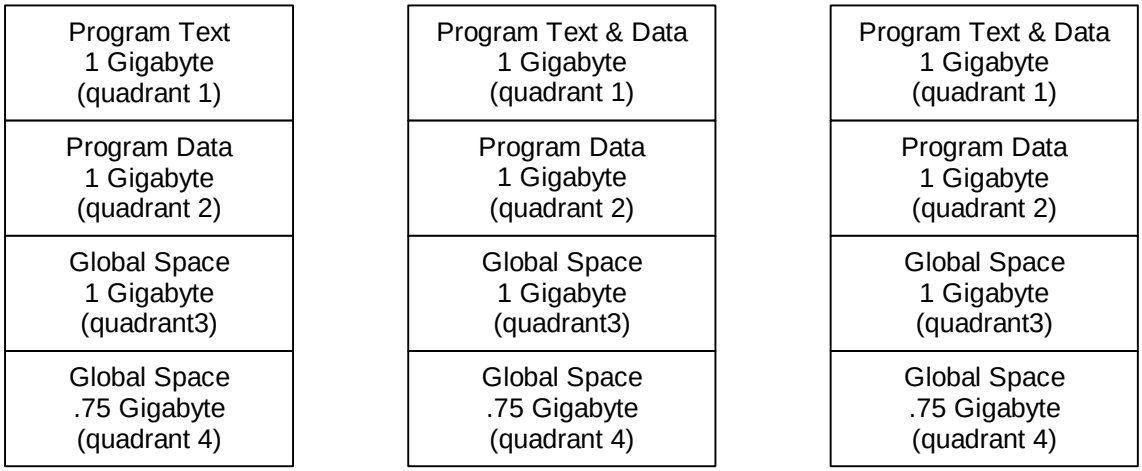
Memory Windows White Paper

1. Objectives

HP-UX release 11.0 is the first to support the new HP V-class machine. Targeted for high end OLTP, decision support, server consolidation and ERP, the V-class can support up to 16 gigabytes of physical memory. The 32-bit virtual address space limits applications from taking full advantage of systems with a large amount of physical memory.

Not all of HP's software providers have ported their applications to 64-bits, nor is it necessary in all cases. But 32-bit virtual addressing has limitations for shared resources on 32-bit applications. All applications in the system are limited to a total of 1.75 gigabytes of global space (shared memory), 2.75 gigabytes if compiled, linked as EXEC_MAGIC and chatr'd as SHMEM_MAGIC. In a system with 16 gigabytes of physical memory, only 1.75 can be used for shared resources such as shared memory.

To address this limitation a functional change has been made to the 11.00 release of HP-UX. This feature allows 32-bit processes to create unique Memory Windows for shared objects like shared memory. There are three types of executables in HP-UX. Their layout *WITHOUT* Memory Windows is the following:



The default HP-UX memory management is based on quadrants where each process has its own space. The process text (executable code) is mapped to quadrant 1 and its private data is mapped to quadrant 2 in its individual space. Quadrant one and two exists n times (once for each process). All shared objects are mapped to quadrant 3 and 4 (and quad 2 for SHMEM_MAGIC) of one single space shared by all processes. This is where the system wide limit of 1.75 GB's comes from. Each quadrant is 1gigabyte in size. The 4th quadrant is only .75 gigabytes in size because the last 1/4 of the quadrant is reserved for architected I/O space.

By default all executables are SHARE_MAGIC, the default mode. EXEC_MAGIC and SHMEM_MAGIC require special binaries.

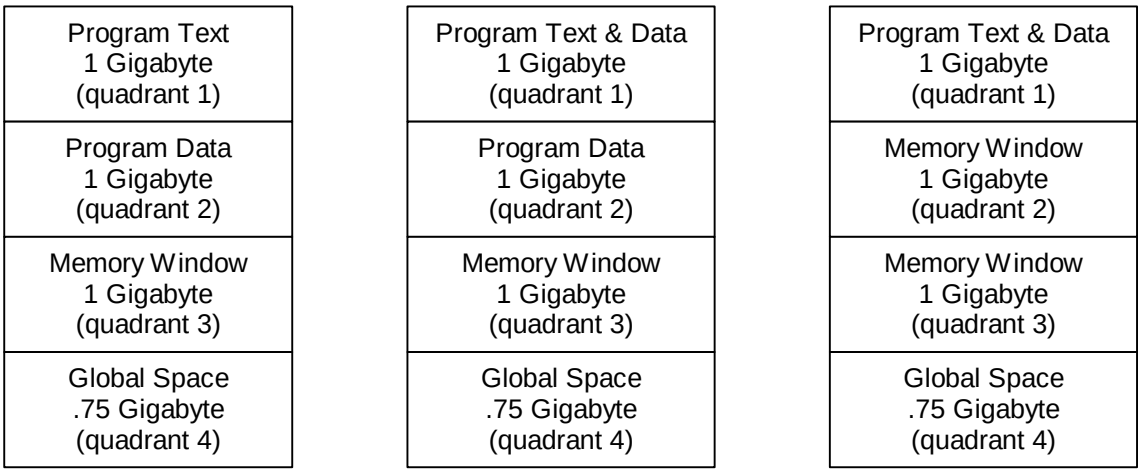
If no special options are given to the compiler/linker, the default executable format is SHARE_MAGIC. Text is shared for all copies running on the system and data is private. EXEC_MAGIC is the output of using the -N option to the linker. It results in the data starting immediately after the text allowing for a greater sized data space (approximately 1.9 GB's). SHMEM_MAGIC combines all the pieces of a process into the first quadrant (text, data, and stack). SHMEM_MAGIC was done at a time when only 32-bit kernels were supported. Applications requiring more than 1.75 GB's of shared space should recompile as 64-bit and then Memory Windows is not needed.

Unless otherwise noted, the remainder of this paper assumes the default executable format of SHARE_MAGIC.

Memory Windows allow each process to define a unique global space of up to 1 gigabyte of shared memory. Shared memory segments placed in the unique space can only be accessed by processes in the same memory window. But different applications, or distinct instances of a single application, can be placed in different Memory Windows and consume more of the available physical memory within the system.

Memory Windows extends system wide virtual capacity for 32-bit applications only. By extending the virtual capacity, more of the underlying physical memory can be used for shared objects. Without Memory Windows a 16 GB system could only consume a maximum of 1.75 GB's of physical RAM. With Memory Windows this limitation does not exist.

With Memory Windows the executable formats look like:



The default executable's maximum size memory window is 1 gigabyte. Any consumption beyond 1 gigabyte consumes space from the 4th quadrant which is shared across *ALL* processes in the system. This is important, any application within a memory window that uses more than 1 gigabyte of shared memory consumes quadrant 4 resources that are shared by all processes no matter what memory window they occupy.

This allows cooperating applications to create 1 gigabyte of shared resources without exhausting the system wide shared memory in the 4th quadrant. The 4th quadrant remains globally visible to all processes to allowing sharing for objects across different Memory Windows. These objects may be application specific or system dependent such as shared libraries.

Creating unique Memory Windows removes the current 1.75-gigabyte limitation. This Document attempts to answer:

- What are the requirements for Memory Windows?
- Who needs Memory Windows?
- How can I change the number of Memory Windows?
- How do I use Memory Windows?
- What are the drawbacks/limitations?
- What tools/statistics exist for debugging?

2.0 Compatibility Disclaimer

Incorrect use of Memory Windows can lead to application failure. Memory Windows can be applied to any application, but that does not mean the application is able to run in Memory Windows. Documented later in this paper, there are interfaces that may break when used under Memory Windows. Only the application owner or software provider can say whether or not all aspects of the application will function under Memory Windows. Furthermore, the application may have a sophisticated environment requiring a complex change to make sure all the processes are in the correct memory window. Placing random processes in Memory Windows can result in application failure. **HP does not consider this failure as a compatibility failure as only the application owner or software provider can certify how and if the application can run under Memory Windows.** Users of Memory Windows need to determine if their application is supported under Memory Windows. They can do this by contacting Hewlett-Packard and/or the application developer for a support statement regarding the application and Memory Windows.

Memory Windows should only be used by application owners or the software provider should be contacted about any potential issues. The errors seen because of incorrect usage may be subtle and non-easily associated with Memory Windows.

In many cases software providers may have already certified their applications with Memory Windows. Contact HP to see if this is the case and for any white papers specifically written for that application/product.

3.0 Hardware/Software requirements

Any machine running HP-UX (32-bit or 64-bit) and any hardware supporting HP-UX release 11.00 can use Memory Windows. Memory Windows is an extension to the 11.00 release and is installed as a patch. The patch id is PHKL_13810/PHCO_13811 (or superseded equivalent). Future releases of HP-UX will include Memory Windows in the base implementation.

The change itself extends the amount of shared memory beyond the current limitations, so it stands larger configurations are more likely to benefit from this change.

4.0 Who needs Memory Windows?

Memory Windows may benefit customer configurations where the system 1.75 gigabytes shared memory limitation prevents full utilization of available physical memory. In the area of server consolidation this may be the result of placing multiple distinct database instances on a single machine. The SGA(System Global Area) requirements for all those databases may exceed the 1.75-gigabyte limitation.

The same limitation can be true for any software provider application/programs or customer specific applications. The consolidation of these applications onto a single machine may result in the need for greater than 1.75 gigabytes of shared resources(shared memory) for the entire system. Or a single application, such as Informix's XPS, may be designed to take advantage of available resources in the system.

Memory Windows allows the creation of more than 1.75 gigabytes of shared memory, **but it does not extend how much shared resources a single process can create! Default executables are still limited to 1.75 gigabytes and SHMEM_MAGIC executables are limited to 2.75 gigabytes.**

5.0 How can I change the number of Memory Windows?

HP-UX ships Memory Windows disabled. To enable Memory Windows, the kernel tunable parameter **max_mem_window** must be set to the desired number of Memory Windows. Customers can change the value by placing the desired value into their kernel configuration file. The system must be rebooted for the new value to take effect.

For each memory window, the kernel allocates resources to allow for a unique space and the management of objects placed in the memory window. Because spaces are also used by processes for text and data segments, there is a maximum limit on the number of Memory Windows allowed. The value of max_mem_window is not allowed to exceed 8192.

The penalty for specifying a larger value is not zero, but it isn't too expensive either. The limitation is to prevent all the possible spaces from being consumed by Memory Windows and thereby preventing normal processes from starting up and executing.

max_mem_window represents the number of Memory Windows beyond the global default window. Setting max_mem_window to one creates a single memory window to accompany the existing global memory window. With a value of one there are a total of two Memory Windows, one default and one user defined. Setting max_mem_window to two would produce a total of three Memory Windows, the default and two user defined. Setting max_mem_window to 0 leaves only one memory window, the default or global memory window.

6.0 How many windows do I need?

What should the value be? That depends on the application requirements and the applications installed on the system. The new value should be dependent on current application requirements. Each software provider should document how may windows they intend to use.

The number of Memory Windows a system needs depends on the number of applications needing "separation" and the amount of resources needed or consumed by the application. For example, say

there are 3 applications being consolidating to a single V-class machine with 16 gigabytes of memory. Suppose application 1 and 2 are single instances and need .75 GB's of shared memory each. Now suppose the third application is the type where each instance needs its own window. And each instance is expected to consume up to .5 GB's of shared memory. How many windows to configure?

First, application 1 and 2 need a window a piece. This results in a total need of 2 windows. Now lets assume that memory pressure is something we don't want. Assume our budget of physical memory for shared objects from the example is 12 gigabytes. Application 1 and 2 will consume 1.5 gigabytes of that. That leaves 10.5 gigabytes of space left over for application 3. Since each instance wants up to .5 gigabytes, that means approximately 21 instances can run before a lack of physical memory is an issue. This results in total of 23 Memory Windows for the system. Since each application 3 may not take its .5 GB's of space, it would be prudent to allocate a number slightly higher.

The penalty for allocating more Memory Windows than needed is negligible. The difference in kernel resources consumed with 256 Memory Windows versus 100 is quite small when compared to the physical size of the machine. When in doubt, specify a number that comfortably covers the application needs.

7.0 How do I use Memory Windows?

To properly use Memory Windows, all aspects of the application must be properly placed in the correct memory window. However there are cases where simply placing the executable in a memory window does not work.

If through normal execution an executable accesses/shares data between different processes, then all the executables must be in the same memory window.

If two processes communicate via shared memory then they must run in the same memory window or the application may fail. For example, a system is configured with 16GB of physical memory and has five database instances each running in their own memory window. In addition, there is a backup application running on the system that communicates with the each of the databases via shared memory communication buffers.

If the backup application does not support running multiple separate instances, then the backup application may fail. The reason is that when the databases are started within a memory window, HP-UX will first attempt to allocate the shared memory communication buffers in quadrant 3. However, quadrant 3 is only shared within a window and not among all Memory Windows. Therefore, if the backup process daemon is not running in each of the Memory Windows, it will not be able to access the shared memory communication buffers and will fail.

A process joins a memory window by using the setmemwindow command. The command itself is described later.

There are two ways a process can associate with a memory window. They are:

1. Private to process and its children. Only the parent process and its children share the window. This is referred to as **Pure Inheritance**. No other processes in the system can attach to the memory window. The memory window remains active until the process, its children, and all objects created in that window are no longer in use. An example is given later.

A process creates a private window by not specifying a window id to the setmemwindow command.

2. A process creates/joins a window associated with a particular key. Any process can join this memory window as long as it knows/specifies the proper key. This is referred to as **Key Associative**. The window associated with the user key remains active until all processes and all objects created in that window are no longer in use. An example is given later.

   A process associates itself with a particular window by specifying a user key or id to the setmemwindow command.

7.1 Memory Window commands and environment

7.1.1 /etc/services.window (file)

/etc/services.window is only used by **Key Associative** processes. A group of processes wishing to use a common memory window and not started from a common parent node(process) must associate themselves with a unique key. The key allows the separate processes to select the same memory window and access/attach to shared resources in that memory window.

The file **/etc/services.window** is a centrally located file to avoid applications from hard coding id's in startup or control scripts. Id's could be embedded in various control scripts and passed directly to the **setmemwindow** command, but conflicts between applications would be difficult to find and correct. Id's placed in a central repository can be easily changed by the system administrator in the event two applications collide on a user key. While it's easy to change user keys, it may not be so easy to change the associated string. Care must be taken in selecting a unique string for the particular application/product. Either installation scripts or system administrators can update/modify entries in this file.

An example /etc/services.window file

```
#
# /etc/services.window format:
#
# Name          <user_key>

informix            20
oracle      30
sybase              40
```

7.1.2 getmemwindow (command)

getmemwindow is used to extract the user id's/keys from the **/etc/services.window** file given a particular string. Applications use getmemwindow to extract their unique id and pass that id to the setmemwindow command.

syntax: getmemwindow <name>

Given a name, getmemwindow extracts the user key from the /etc/services.window file associated with name. getmemwindow is only needed for key associative processes. getmemwindow is intended for application startup scripts to extract the unique user key from the /etc/services.window file.

Example code snippet:

```
# Some sh/ksh script
#
# This starts the executable “Program” with arg1 and arg2 in the
# memory window associated with the user key in $WinId
#
WinId = $(getmemwindow “informix”)
setmemwindow -i $WinId Program arg1 arg2 ....
```

7.1.3 setmemwindow (command)

setmemwindow starts a particular process in a memory window.

syntax: setmemwindow [-i WindowId] [-ncjf] [-p pid] || program arg1 arg2

setmemwindow provides the ability to change the memory window id of a running process or start a particular **program** in a memory window. **program** is only used if the process id **pid** is either “0” or **-p** is unspecified. If a **pid** is specified and the id is non-zero, only the desired window id is changed in the process, program is ignored.

What does it mean to only change the id? Setting the window for a process does not mean a process immediately attaches or creates objects in a memory window. The targeted process does not begin using the specified window until it exec’s a new image. This is why setmemwindow() for pid “***0***” forks, sets the window id and exec’s **program**. If a currently running process is specified the memory window is not changed until the process exec’s. Any children created by that process inherit the new window id and when they exec() the new window id takes effect.

If setmemwindow exec’s **program**, the default behavior for setmemwindow is to wait until **program** finishes. setmemwindow was intended as a wrapper for an existing executable. If waiting is not desired, there is the -n option to override the default wait behavior.

If **-c** or **-j** is unspecified, the default behavior is to place the process in the window specified by **WinId**. If WinId exists the process is placed in the corresponding memory window. If no memory window exists with WinId, an unused window is allocated and associated with “ WinId”. -c and -j allow the caller to change the behavior. Both are described below.

If **-i** is unspecified the program is placed into a private window. Only **program** and its descendents have access to the memory window. No other processes can *ever* join the memory window.

Options:

- **-i <WinId>**

The value of **WinId** is a key to the desired memory window. WinId is user specified and should be one of the values contained in /etc/services.window. Applications extract the user key from /etc/services.window through the getmemwindow command. The kernel tries to locate an existing window with WinId. If an existing entry is found, that memory window is used. If no window is found then depending on the options specified (-c or -j) an unused entry is located and assigned to WinId.

If the caller does not specify a window id **program** is placed in a private memory window. The first available memory window is located and assigned to the process. Only children created by the program are in the same memory window and no other process can ever join the memory window.

The value "**0**" is special. If "0" is specified the process/program is placed into the default global window instead of a unique window with id WinId. By default all processes are in the memory window "0".

? **-p <ProcessId>**

Change the window in process <**ProcessId**>. If -p is unspecified or the value of pid is "0", the calling process has its window id changed and then program is exec'd. An unspecified pid or "0" requires the caller to specify a **program**. If a non-zero process is specified, only the desired window is changed. No program is exec'd, even if one is specified.

? **-b**

Create a memory window where both memory window quadrants use the same space id. For SHMEM_MAGIC executables this generates two quadrants with the same space id and both quadrants being contiguous. Applications can use this to generate larger contiguous shared memory ranges. An application can generate a 1 Gigabyte shared memory segment (2nd quadrant) then create another 1-Gigabyte segment (3rd quadrant), yielding two contiguous 1-Gigabyte quadrants. To the application it appears to be one single 2-Gigabyte quadrant.

Note: This option only benefits SHMEM_MAGIC applications as they are the only type of executable able to access the 2nd quadrant for shared memory purposes.

? **-c**

Create a window with id **WinId** and attach the specified process to it. If a memory window corresponding to WinId already exists, the call fails.

? **-j**

Join an existing window with id **WinId**. The specified process attaches to an existing memory window. If no window with WinId exists, the call fails.

? **-n**

If program is exec'd, the default behavior is to waitpid(2) for the process to terminate. **-n** causes setmemwindow() to return after starting **program** in the specified window.

? **-f**

If -f is specified, setmemwindow still exec's the defined program if unable to set the memory window. This allows the caller to exec even though the process is not in the desired memory window. This would allow a script to start an application whether or not Memory Windows were present/installed in the system.

Examples:

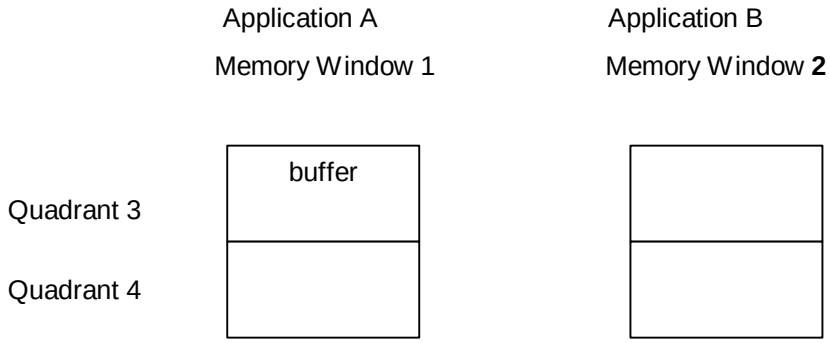
```
# Start a program and all its children in a private memory window
setmemwindow program arg1 arg2

# Start a program in the memory window associated with user key “informix”
WinId = getmemwindow “informix”
setmemwindow -i $WinId program arg1 arg2

# Start a program in the memory window associated with user key “informix” and make sure this
# process is the first one to enter the memory window.
WinId = getmemwindow “informix”
setmemwindow -i $WinId -c program arg1 arg2
```

8.0 What are the drawbacks/limitations?

8.1 Processes must be in same memory window to share data.

Processes wanting to share global data, such as shared memory or MAP_SHARED memory mapped files, must make sure all processes are in the same memory window. If they are in separate Memory Windows and the data is located in the 3rd quadrant, only processes in that memory window can access/attach to the data.

If the condition occurs and a shared object resides in the 3rd quadrant of a memory window and a process in a different memory window tries to attach, the second process fails and receives an error. This is common for all objects occupying the 3rd quadrant, except in the case of shared libraries. For example:



If application B tries to attach to the buffer created by application A the shmat() would fail. That’s because the buffer address is in application #A’s memory window space.

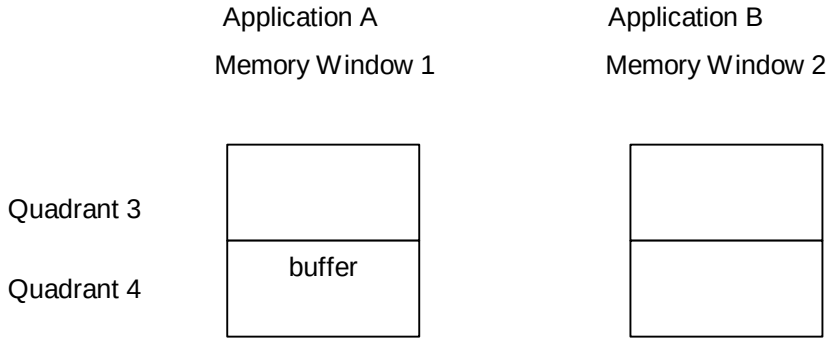
If processes in different Memory Windows wish to share data reliably, the creator of the data must take steps to guarantee the data is placed in a location accessible to all processes.

Shared memory and memory mapped files have introduced two new options to force the creation of objects into locations globally visible across all Memory Windows. For memory mapped files the option is

MAP_GLOBAL. If MAP_GLOBAL is specified, the object is created visible to all Memory Windows. If no space is available, mmap(2) fails.

For shared memory the option is **IPC_GLOBAL**. If IPC_GLOBAL is specified to shmget(2), the shared memory segment that is created is visible to all Memory Windows. If no space is available, shmget(2) fails.

Using the example above the picture if the caller had specified IPC_GLOBAL on the shared memory segment:



Now when application B calls shmat() the call succeeds as buffer is in 4th quadrant where all processes shared the same space.

8.2 Children inherit memory window id.

Children inherit their window id from their parent. In most cases this is the desired behavior, but in some cases it may be unexpected. A program in a non-default memory window may fork and exec a system command, another application, or some utility. The exec'd program may have an assumption about what memory window its in and the caller could have violated that assumption.

This shouldn't be an issue, but if it is a problem an easy solution is to place a wrapper around the failing executable using setmemwindow. Using setmemwindow the executable is placed in the required window. Most likely this will be the default or system global window, for example, "**setmemwindow -i 0 program arg1 arg2**".

8.3 Shared Memory shmget()/shmat()

When Memory Windows is enabled, shared memory objects created via the shmget() interface are placed by default in the processes memory window. Only processes from that memory window are able to shmat() the segment as well. Any processes from different Memory Windows will receive an error.

8.4 Memory Mapped Files mmap().

Memory mapped files are similar to shared memory segments where any object mapped as MAP_SHARED is located in the 3rd or 4th quadrants. If a portion of a file is mapped using the mmap() in

the 3rd quadrant, only processes from that memory window are able to map that file as well. Any processes from different Memory Windows will receive an error.

8.5 Shared Libraries mapped privately.

Shared libraries are just memory mapped files. As such, a shared library is either located in the 3rd or 4th quadrant, the same as a shared memory segment. But unlike a shared memory segment, a process attaches itself to a shared library based on which libraries a process needs. Different applications commonly share the same libraries such as libc(3).

A premise of memory windowing is any process wishing to share an object must be in the same memory window space. If two processes are not in the same memory window and the object they wish to share occupies the 3rd quadrant, a failure occurs for the second process.

This behavior may be ok for a cooperating application, but one application running in one window shouldn't impact a completely separate application running in a completely different memory window. Rather than returning an error and failing the shared library load, HP-UX handles shared libraries and Memory Windows in a special way.

If a shared library is mapped into the 3rd quadrant of the global space and a memory windowed process attempts to load(i.e. map) that library, the process cannot map the library shared, so instead the library is mapped privately. This of course has some drawbacks, for instance private memory mapped files must allocate swap space. But private mapping should be a rare condition. HP-UX prevents this from occurring in most systems by defaulting shared libraries into the 4th quadrant. The 4th quadrant is the same(visible) to all Memory Windows. Only if the 4th quadrant is completely filled are shared libraries mapped in the 3rd quadrant. Other than the swap space consumption, mapping privately should have no effects on the process performance.

The rules for shared libraries are:

a. Shared library tries quadrant 4 first.
b. If quadrant 4 is filled, the library is mapped into the 3rd quadrant if the process is in the global memory window. Otherwise, non-global memory windowed processes map the library privately.

8.6 Posix Message Queues mq_open()

POSIX message queues are built on a special form of shared memory. Therefore message queues are subject to the same constraints as a memory mapped file. If the POSIX message queue is located in the 3rd quadrant (the user has no control of this), only processes in the same memory window can use/access the message queue. By use/access I refer to the sending and receiving of messages.

8.7 Change in Policy allocation

In previous HP-UX releases the location policy for shared objects(shared memory, MMF's, shared libraries) is as follows:

1. First try allocating address from quadrant 4.
2. If #1 fails, try allocating from quadrant 3.
3. If #2 fails return an error.

This policy is done to preserve the 3rd quadrant for applications needing large contiguous segment(s). However, a memory windowed application wants the shared data to occupy the memory window space (quadrant 3). In a memory windowed system the 4th quadrant becomes a more precious resource for shared libraries or objects mapped to span Memory Windows. To reflect the importance of preserving quadrant 4, Memory Windows changed the default location policy to the following:

1. If the object is a shared library, allocate it quadrant 4.
2. If the object isn't a shared library or the allocation for a shared library failed in #1, try allocating from quadrant 3
3. If #2 fails, try allocating from quadrant 4.

Please note, the change in policy location only occurs if Memory Windows are enabled in the system (i.e. max_mem_window != 0). Instead of shared memory addresses populating quadrant 4 first, now its quadrant 3. Its not anticipated this will cause problems since shared libraries default to quadrant 4 and shared libraries are the common consumers of shared space. Customers/applications having capacity problems with the old policy may be adversely affected with this change, but they are most likely the ones who need this functionality to solve the capacity problem anyway.

9.0 What tools/statistics exist for debugging.

Memory Windows functionality is installed into HP-UX 11.00 through a patch process. Memory Windows is not part of the release base and as such there are restrictions on what changes are acceptable. For instance, changing visible header files can cause unforeseen compatibility problems. HP recognizes the need to return status information and not to break binary compatibility. For the patch this meant return status information was not placed in the appropriate places and was reported through a different means. **Because HP wants to place the information in the appropriate place, the command to provide memory window information is an unsupported tool at the 11.00 release.**

memwin_stats is available at (browser):

- ftp://contrib:9unsupp8@hprc.external.hp.com/sysadmin/memwin/

memwin_stats is the **Unsupported** command to display information about shared memory segments, processes and the Memory Windows themselves. Using the user window id from process and the output of all the Memory Windows in the system, you should be able to determine which memory window a process is in and then look at the status of that window. The reference count is an indication of how many "objects" are in the memory window, where an object can be, the process itself, a memory mapped file or a shared memory segment. The amount of available virtual space is returned. This is the amount of virtual space left for the memory window, it does not reflect the amount of physical space consumed. And lastly the user id's for shared memory segments are returned. Using the id and other information from ipcs(), the specific memory window a shared memory segment can be determined. At this time there is no way to locate the memory window for a memory mapped file.

memwin_stats returns the number of 4K pages still available in a given quadrant. This means the window has ‘X’ pages of total virtual space. What it doesn’t show or report is how contiguous those pages are. A memory window may have 100 Megabytes of available space, but if the largest contiguous chunk in the 100 Megabytes is 20 megabytes, only shared objects up to 20 megabytes in size can be created. If a shmget() call for 40 Megabytes was issued, the call would fail.

syntax: memwin_stats [-m] [-w] [-p pid]

? **-m**

Display information about shared memory segments and which window they occupy. The basic “ipcs” information is displayed along with the window where the shared memory segment was created. The two new fields added to the display are the “user key” and “kernel key” associated with the memory window.

The User key is the window id of the process that created the shared memory segment. The value should be one of the keys found in the file /etc/services.window. If the user was in a private window, the user key is meaningless, that’s where the kernel key comes in. Using the kernel key and the output from the -w option, you can determine which memory window the shared memory segment comes from.

? -**p <ProcessId>**

Displays the memory window information, in this case the user key and kernel key, for a particular process.

? **-w**

Displays information about the Memory Windows contained in the system. Each memory window configured is displayed. The window can be in 1 of X states. It’s either:

1. Unused
2. The default Global Window
3. Active user window.

For each window the amount of available space in each memory window is displayed, the kernel keys(see above), the user key, and the number of references to the window.