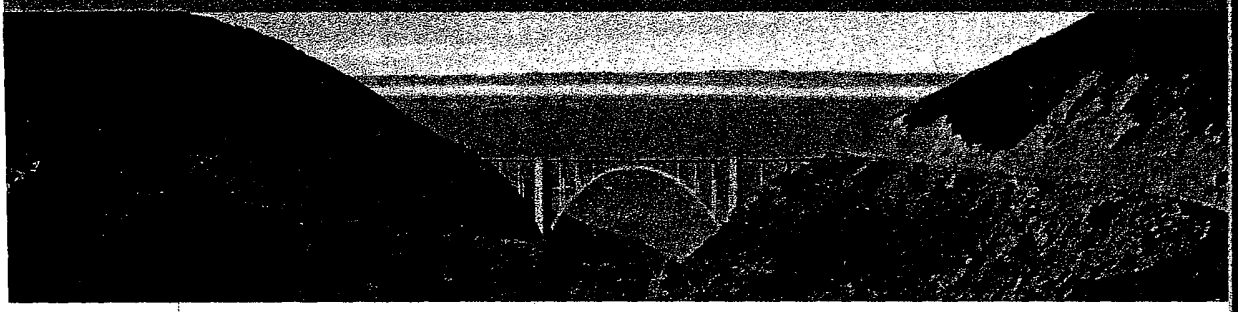
565

HP-UX

TUNING AND PERFORMANCE

CONCEPTS, TOOLS, AND METHODS



- Maximize the performance of your HP-UX system!
- Covers application design and system administration
- Covers every hardware bottleneck
- Choosing and using the best measurement tools—free and commercial

Robert F. Sauers and Peter S. Weygant

Hewlett-Packard® Professional Books

Library of Congress Catalog-in-Publication Data
Sauers, Robert (Robert F.)
HP-UX tuning and performance : concepts, tools, and methods / Robert Sauers, Peter Weygant.
p. cm. -- (Hewlett-Packard professional books)
ISBN 0-13-102716-6
1. Hewlett-Packard computers--Programming. 2. UNIX (Computer file) I. Weygant, Peter. II. Title. III. Series.
QA76.8.H48 S28 1999
005.2'82--dc21

99-29396
CIP

Editorial/production supervision: *Vanessa Moore*
Cover production: *Talar Agasyan*
Cover design: *Design Source*
Cover design director: *Jerry Votta*
Manufacturing manager: *Alexis R. Heydt*
Marketing manager: *Lisa Konzelmann*
Acquisitions editor: *Jill Pisoni*
Editorial assistant: *Linda Ramagnano*
Project coordinator: *Anne Trowbridge*
Manager, Hewlett-Packard Retail Book Publishing: *Patricia Pekary*
Editor, Hewlett-Packard Professional Books: *Susan Wright*

Published by Prentice-Hall, Inc.
Upper Saddle River, New Jersey 07458

Prentice Hall books are widely used by corporations and government agencies for training, marketing, and resale.
The publisher offers discounts on this book when ordered in bulk quantities.
For more information, contact Corporate Sales Department, Phone: 800-382-3419;
Fax: 201-236-7141; E-mail: corpsales@prenhall.com
Or write: Prentice Hall PTR, Corp. Sales Dept., One Lake Street, Upper Saddle River, NJ 07458.

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

ISBN 0-13-102716-6

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Prentice-Hall (Singapore) Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

CHAPTER 9

Memory Bottlenecks

This chapter describes major memory bottlenecks, starting with a review of some important concepts in the area of memory management. This is followed by a description of typical bottleneck symptoms and some techniques for diagnosing and tuning them. Chapter 9 covers the following topics:

- Virtual address space
- Types of magic
- fork() and vfork()
- Dynamic buffer cache
- Sticky bit
- Memory-mapped files and semaphores
- Shared libraries
- Paging, swapping, and deactivation
- Memory management policies
- Sizing memory and the swap area
- Memory metrics
- Types of memory management bottlenecks
- Expensive system calls
- Tuning memory bottlenecks
- Memory-related tunable parameters

9.1 Virtual Address Space

In order to understand the major bottlenecks that affect memory, it is necessary to know how virtual addressing works on a Precision Architecture machine, particularly at the individual process level.

While the amount of actual RAM available for HP-UX is determined by the number of memory chips installed on the computer, the system can make a much larger amount of space available to each process through the use of *virtual memory*. On 32-bit PA-RISC machines or on 64-bit PA-RISC 2.0 machines running in narrow mode, the virtual address space available to each process is 4 GB, spread over four 1-GB quadrants that are used for various kinds of memory objects. On 64-bit PA-RISC 2.0 systems (running HP-UX 11.0), the address space is 16 TB, spread over four 4-TB quadrants. Space registers (SRs) are used for short pointer addressing of these quadrants. Figure 9-1 shows the 32-bit implementation, and Figure 9-2 shows the 64-bit implementation.

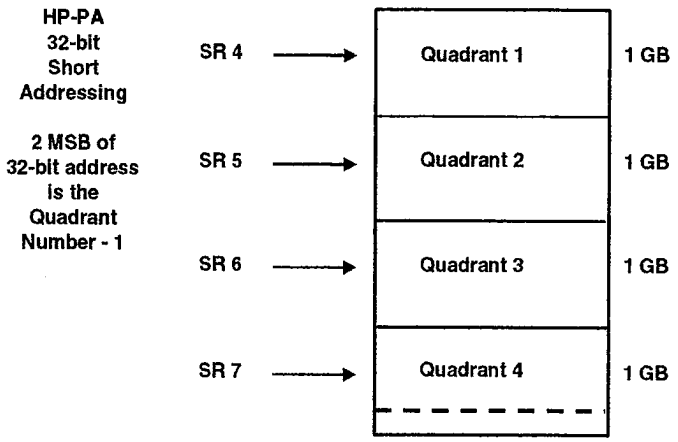


Figure 9-1 PA-RISC Per-Process Virtual Address Space (32 Bits)

Individual processes make use of areas of memory within all these quadrants. Specific areas are normally accessed by the use of short pointers consisting of a two-bit quadrant reference, and an offset into the quadrant where the required memory area starts.

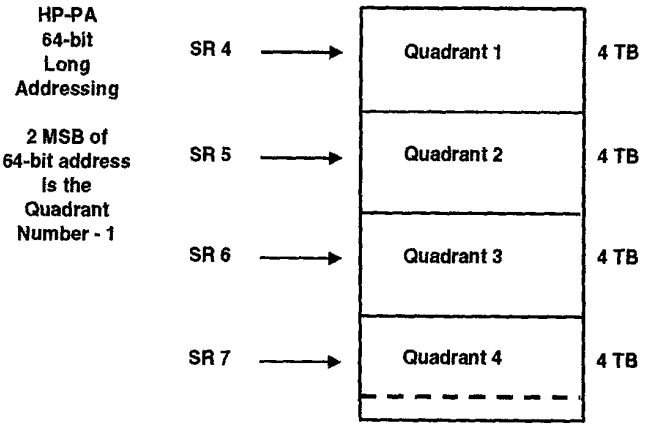


Figure 9-2 PA-RISC Per-Process Virtual Address Space (64 Bits)

Different levels of HP PA provide different amounts of virtual address space, as shown in Figure 9-3.

Level 0	**32-bit physical addressing only** **(Note: HP has never made a Level 0 system)**
Level 1	**48-bit virtual addressing** **2 ** 48 total VAS (272 TB)** **2 ** 16 (32768) spaces of 4 GB each**
Level 2	**64-bit virtual addressing** **2 ** 64 total VAS** **2 ** 32 spaces of 4 GB each**

Figure 9-3 PA-RISC VAS Levels

Pages of memory in HP PA can have two types of protection assigned to them: authorization, which is granted with a Protection ID, equal to the space number associated with the owner; and access rights (read/write/execute), which are the same as the actual file permissions for shared libraries and memory-mapped files. Space number 0 is always reserved for the kernel; non-zero space numbers are assigned to user processes. Note that in PA-RISC, text does not normally have write permission, which means that code cannot be modified in memory.

9.1.1 Variable Page Size

In HP-UX 11.0 and later on PA 2.0 systems, the size of a page of physical memory can be changed from the default of four KB. Because each page translation requires space in the TLB, one would want to define a larger page size when larger ranges of memory are accessed sequentially, or when application performance is poor due to a large number of TLB misses. PA 2.0 systems typically have smaller TLBs than earlier systems. They no longer have a block TLB, nor do they have a hardware TLB walker. These changes in the design were made to lower the cost and reduce the portion of the chip physically required for large TLBs.

The following page sizes are available by user request:

- 4 KB (the default)
- 16 KB
- 64 KB
- 256 KB
- 1 MB
- 4 MB
- 16 MB
- 64 MB

The page size associated with an executable can be specified by using the *chatr(1)* command. The kernel may also increase or decrease the page size according to access patterns that the application exhibits. If an application executes sequentially, the page size will be increased. If an application executes randomly, the page size will be decreased. When there is severe memory pressure, the memory management system may also reduce the page size rather than force large size page-outs with subsequent page-ins when the pages are needed again.

9.2 Types of Magic

Executable programs compiled for PA-RISC processors include a *magic number* in their *a.out* file. This number tells the operating system how to interpret references in the code to the four VAS quadrants described in the previous section. Different types of magic number have been used over time. They include SHARE_MAGIC, DEMAND_MAGIC, EXEC_MAGIC, and SHMEM_MAGIC.

9.2.1 SHARE_MAGIC

By default, a process is compiled with SHARE_MAGIC, which means that the address space is divided into four 1-GB or 4-TB quadrants in 32-bit and 64-bit versions of HP-UX, respectively. In the SHARE_MAGIC format, each quadrant has a specific purpose—such as text, data, or shared objects. *Shared text*, which is code that can be used by many processes, is in Quadrant 1. *Private data* is in Quadrant 2. This data includes

- Initialized data
- Uninitialized data (BSS)
- Dynamically allocated memory
- u_area
- Kernel stack
- User stack
- All data from shared libraries
- Private memory-mapped files

Originally, shared libraries occupied Quadrant 3 while Quadrant 4 was for shared memory. The last portion of Quadrant 4 is reserved for hard and soft physical address space, which is used for addressing hardware devices based on slot number.

In HP-UX 10.0 and in HP-UX 11.0 in 32-bit mode, the VAS with SHARE_MAGIC appears as in Figure 9-4.

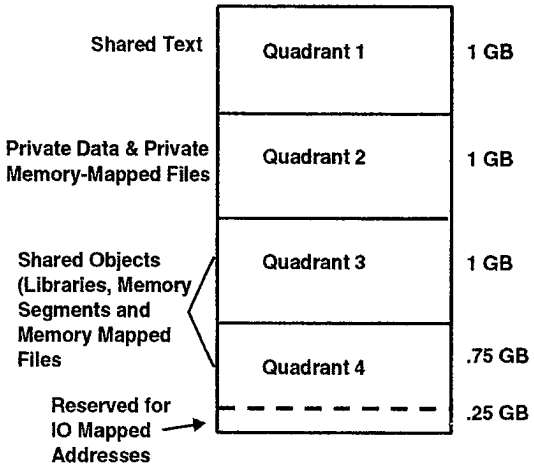


Figure 9-4 HP-UX VAS SHARE_MAGIC Format for 32-Bit HP-UX

In these versions, Quadrants 3 and 4 are globally allocated. Shared libraries, shared memory mapped files and shared memory can go anywhere within Quadrants 3 or 4, but no single segment can cross the boundary between them. This limits any one segment size to a 1 GB maximum on 32-bit HP-UX or 4 TB maximum on 64-bit HP-UX. Note that shared objects are mapped globally in HP-UX 10.x. Therefore, the total size of all shared objects must fit within Quadrants 3 and 4.

Note also the following limitations with the 32-bit SHARE_MAGIC format:

- Text is limited to 1 GB.
- Data is limited to 1 GB.
- Text is shared.
- Text is read-only.
- Text and data are demand paged in.
- Swap space is reserved only for the data.

In HP-UX 11.0 64-bit mode, the SHARE_MAGIC format is as shown in Figure 9-5.

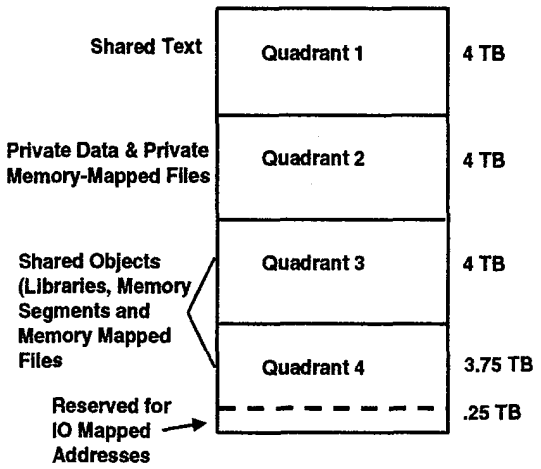


Figure 9-5 HP-UX VAS SHARE_MAGIC Format for 64-Bit HP-UX

While 64-bit architecture allows a far greater number of addresses, only a subset of the possible addresses within the 4 TB are normally used. The actual number of addresses used is constrained by the total swap space available.

9.2.2 DEMAND_MAGIC

DEMAND_MAGIC is functionally identical to SHARE_MAGIC, except for the fact that page alignment is guaranteed between memory and disk. A consequene of this alignment is that only exact page size I/Os are needed between memory and disk images. Very few users take advantage of this feature. Other characteristics and limitations of SHARE_MAGIC also apply to DEMAND_MAGIC.

9.2.3 E

The EXEC
9.01 on the Serie
GB, as shown in

In the 10.0
Also, text (code)
is writable, swap
which swap spa
wasteful, since
entirely at *exec()*
need all the code

Because o
is neither needed

9.2.4 S

Shared me
must be placed i
32-bit applicatio
systems, especia

9.2.3 EXEC_MAGIC

The EXEC_MAGIC format was introduced with HP-UX 10.0 on the Series 800 (HP-UX 9.01 on the Series 700). In this format, the data segment and private text segment could exceed 1 GB, as shown in Figure 9-6.

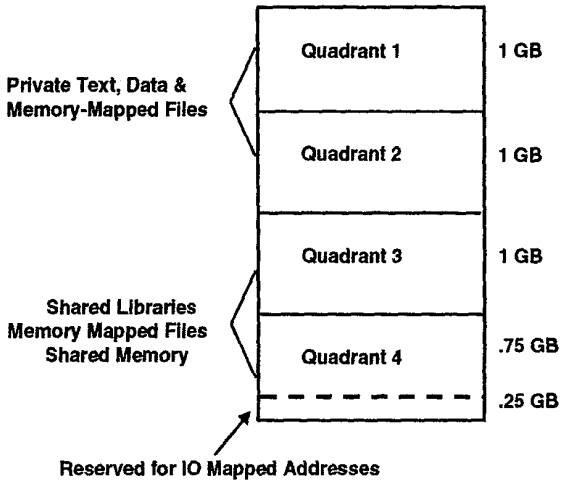


Figure 9-6 HP-UX VAS EXEC_MAGIC Format at HP-UX 10.0

In the 10.0 (and 11.0 32-bit) implementation, text and data together cannot exceed 2 GB. Also, text (code) is private, which means that there can be multiple copies in memory. Since text is writable, swap space must be reserved for it, although a lazy swap allocation scheme is used in which swap space is allocated only for text pages that are modified. Even this lazy swap is wasteful, since virtually no modern application uses modifiable code. Also, text is loaded entirely at *exec()* time rather than being paged in. This loading is inefficient, because you seldom need all the code in memory.

Because of the vastly larger quadrant size available on 64-bit processors, EXEC_MAGIC is neither needed nor allowed for 64-bit applications.

9.2.4 Shared Memory Windows for SHARE_MAGIC

Shared memory is normally a globally visible resource to applications. All shared objects must be placed in memory quadrants 3 and 4; therefore, the total size of all shared objects that 32-bit applications can access is 1.75 GB. This may be acceptable on smaller systems, but large systems, especially those where the total size of shared memory segments for different applica-

tions exceeds 1.75 GB, or those that are used for large databases, may need a much larger shared memory segment.

For HP-UX 11.0, an extension pack has been released that will allow 32-bit applications to access shared memory in windows that are visible only to the group of processes that are authorized through the use of a unique key. It is expected that this feature will be standard starting with HP-UX 11.10. Shared memory windows provide up to 2 GB of shared object space, which is visible only to the group of 32-bit processes configured to access it with the *setmemwindow(1m)* command. The total amount of shared memory in a shared memory window depends on the magic number of the executable. SHARE_MAGIC executables can use a shared window of up to 1 GB, whereas SHMEM_MAGIC executables can use a shared window of up to 2 GB.

The number of shared memory windows is configurable with the tuneable parameter *max_mem_window*. Each group of applications can access its own private memory window. The shared objects placed in Quadrant 4 remain globally visible. Therefore, HP-UX tries to load all shared libraries into Quadrant 4 when shared memory windows are used. Using memory windows has several side effects:

- Shared libraries that cannot be placed into Quadrant 4 are placed in Quadrant 3 and must be mapped into each shared memory window.
- The IPC_GLOBAL attribute must be used to force a shared memory segment into the shared memory window using shmat(2).
- The MAP_GLOBAL attribute must be used to force a memory-mapped file into the shared memory window using mmap(2).
- Processes must be in the same memory window to share data.
- Child processes inherit the shared memory window ID.
- The shared memory window ID may be shared among a group of processes by inheritance or by use of a unique key referred to by the processes.

The per-process Virtual Address Space (VAS) for processes that use SHARE_MAGIC shared memory windows is shown in Figure 9-7. Use of this feature constrains the globally accessible shared object VAS to .75 GB. This means that all shared libraries, memory-mapped files and shared memory segments that must be accessible to all processes on the system must fit into the .75 GB Quadrant 4. Therefore, this feature should be used with care.

Shared memory windows are not needed, at least today, for 64-bit applications, because the total shared object space may be as much as 3.75 TB.

9.2.5 SHMEM_MAGIC

SHMEM_MAGIC provides a means of extending the VAS available to global shared objects. SHMEM_MAGIC achieves this goal at the expense of the VAS available for process text and private data, which is limited to 1 GB together when using this option. Figure 9-8 shows this format.

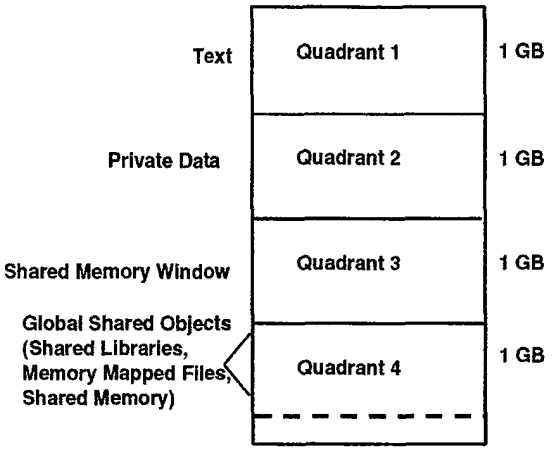


Figure 9-7 32-Bit SHARE_MAGIC Format with Shared Memory Windows

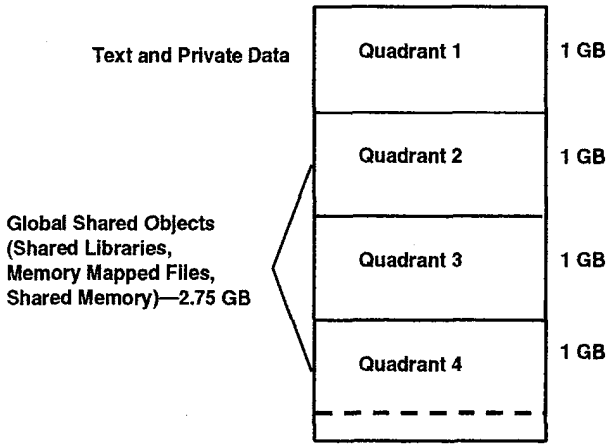


Figure 9-8 SHMEM_MAGIC Format

9.2.6 Shared Memory Windows for SHMEM_MAGIC

In a manner similar to that for SHARE_MAGIC, shared memory windows can be used to allocate VAS for shared memory segments that are visible only to a group of cooperating processes that are linked as SHMEM_MAGIC. Doing so makes 2 GB of VAS available for shared objects that are accessible only to the cooperating processes. Using this feature will constrain the VAS available for globally accessible shared objects to .75 GB, and it should therefore be used only when absolutely necessary. Figure 9-9 shows the SHMEM_MAGIC format with memory windows.

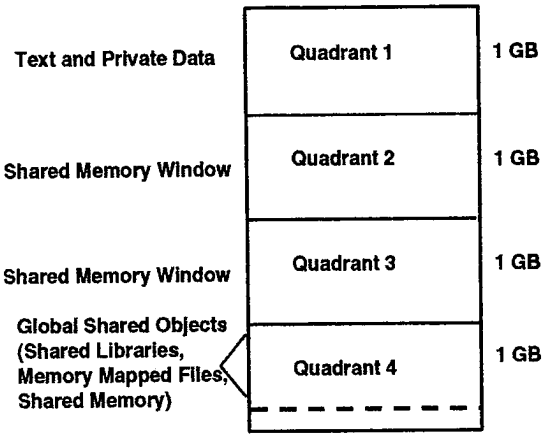


Figure 9-9 32-Bit SHMEM_MAGIC Format with Shared Memory Windows

9.3 fork() and vfork()

Virtual memory bottlenecks may also occur as the result of the *fork()* system call, which creates a new process derived from a currently running one. With *fork()*, the VAS data structures—virtual frame descriptors (VFDs) and disk block descriptors (DBDs)—are copied for use by the child process; this may even require page-outs to provide enough additional memory. Full swap space reservation is made for the child process. Furthermore, copy-on-write is implemented for the parent—that is, the child process receives a copy of the data page before the parent writes. Moreover, copy-on-access is implemented for the child, which means that a fresh copy of data is created whenever the child writes or reads data.

Vfork() allows VAS structures to be shared, which makes it much more efficient than *fork()*. No swap space reservation is necessary for the child. When using *vfork()*, the parent waits, suspending execution, and the child must be coded to call *exec()* immediately; these call-

ing conventions mus
copying the VAS st
saved.

From Bob's C
problem. The
application rar
type a systerr
drive could be
prompt appeai

What I f
neer was callir
resulted in phy
the swap area

When th
ied all of the \
were quite lar
some time, bu
the VAS struc
exec() to start
copied and cre

The solu
fork() call to a
were no longe

9.4 Dynamic B

The use of th
necks can be obser
second and successi
size of the file syste

A dynamic bu
both set to zero; this
fault requests, whicl
vhand reclaims page
syncer process atten

An advantage
memory they occup
most sense in scient
intensive virtual me

Dynamic cacl
other applications w

ing conventions must be used. *Vfork()* saves the resources of CPU time and required memory for copying the VAS structures. The larger the process, the greater the amount of overhead that is saved.

From Bob's Consulting Log— I was called in to diagnose a workstation performance problem. The customer was running a large engineering design application. The application ran well until the engineer used a feature that allowed a shell escape, to type a system command. Performance suddenly degraded significantly: the disk drive could be heard performing I/Os for about 90 seconds, and then, finally, the shell prompt appeared.

What I found was that the system had only 128 MB of memory, and the engineer was calling up a model of a satellite that took over 150 MB of data space. This resulted in physical memory being totally filled, plus some of the data being written to the swap area.

When the engineer did a shell escape, the application called *fork()*, which copied all of the VAS structures associated with the application. These VAS structures were quite large, because of the 150 MB of data space. The copying not only took some time, but it also caused paging out to occur to make room for the new copy of the VAS structures. However, to do the shell escape, the application now called *exec()* to start up the shell. The *exec()* threw away the VAS structures that were just copied and created new ones for the shell process!

The solution was to convince the third party software supplier to change the *fork()* call to a *vfork()* call. When they made this change, the large VAS structures were no longer copied, and the shell prompt appeared in less than a second.

9.4 Dynamic Buffer Cache

The use of the file system buffer cache is another important area where memory bottlenecks can be observed. The storage of pages in buffers means quicker access to the data on the second and successive reads, because they do not have to be read in again. It is possible to set the size of the file system buffer cache, and the cache can be set up as either dynamic or static.

A dynamic buffer cache is enabled when the system parameters *bufpages* and *nbuf* are both set to zero; this is the default for HP-UX 10.0. The buffer cache grows as the result of page fault requests, which in turn result in pages being added. The cache shrinks as the page daemon *vhand* reclaims pages and *syncer* trickles out dirty buffers. (Trickling is a process by which the *syncer* process attempts to avoid large spikes of I/O by writing out pages in small groups.)

An advantage of the dynamic buffer cache is the fact that when buffers are not in use, the memory they occupy can be reclaimed for other purposes. The use of dynamic caches makes the most sense in scientific and engineering applications, which alternate between intensive I/O and intensive virtual memory demands.

Dynamic caches are not always the best strategy: most database environments and many other applications will run better with a fixed cache size. Growth and shrinkage of the cache may

in fact cause performance degradation, as well as make application performance less predictable.

9.5 Sticky Bit

The *sticky bit* is a mode setting on executable files indicating that shared code is to be paged or swapped out to the swap area. Under these circumstances, startup of the executable may be faster from the swap area than from the *a.out* file itself, which is beneficial for frequencly executed programs like *vi*. However, text pages are almost always shared, non-modifiable, and merely deallocated when memory pressure occurs. In current implementations of the sticky bit (10.0 and later), the bit is honored when *a.out* is remote (for example, when it is mounted across an NFS mount), when a local swap area is present, and when the *page_text_to_local* parameter is enabled. This may be useful in distributed environments where programs are executed remotely.

9.6 Memory-Mapped Files and Semaphores

An alternative to the use of the file system buffer cache for file I/O is the use of memory-mapped files. The *mmap()* system call creates a mapping of the contents of a file to the process's virtual address space. Private memory-mapped files are mapped into the Private Data space; shared memory-mapped files are mapped into shared quadrants. With memory-mapping, I/O is not buffered through the buffer cache but is page-faulted in and paged out. The backing store is the original file, and *vhand* writes the pages to the original file, not the swap area.

The *madvise()* system call can be used to specify random or sequential access to a memory-mapped file; sequential access results in clustered reads. Starting in HP-UX 10.0, the kernel clusters the reads if sequential access is detected.

Advantages (+) of using memory-mapped files include the following:

+ After setup, data is accessd by pointer, so no system calls (such as *read()* and *write()*) are used to access the data; thus there may be fewer context switches.
+ Dirty data can be flushed by request. Reads cause page faults unless they are already in memory, and page-outs of dirty data directly to the file are initiated by *vhand*.
+ The data is not double buffered (meaning that it is stored only once in memory, in the process address space), and no swap space is allocated for the pages associated with the memory-mapped files. When memory-mapped files are not being used, data is stored in both the process VAS and in the buffer cache.

The use of memory-mapped files does not necessarily mean better performance. Here are some of their disadvantages (-):

- They require significant coding changes to the application. The most appropriate use is when you have a lot of data and do not want the swap area to be too large.

- Memory-mapped pages cannot be locked into memory, and are not trickled out to the disk by *syncer* as is the case with the buffer cache. Instead, they are written all at once.
- There may be protection ID fault thrashing.

Table 9-1 shows the maximum file size that can be mapped as well as the maximum combined size of all shared mapped files, all shared memory, and all shared libraries.

Table 9-1 Memory Mapped File Size Limits

	32-bit systems	64-bit systems
Maximum memory-mapped file size	1 GB	4 TB
Combined maximum size for shared mapped files, all shared memory, and all shared libraries without shared memory windows	1.75 GB	7.75 TB

Use of memory-mapped files requires considerable care. Concurrent access to the same file by traditional file system calls and by memory-mapping can produce inconsistent data at best and corrupted data at worst. The user cannot specify the mapped address range, and only one fixed contiguous mapping of a shared mapped file can exist; no address aliasing is possible. Even if more than one process is accessing the file, both processes must map the file in the same way. Separate calls to *mmap()* for the same file might result in non-contiguous virtual addresses, depending on what occurred on the system between calls; this may cause application problems. Finally, extending a memory-mapped file might result in an ENOMEM error to the application if global shared memory space is full.

9.6.1 Memory-Mapped Semaphores

Memory-mapped semaphores may be used with memory-mapped files or with an anonymous memory region created with *mmap()*. These semaphores require additional memory (at least one page), although multiple semaphores may be located on the same page. Memory-mapped semaphores are binary (set or clear) instead of counting. They are managed with the *msem_init()*, *msem_lock()*, *msem_unlock()*, and *msem_remove()* system calls. Although memory-mapped semaphores consume more memory than do traditional System V semaphores, they are much more efficient. Their use will usually improve both application and system performance.

Memory-mapped semaphores are mentioned here for consistency and completeness, since they are implemented as part of the memory-mapped file implementation. Their employment in a user process is really an application design or optimization choice that may improve CPU utilization. Application tuning is discussed in Part 4.

9.7 Shared Libraries

Shared libraries are used by default on HP-UX systems. With the use of shared libraries, the library on the disk is not made a part of the *a.out* file, and memory or disk space is saved as a result. However, shared libraries may consume more CPU and require more I/O activity.

With shared libraries, deferred binding is the default. Binding is done upon the first call to a procedure in the library. This means that unresolved symbols may not be detected until *exec()* time. Shared libraries are compiled as position-independent code (PIC), which results in reduced performance compared with executables linked with archive libraries. This is because the code is bigger, and uses more CPU. Also, shared libraries cannot be locked in memory, and swap space is reserved for data required by every procedure in the library, even those that are not called.

Shared libraries are favored by software vendors because updates to the shared library do not require relinking of the *a.out* file. They are favored from a system perspective because they consume less memory and disk space; however, they consume more CPU. Performance tradeoffs with shared libraries are discussed further in the chapter on "Application Optimization."

9.8 Paging, Swapping, and Deactivation

Unix systems use a variety of strategies for handling high demand on memory resources. These include paging, swapping, and deactivation.

9.8.1 Paging

Paging is the process by which memory pages are brought into memory and removed from memory. Various algorithms for paging have been used in different HP-UX systems. *Page-ins* occur when a process starts up, when a process requests dynamic memory, and during page faults after a page-out, a swap-in or a reactivation. Page-ins are always done as needed. Code that is never executed never gets paged in unless the program is linked as an EXEC_MAGIC program.

Page-outs and *page-frees* occur when memory is scarce. The page daemon *vhand* (further described below) does page-outs only for dirty data pages; text (code) pages and unmodified data pages are simply freed.

9.8.2 Operation of vhand

Vhand, also known as the page daemon, is the system process that manages the paging out and freeing of data pages in a dynamic buffer cache. The name *vhand* was suggested by the two parts ("hands") of the daemon. The *age hand* cycles through memory structures, clearing the Recently Referenced bit in the PDIR (page directory) and flushing the TLB entry. The *steal hand* follows, freeing or paging out those pages that the age hand has cleared, and which the application has not accessed since the time the Recently Referenced bit was cleared. See Figure 9-10.

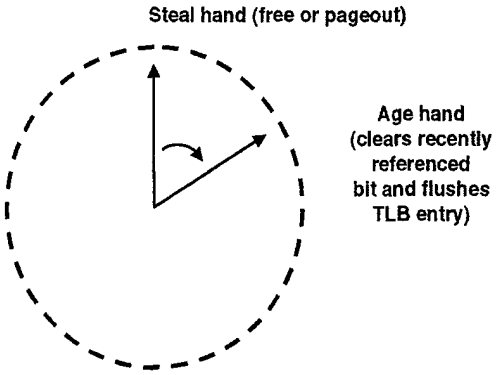


Figure 9-10 Two-Handed Clock Algorithm of vhand, the HP-UX Page Daemon

The TLB must be flushed to force a page fault, even though the page may still be in memory. Memory pressure has a negative impact on CPU utilization, because the CPU must deal with more TLB misses, page faults, and other memory management housekeeping.

9.8.3 Swapping

Swapping is an event that occured with user processes in HP-UX systems earlier than 10.0 when memory was very scarce or when the virtual memory system was thrashing. The algorithm for swapping depended on paging rates and on the amount of free memory in the system. In *swapping out*, the entire process, including private data, the u-area, and the *vfd/dbd* data structures, were written to the swap space in a series of large (up to 256 K) I/O operations. Shared text (code) was not swapped, but freed, and the process was not removed from the run queue. The opposite process, *swapping in*, required a large number of page faults to bring back code and data as the process started executing again.

Since a swapped process remains on the run queue, its priority will soon be improved to the point where it is the highest priority process again, and then it will then be swapped back in. This potentially starts the severe memory pressure again as the process that was swapped (and probably had a very large RSS) executes again, page faulting in its pages.

The swap area is used for both paging and swapping (pre-10.0). It is the backing store for private data, pageable process structures, shared memory, private text (executables with the `EXEC_MAGIC` format), and shared text (where the sticky bit is set and the executable is remotely accessed). *Swapper* is the process responsible for swapping processes out. Swapping is no longer implemented as all-or-nothing, but deactivated processes are the first to be paged out.

9.8.4 Deactivation

Swapping out has a tremendous negative impact on the system. Because of this, "swapping" has been implemented through *deactivation*, starting with HP-UX 10.0. Deactivation occurs when memory is very scarce, or when the virtual memory system is thrashing. The algorithm determines when deactivation is needed based on paging rates, the number of running processes, the amount of CPU idleness, and the amount of free memory.

In deactivation, a user process may be removed from the run queue for up to 20 minutes. Process structures (*u_area*) are written to the swap area after all the pages have been paged out. A candidate for deactivation is chosen based on the process size, priority, and time in memory, as well as on whether or not the process is interactive, whether or not it is serialized, and whether it is running or sleeping. *Glance* shows deactivation and reactivation statistics, but there is no utility that can give a list of processes that are currently deactivated. *Sar* and *vmstat* continue to refer to "swapping," although the term now means deactivation. The process *swapper* is now responsible for deactivating processes rather than for swapping them out.

The biggest advantages (+) of deactivation over swapping, from a system perspective, are:

+ Deactivation causes pages to be trickled out to the disk by *vhand* rather than all at once with multiple large I/Os.
+ With deactivation, the process stops executing for a while, so it does not soon cause its pages to be brought in again.

Of course, the user of the deactivated process may not like having to wait up to 20 minutes before forward progress continues.

9.8.5 Serialization

Serialize() is a system call and command that was introduced with HP-UX 10.0. It can improve performance when the overhead for paging in and out would be excessive. The use of the *serialize()* call provides a hint to the memory management system that the process is large, and that throughput will probably increase if the process is run serially with respect to other "serialized" processes. Serialization has no effect on processes that are not serialized.

Serialize() lets a process run for up to one hour before another serialized process is allowed to run; it is effective only when there is a shortfall of memory and when there are several serialized processes. There is no tool that shows you whether or not a process has been serialized.

The following is an example of using *serialize()*.

From Bob's Consulting Log—Five engineers were running a compute-bound application that used as its input a very large data set, each engineer supplying a different data set. I/O was only done at the end to write out the results. Normally, the five copies of the application would run concurrently and be timesliced, causing forced context switches. Accessing five very large data sets caused such severe memory

pressure
for one c
Ea
time. Aft
total exe
the last
overhea

9.9 Memor

The mem
consistent poli
gpgslim—are
A descri
Table 9-3 on p

Table 9-2

Parameter
lotsfree
desfree
gpgslim
minfree

9.9.1 F

The HP-U
handling or regi
that are all of a c

pressure that the applications actually ran for a longer time than 5 X the average time for one copy because of this additional overhead.

Each user saw consistent slow performance and roughly the same execution time. After serializing these processes, the users saw inconsistent performance and total execution times. The first process would execute in one fifth the average time; the last process would execute in 5 X the average time. The important thing is that *overhead on the system had been significantly reduced.*

9.9 Memory Management Thresholds

The memory management system within HP-UX uses a variety of thresholds to enforce a consistent policy for memory management. Some of these thresholds—*lotsfree*, *desfree*, and *gpgslim*—are used in relation to *freemem*, the amount of memory currently free on the system.

A description of the basic HP-UX memory management parameters is in Table 9-2. Table 9-3 on page 197 shows how the default values of the variable parameters are calculated

Table 9-2 HP-UX Parameters for Memory Management

Parameter	Tunable	Description	Comment
lotsfree	Yes	Upper bound where paging starts and the threshold at which paging stops	The default is a variable number of pages based on physical memory size.
desfree	Yes	Lower bound where paging starts	The default is a variable number of pages based on physical memory size.
gpgslim	No—dynamic	The current threshold between *lotsfree* and *desfree* where paging actually occurs	Default = (*lotsfree* + 3**desfree*). Recalculated every time *vhand* runs based on how often *freemem* = 0.
minfree	Yes	Threshold where deactivation occurs. Any process is chosen. VM system is thrashing and cannot keep up to provide enough free pages.	The default is a variable number of pages based on physical memory size.

9.9.1 Regions and Pregions

The HP-UX 10.0 and later memory management policy also has positive effects on the handling or regions and pregions. A *region* is a collection of pages belonging to all processes that are all of a certain type—for example, text, private data, stack, heap, shared library text, and

shared memory segments. A *pregion* is a collection of pages belonging to a particular process that are all of a certain type. In HP-UX since version 10.0, the following policies are used:

- All regions are treated equally no matter the size.
- Shared regions are not more likely to be aged.
- All pages of a pregion are eventually scanned.

Pages belonging to lower priority ("niced") processes are more likely to be aged and stolen; pages belonging to higher priority processes are less likely to be aged and stolen. Processes blocked for memory are awakened in CPU priority order rather than in FIFO order, with interactive processes usually being favored. Page-ins are clustered unless too little memory is available. Page-ins cause process blocking as available memory approaches zero (amount depends on process priority). Finally, the buffer cache can shrink as well as expand.

9.9.2 Thresholds and Policies

Memory management thresholds vary based on the amount of physical memory and CPU speed, and are set at boot time. The value of *gpgslim* floats between *lotsfree* and *desfree*, depending upon demands on memory. When *freemem* < *gpgslim*, *vhand* runs eight times per second and scans a set number of pages (depending on need and swap device bandwidth) and uses no more than 1/16 of a particular pregion at a time and no more than 10% of the CPU cycles for that interval. Each time *vhand* scans a pregion, it starts scanning pages at the point where it left off the previous time. The *nice* value affects the probability that a page will be aged. When *freemem* < *minfree*, *swapper* runs to free up large blocks of memory by deactivating processes. (Although the name is still *swapper*, HP-UX no longer swaps.)

Starting with HP-UX 10.20, *lotsfree*, *desfree* and *minfree* are tuneable. However, unless you really understand the needs of the application and how it is affected by these parameters, it is highly recommended that you accept the default values. In 11.x and later versions, the default values for *lotsfree*, *desfree*, and *minfree* have been adjusted, especially for systems with large amounts of physical memory (> 2GB). This was done because it is much better to start paging sooner on such systems, so that the paging process can meet demands more effectively.

9.9.3 Va

In these sam
N is the number o

Table 9-3 Cal

Parameter	
lotsfree	M
desfree	M
minfree	M

9.10 Sizing M

Choosing th
good memory per
lar installation, bu

9.10.1 Si:

For the swa
memory, with a m
large memory co
75% of available
physical swap spa
When physi
25% of physical r

- The sum of
  and shared l
  segment is l
- N times the
  N = the nun
- The sum of
- 10% overhe

Beyond this

9.9.3 Values for Memory Management Parameters

In these sample calculations for the default values of the memory management parameters, N is the number of non-kernel free pages at boot time.

Table 9-3 Calculations for Default Values of Memory Management Parameters

Parameter	N <= 8K and physical memory size is 32 MB	8K < N <= 500K and physical memory size is 2 GB	N > 500K and physical memory size is 2 GB
lotsfree	MAX (N/8, 256)	MAX (N/16, 8192)	16384 [64 MB]
desfree	MAX (N/16, 60)	MAX (N/64, 1024)	3072 [12 MB]
minfree	MAX (*desfree*/2, 25)	MAX (*desfree*/4, 256)	1280 [5 MB]

9.10 Sizing Memory and the Swap Area

Choosing the right memory size and configuring the right swap area size can contribute to good memory performance. Only experience can determine the right values are for any particular installation, but some initial guidelines are provided in the next paragraphs.

9.10.1 Sizing the Swap Area

For the swap area, the old rule of thumb was to use two to three times the size of physical memory, with a minimum of 1 times physical memory. However, this is not always realistic. For large memory configurations, the use of *pseudo-swap* in addition to normal swap allows up to 75% of available memory to be used once the swap devices are full, without the need to reserve physical swap space.

When physical memory size is greater than 512 MB, a more realistic guideline is to use 25% of physical memory as a minimum, plus the following:

- The sum of all shared memory requirements (not including text, memory mapped files, and shared libraries) minus the amount of locked memory. (Note: if the shared memory segment is locked into memory, do *not* count it.)
- N times the private virtual memory requirements for each application (private VSS) where N = the number of users; use glance (Memory Regions) to calculate this for each process.
- The sum of shared text VSS requirements when accessing remotely with the sticky bit set
- 10% overhead for VAS structures and fudge factor

Beyond this, pseudo-swap should allow for peak periods.

9.10.2 Sizing Memory

The following determine physical memory size:

- The sum of all resident shared memory requirements (text, shared libraries, shared memory, memory-mapped files), including the amont of locked memory (shared RSS)
- N times the private resident memory requirements for each applcation (private RSS) where N= the number of users
- 10 to 20 MB for the kernel and static tables
- The size of the fixed buffer cache, if applicable
- Initial allocation for the dynamic buffer cache, if applicable (a minimum of 10% of physical memory is required; 20% is recommended)
- An estimate for networking needs (10% of physical memory)
- Additional memory for NFS

9.10.3 Controlling Memory Allocation with PRM

On HP-UX 10.20 and later systems, Process Resource Manager (PRM), working with the standard memory manager, lets you allocate memory amounts or percentages independent of CPU allocations. Process groups are guaranteed a minimum percentage of memory and optionally a maximum percentage. This guarantees a fair share to a process group, but not necessarily to a given process. Shares are enforced when paging is occurring; processes are suppressed by the requested method (today, only SIGSTOP is available). You can choose to suppress all the processes in a process group, or just the largest.

Some side effects of using PRM for memory allocation are:

- PRM reports available memory—the maximum amount of memory that is available for allocation to user processes.
- PRM does not suppress a process that locks memory; however, use of locked memory will affect other processes in the process group.
- Allocations may interact with CPU allocations in such a way that a process group may not use all of the CPU it is allocated if it cannot use any more memory.
- If the PRM memory daemon dies unexpectedly, processes will remain suppressed until prmrecover is used.
- Process groups will exceed their allocations, even with a cap, in the absence of memory pressure.

9.11 Memory Metrics

A variety of global and per-process metrics are available for identifying potential bottlenecks in memory.

9.11.1 Gl

Global mer
memory system a

- Free memo
- Active virtu
- Available n

The most useful

9.11.2 Gl

The only q
Ware gives this a

9.11.3 O

Other glob

- Page-in/pa
- Page-in/pa
- Swap-in/pa
- Swap-in/pa
- Deactivatic
- Deactivatic
- Number of
- Number of

The following gl

- Page-out ra
  occur only
- Deactivatic
  sure, and w

9.11.4 P

Per-proces
dent Set Size (R
percentage block

- Number of
- Number of

9.11.1 Global Memory Saturation Metrics

Global memory saturation metrics (provided by *glance*, *gpm*, and *vmstat*) tell whether the memory system as a whole is saturated. These include:

- Free memory in KB or pages
- Active virtual memory (avm) in the last 20 seconds
- Available memory (physical memory, kernel memory, fixed buffer cache memory)

The most useful global saturation metric is free memory.

9.11.2 Global Memory Queue Metrics

The only queue relating to memory is the number of processes blocked on VM. *MeasureWare* gives this as a count; *glance* and *gpm* show this as a percentage of time blocked on VM.

9.11.3 Other Global Memory Metrics

Other global metrics include:

- Page-in/page-out rate
- Page-in/page-out quantity
- Swap-in/page-out rate (before 10.0)
- Swap-in/page-out quantity (before 10.0)
- Deactivation/reactivation rate (10.0 and greater)
- Deactivation/reactivation quantity (10.0 and greater)
- Number of page faults and paging requests
- Number of VM reads and VM writes (clustered)

The following global metrics are the most useful in diagnosing memory bottlenecks:

- Page-out rate. Page-ins are normal, even when there is no memory pressure. Page-outs occur only when memory pressure exists.
- Deactivations. Deactivations only occur as a last resort when there is severe memory pressure, and when the paging system cannot keep up with demands.

9.11.4 Per-Process Memory Metrics

Per-process memory saturation metrics (provided by *top*, *glance*, and *gpm*) include Resident Set Size (RSS) and Virtual Set Size (VSS). Per-process memory queue metrics include the percentage blocked on VM. Other per-process memory metrics are:

- Number of VM reads and VM writes
- Number of page faults from memory

- Number of page faults from disk
- Number of swaps (before 10.0)
- Number of deactivations (10.0 and later)

Looking at the RSS will show you how much of a process tends to occupy memory. VSS shows you how large the process is, including:

- Text in memory as well as text not yet referenced from the *a.out* file (error routines may never be paged in if not needed)
- Data in memory and data not yet paged in from the a.out file
- Shared libraries in memory and not yet paged in from the .sl file
- Shared memory
- Memory-mapped files in memory and not yet paged in from the original file
- Private data that has been paged out to the swap area
- Shared memory that was not locked, and that was paged out to the swap area

9.11.5 Typical Metric Values

Page-ins occur normally, and thus do not indicate memory pressure. However, page-outs are an indicator of memory pressure. Page-outs of the following can cause pressure:

- Process data pages
- Process text pages for EXEC_MAGIC format executables
- Shared memory pages
- Writes to memory-mapped files (MMFs)
- Shrinkage of the dynamic buffer cache

Swapping or deactivation is an indicator of severe memory pressure.

9.12 Types of Memory Management Bottlenecks

What are the symptoms of a memory bottleneck? The four major ones are:

- Saturation of memory
- A large VM queue
- Resource starvation
- User dissatisfaction with response time

Saturation is indicated by low free memory, and by process deactivation (swapping in systems before 10.0). A large VM queue sustained over time is also indicated by a high percentage of processes blocked on VM, as well as by large disk queues on swap devices. Resource starvation occurs when a high percentage of CPU utilization is used for VM activity, or when the disk

subsystem is consumed by VM activity. User dissatisfaction with the system results from poor transaction response time.

Lack of memory often results in other problems with the CPU and disk systems, and these problems tend to mask the true cause, which lies inside the memory-management subsystem.

From Bob's Consulting Log—One client had recently added 20% more users to an OLTP system, and performance degraded significantly compared to the state before adding the users. I was asked to recommend a CPU upgrade. On investigation, we found that the degradation of performance was much more severe than what you would expect for the number of users being added. I looked at how much memory each new user needed and found that the new users increased the memory beyond what was physically in the system. Memory was thrashing, and performance was degrading much more than expected. The actual solution to the problem—a memory upgrade—turned out to be a lot less expensive than the CPU upgrade the client thought he needed.

9.13 Expensive System Calls

The most expensive system calls from the standpoint of memory are *fork()* and *exec()*, *malloc()*, and *mmap()*. *Fork()* and *exec()* require extensive new memory allocation for VAS/pregion structures; *vfork()* offers a partial remedy (see the previous section on *vfork()*).

Malloc() and *mmap()* also are expensive calls simply because they are likely to increase memory utilization substantially.

9.14 Tuning Memory Bottlenecks

As with CPU bottlenecks, there are several ways of tuning a memory bottleneck:

- Hardware solutions
- Software solutions
- Application optimization
- Adjusting memory-related operating system tunable parameters

9.14.1 Hardware Solutions

The simplest hardware solution may be to increase the amount of physical memory. Another strategy is to use multiple interleaved swap devices if not enough physical memory can be used to prevent page-outs.

9.14.2 Software Solutions

Typical software solutions include the following:

- On small systems, reduce the size of the kernel (subsystems and tables).
- Carefully reduce the size of the fixed buffer cache.

- Use a dynamic buffer cache, and tune it carefully.
- Reduce and/or restrict the use of memory locking by defining the system parameter unlockable_mem.
- Use privileges (see *setprivgrp(1m)*) to regulate user access to memory.
  - Use the `MLOCK` privilege to lock processes into memory.
  - Use the `SERIALIZE` privilege on large processes and batch processes.
- Nice less important, large, or batch processes.
- Move work to other time periods, or run them as batch jobs.
- Reduce the number of workspaces in a VUE environment.
- Restrict maximum process size by setting the following parameters:
  - *maxdsiz*
  - *maxssiz*
  - *maxtsiz*

  Keep in mind that setting these values affects all processes on the system.
- Switch from hpterm to xterm or dtterm.
- Use the sticky bit for NFS-mounted executables (Doing this requires setting the `PAGE_TEXT_TO_LOCAL` parameter).
- Use *setrlimit(2)* starting in HP-UX 10.10.

It is recommended that database shared memory segments be locked into memory. These segments are caches, and it makes no sense to allow a portion of a cache to be paged out.

9.14.3 Application Optimization

Here are some suggestions for optimizing applications to best use memory:

- Minimize the use of expensive system calls:
  - Switch from *fork()* to *vfork()* if appropriate
  - Minimize the use of *mmap()*
- Use memory leak analysis software (for example, *Purify* from Rational Software).
- Use malloc() carefully, because it allocates memory in such a way that the virtual space cannot be returned to the system until the process exits. Using free() releases memory only at the process level; such memory is still considered to be in use by the system. Also, watch for malloc pool fragmentation.
- Minimize the use of resources that consume memory indirectly, such as user-space threads and semaphores.

9.15 Memory-Related Tunable Parameters

The following memory-related parameters may be tuned. These are found in the file */usr/conf/master.d/**. Items marked with an asterisk (*) are discussed in more detail in Chapter 10 on "Disk Bottlenecks."

- *bufpages* *
- *dbc_max_pct* *
- *dbc_min_pct* *
- *desfree*
- *lotsfree*
- *maxdsiz*
- *maxssiz*
- *maxswapchunks*
- *maxtsiz*
- *maxusers*
- *minfree*
- *msgmax, msgmnb*
- *nbuf* *
- *nclist*
- *netmemmax*
- *nfile* *
- *ninode* *
- *nproc*
- *page_text_to_local*
- *strmsgsz*
- *swapmem_on*. Should be enabled to reduce the amount of swap space required for large memory systems (greater than 512 MB).
- *unlockable_mem*. Can be used to limit the amount of memory that can be locked by processes.

While most of these parameters have only a small effect on memory utilization, system tables should not be sized arbitrarily large.

It is highly recommended that *desfree*, *lotsfree*, and *minfree* not be tuned unless HP support specifically tells you to do so.

For the buffer cache, use either *bufpages* to create a fixed size buffer or *dbc_max_pct* and *dbc_min_pct* to create a dynamic (variable size) buffer cache. These parameters will be discussed further in Chapter 10.

9.15.1 Logical View of Physical Memory Utilization

Figure 9-11 shows a logical summary of the components of physical memory that must be managed in performance tuning.

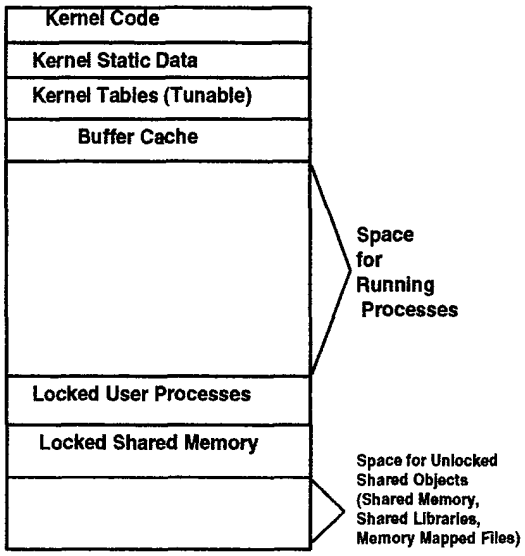


Figure 9-11 Logical View of Physical Memory Utilization in HP-UX

Disk B

HP-UX disk I/
followed by a d
tuning. Here ar

- Review o
- Logical v
- Disk arra
- SCSI acc
- File syste
- File syste
- Disk met
- Types of
- Expensiv
- Tuning di
- Database
- Disk relat

Dealing v
ing always invo

10.1 Revie

The Unix
ing from or wri