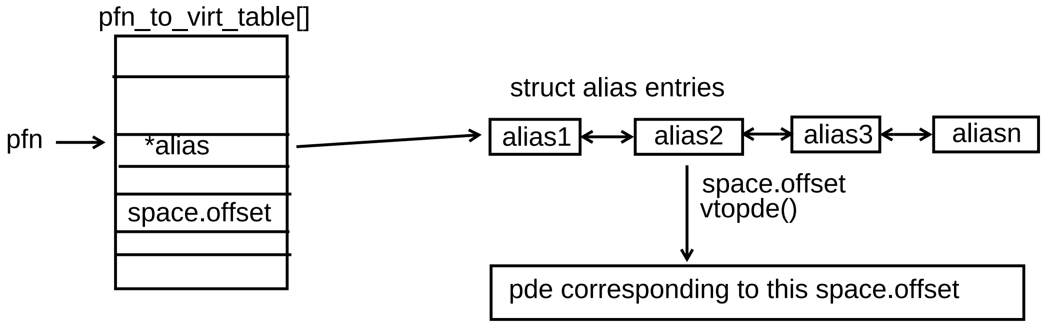
Mapping Physical to Virtual Addresses

HP-UX uses a hashed page directory to translate from virtual to physical address.

The pfdat table maps physical to virtual addresses. Inverse translations from physical to virtual use the pfn_to_virt_table[], an array that contains entries of either space and offset page (in the case of a single translation to a page) or a list of alias structures (when the physical page has more than one virtual address translation).

The hashed pdir stores the physical address in the translation “bucket” (hash table) to disassociate the physical page number from that page’s pde.

Figure 1-23 Physical-to-virtual address translation



The `pfn_to_virt_table` may contain the space.offset (virtual address) corresponding to a physical address or it may have a pointer to a link list of alias structures, each of which has a space.offset pair.

Address Aliasing

HP-UX supports software address aliasing for text only of `EXEC_MAGIC` executables. (Whereas the hardware implements address aliasing on 1MB boundaries, software address aliasing is implemented on a per-page basis; pages are 4KB apart.)

When a text segment is first translated, it has no alias. However, if a process or thread attaches to the same text segment, it may require another translation. Processes sharing text segments do not use aliases. Only processes with private text segments that share data pages using `copy-on-write` use aliases.

When multiple virtual addresses translate to the same physical address, HP-UX uses alias structures to keep track of them. Aliases for a page frame (`pfn`) are maintained via alias chains off the `pfn_to_virt_table[]`. When a `pfn_to_virt_table`'s space field is invalid and the offset field is non-zero, the non-zero value points to the beginning of a linked list of alias structures. Each alias structure contains the space and offset of the alias, and a temporary hold field for a pde's access rights and access ID. The `pfdat_lock` of the alias's `pfn` protects the alias chain from being read and modified.

To locate the `pde` for a particular alias space and offset, the space and offset are hashed for the `pde` chain and its corresponding `pdlock`. Once the `pdlock` is obtained, the `vtopde()` routine walks the `pde` hash chain to find a match of the tag.

The aa_entfreelist is the head of the doubly-linked list of free alias entries. The system gets an alias structure from aa_entfreelist, in which it stores the information for this new virtual-to-physical translation.

The global variable max_aapdir contains the total number of alias pdes on the system. Once a page is allocated for use as alias pdes, it is not returned, so the value of max_aapdir may grow over time but will never shrink.

The number of available alias pdes is stored in aa_pdircnt. When an alias pde is used or reserved (we reserve one if we include an HTBL pde in an alias linked list, in case we have to move it later), aa_pdircnt is decremented. When an alias pde is returned to aa_pdirfreelist or unreserved, aa_pdircnt is incremented.

The number of available alias structures is kept in aa_entcnt. Once a page is allocated for use as a group of alias structures, it is not returned. We do not keep track of the total number of alias structures on the system, just the number of available structures.

MAINTAINING PAGE AVAILABILITY

Two computational elements maintain page availability:

- Paging thresholds trigger the gamut of paging events.
- The vhand and sched daemons (system processes) handle the actual paging and deactivation.

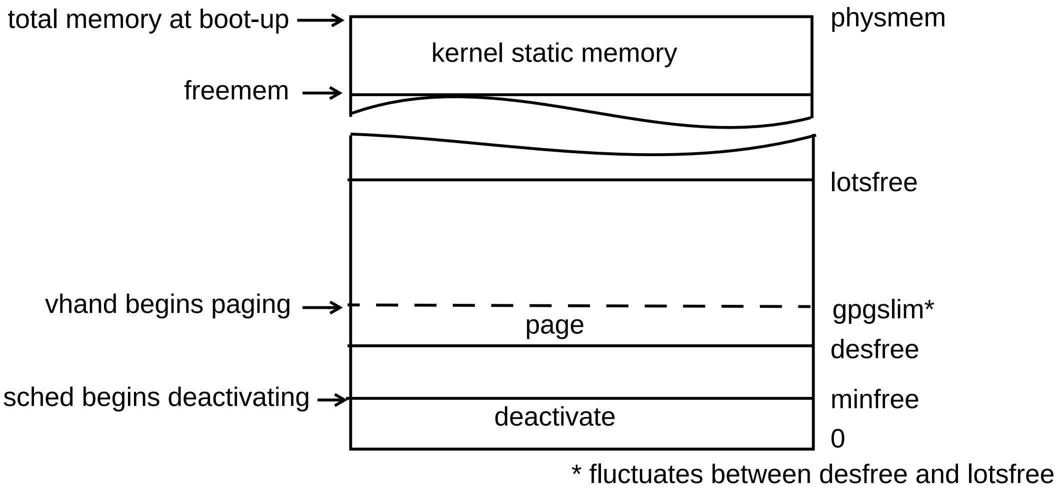
vhand monitors free pages to keep their number above a threshold and ensure sufficient memory for demand paging. vhand governs the overall state of the paging system. sched becomes operative when the number of pages available in memory diminishes below a certain level. vhand and sched will be described in the context of their work shortly.

NOTE The sched process is known colloquially as the swapper.

Paging Thresholds

Memory management uses paging thresholds that trigger various paging activities. The figure shows the full range of available memory and indicates what paging activity occurs when memory level falls below each paging threshold.

Figure 1-24 **Available memory in the system**



The value termed freemem represents the total number of free pages in the phead linked list, which includes all memory available in a system after kernel initialization.

Three tunable paging thresholds are initialized by the setmemthresholds() routine.

Table 1-20 setmemthresholds() paging thresholds

Paging threshold	Meaning
lotsfree	Plenty of free memory, specified in pages. The upper bound from which the paging daemon will begins to steal pages.
desfree	Amount of memory desired free, specified in pages. This is the lower bound at which the paging daemon begins stealing pages.
minfree	The minimal amount of free memory tolerable, specified in pages. If free memory drops below this boundary, sched() recognizes the system is desperate for memory and deactivates entire processes whether they are runnable or not.

The gpgslim Paging Threshold

The gpgslim paging threshold is the point at which vhand starts paging. gpgslim adjusts dynamically according to the needs of the system. It oscillates between an upper bound called lotsfree and a lower bound called desfree. Both lotsfree and desfree are calculated when the system boots up and are based on the size of system memory.

When the system boots, gpgslim is set to 1/4 the distance between lotsfree and desfree (desfree + (lotsfree - desfree)/4). As the system runs, this value fluctuates between desfree and lotsfree. When the sum of available memory and the number of pages scheduled for I/O (soon to be freed) falls below gpgslim, vhand() begins aging and stealing little-used pages in an attempt to increase the available memory above this threshold.

The system wants to keep memory at gpgslim. If the system is not stressed, gpgslim starts rising, because it does not need to have a lot more pages freed. As memory becomes more scarce, the system tries to maintain the pool of free memory, causing gpgslim to fall. If gpgslim decreases to minfree, the system starts to deactivate entire processes.

How Memory Thresholds are Tuned

Performance testing has shown that memory usage differs for a server versus a workstation. Workstations typically run a few large applications whereas servers typically run many applications of varying size. Consequently, the paging and deactivation thresholds on workstations are a smaller fraction of memory than on the servers. In a typical workstation environment, applications start up requiring a large number of pages, which eventually reduce to a smaller working set of pages. By allowing applications to claim more memory before paging or deactivating, the working set is more likely to stay in memory.

Paging and activation algorithms take these and other differences into account. Depending on the physical memory size of the system, the paging thresholds are initialized to either a “small memory” or “large memory” set of values.

Small Memory Thresholds

For small memory systems (that is, systems with 32MB or less of freemem), the paging thresholds are set to a smaller fraction of total memory to allow applications to utilize more memory before the system begins paging and deactivating. The paging thresholds are set as follows:

Table 1-21 Small-memory paging thresholds

Threshold	Limit	Not to exceed
lotsfree	1/8 freemem	1MB
desfree	1/16 freemem	240 KB
minfree	1/2 desfree	100 KB

Large Memory Thresholds

For large memory systems (that is, systems with greater than 32 MB of freemem), the paging thresholds are set to a larger fraction of memory to allow vhand() to start paging earlier so that it can efficiently walk a (potentially) longer active pregions list. This also helps sched() process a potentially longer active process list by starting process deactivation earlier. The paging thresholds are set as follows:

Table 1-22 Large-memory paging thresholds

Threshold	Limit	Capacity if freemem < 2 GB	Capacity if freemem > 2 GB
lotsfree	1/16 freemem	32 MB	64 MB
desfree	1/64 freemem	4 MB	12 MB
minfree	1/4 desfree	1 MB	5 MB

These settings result in a linear increase of the paging thresholds up to a certain memory size, after which the thresholds remain fixed. For example, lotsfree increases linearly and reaches its maximum value of 32 MB when freemem is 512 MB. For memory sizes beyond 512 MB, lotsfree remains fixed at 32 MB. This results in the system paging earlier for smaller memory configurations and later for larger sizes.

When physical memory sizes exceed 2 GB, all the paging thresholds are increased to a larger set of fixed values.

How Paging is Triggered

The rate schedpaging() runs is termed vhandrunrate, a tunable parameter (set to run by default at eight times per second) activated when the sum of free memory and paroled memory (freemem + parolemem) is less than lotsfree.

vhand, the pageout daemon

Programmatically, vhand is awakened by schedpaging() periodically to maintain recently referenced pages and to move pages out when memory is tight. vhand operates on the basis of vhandargs_t, which consists of a pointer to the target pregion, a count of the physical pages visited, and a nice value for preferential aging.

vhand can also be awakened by allocpfd2() (in vm_page.c), a routine that allocates a single page of memory.

If all the pages on the free memory list (phead) are locked, or the routine has been called while using the interrupt control stack (ICS) and all pages on the free list are also in the page cache (phash), allocpfd2() cannot get any pages.

If on the ICS without any available pages, allocpfd2() wakes the page daemon. Regardless of which stack the system is running on, allocpfd2() then wakes up unhashdaemon, which removes pages from the page cache.

If on the ICS, allocpfd2() returns NULL; if not on the ICS, allocpfd2() sleeps waiting for a page to become available, and then retry.

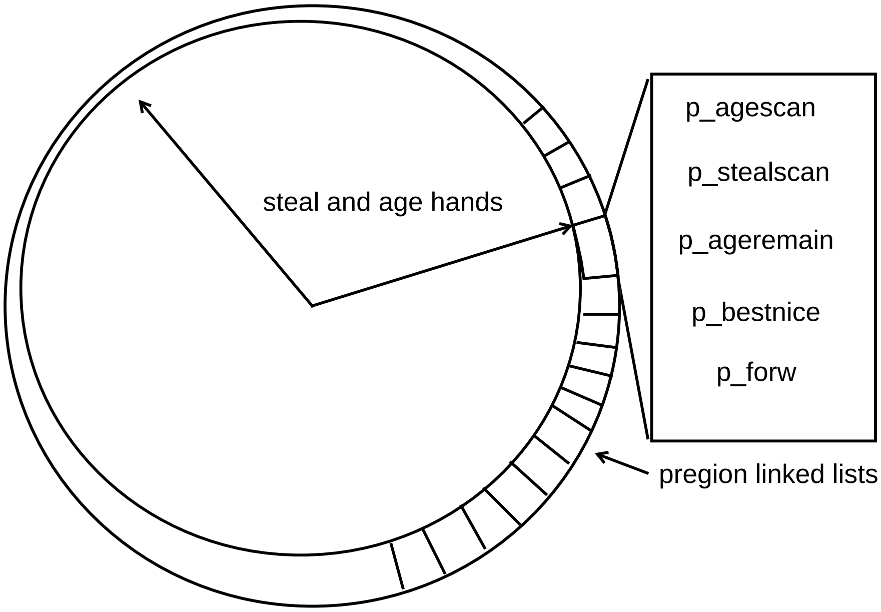
Two-Handed Clock Algorithm

A doubly linked list of pregions, termed the active pregion list, is used by vhand to examine memory availability. Conceptually, the pregions can be visualized as being linked in a circle, in the center of which are two clock-like hands. The two hands function as a steal hand and an age hand.

- A steal hand removes pages whose reference bits remain clear since the most recent pass of the age hand.
- An age hand clears reference bits on in-core pages in an active pregion.

The kernel automatically keeps an appropriate distance between the hands, based on the available paging bandwidth, the number of pages that need to be stolen, the number of pages already scheduled to be freed, and the frequency by which vhand runs.

Figure 1-25 **Two-handed vhand clock algorithm, showing also the factors that affect vhand**



The two hands cycle through the active `pregion` linked lists of physical memory to look for memory pages that have not been referenced recently and move them to secondary storage - the swap space. Pages that have not been referenced from the time the age hand passes to the time the steal hand passes are pushed out of memory. The hands rotate at a variable rate determined by the demand for memory.

The vhand daemon decides when to start paging by determining how much free memory is available. Once free memory drops below the `gpgslim` threshold, paging occurs. vhand attempts to free enough pages to bring the supply of memory back up to `gpgslim`. Between `gpgslim` and `lotsfree`, the page daemon continues to age pages (that is, clear their reference bits) but no longer steals pages.

Factors Affecting vhand

vhand responds to various workloads, transient situations, and memory configurations. When aging and stealing from regions, vhand

- ages some constant fraction of each `pregion`.

- uses the pregion field p_agescan to track the last age hand location.
- uses the pregion field p_ageremain to track remaining pages to be aged.
- uses the pregion field p_stealscan to track the last steal hand location.
- pushes vfd/dbd pairs to swap if they have no valid pages.

When the age hand arrives at a region, it ages some constant fraction of pages before moving to the next region (by default 1/16 of the region's total pages). The p_agescan tag enables the age hand to move to the location within a pregion where it left off during its previous pass, while the p_ageremain charts how many pages must be aged to fill the 1/16 quota before moving on to the next pregion.

The steal hand uses the pregion field p_stealscan to locate itself within a pregion and resume taking pages that have not been referenced since last aged. If no valid page remain, vhand pushes out of memory the vfd/dbd pairs associated with the region.

How much to age and steal depends on several factors:

- frequency of vhand runs (by default eight times per second).
- available paging bandwidth (based on comparison with a global rate of pageouts completed within an interval of time).
- how often the system falls to zero free memory.
- position of the paging threshold gpgslim.
- number of pages already scheduled to be freed.

vhand is biased against threads that have nice priorities: the nicer a thread, the more likely vhand will steal its pages. The pregion field p_bestnice reflects the best (numerically, the smallest value) nice value of all threads sharing a region.

What Happens when vhand Wakes Up

Refer to the table that follows for explanations of the vhand variables.

- vhand establishes pagecounts for pages to age and pages to steal, and sets the coalescecnt to zero.

- vhand uses the SCRITICAL flag to get access to the system critical memory pool. (The SCRITICAL flag for the vhand process is set when the process starts running for the first time.)
- vhand increments the value coalescecnt and compares it to the value coalescerate. If coalescecnt is higher, vhand attempts to remove pages from kernel allocation buckets until freemem is above lotsfree. Then vhand resets coalescecnt to zero.
- Next vhand updates the value of gpgslim, based on value of memzeroperiod.
- vhand updates pageoutrate, using pageoutcnt.
- vhand updates targetlaps, the number of desired laps between the age and steal hands. If less CPU cycles are being used than the value of targetcpu, vhand increases the value of targetlaps (up to a maximum of 15); if more CPU cycles are being used than targetcpu, targetlaps is decreased.
- vhand updates agerate, the number of pages to age per second.
- If vhandinfoticks is non-zero, diagnostic information prints to the console.

NOTE None of the variables in the table that follows may be tuned.

Table 1-23 Variables affecting vhand

Variable	Purpose
`coalescerate`	How often `vhand()` attempts to reclaim unused memory from the kernel allocation buckets, beginning at 128; that is, every 128th time `vhand` runs, it attempts to return memory to the system. • If successful, `vhand` resets `coalescerate` to every 128th time. • If unsuccessful, `vhand` multiplies `coalescerate` by two (checks memory half as often) up to every 512th time.
`memzeroperiod`	Minimum time period (default=3 seconds) permissible for `freemem` to reach zero events; determines how often `gpgslim` is adjusted when `vhand()` is running. • `gpgslim` is incremented if `freemem` does not reach zero twice within `memzeroperiod.` • `gpgslim` is decremented if `freemem` reaches zero twice within `memzeroperiod` slightly above `lotsfree`.
`pageoutrate`	Current pageout rate, calculated empirically from number of pageouts completed.
`pageoutcnt`	Recent count of pageouts completed
`targetlaps`	Ideal gap between steal and age hands for `handlaps`; adapts at run time. During normal operation, the hands should be as far apart as possible to give processes maximum time to reset a cleared reference bit being used by a page. `targetlaps` is defined in the kernel as a static variable; it does not appear in the symbol table.

Variable	Purpose
`targetcpu`	Maximum percentage of CPU vhand should spend paging. (default value is 10%.)
`handlaps`	Actual number of laps between the age and steal hands.
`agerate`	Number of pages the age hand visits to age per second; adapts continually to system load. These are defined in the kernel as static variables (meaning they do not appear in the symbol table).
`stealrate`	How many pages the steal hand visits per second; adapts continually to system load. These are defined in the kernel as static variables (meaning they do not appear in the symbol table).

vhand Steals and Ages Pages

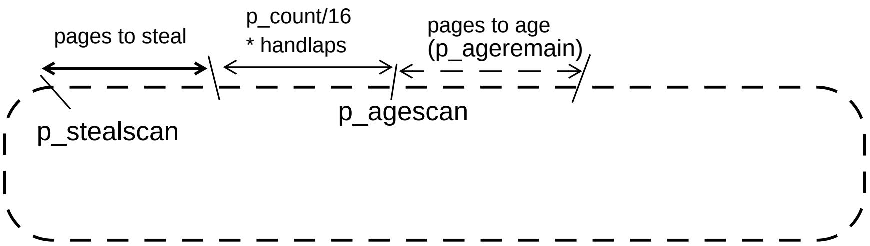
Once vhand establishes its criteria, it proceeds to traverse the linked list of pregions. Continuing in the clock-hands analogy, vhand is ready to move its hands.

- vhand determines how many pages and what pages are available to steal.
- Next, vhand moves the age hand to clear the reference bit from a selected number of pages.
  - If the steal hand is pointing to `bufcache_preg`, vhand steals buffers from the buffer cache with the `stealbuffers()` routine. The global parameter dbc_steal_factor determines how much more aggressively to steal buffer cache pages than `pregion` pages. If `dbc_steal_factor` has a value of 16, buffer cache pages are treated no differently than `pregion` pages; the default value of 48 means that buffer cache pages are stolen three times as aggressively as `pregion` pages.

- If the steal hand points to a pregion whose region has no valid pages (that is, r_nvalid == 0), vhand pushes its B-tree out to the swap device. If none of the processes using the region are loaded in memory (that is, r_incore == 0), the entire region may be swapped out.
- Otherwise, vhand steals all pages between p_stealhand and (p_agescan - p_count/16 * handlaps), up to the steal quota (calculated from stealrate).
- vhand updates p_stealscan to the page number following the last stolen page of the affected pregion.
- If vhand has not stolen as many pages as permissible (calculated from stealrate), it moves to the next pregion and repeats the process until it satisfies the system's demand.
- If the age hand points to bufcache_preg, vhand ages one sixteenth of the pages in the buffer cache with the agebuffers() routine.
- vhand determines the best nice value (that is, the lowest number) of all the pregions using the region. For each page in the region, if the nice value exceeds a randomly generated number, vhand does not age the page.
- Otherwise, vhand ages all pages between p_agehand and (p_agehand + p_ageremain) by clearing the pde_ref bit and purging the TLB.
- Finally, vhand updates p_agehand to be the page number after the last aged page in the affected pregion.

Note, the steal hand is moved first to keep it behind the age hand and prevent aging and stealing a page in the same cycle.

Figure 1-26 Ranges within which pregion pages are aged and stolen



The sched() routine

The sched() routine (colloquially termed “the swapper”) handles the deactivation and reactivation of processes when free memory falls below minfree, or when the system appears to be thrashing.

NOTE Deactivation occurs on a per-thread basis. sched() chooses to deactivate on a process level and then deactivates each thread.

Deactivation occurs when sched() determines the system:

- is low on memory; that is, if freemem falls below the deactivation threshold minfree and more than one process is running.
- appears to be thrashing; that is, if the system has a high paging rate and low CPU usage.

Reactivation occurs when the system is no longer low on memory or thrashing.

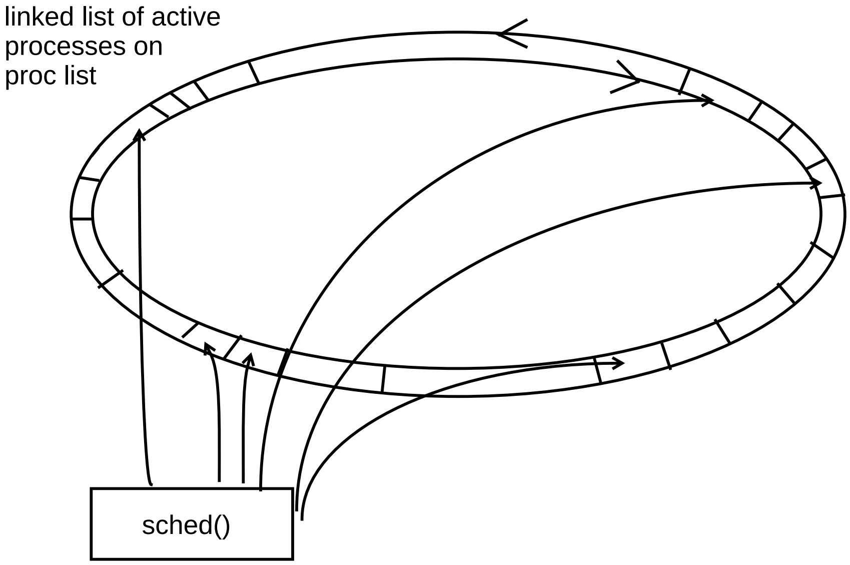
What to Deactivate or Reactivate

Deactivation and reactivation are determined by:

- process priority; the lower the process priority (meaning the higher the nice value), the more likely it will be deactivated. The higher the process priority, the more likely it will be reactivated. Real-time processes are ineligible for deactivation.
- process size; the larger the process resident set size, the more likely it will be deactivated.
- process state; a process that has been sleeping or has been in memory for some time is likely to be deactivated. A process deactivated for a while and is now ready to run is likely to be reactivated.
- process type. A batch process (one that works continuously) or one marked for serialization is more likely than an interactive process (one that works in spurts) to be deactivated. Interactive processes are more likely to be reactivated than batch or serialized processes.
- time in current state

The swapper deactivates processes and prevents them from running, thus reducing the rate at which new pages are accessed. Once swapper detects that available memory has risen above `minfree` and the system is not thrashing, the swapper reactivates the deactivated processes and continues monitoring memory availability.

Figure 1-27 **`sched()` chooses processes to deactivate based on size, nice priority, and how long it has been running.**



`sched()` walks the chain of active processes, examining each, and deciding the best candidate to be deactivated based on size, nice priority, and how long it has been running.

Programmatically, `sched()` deactivates and reactivates processes.

If the system appears to be thrashing or experiencing memory pressure, the sched routine walks through the active process list calculating each process's deactivation priority based on type, state, size, length of time in memory, and how long it has been sleeping. (Batch and processes marked for serialization by the `serialize()` command are more likely to be deactivated than interactive processes.) The best candidate is then marked for deactivation.

If the system is not thrashing or experiencing memory pressure, the `sched` routine walks through the active proc list calculating each deactivated process' reactivation priority based on how long it has been deactivated, its size, state, and type. Batch processes and those marked by the `serialize()` command are less likely to be reactivated than is an interactive process. Once the most deserving process has been determined, it is reactivated.

When a process is deactivated

Once a process and its pregions are marked for deactivation, `sched()`

- removes the process from the run queue.
- adds its `uarea` to the active `pregion` list so that `vhand` can page it out.
- moves all the `pregions` associated with the target process in front of the steal hand, so that `vhand` can steal from them immediately.
- enables `vhand` to scan and steal pages from the entire `pregion`, instead of 1/16.

Eventually, `vhand` pushes the deactivated process's pages to secondary storage.

When a process is reactivated

Processes stay deactivated until the system has freed up enough memory and the paging rate has slowed sufficiently to return processes to the run queue. The process with the highest reactivation priority is then returned to the run queue.

Once a process and its pregions are marked for reactivation, `sched()`:

- removes the process's `uarea` from the active `pregion` list.
- clears all deactivation flags.
- brings in the `vfd/dbd` pairs.
- faults in the `uarea`.
- adds the process to the run queue.

Self-Deactivation

Earlier HP-UX implementations did not permit a process to be swapped out if it was holding a lock, doing I/O, or was not at a signalable priority. Even if priority made it most likely to be deactivated, `vhand` bypassed the process.

Now, if the most deserving process cannot be deactivated immediately, it is marked for self-deactivation; that is, the process sets a self-deactivation flag. The next time the process must fault in a page, it deactivates itself.

NOTE `sched()` deactivates and reactivates processes. As part of a process's deactivation or reactivation, all its threads get deactivated or reactivated. `sched()` does not deactivate or reactivate threads individually.

Thrashing

Thrashing is defined as low CPU usage with high paging rate. Thrashing might occur when several processes are running, several processes are waiting for I/O to complete, or active processes have been marked for serialization.

On systems with very demanding memory needs (for example, systems that run many large processes), the paging daemons can become so busy deactivating/reactivating, and swapping pages in and out that the system spends too much time paging and not enough time running processes.

When this happens, system performance degrades rapidly, sometimes to such a degree that nothing seems to be happening. At this point, the system is said to be thrashing, because it is doing more overhead than productive work.

If your working set is larger than physical memory, the system will thrash. To solve the problem,

- reduce the working set of running processes by deactivation, or
- increase the size of physical memory.

If you are left with one huge process constrained with physical memory and the system still thrashes, you will need to rewrite the application so that it uses fewer pages simultaneously, by grouping data structures according to access, for example.

Serialization

All processes marked by the serialize command are run serially. This functionality unjams the bottleneck (recognizable by process throughput degradation) caused by groups of large processes contending for the CPU. By running large processes one at a time, the system can make more efficient use of the CPU as well as system memory since each process does not end up constantly faulting in its working set, only to have the pages stolen when another process starts running.

As long as there is enough memory in the system, processes marked by `serialize()` behave no differently than other processes in the system. However, once memory becomes tight, processes marked by serialize are run one at a time in priority order. Each process runs for a finite interval of time before another serialized process may run. The user cannot enforce an execution order on serialized processes.

`serialize()` can be run from the command line or with a `PID` value. `serialize()` also has a timeshare option that returns the `PID` specified to normal timeshare scheduling algorithms.

If serialization is insufficient to eliminate thrashing, you will need to add more main memory to the system.

SWAP SPACE MANAGEMENT

Swap space is an area on a high-speed storage device (almost always a disk drive), reserved for use by the virtual memory system for deactivation and paging processes. At least one swap device (primary swap) must be present on the system.

During system startup, the location (disk block number) and size of each swap device is displayed in 512-KB blocks. The swapper reserves swap space at process creation time, but does not allocate swap space from the disk until pages need to go out to disk. Reserving swap at process creation protects the swapper from running out of swap space. You can add or remove swap as needed (that is, dynamically) while the system is running, without having to regenerate the kernel.

HP-UX uses both physical and pseudo swap to enable efficient execution of programs.

Pseudo-Swap Space

System memory used for swap space is called pseudo-swap space. It allows users to execute processes in memory without allocating physical swap. Pseudo-swap is controlled by an operating-system parameter; by default, `swapmem_on` is set to 1, enabling pseudo-swap.

Typically, when the system executes a process, swap space is reserved for the entire process, in case it must be paged out. According to this model, to run one gigabyte of processes, the system would have to have one gigabyte of configured swap space. Although this protects the system from running out of swap space, disk space reserved for swap is under-utilized if minimal or no swapping occurs.

To avoid such waste of resources, HP-UX is configured to access up to three-quarters of system memory capacity as pseudo-swap. This means that system memory serves two functions: as process-execution space and as swap space. By using pseudo-swap space, a one-gigabyte memory system with one-gigabyte of swap can run up to 1.75 GB of processes. As before, if a process attempts to grow or be created beyond this extended threshold, it will fail.

When using pseudo swap for swap, the pages are locked; as the amount of pseudo-swap increases, the amount of lockable memory decreases.

For factory-floor systems (such as controllers), which perform best when the entire application is resident in memory, pseudo-swap space can be used to enhance performance: you can either lock the application in memory or make sure the total number of processes created does not exceed three-quarters of system memory.

Pseudo-swap space is set to a maximum of three-quarters of system memory because the system can begin paging once three-quarters of system available memory has been used. The unused quarter of memory allows a buffer between the system and the swapper to give the system computational flexibility.

When the number of processes created approaches capacity, the system might exhibit thrashing and a decrease in system response time. If necessary, you can disable pseudo-swap space by setting the tunable parameter `swapmem_on` in `/usr/conf/master.d/core-hpux` to zero.

At the head of a doubly linked list of regions that have pseudo-swap allocated is a null terminated list called `pswaplist`.

Physical Swap Space

There are two kinds of physical swap space: device swap and file-system swap.

Device Swap Space

Device swap space resides in its own reserved area (an entire disk or logical volume of an LVM disk) and is faster than file-system swap because the system can write an entire request (256 KB) to a device at once.

File-System Swap Space

File-system swap space is located on a mounted file system and can vary in size with the system's swapping activity. However, its throughput is slower than device swap, because free file-system blocks may not always be contiguous; therefore, separate read/write requests must be made for each file-system block.

To optimize system performance, file-system swap space is allocated and de-allocated in `swchunk`-sized chunks. `swchunk` is a configurable operating system parameter; its default is 2048 KB (2 MB). Once a chunk of file system space is no longer being used by the paging system, it is released for file system use, unless it has been preallocated with swapon.

If swapping to file-system swap space, each chunk of swap space is a file in the file system swap directory, and has a name constructed from the system name and the swaptab index (such as becky.6 for swaptab[6] on a system named becky).

Swap Space Parameters

Several configurable parameters deal with swapping.

Table 1-24 **Configurable swap-space parameters**

Parameter	Purpose
swchunk	The number of DEV_BSIZE blocks in a unit of swap space, by default, 2 MB on all systems.
maxswapchunks	Maximum number of swap chunks allowed on a system.
swapmem_on	Parameter allowing creation of more processes than you have physical swap space for, by using pseudo-swap.

Swap Space Global Variables

When the kernel is initialized, conf.c includes globals.h, which contains numerous characteristics related to swap space, shown in the next table. The most important to swap space reservation are swapspc_cnt, swapspc_max, swapmem_cnt, swapmem_max, and sys_mem

Table 1-25 **Swap-space characteristics in globals.h**

Element	Meaning
bswlist	head of free swap header list.
*pageoutbp	pointer to swbuf header used by pageout when swapping.
ref_hand	current reference hand used by pageout daemon.

Element	Meaning
maxmem	page count of actual max memory per process.
physmem	page count of physical memory on this CPU.
nswdev	number of swap devices.
nswap	page count of size of swap space.
*fswdevt	pointer to file system swap table.
*swaptab	pointer to the table of swap chunks.
swapphys_buf	pages of physical swap space to keep available.
swapphys_cnt	pages of available physical swap space on disk.
swapspc_cnt	Total amount of swap currently available on all devices and file systems enabled in units of pages. Updated each time a device or file system is enabled for swapping.
swapspc_max	Total amount of device and file-system swap currently enabled on the system in units of pages. Updated each time a device or file system is enabled for swapping.
swapspc_debit	number of swap blocks by which to adjust swapspc_cnt.
swapspc_sparing	number of swap blocks unavailable to swap.
swapmem_max	Maximum number of pages of pseudo-swap enabled. Initialized to 3/4 available system memory.
swapmem_cnt	Total number of pages of pseudo-swap currently available. Initialized to 3/4 available system memory.
maxfs_pri	highest available device priority.
maxdev_pri	highest available swap prioirity.

Element	Meaning
`sys_mem`	Number of pages of memory not available for use as pseudo-swap. Initialized to 1/4 available system memory.
`sysmem_max`	maximum pages not available for swap.
`freemem`	page count of remaining blocks of free memory.
`freemem_cnt`	Number of processes waiting for memory.

Swap Space Values

System swap space values are calculated as follows:

- Total swap available on the system is `swapspc_max` (for device swap and file system swap) + `swapmem_max` (for pseudo-swap).
- Allocated swap is `swapspc_max - [sum(swdevt[n].sw_nfpgs) + sum(fswdevt[n].fsw_nfpgs)]` (for device swap and file system swap) `+ (swapmem_max - swapmem_cnt)` (for pseudo-swap).

In HP-UX, only data area growth (using `sbrk()`) or stack growth will cause a process to die for lack of swap space. Program text does not use swap.

Reservation of Physical Swap Space

Swap reservation is a numbers game. The system has a finite number of pages of physical swap space. By decrementing the appropriate counters, HP-UX reserves space for its processes.

Most UNIX systems and UNIX-like systems allocate swap when needed. However, if the system runs out of swap space but needs to write a process' page(s) to a swap device, it has no alternative but to kill the process. To alleviate this problem, HP-UX reserves swap at the time the process is `fork`ed or `exec`'d. When a new process is forked or executed, if insufficient swap space is available and reserved to handle the entire process, the process may not execute.

At system startup, `swapspc_cnt` and `swapmem_cnt` are initialized to the total amount of swap space and pseudo-swap available.

Whenever the swapon( ) call is made to a device or file syste, the amount of swap newly enabled is converted to units of pages and added to the two global swap-reservation counters swapspc_max (total enabled swap) and swapspc_cnt (available swap space).

Each time swap space is reserved for a process (that is, at process creation or growth time), swapspc_cnt is decremented by the number of pages required. The kernel does not actually assign disk blocks until needed.

Once swap space is exhausted (that is, swapspc_cnt == 0), any subsequent request to reserve swap causes the system to allocate addition chunk of file-system swap space. If successful, both swapspc_max and swapspc_cnt are updated and the current (and subsequent requests) can be satisfied. If a file-system chunk cannot be allocated, the request fails, unless pseudo-swap is available.

When swap space is no longer needed (due to process termination or shrinkage), swapspc_cnt is incremented by the number of pages freed. swapspc_cnt never exceeds swapspc_max and is always greater than or equal to zero. If a chunk of file-system swap is no longer needed, it is released back to the file system and swapspc_max and swapspc_cnt are updated.

If no device or file system swap space is available, the system uses pseudo-swap as a last resort. It decrements swapmem_cnt and locks the pages into memory. Pseudo swap is either free or allocated; it is never reserved.

Swap Reservation Spinlock

The rswap_lock spinlock guards the swap reservation structures swapspc_cnt, swapspc_max, swapmem_cnt, swapmem_max, sys_mem, and pswaplist.

Reservation of Pseudo-Swap Space

Approximately 3/4 of available system memory is available as pseudo-swap space if the tunable parameter swapmem_on is set to 1. Pseudo-swap is tracked in the global pseudo swap reservation counters swapmem_max (enabled pseudo-swap) and swapmem_cnt (currently available pseudo-swap). If physical swap space is exhausted and no additional file-system swap can be acquired, pseudo swap space is reserved for the process by decrementing swapmem_cnt.

For example, on a 64MB system, swapmem_max and swapmem_cnt track approximately 48MB of pseudo-swap space, the remainder tracked by the global sys_mem, which represents the number of pages reserved for system use only.

Processes track the number of pseudo swap pages allocated to them by incrementing a per region counter r_swapmem. All regions using pseudo swap are linked on the pseudo swap list pswaplist. Once pseudo swap is exhausted (that is, swapmem_cnt==0), attempts at process creation or growth will fail.

Because the swapper competes with the operating system for use of memory, swapmem_cnt can also be decremented by the operating system for any dynamically allocated memory. Once swapmem_cnt is exhausted, subsequent requests for swap space fail; however, the operating system can still reserve memory out of the malloc pool.

Once a process no longer needs its allocated pseudo swap space, swapmem_cnt is incremented by the amount released and r_swapmem is updated. If the system returns the pseudo swap space used for dynamically allocated kernel memory, the amount being released is firtst added to sys_mem. Once sys_mem grows to its maximum value, any additional pages returned are used to update swapmem_cnt.

swapmem_cnt must be less than or equal to swapmem_max and greater than or equal to zero.

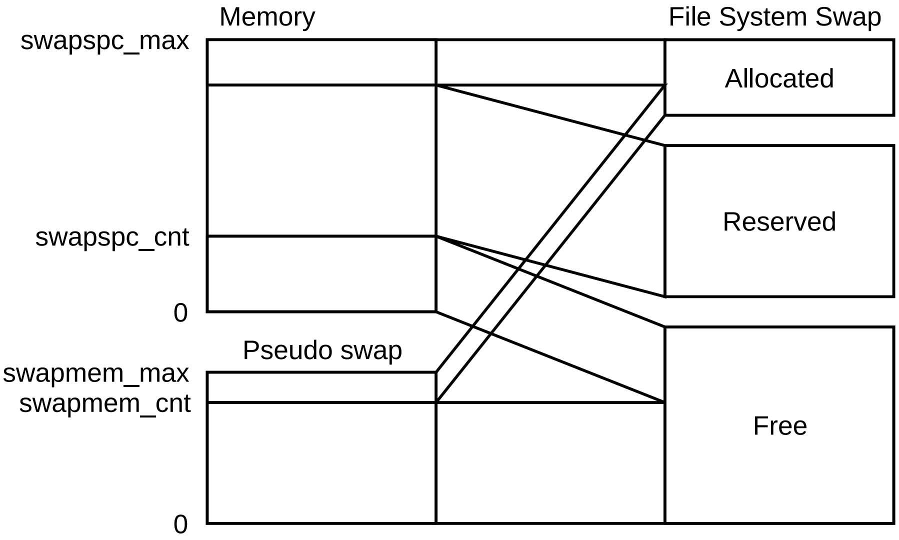
Because pseudo swap is shared by the swapper and memory allocation routines, it is used sparingly. The operating system periodically checks to see if physical swap space has been recently freed. If it has, the system attempts to migrate processes using pseudo swap only to use the available physical swap by walking the doubly linked list of pseudo swap regions. swapspc_cnt is decremented by the r_swapmem value for each region on the list until either swapspc_cnt drops to zero or no other regions utilize pseudo swap. swapmem_cnt is then incremented by the amount of pseudo swap successfully migrated.

Pseudo Swap and Lockable Memory

Because pseudo swap is related to system memory usage, the swap reservation scheme reflects lockable memory policies.

Although the system is not necesarily allocating additional memory when a process locks itself into memory, locked pages are no longer available for general use. This causes swapmem_cnt to be decremented to account for the pages. swapmem_cnt is also decremented by the size of the entire process if that process gets plocked in memory

Figure 1-28 **Reserving swap space from file-system swap to memory**



How Swap Space is Prioritized

All swap devices and file systems enabled for swap have an associated priority, ranging from 0 to 10, indicating the order that swap space from a device or file system is used. System administrators can specify swap-space priority using a parameter of the swapon(1M) command.

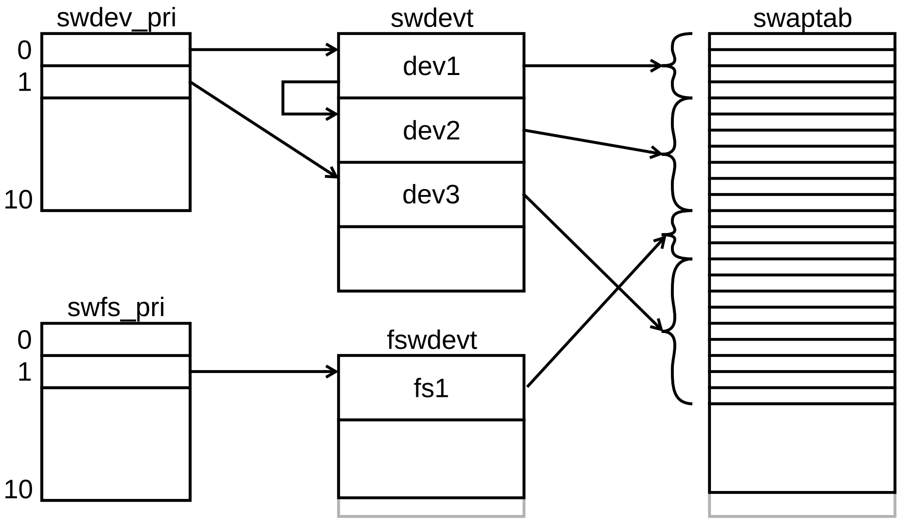
Swapping rotates among both devices and file systems of equal priority. Given equal priority, however, devices are swapped to by the operating system before file systems, because devices make more efficient use of CPU time.

We recommend that you assign the same swapping priority to most swap devices, unless a device is significantly slower than the rest. Assigning equal priorities limits disk head movement, which improves swapping performance.

Three Rules of Swap Space Allocation

- Start at the lowest priority swap device or file system. The lower the number, the higher priority; that is, space is taken from a system with a zero priority before it is taken from a system with a one priority.
- If multiple devices have the same priority, swap space is allocated from the devices in a round-robin fashion. Thus, to interleave swap requests between a number of devices, the devices should be assigned the same priority. Similarly, if multiple file systems have the same priority, requests for swap are interleaved between the file systems. In the figure, swap requests are initially interleaved between the two swap devices at priority 0.
- If a device and a file system have the same swap priority, all the swap space from the device is allocated before any file-system swap space. Thus, the device at priority 1 will be filled before swap is allocated from the file system at priority 1.

Figure 1-29 Choosing a swap location



Swap Space Structures

Swapping is accomplished on HP-UX using the following data structures:

- Device swap priority array (swdev_pri[]), used to link together swap devices with the same priority. That is, the entry in swdev_pri[n] is the head of a list of swap devices having priority n. The first field in swdev_pri[] structure is the head of the list; the sw_next field in the swdevt[] structure links each device into the appropriate priority list.
- File system swap priority array (swfs_pri[]), which serves the same purpose as swdev_pri[], but for file system swap priority.
- Device swap table (struct swdevt), defined in conf.h to establish the fundamental swap device information.
- File system swap table (struct fswdevt), defined in swap.h for supplimentary file-system swap.
- Swap table of available chunks (struct swaptab), which keeps track of the available free pages of swap space.
- Mapping of swap pages (struct swapmap), whose entries together with swaptab combine for a swap disk block descriptor.

The following table details the elements of the struct swdevt.

Table 1-26 Device swap table (struct swdevt)

Element	Meaning
sw_dev	Actual swap device, as defined by its major (upper 8 bits) and minor (lower 24 bits) numbers.
sw_enable	Enabled flag. Zero if device swap is disabled; one if enabled.
sw_start	Offset into the swap area on disk, in kilobytes.
sw_nblksavail	Size of swap area, in kilobytes.
sw_nblksenabled	Number of blocks enabled for swap. Must be a multiple of swchunk (2MB default).
sw_nfpgs	Number of free swap pages on the device. Updated whenever a page is used or freed.

Element	Meaning
sw_priority	Priority of swap device (1-10).
sw_head, sw_tail	First and last swaptab[] entry associated with swap device.
sw_next	Pointer to the next device swap entry (swdevt) at this priority; implemented as a circular list used to update the pointer in swdev_pri for round-robin use of all devices at a particular priority.

The following table details the principle elements of the struct fswdevt.

Table 1-27 **File system swap table (struct fswdevt)**

Element	Meaning
fsw_next	Pointer to next file system swap (fswdevt entry) at this priority; implemented as a circular list.
fsw_enable	Enabled flag. Zero if file-system swap is disabled; one if enabled.
fsw_nfpgs	Number of free swap pages in this file system swap; updated whenever a page is used or freed.
fsw_allocated	Number of swchunks (2MB default) allocated on this file-system swap.
fsw_min	Minimum swchunks to be preallocated when the file-system swap is enabled.
fsw_limit	Maximum swchunks allowed on file system; unlimited if set to zero.
fsw_reserve	Minimum blocks (of size fsw_bsize) reserved for non-swap use on this file system.
fsw_priority	Priority of device (0-10). Priority can also be determined by identifying swfs_pri[] linked list.

Element	Meaning
fsw_vnode	vnode of the file system swap directory (/paging) under which the swap files are created.
fsw_bsize	Block size used on this file system; used to determine how much space fsw_reserve is reserving
fsw_head fsw_tail	Index into swaptab[] of first, last entry associated with this file system swap.
fsw_mntpoint	File system mount point; character representation of fsw_vnode, used for utilities (such as swapinfo(1M)) and error messages.

swaptab and swapmap Structures

Two structures track swap space. The swaptab[] array tracks a chunk of swap space. swapmap entries hold swap information on a per-page level. swaptab defaults to track a 2MB chunk of space and swapmap tracks each page within that 2MB chunk.

Each entry in the swaptab[] array has a pointer (called st_swpmp) to a unique swapmap. swapmap entries have backwards pointers to the swaptab index. There is one entry in the swapmap for each page represented by the swaptab entry (default 2 MB, or 512 pages); that is, swapmap conforms in size to swchunk.

A linked list of free swap pages begin at the swaptab entry's st_free and use each free swapmap entry's sm_next. When a page of swap is needed, the kernel walks the structures (using the getswap() routine in vm_swalloc.c), which calls other routines that actually locate the chunk, and so forth.

- Beginning with the lowest priority, we begin by examining swdev_pri[].curr, which points to a swdevt entry.
- If sw_nfpgs is zero (no free pages), we follow the pointer sw_next to get the next swdevt entry at this priority.
- If none of these have free pages, we move on to swfs_pri[].curr, the file system swap at this priority, checking fsw_nfpgs for free pages.

- If we are still unsuccessful, we move to the next priority and try again.
- Once we find a swdevt or fswdevt with free pages, we walk that device's swaptab list, starting with sw_head or fsw_head, and using st_next in each swaptab entry, until we find a swaptab entry with non-zero st_nfpgs.
- st_free points to the first free swapmap entry (and thus first free page) in this swaptab chunk.
- The swalloc() routine creates a disk block descriptor (dbd) using 14 bits of dbd_data for the swaptab index and 14 bits for the swapmap index. The r_bstore in the region is set to the disk device vnode or the file system directory vnode, and the dbd is marked DBD_BSTORE.

  When faulting in from swap, the same process is followed as for faulting in from the file system: r_bstore and dbd_data are hashed together and checked for a soft fault, then devswap_pagein() is called. The devswap_pagein() routine uses the dbd_data as a 14-bit swaptab index and a 14-bit swapmap index to determine the location of the page on disk.

Now all information needed to retrieve the page from swap has been stored.

Figure 1-30 The swaptab and swapmap structures

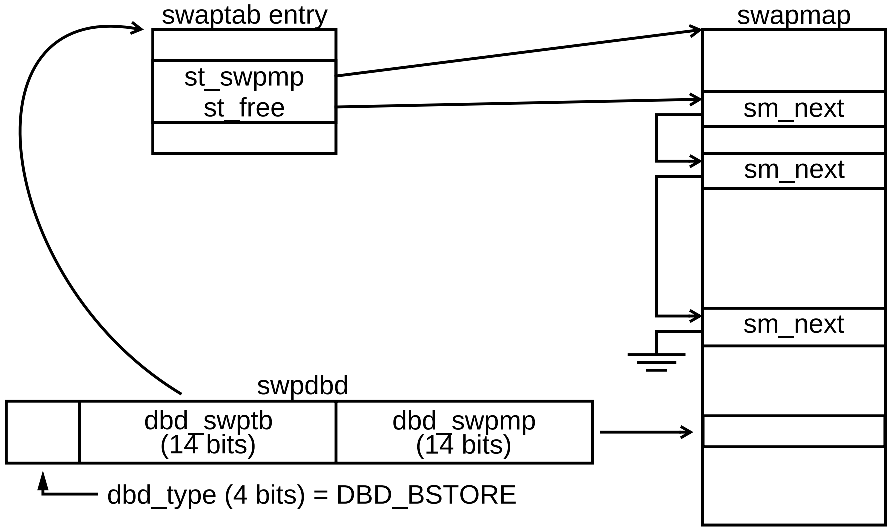


Table 1-28 Swap table entry (struct swaptab)

Element	Meaning
st_free	Index to the first free page in the chunk. Each entry maps to a 4KB-age of swap.
st_next	Index to next swaptab entry for same device or file-system swap; at end of list, st_next is -1.
st_flags	ST_INDEL: File-system swap flag, indicating chunk is being deleted; do not allocate pages from it. Set only by the realswapoff() routine. ST_FREE: File-system swap flag, indicating chunk may be deleted, because none of its pages are in use. In the case of remote swap, the chunk should not be deleted immediately; set st_free_time to current time plus 30 minutes (1800 seconds) when setting this flag. Once 30 minutes has elapsed, the chunk can be freed. If the chunk is needed during the interim, the flag can be cleared using chunk_release(), called from lsync(). ST_INUSE: swaptab entry is being changed.
st_dev, st_fsp	Pointers to swdevt entry that references the swaptab entry.
st_nfpgs	Number of free pages in this (swchunk) swaptab entry.
st_swpmp	Pointer to swapmap[] array that defines this swchunk of swap pages.
st_free_time	Indicates when remote fs chunk can be freed (see explanation of ST_FREE flag).

Table 1-29 swap map entry (struct swapmap)

Element	Meaning
sm_ucnt	Number of threads using the page. When decremented to zero, the swap page is free and the free pages linked list can be updated.
sm_next	Index of the next free page in the swapmap[]. This is valid only if sm_ucnt is zero; that means that this swapmap entry is included in the linked list beginning with swaptab's st_free.

Deactivation using the pager

Since vhand()is tuned to be nice regarding I/O usage and CPU usage, it allows the pager to fault out swapped processes. The swapper marks the process to be swapped for deactivation, which takes it off the run queue. Since it cannot run once its pages are aged, they cannot be referenced again. When the steal hand comes around, it steals all the pages in the region.

When memory pressure is high, sched() selects a process to swap using the routine choose_deactivate(). This routine is biased to choose non-interactive processes over interactive ones, sleeping processes over running ones, and long-running processes over newer ones.

Once a process has been chosen to be deactivated, the following actions occur:

- The process's SDEACT flag and its threads' TSDEACT flags are set.
- The process's threads are removed from the run queue. It the process is waiting for I/O, its SDEACTSELF flag and its threads' TSDEACTSELF flags are set. When I/O completes, the process deactivates in the paging routines.
- The process's p_deactime in the proc structure is set to the current time to establish a record of how long the process is deactivated.
- The process is positioned in the active pregion chain to ready it for the steal hand.
- The uarea pregion is added to the list of active regions for it to get paged out.

- The global counter deactive_cnt is incremented.

A process that has been inactive long enough for all its pages to have been aged and stolen is virtually swapped out already. The global deactprocs points to the head of a list of inactive processes, its chain running through the pregion element p_nextdeact. If the average number of free pages drops below lotsfree, these pages are swapped out.

When memory pressure eases, a deactivated process is reactivated. The choose_reactivate() routine is biased to choose interactive over non-interactive ones processes, runnable processes over sleeping ones, and processes that have been deactivated longest over those more recently deactivated.

Overview of Demand Paging

Recall that for a process to execute, all the regions (for data, text, and so forth) have to be set up; yet pages are not loaded into memory until the process demands them. Only when the actual page is accessed is a translation established.

A compiled program has a header containing information on the size of the data and code regions. As a process is created from the compiled code by fork and exec, the kernel sets up the process's data structures and the process starts executing its instructions from user mode. When the process tries to access an address that is not currently in main memory, a page fault occurs. (For example, you might attempt to execute from a page not in memory.) The kernel switches execution from user mode to kernel mode and tries to resolve the page fault by locating the `pregion` containing the sought-after virtual address. The kernel then uses the pregion's offset and region to locate information needed for reading in the page.

If the translation is not already present and the page is required, the `pdapage()` routine executes to add the translation (space ID, offset into the page, protection ID and access permissions assigned the page, and logical frame number of the page), and then on demand brings in that page and sets up the translation, hashes in the table, and all the rest.

In main memory, the kernel also looks for a free physical page in which to load the requested page. If no free page is available, the system swaps or pages out selected used pages to make room for the requested page. The kernel then retrieves (pages in) the required page from file space on disk. It also often pages in additional (adjacent) pages that the process might need.

Then the kernel sets up the page's permissions and protections, and exits back to user mode. The process executes the instruction again, this time finding the page and continuing to execute.

The flexibility of demand paging lies in the fact that it allows a process to be larger than physical memory. Its disadvantage lies in the degree of complexity paging requires of the processor; instructions must be restartable to handle page faults.

By default, all HP-UX processes are load-on-demand. A demand paged process does not preload a program before it is executed. The process code and data are stored on disk and loaded into physical memory on demand in page increments. (Programs often contain routines and code that are rarely accessed. For example, error handling routines might constitute a large percentage of a program and yet may never be accessed.)

copy-on-write

HP-UX now implements copy-on-write of `EXEC_MAGIC` processes, to enable the system to manipulate processes more efficiently. The system used to copy the entire data segment of a process every time the process `fork`'d, increasing `fork` time as the size of the data and code segments increased. Only one translation of a physical page is maintained; a parent process can point to and read a physical page, but copies it only when writing on the page. The child process does not have a page translation and must copy the page for either read or write access.

Copy-on-write means that pages in the parent's region are not copied to the child's region until needed. Both parent and child can read the pages without being concerned about sharing the same page. However, as soon as either parent or child writes to the page, a new copy is written, so that the other process retains the original view of the page.

For more information about the implementaton of `EXEC_MAGIC`, see the HP-UX Process Management white paper.

HOW PROCESS STRUCTURES ARE SET UP IN MEMORY

When a process is fork'd, a duplicate copy of its parent process forms the basis of the child process. .

Region Type Dictates Complexity

Under the kernel procdup() routine, the system walks the pregion list of the parent process, duplicating each pregion for the child process. How this is done is dictated by the region type.

- If the region is type RT_SHARED, a new pregion is created that attaches to the parent's region.
- If the region is type RT_PRIVATE, the region is duplicated first, and then a new pregion is created and attached to the new region.

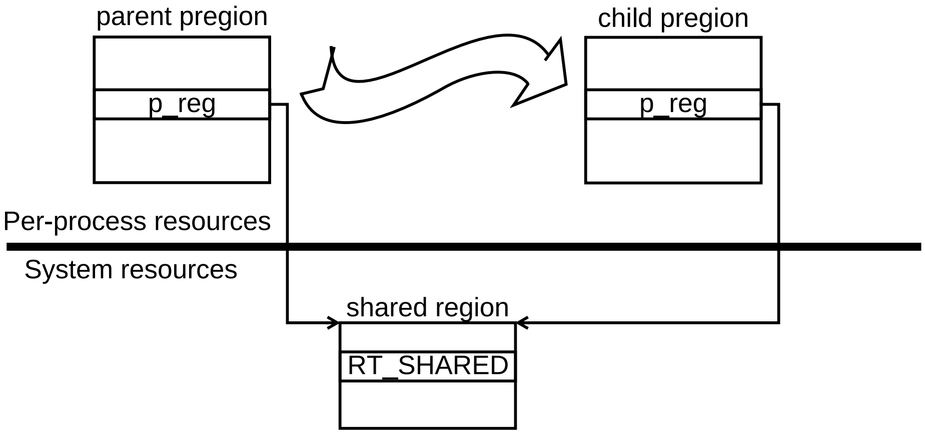
Duplicating pregions for Shared Regions

Because a region of type RT_SHARED is shared by parent and child, fewer changes occur to the pregions and region: Only a new pregion must be created and attached to the shared region.

- A new pregion is allocated and fields copied from the parent pregion to the child pregion.
- The pregion elements used by vhand (p_agescan, p_ageremain, and p_stealscan) are initialized to zero and the child pregion is added to the active pregion chain just before the stealhand, to prevent it from being stolen yet.
- The region elements r_incore and r_refcnt are incremented to reflect the number of in-core pregions accessing the region and the number of pregions, in-core or paged, accessing the region.

The procedure is considerably more complex when an RT_PRIVATE region is copied.

Figure 1-31 Duplicating pregions with shared regions

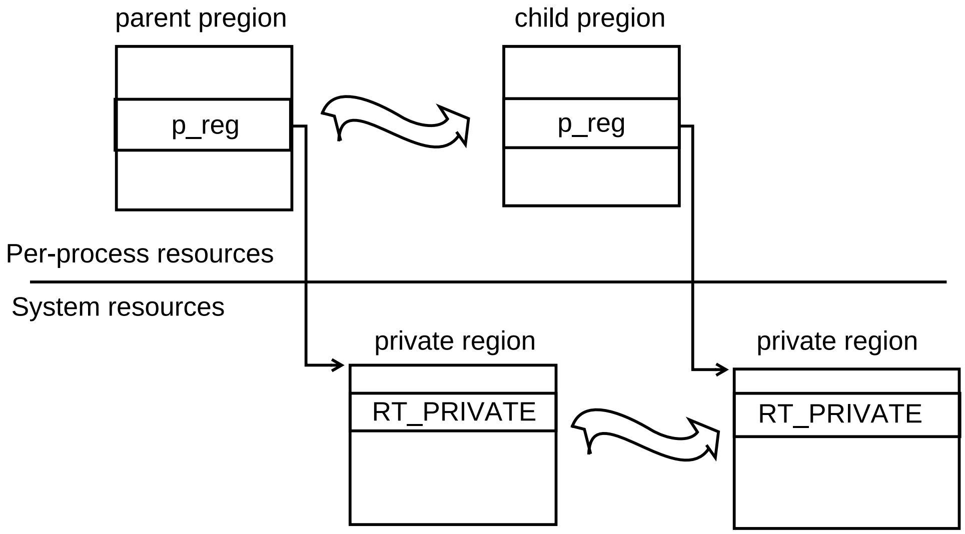


Duplicating pregions for Private Regions

Forking a process with a region of type RT_PRIVATE requires that a new child region be allocated first.

- The child region's pointers are set:
  - r_fstore, the forward store pointer is pointed to the same value as the parent's, and the vnode's reference count (v_count) is incremented.
  - r_bstore, the backward store pointer is set to the kernel global swapdev_vp, and its v_count is incremented also.
- The child region is attached to the end of the linked list of active regions.
- Swap is reserved. If insufficient swap space is available, fork() fails and returns the error ENOMEM.
- The child region's B-tree structures are initialized and sufficient swap space is reserved for a completely filled B-tree.

Figure 1-32 Duplicating a child process of type RT_PRIVATE



- The parent's vfd and dbd proto values are copied to the child's B-tree root.
- The vfd proto in both the parent region and the child region are set so that all pages of the region are copy-on-write.
- The B-tree element b_vproto is set to indicate that the copy-on-write flag (pg_cw) must be set in the vfd for any new vfddbd pair added to the B-tree.
- A chunk of vfddbds is created for the child's B-tree (equal to each chunk of vfddbds in the parent's B-tree) and filled with proto values. The pg_cw bit is already set to copy-on-write for all default vfds in the child B-tree's chunk.

Setting copy-on-write when the vfd is valid

Before the chunks of vfddbds in the child region can be used, the validity of every entry must be checked.

- If a vfd is not valid (that is, its pv_v is not set), the pg_cw of the parent's vfd must be set and copied to the child. If pg_lock is set in the parent, it must be unset in the child, as locks are not inherited.

Once the vfd is valid, further modifications are made to the low-level structures:

- The r_nvalid element in the child region is incremented to reflect the number of valid pages.
- The vfd contains a pfn (page frame number), which indexes into the pfdat[] array. The pfdat entry pf_use count (number of regions using this page) must be incremented.
- If the parent vfd's copy-on-write bit isn't set, the pde must be set for translations to the page to behave as copy-on-write.

Reconciling the Page and Swap Image

If a page has been written to a swap device, but has since been modified, the swap-device data now differs from the data in memory. The disk page must be disassociated from the page in memory by setting the dbd type to DBD_NONE. Then, the next time the page is written to a swap device, it will be assigned a new location.

Everything is now set up from the perspective of the parent's B-tree for copy-on-write.

Setting the child region's copy-on-write status

- The child's r_swalloc is set to the number of region and B-tree pages reserved.
- The r_prev and r_next are set to link the child region to the parent region.
- The kernel chooses new space for the pregion, rather than copying it from the parent pregion. This establishes two ranges of virtual addresses (different space, same offset) translating to the single range of physical address.
  - If a parent process accesses its virtual addresses, it willl get a TLB miss fault because the addresses have been purged from the TLB.
  - If a child process accesses any of its virtual addresses, it will also get a TLB miss fault because the addresses did not previously exist in the TLB, and do not exist in HTBL.

Duplicating a Process Address Space to Make the Process copy-on-write

- procdup() creates a duplicate copy of a process based on forktype, parent process (pp), child process (cp), and parent thread (pt) and child thread (ct).

  procdup() allocates memory for the uarea of the child. (In fact, procdup() is the routine that calls createU() to create the uarea too.)

  procdup() calls dupvas to duplicate the parent's virtual address space, based on the kind of process (fork vs vfork) being executed.

- If the process was created by fork, dupvas duplicates the parent process's virtual address space; if the process was vfork'd the parent's virtual address space is used.

  dupvas looks for and finds each private data object, does whatever each requires to be duplicated (there are special considerations required for text, memory mapping, data objects, graphics), and when it finishes duplicating the special objects, calls private_copy or shared_copy, depending on whether it is dealing with a private or shared region.

  - If the region is shared, shared_copy increments the reference count on the region to indicate it is being shared.
  - If the region is private, private_copy locks the region and enables the region to be duplicated by calling dupreg().

- dupreg() allocates a new region for the child, duplicates the parent's vfds and the entire region structure, then calls do_dupc to duplicate entries under the region.

- do_dupc() sets up a parent-child relationship, and by duplicating the relationship, sets up the child to be copy-on-write. It makes sure the parent's region is valid, sets copy on write for the child, sets the translation as rx (read-execute) only, duplicates information for every vfddbd combination in the region.

  once do_dupc() completes, the child process exists as a duplicated version of the parent process. The child process is attached to the child's address space and is no longer dependent on the parent.

- do_dupc then calls hdl_cw() to update the child's access rights and make the child copy on write.

Duplicating the uarea for the Child's Process

The createU() routine builds a uarea and address space for the child process. The uarea is set up last for a fork'd process, to prevent the child process from resuming in the middle of pregion duplication code. If the process is vfork'd, the uarea is created during exec(). Until then, the child uses the parent's uarea.

- When a user process is created with FORK_PROCESS, a temporary space is allocated for a working copy of the parent's uarea to be modifed into the child's uarea. The temporary space will be freed after the uarea is copied to the new region. fork() updates the savestate in the parent uarea's u_pcb just before copying the data. (vfork() does not do this because it creates the uarea during exec(), and the savestate will change immediately.)
- A region is allocated for the new uarea, its data structure is initialized, its r_bstore value set back to the swap device, and the new region is added to the list of active regions. The uarea has no r_fstore value, since it comes with ready-made data.
- Space is allocated for the uarea's pregion, which is initialized. Each uarea has a unique space ID. The new pregion is marked with the PF_NOPAGE flag. uarea pregions are unaffected by vhand because they are not added to the list of active pregions. Only if an entire process is swapped out are the uarea's pages written to a swap device.
- Once created, the pregion is attached into the linked list of pregions connected to the vas. Its pointer is stored in r_pregs, its p_prpnext set to NULL, and its r_incore and r_refcnt set to one.
- Once swap space is reserved for the uarea and B-tree pages and the default dbd is set to DBD_DFILL, the uarea pages (UPAGES) are allocated. Each page requires a pfdat entry from phead (sleeping if none is available immediately). The pfn is stored in the vfd, the pg_v is set as valid, r_nvalid is incremented, and a pde is created for the physical-to-virtual translation. The pfdat entry's P_UAREA and HDLPF_TRANS flags are set, and the dbd is set to DBD_NON.
- The pointers u_procp (to the child process) and u_kthreadp (to the child thread) are pointed to the child uarea.

Conceivably, the child can now run successfully. The current state is therefore saved in the copied uarea with a setjmp() call and pointed to with pcb_sswap. Thus, when the child first calls the resume()

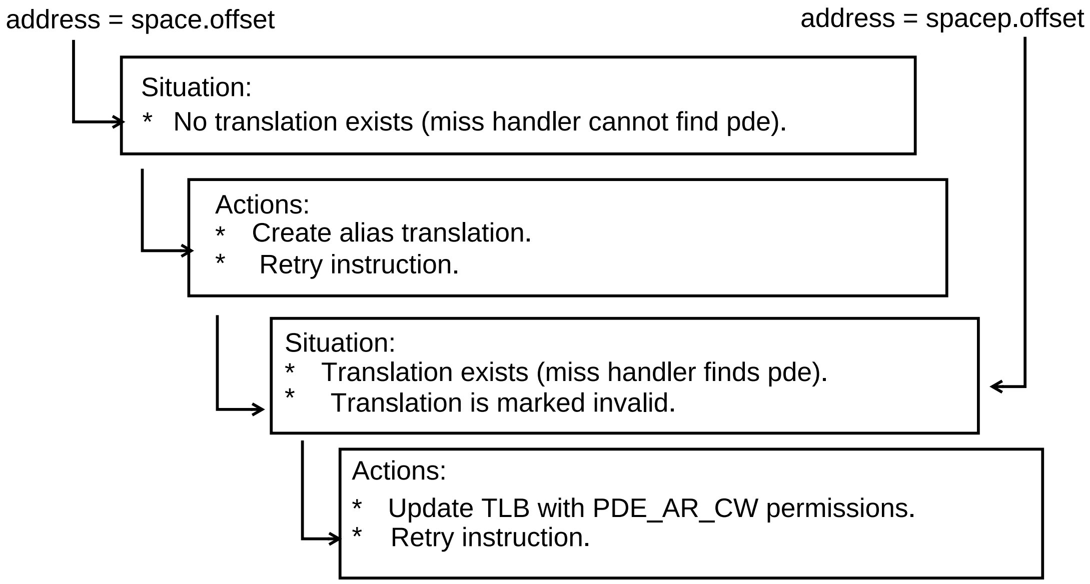
routine, it detects that pcb_sswap is non-zero and does a longjmp() to get back here. The child then return from procdup() with the value FORKRTN_CHILD.

The parent's open file table is copied to the child and the copied uarea is copied into the actual pregion. This copy causes TLB miss faults that cause the pregion's pdes to be written to the TLB, thus associating the uarea's virtual address with the physical pages just set up. The process completes by returning from procdup with the return value FORKRTN_PARENT.

Reading from the parent's copy-on-write page

When the parent region accesses one of its RT_PRIVATE pages for read, the processor generates a TLB miss fault, which the kernel handles as an interrupt. The TLB miss fault handler finds the pde and inserts the information (including the new access rights) into the processor's TLB. On return from the interrupt, the processor retries the read and is successful, since PDE_AR_CW allows user-mode read and execute access

Figure 1-33 The first time a read is done to a copy-on-write page


address = space.offset
address = spacep.offset
Situation:
* No translation exists (miss handler cannot find pde).
Actions:
* Create alias translation.
* Retry instruction.
Situation:
* Translation exists (miss handler finds pde).
* Translation is marked invalid.
Actions:
* Update TLB with PDE_AR_CW permissions.
* Retry instruction.

Reading from the child's copy-on-write page

When the child region accesses one of its pages for read, the TLB miss handler does not find a pde for the virtual address, because none has been set one up yet. The virtual address was set up in the pregion structure. If you are not doing copy-on-access (which is now the default) and the page is needed, the aliased translation must be made.

- First a save state is created.
- The vas pointer is taken and the skip list searched to find the pregion containing the page with this address.
- If the page translates to more than one virtual address, the appropriate alias is acquired.
- The child region fails to access a page for read and gets a TLB miss, but the miss handler finds a translation and loads it into the TLB.
- The routine returns from interrupt and succeeds in reading the page.

Faulting In A Page

When regions are initialized, the disk block descriptor (dbd) dbd_data field of the is set to DBD_DINVAL (0xfffffff) in all cases. The prototype dbd_type values are set as follows:

- DBD_FSTORE for text and initialized data,
- DBD_DZERO for stack and uninitialized data.

When a page is read for the first time, a TLB miss fault results because the physical page (and therefore its translation in the sparse PDIR) does not yet exist. The fault handler is responsible for bringing in the page and restarting the instruction that faulted. In determining whether or not the page is valid, the fault handler determines which pregion in the faulting process contains the faulting address. The fault code eventually calls virtual_fault(), the primary virtual-fault handling routine . The arguments passed to this routine are the virtual address causing the fault, the virtual address and virtual space of the pregion, and a flag indicating read or write access.

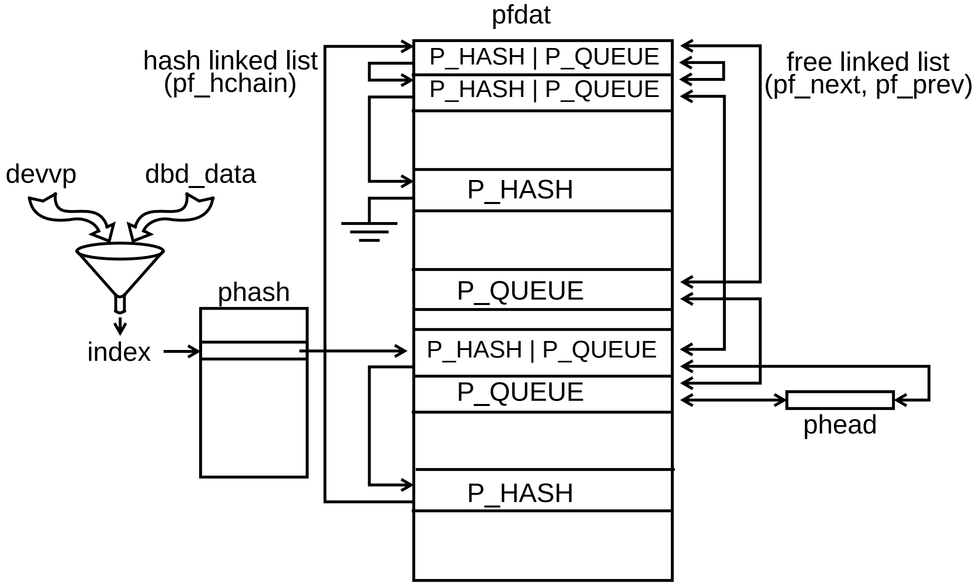
The kernel searches the B-tree for the vfd and dbd of the page. If the valid bit in the vfd flag is set, another process has read the address into memory already. If the r_zomb flag is set in the region, the program prints Pid %d killed due to text modification or page I/O error message and returns SIGKILL, which the handler sends to the process.

Faulting In a Page of Stack or Uninitialized Data

If the dbd_type value is set to DBD_DZERO (as is the case for stack and uninitialized data), the process sets the copy-on-write bit to zero. The kernel then checks to determine whether the page pertains to a system process or to a high-priority thread. If neither and memory is tight, the process sleeps until free memory is driven down to the priority associated with the process. (In worst case, a thread might wait until memory is above desfree.)

Once the process is restarted, vfd and dbd pointers are examined to ensure their continued accuracy. A free pfdat entry is acquired from phead, its pfn (pf_pfn) placed in the vfd, the vfd's valid bit set, and the region's r_nvalid counter (number of valid pages) incremented. The process changes dbd_type to DBD_NONE and dbd_data to 0xfffff0c. Finally, the virtual-to-physical translation of the page is added to the sparse PDIR and the page is zeroed.

Figure 1-34 Checking the free list to fault in a DBD_FSTORE page



Faulting in a Page of Text or Initialized Data

If a process has a virtual fault on a DBD_FSTORE page, the kernel uses the r_fstore pointer of the region's vnode, to determine which file-system specific pagein() routine (for example, ufs_pagein(), nfs_pagein(), cdfs_pagein(), vx_pagein()) to call. The pagein() routines are used to recover the correct page from a free list of memory pages or to read in a correct page from disk.

The pagein routine gets information about the page being faulted from the vm_pagein_init() routine, which gets the vfd/dbd pairs, sets up the region index, and ascertains that no valid page already exists.

One page must be reserved. Then vm_no_io_required() is called to determine if the page can be satisfied locally, either by a zero-filled page (sparse file) or from the page cache.

vm_no_io_required() checks for the faulted page in the page cache:

- vm_no_io_required acquires the device vnode pointer (devvp) that points to the actual disk device (such as /dev/vg00/lvol5) rather than to the file referenced by r_fstore.
- If the dbd data field is DBD_DINVAL, vm_no_io_required gets the actual location of the disk block on the disk device and stores this value in the dbd data field.
- vm_no_io_required calls pageincache() with the device vnode pointer and the dbd_data to determine whether the faulted page is on the hash list.
- The pageincache() routine hashes on the vnode pointer and data to choose a pfdat pointer in phash[]. The routine walks the pf_hchain chain of pfdat entries looking for a matching vnode pointer (pf_devvp) and data value (pf_data). If it finds a match, it removes it from the free list.
- If pageincache() returns a pfdat entry, the region's valid page count (r_nvalid) is incremented, the vfd is updated with the pfn (pf_pfn), and a virtual-to-physical translation for the page to the sparse PDIR is added (if it had been removed).

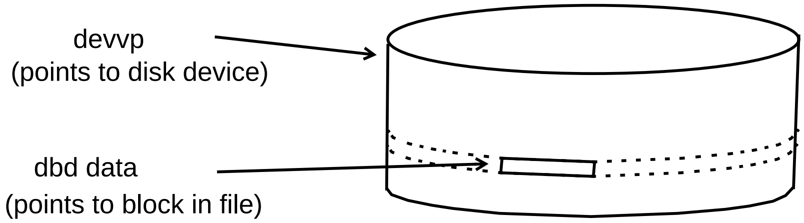
On successfully finding the page in the free list, vm_no_io_required() returns a 1, meaning that no I/O is required to retrieve the page. This is called a soft page fault.

If vm_no_io_required() cannot find the page locally, it returns 0, meaning the page must be faulted in from disk.

Retrieving the Page of Text or Initialized Data from Disk

If the required page is not found in the free list, the pagein() routines refer to dbd to ascertain which page to fetch. (The information had been stored in the dbd by vm_no_io_required().) The pagein() routines also schedule read-ahead pages for I/O, the number of read-ahead pages based on the value of p_pagein in the pregion. This value is adjusted based on whether the file is being accessed at random or sequentially.

Figure 1-35 DBD_FSTORE fault of data not in the free list



If it is being accessed at random, a minimal number of read-ahead pages are required; if sequentially, a maximal number of read-ahead pages are desired, up to the end of the pregion's pages.

- Each time I/O is scheduled for the pregion, the p_nextfault bit in the pregion structure is set to the page expected to be read next if further reading is required.
- If the next page fault matches p_nextfault, the file is being accessed sequentially. In this case, the value of p_pagein is multiplied by two, up to maxpagein_size, a global set to 64. If p_strength is less than 100 (defined as PURELY_SEQUENTIAL), it is also incremented.
- If the next fault does not match p_nextfault, the file is being accessed at random. In this case, the value of p_pagein is divided by two, down to no less than minpagein_size, a global set to 1. If p_strength is greater than -100 (defined as PURELY_RANDOM), it is also decremented.

A page of memory is allocated from phead, a virtual-to- physical translation added to the sparse PDIR, the I/O scheduled from the disk to the page, and the process put to sleep awaiting the non-read-ahead I/O to complete (the process does not await read-ahead I/O to complete). The vfd is marked valid. The dbd is left with dbd_type set to DBD_FSTORE and dbd_data set to the block address on the disk.

Regardless of whether the page data is retrieved from zero-fill, free list, or disk, the page directory entry (pde) has been touched. The instruction is retried and gets a TLB miss fault; the miss handler writes the modified pde data into the TLB; the instruction is retried again and succeeds.

p_strength varies between -100 and 100; p_pagein varies by powers of two between 1 and 64.

VIRTUAL MEMORY AND exec()

When the system performs an exec(), the virtual memory system concerns itself with cleaning up old pregions/regions and setting up new ones.

Cleaning up from a vfork()

Cleanup in the vfork() case is simple.

- The child process is executing but borrowing its resources from the parent process.
- The routine creates its own uarea and returns the parent's resources.
- Then the routine adds text, data, and so on.
  - The routine gets a new vas and attaches it to the child process (p_vas).
  - The uarea and stack of the parent process are copied and the pregions and regions are created for the child uarea, just as for a FORK_PROCESS fork type.
  - The uarea is copied into the child's uarea region, which is pointed to the now-complete uarea from the thread, and the thread switches from using the parent's kernel stack to the new child kernel stack.

Disposing of the old pregions: dispreg()

If exec() is called after a FORK_PROCESS fork, several regions must be disposed of first. Typically, all pregions are disposed of except for the PT_UAREA pregion, which is still needed. If the file is calling exec() on itself, we save a little processing and keep the PT_TEXT and PT_NULLDREF regions, too.

- deactivate_preg() is used to deactivate the pregion by removing it from the active pregion list. If the agehand is pointing to the pregion being deactivated and stealhand is pointing to the next region in the active pregion list, the agehand is moved back one pregion to prevent the agehand from exceeding the stealhand in sequence. Otherwise if the agehand or stealhand is pointing to the pregion being deactivated, both hands are moved forward one pregion.

- If the region is type RT_PRIVATE or the pregion being discarded is the last attached, its resources must be freed up.
  - wait_for_io() awaits completion of any pending I/O to the region (that is, r_poip = 0), so that no I/O request returns to modify a page now assigned a different purpose.
  - The region's B-tree is traversed to delete all the virtual address translations. (That is, for each valid vfd, the TLBs are purged, the cache flushed, and the pde entry invalidated (set space to -1, address to 0, pfn to 0, valid to 0, ref to 0, and clear the bit from pde_os).
- If the pde is not the HTBL entry, the pde is moved from hash list to free list. If it is the HTBL pde and it is unused, an effort is made to fill it with a translation down its linked list, and then free the copied pde.
- The physical-to-virtual translation is removed from pfn_to_virt_table. If it was the last virtual translation for this physical page, the HDLPF_TRANS is cleared in the pfdat entry.
- The pregion pointer is removed from the rpregs list and the memory used by the pregion is freed (that is, returned to its kernel memory bucket).
- The region's r_incore and r_refcnt elements are decremented. If r_refcnt equals zero, the region is freed also.
  - Again, r_poip must decrement to zero before a region can be freed, to prevent any unexpected I/O to its pages.
  - The B-tree is walked again, and for each valid page found, r_nvalid and pf_use are decremented in the pfdat entry. If the physical page is not aliased, its pf_use will now be 0; it can be freed for other uses.
  - Its P_QUEUE flag is set and the page is put on the pfdat free list (phead). The kernel global freemem is incremented. If any other processes are waiting for memory, we wake them all up so that the first one here can have the page (the losers of the race will go to sleep again).

- If r_bstore is swapdev_vp, the reserved swap pages (r_swalloc) are released, as are the swap pages reserved for the B-tree structure (r_root->b_rpages).
- The pages themselves are freed by invalidating their pdes, purging the TLBs, flushing the caches, moving the non-HTBL pdes from the hash list to the free list, and linking the pfdat entry into phead.
- r_root and r_chunk region elements are moved back to the buckets rather than being freed.
- activeregions is decremented; the region is removed from the r_forw / r_back region chain, and the region memory returned to its memory allocation bucket.

Building the new process

If the process for which memory structures are being created is the first to use the a.out as an executable, the a.out vnode's v_vas is NULL, and requires creating the pseudo-vas, pseudo-pregion, and region. Otherwise, the pseudo-vas' reference count is updated.

- To what region a PT_TEXT pregion is attached depends on the type of executable.
  - If the executable is non-EXEC_MAGIC, a PT_TEXT pregion is attached to the pseudo-vas region.
  - If the executable is EXEC_MAGIC, VA_WRTEXT is set in the process vas, the pseudo-vas' region is duplicated as a type RT_PRIVATE region (performing all the steps discussed for an RT_PRIVATE region), RF_SWLAZYWRT is set in the new region so that no swap is reserved before needed, and a PT_TEXT pregion is attached to it.
  - In both cases, a new space is attached to the pregion's virtual address.
  - A PT_NULLDREF pregion is attached to the global region (globalnullrp), using the same space as PT_TEXT.
  - The pseudo-vas' region is duplicated as a type RT_PRIVATE region using r_off to point to the beginning of the data portion of the a.out file. A PT_DATA pregion is attached to it. If this is an EXEC_MAGIC executable, we use the PT_TEXT pregion's space, otherwise a new space is assigned.

- The PT_DATA pregion is incremented by the size of bss (uninitialized data area), using dbd type DBD_DZERO. This sets b_protoidx to the end of the inititialized data area and b_proto2 to DBD_ZERO. More swap is reserved.
- A private region of three pages (SSIZE +1) is created for the user stack. The dbd proto value is set to DBD_DZERO, and a PT_STACK pregion is attached at USRSTACK. The PT_DATA pregion's space is used.
- When a shared library is linked to the process, two PT_MMAP pregions are created: an RT_SHARED pregion containing text mapped into the third quadrant with a space of KERNELSPACE and an RT_PRIVATE pregion containing associated data (such as library global variables) with the PT_DATA pregion's space.
- If VA_WRTEXT is set, the data pregion takes the first available address above where the text ends (in the first or second quadrant); othwerwise it is assigned the first available address above 0x40000000 (the second quadrant).

Virtual memory and exit()

From the virtual memory perspective, an exit() resembles the first part of an exec(). All virtual memory resources associated with the process are discarded, but no new ones are allocated.

Thus, when exiting from a vfork child before the child has performed an exec(), nothing needs to be cleaned up from virtual memory except to return resources to the parent process. If exiting from a non-vfork child, the virtual memory resources are discarded by calling dispreg().