Shared Memory in the HP-UX 32-bit Environment

by
Jeff Kleckley
Hewlett-Packard Company

20 Perimeter Summit Blvd., MS 907
Atlanta, GA 30319-1417

Jeff_Kleckley@hp.com

Introduction

This paper will describe the flexibility of virtual memory in the HP-UX 32-bit environment. Specifically, how this flexibility relates to shared memory. Various 32-bit virtual memory limitations will be discussed as well as the methods used to overcome them. However, before these methods are discussed, a description of the HP-UX 32-bit virtual memory addressing is necessary.

Each process in HP-UX views a 4Gb address space which is divided into 4 1Gb quadrants. HP-UX accomplishes this with virtual memory addressing. Each process has its own view of how virtual memory is laid out independent of the underlying physical memory. HP-UX creates a mapping of a process' view of virtual memory and the actual physical memory. This mapping is managed by HP-UX and implemented in the underlying hardware so it can handle the address translation from virtual memory adresses to physical memory addresses.

The upper bound for all the virtual memory on the system is the amount of total swap space (pseudo swap + device swap + filesystem swap). The virtual address space layout for HP-UX uses 4 quadrants which observe several kernel parameters. For a 32-bit address space, each quadrant has a hard limit of 1Gb regardless of available virtual memory (swap space). Depending on their memory needs, many 32-bit applications will find the standard virtual address layout to be restrictive. However, HP-UX allows for flexibility in regards to the virtual address space for 32-bit processes.

While HP-UX always adhere to the 4 quadrant model, how these quadrants are used can be changed by a process's magic number. The magic numbers we will discuss are:

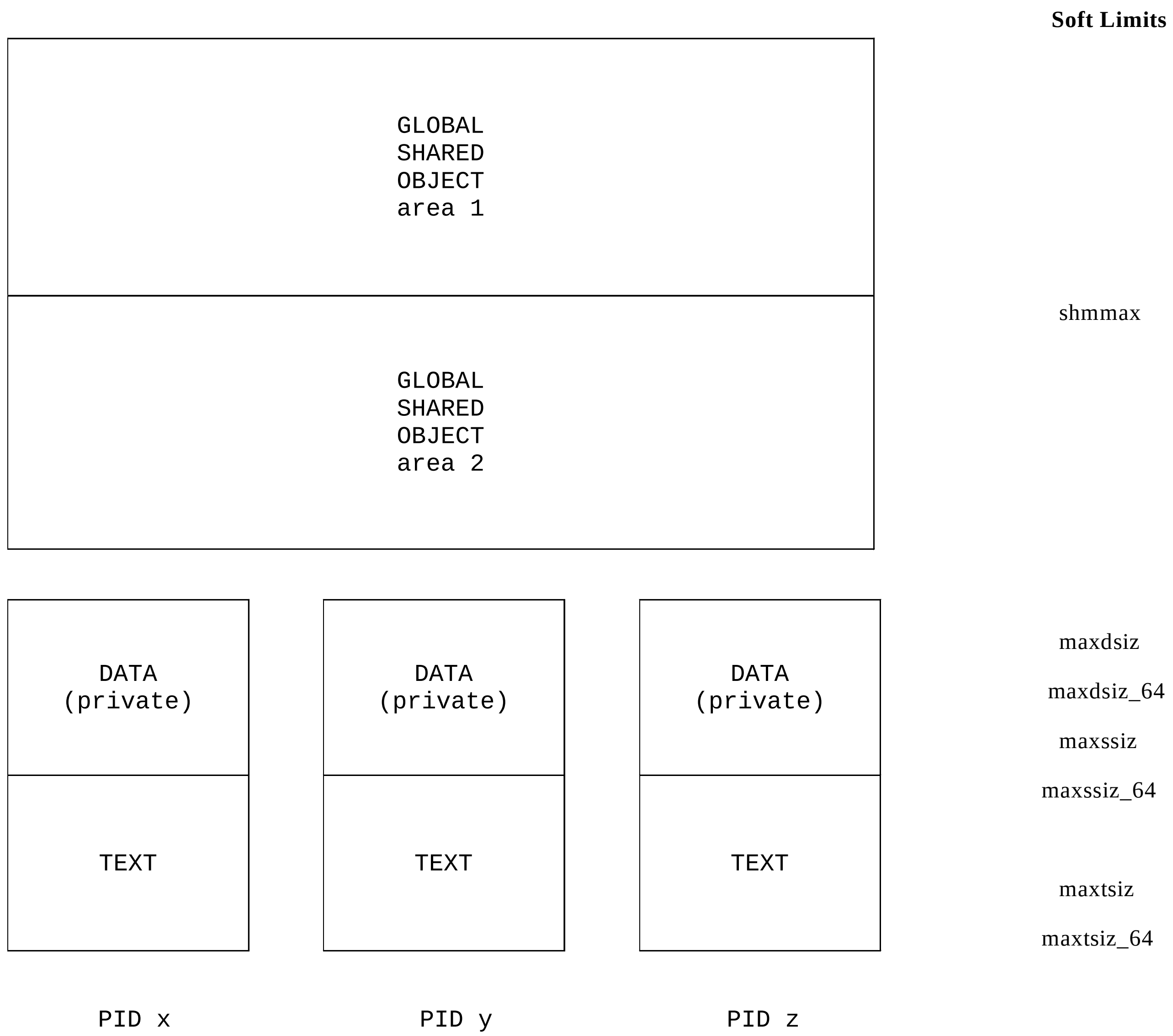
EXEC_MAGIC
SHARE_MAGIC
DEMAND_MAGIC
SHMEM_MAGIC

When an executable is linked using any of these magic numbers, this tells the kernel to select a different memory map.

How do you use these magic numbers? Better yet, how do you check to see if you did it correctly? Before answering these questions, a little background on HP-UX memory addressing is in order.

Background

The process address space is first limited by the amount of available virtual memory (total swap space) which is observed with **swapinfo –tam**. The address space layout for HP-UX uses 4 quadrants which observe several kernel parameters that act as soft limits:



For a 32-bit address space, each quadrant has a hard limit of 1Gb regardless of available virtual memory (swap space). For a 64-bit address space, each quadrant is hard limited to 4Tb. With 64-bit addressing, programs will be able to access a maximum of 4Tb of text, 4Tb of private data, and 8Tb of shared objects. Current 64-bit programs use nowhere near 4Tb in any of the four quadrants in a 64-bit address space; however, many programs using the 32-bit address space find the 1Gb quadrants to be a limitation. Therefore, the rest of this document refers to a 32-bit address space and ways to work around its limitations.

HP-UX allows for flexibility in regards to the virtual address space for 32-bit processes.

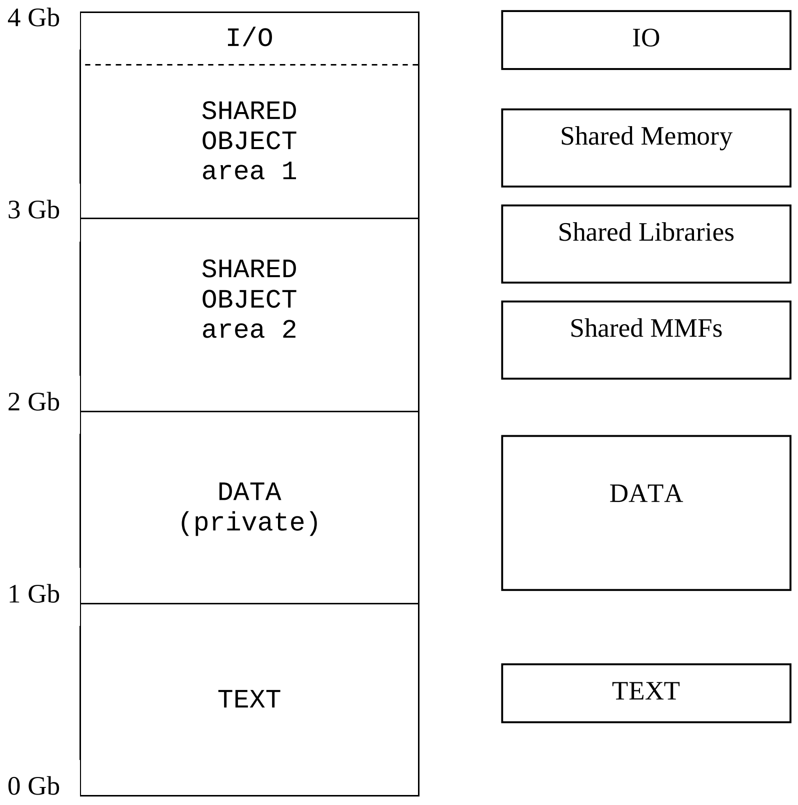
While HP-UX always adhere to the 4 quadrant model, how these quadrants are used can be changed by a process's magic number.

If you check the magic(4) man page, you will see the following table documenting these magic numbers among others:

```
               #define EXEC_MAGIC      0x107   /* normal executable */
               #define SHARE_MAGIC     0x108   /* shared executable */
               #define DEMAND_MAGIC    0x10B   /* demand-load executable */
```

SHARE_MAGIC Executables

With HP-UX, the magic numbers for SHARE_MAGIC and DEMAND_MAGIC executables have the same meaning. The address space for a process is laid out with a maximum of 1Gb of text, 1Gb of data, 1.75Gb of shared items (shared libraries and shared memory). The last .25Gb is reserved for IO. So the memory map looks like this:



Code Example

It is important to remember that the cc(1) compiler will call the linker ld(1) and assign a magic number of 108 or SHARE_MAGIC by default.

Construct a simple c program source:

```
$ more test.c
main(){}
```

Compile using defaults:

```
$ cc test.c
```

Use chatr to see the characteristics of the resulting a.out file:

```
$ chatr a.out |more
a.out:
         shared executable
```

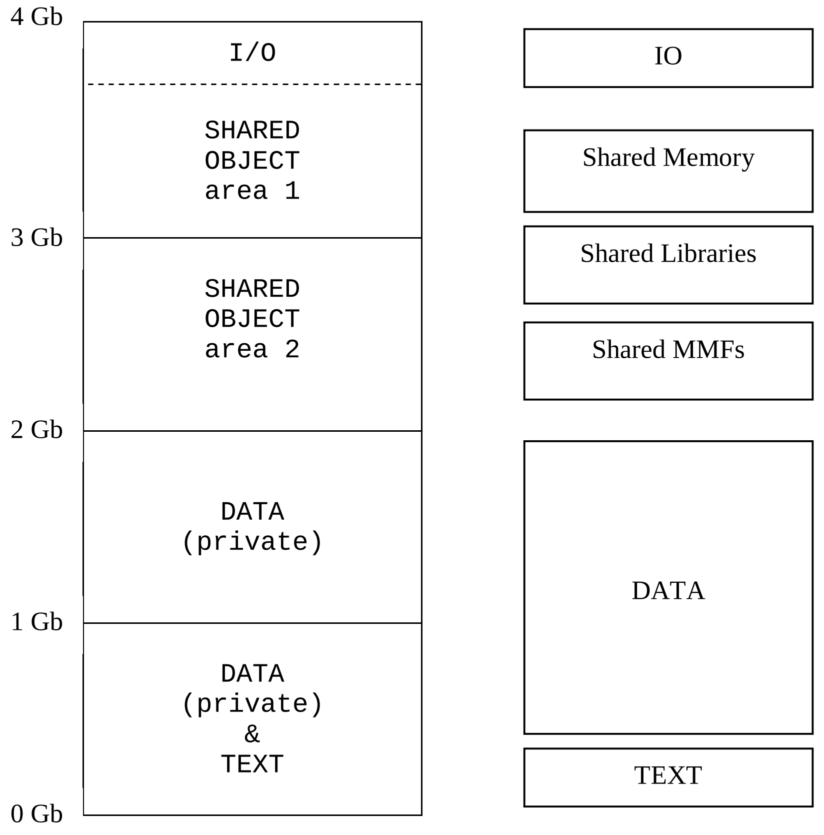
Check the resulting a.out file with odump and see that the magic number is 108:

```
$ /usr/contrib/bin/odump a.out |head
Header record for : a.out

version:            85082112
system id:          0210 (PA-RISC 1.1)    magic number:   0108
```

Many applications will have some problems with this layout. There is only one quadrant available for the private data of a process and it is limited to 1Gb. Also, there is only ~1.75Gb of shared space available for the whole system. These limitations are further magnified by today's large memory systems. Imagine buying a system with 4Gb of memory and having the operating system limit several Oracle database instances to only ~1.75Gb total shared memory. Because applications were limited to only 1Gb of private data space, a magic number, EXEC_MAGIC, was introduced to allow for larger data areas....

EXEC_MAGIC Executables

When an executable uses the EXEC_MAGIC magic number, this tells the kernel to select a different memory map. Since the text of a program does not change at runtime and it probably does not use anywhere near the full 1Gb of the text quadrant, then it is safe to start the private data area immediately on top of the program's text in the text area. This allows the data area to make use of the normally wasted virtual address space above the process text in the first quadrant. Given a small text area, an EXEC_MAGIC executable data space can reach about 1.9Gb. So the memory map for an EXEC_MAGIC executable will look like this:



Code Example

How to set the magic number to EXEC_MAGIC?

The program must be recompiled with an option passed to the linker (ld) to specify the magic number to EXEC_MAGIC. The ld(1) man page states that this is done with the -N option:

```
           -N             Generate an executable output file with file type
                          EXEC_MAGIC.
```

Now let’s recompile our example program and pass this option to the linker:

```
$ cc -Wl,-N test.c
```

What does chatr(1) say about the resulting a.out file?

```
$ chatr a.out |more
a.out:
         normal executable
```

The a.out file is further checked with odump:

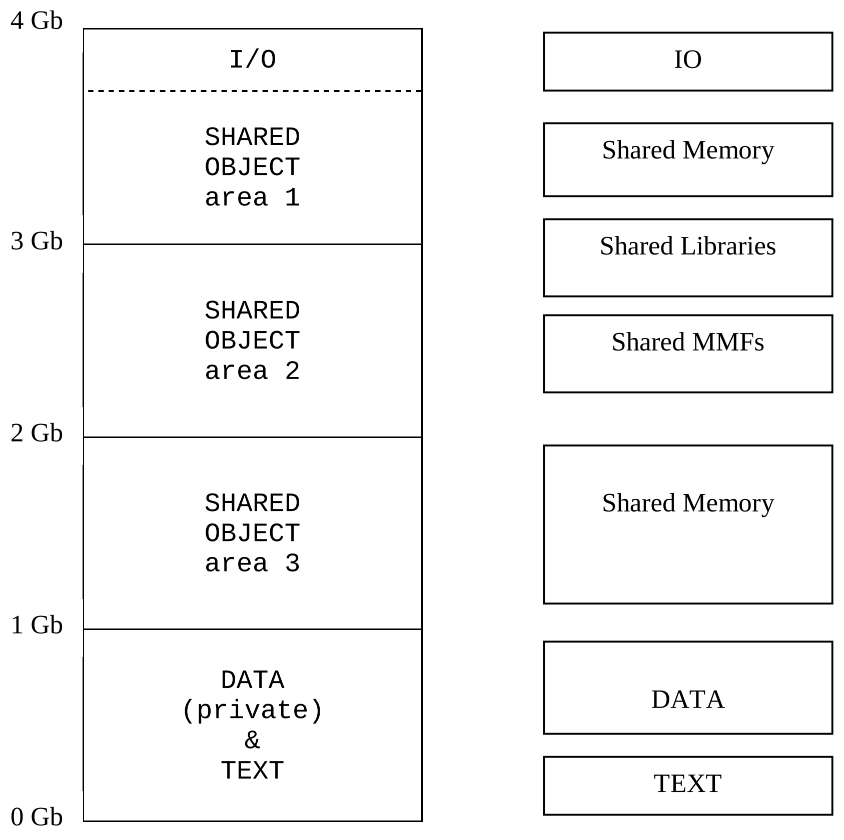
```
$ /usr/contrib/bin/odump a.out |head
Header record for : a.out

version:          85082112
system id:        0210 (PA-RISC 1.1)    magic number:   0107
```

SHMEM_MAGIC Executables

Because a 32-bit HP-UX system is further limited to only 1.75Gb of total shared memory space, applications that rely on shared memory (ie: databases) are further restricted. Conversely, 64-bit processes running on 64-bit HP-UX do not see this restriction since they have approximately 8Tb of space available for shared objects. So another magic number, SHMEM_MAGIC, was introduced to allow for a larger shared memory space for 32-bit processes. The following patches are required to use this magic number on 10.20 systems:

PHKL_16751 (s800)
PHKL_16750 (s700)
PHSS_17903

Since many applications that use large amounts of shared memory rarely use large amounts of private memory they are likely to see large amounts of wasted address space in both of the first two quadrants. The SHMEM_MAGIC magic number causes all of the program's text and data to be fitted into the first quadrant. The second quadrant now becomes a third area for holding shared memory and is only available for programs compiled using SHMEM_MAGIC. Using SHMEM_MAGIC it is then possible for a program to access ~2.75Gb of shared memory. The memory map for an SHMEM_MAGIC executable will look like this:



Shared memory usage can be checked with **ipcs –mob**. A process will request a shared memory segment with the shmget(2) function call. Two things to remember:

1) shared memory segments must be made of contiguous memory pages.
2) no single shared memory segment can be larger than a quadrant boundary or 1Gb.

Code Example

We use chatr(1) instead of a recompile/relink to set SHMEM_MAGIC for an executable. However, its important to remember: an executable must have a magic number of EXEC_MAGIC before using chatr(1) to change it's magic number further to SHMEM_MAGIC.

Let’s try granting our example program set with the default SHARE_MAGIC the ability to access 2.75Gb shared memory.

```
$ cc test.c
$ chatr -M a.out
....
chatr:(error) - only EXEC_MAGIC files can be made SHMEM_MAGIC
```

So the error simply confirms that the executable has a magic number of 108 (SHARE_MAGIC) when it needs to have a magic number of 107 (EXEC_MAGIC).

Let’s follow the previous code example and recompile our program again using EXEC_MAGIC.

```
$ cc -Wl,-N test.c
```

Now use chatr to further change the magic number to SHMEM_MAGIC:

```
$ chatr -M a.out
```

Checking the a.out file now with chatr(1) and odump:

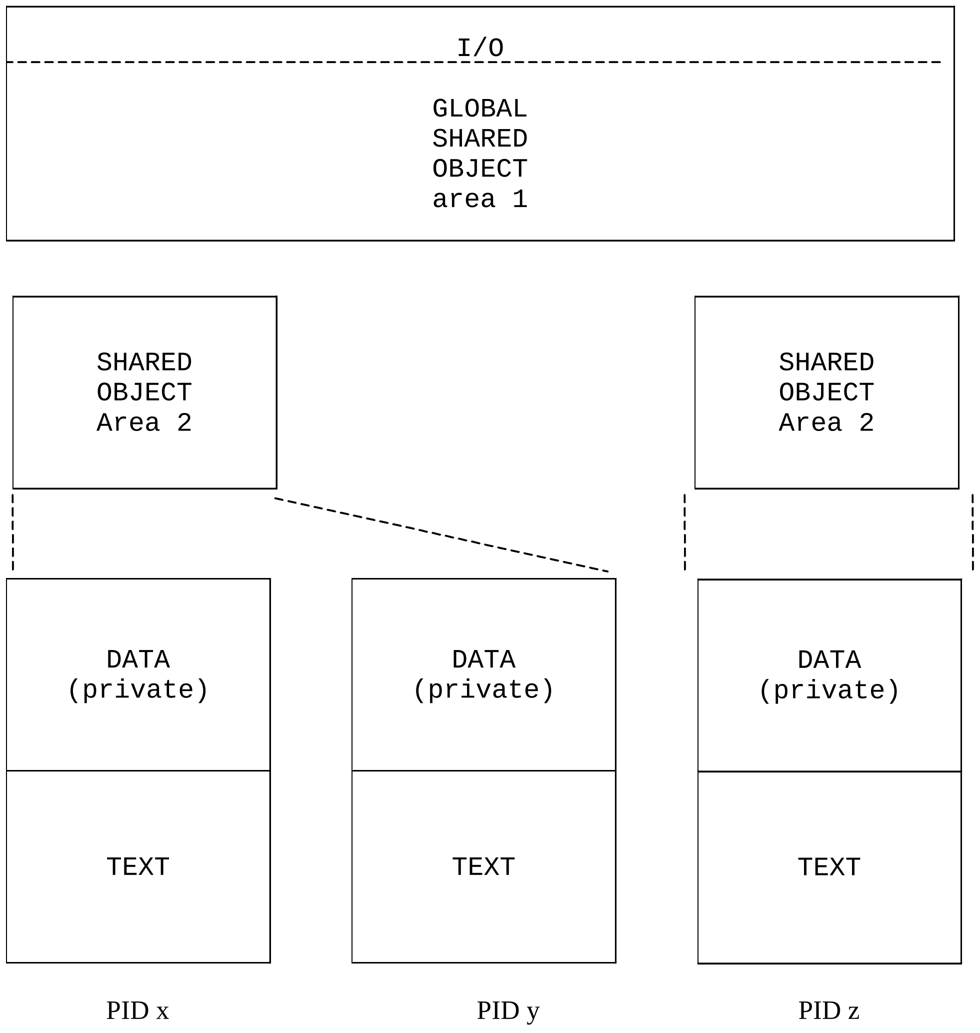
```
$ chatr a.out |more
a.out:
         normal SHMEM_MAGIC executable

$ /usr/contrib/bin/odump a.out |head
Header record for : a.out

version:          85082112
system id:        0210 (PA-RISC 1.1)    magic number:   0109
```

This indicates that the executable is set correctly to take advantage of 2.75Gb shared memory!

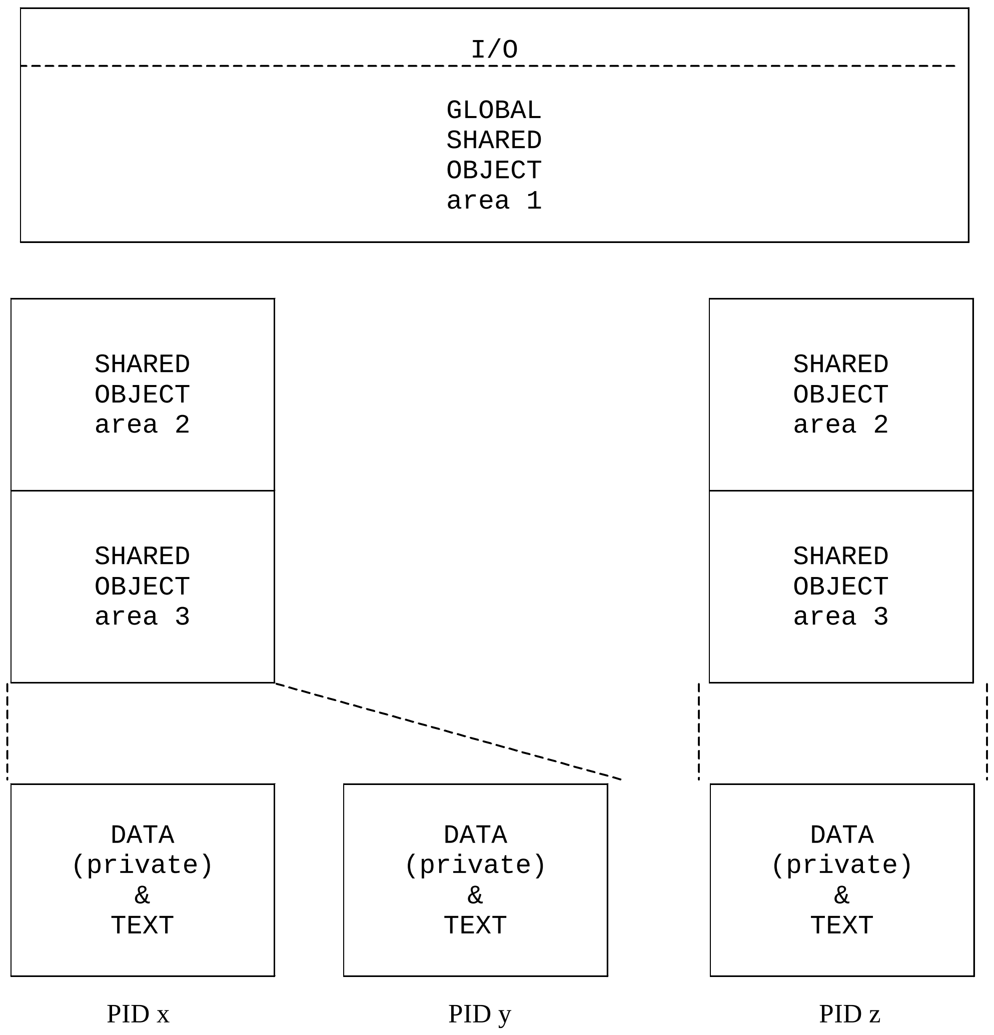
HP-UX 11.x Memory Windows

All 32-bit applications in the system are limited to a total of 1.75Gb of global shared memory, 2.75Gb if compiled with SHMEM_MAGIC. This limitation applies to 32-bit applications running in either 32-bit or 64-bit HP-UX 11.0. Many sites running multiple instances of popular 32-bit database applications found this global space used for shared resources to be limiting. Even with SHMEM_MAGIC exectuables that allow up to 2.75Gb of shared memory, applications on systems with extremely large amounts of memory found this a restriction. Memory windows is an 11.x only feature that allows 32-bit applications to get around the 1.75Gb/2.75Gb limitation for global shared memory.

A 32-bit process can create a unique memory window for shared objects such as shared memory. Other 32-bit processes can attach to this window to access these shared objects. This allows each cooperating process to create 1-2 gigabytes of shared resources without exhausting the system-wide global space. The 4th quadrant still remains globally visible to all processes for shared libraries and shared objects requiring access by all processes, no matter what memory window they are in. The ability to create a unique memory window removes the system-wide 1.75Gb/2.75Gb global shared memory limitation. So the memory layout for SHARE_MAGIC processes using memory windows will look like this:



The memory layout for SHMEM_MAGIC processes using memory windows:



While memory windows allow for more than 1.75Gb/2.75Gb of system-wide global shared memory, it does not extend how much shared resources a single process can create! SHARED_MAGIC executables are still limited to 1.75 gigabytes and SHMEM_MAGIC executables are limited to 2.75 gigabytes themselves. But different applications, or distinct instances of a single application, can attach to different memory windows and consume more than than 1.75Gb/2.75Gb of system-wide global shared memory.

For more information on the concepts and implementation of memory windows, see the memory windows white paper found at /usr/share/doc/mem_wndws.txt on 11.0 systems. Also, it is a good idea to consult with your application vendor for any known issues with HP-UX memory windows before attempting to use them.

What is required to run memory windows?

1) HP-UX 11.x 32-bit or 64-bit installation
2) Two patches were released to enable memory windows:
   PHKL_13810
   PHKL_13811

   These two patches have since been superseded. The current list is:
   PHKL_18543
   PHCO_19047
   PHCO_20179
   PHKL_20995
   PHCO_20443

3) The kernel tunable, max_mem_window, must be set to the desired number of virtual memory windows. The default value is 0. 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.

4) The /etc/services.window file must be created. A group of processes wishing to use a common memory window must associate themselves with the unique key for that window. The file /etc/services.window is a centrally located file used so applications can avoid hard coding id's in startup or control scripts. This file will have entries in the format of "<associated string> <window key>". Each string/key pair must be unique. A sample /etc/services.window file:

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

informix    20
oracle      30
sybase      40
database1   50
database2   60
database3   70
```

How do we start a process in a memory window?

The getmemwindow(1) command was introduced to extract name/window_key from the /etc/services.window file. The setmemwindow(1) command is used to change the window id of a running process or start a process in a specific memory window. The startup for an application would use getmemwindow to identify the window id for the application to use and setmemwindow to execute the application within that window. For example, the startup script may read something like this:

```
$ cat startDB1.sh
WinId=$(getmemwindow database1)
setmemwindow –i $WinId /home/user/executable
```

So based on our /etc/services.window file, the startDB1 executable will be started using a memory window with an ID of 50.

How can we tell if the memory window was declared and what shared memory segments are found in that window?

The memwin_stats is the command to display information about shared memory segments and the memory window being used.

```
# ./memwin_stats -m
Shared Memory:
T      ID     KEY        MODE        OWNER     GROUP   UserKey   KernId
m       0 0x2f100002 --rw-------      root       sys    Global       0
m       1 0x411c36c1 --rw-rw-rw-      root      root    Global       0
m       2 0x4e0c0002 --rw-rw-rw-      root      root    Global       0
m       3 0x412041c9 --rw-rw-rw-      root      root    Global       0
m       5 0x06347849 --rw-rw-rw-      root      root    Global       0
m   19806 0x52140128 --rw-r—-r--      root       sys       50        1
```

In this example, we see the shared memory segments associated with the global default memory window. We also see there is a shared memory segment associated with a user defined memory window which has a window ID of 50.

Code Example

Our program creates and attaches to a 1K shared memory segment. It takes a user string as an argument and writes this string to the shared memory segment. Also, the key that is generated to assign to the shared memory segment is sent to stdout.

We execute our program using a startup script to implement memory windows:

```
# cat startDB1.sh
WinId=$(getmemwindow database1)
setmemwindow –i $WinId /home/user/startDB1 “Hello World!”
```

```
# ./startDB.sh
writing to segment: "Hello World!"
Key is 1377042553
```

```
# ./memwin_stats -m
Shared Memory:
T      ID     KEY        MODE        OWNER     GROUP   UserKey   KernId
m       0 0x2f100002 --rw-------      root       sys    Global       0
m       1 0x411c36c1 --rw-rw-rw-      root      root    Global       0
m       2 0x4e0c0002 --rw-rw-rw-      root      root    Global       0
m       3 0x412041c9 --rw-rw-rw-      root      root    Global       0
m    1204 0x52140079 --rw-r--r--      root       sys      50         1
```

We see a shared memory segment with a key of 0x52140079 (1377042553 decimal) using a memory window with ID 50.

Now we edit several copies of the startup script and our program then execute them. The copied programs will have a slight change so that it uses a different key and therefore generates a new segment. We will also change the startup script to specify different user ID’s of “database2” and “database3” for getmemwindow(1) so that the new shared memory segments are attached to different memory windows.

```
# ./startDB2.sh
writing to segment: "DB2 says Bon Jour!"
Key is 1377042733

# ./startDB3.sh
writing to segment: "DB3 says Ola!"
Key is 1377042734

# ./memwin_stats -m
Shared Memory:
T      ID     KEY        MODE        OWNER     GROUP   UserKey   KernId
m       0 0x2f100002 --rw-------      root       sys    Global       0
m       1 0x411c36c1 --rw-rw-rw-      root      root    Global       0
m       2 0x4e0c0002 --rw-rw-rw-      root      root    Global       0
m       3 0x412041c9 --rw-rw-rw-      root      root    Global       0
m    1404 0x52140079 --rw-r--r--      root       sys      50         1
m     205 0x5214012d --rw-r--r--      root       sys      60         2
m   10206 0x5214012e --rw-r--r--      root       sys      70         3


# ipcs -mob
IPC status from /dev/kmem as of Fri May  5 16:09:24 2000
T      ID     KEY        MODE        OWNER     GROUP NATTCH  SEGSZ
Shared Memory:
m       0 0x2f100002 --rw-------      root       sys      8 1286144
m       1 0x411c36c1 --rw-rw-rw-      root      root      0    348
m       2 0x4e0c0002 --rw-rw-rw-      root      root      1  31040
m       3 0x412041c9 --rw-rw-rw-      root      root      1   8192
m    1404 0x52140079 --rw-r--r--      root       sys      0   1024
m     205 0x5214012d --rw-r--r--      root       sys      0   1024
m   10206 0x5214012e --rw-r--r--      root       sys      0   1024
```

In our example, the three shared memory segments are 1K in size apiece, but each of them could be 1Gb in size as long as they were created using memory windows. The ipcs(1) output shows NATTCH as 0 for the three segements because our sample program detaches from the segment after creating and writing to it.

The example program source:

```
/*
** startDB1 -- read and write to a shared memory segment
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_SIZE 1024  /* make it a 1K shared memory segment */

int main(int argc, char *argv[])
{
    key_t key;
    int shmid;
    char *data;
    int mode;

    if (argc > 2) {
        fprintf(stderr, "usage: startDB1 [data_to_write]\n");
        exit(1);
    }

    /* make the key: */
    if ((key = ftok("/home/user/startDB1.c", 'R')) == -1) {
        perror("ftok");
        exit(1);
    }
    printf("Key is \"%d\"\n", key);

    /* connect to (and possibly create) the segment: */
    if ((shmid = shmget(key, SHM_SIZE, 0644 | IPC_CREAT)) == -1) {
        perror("shmget");
        exit(1);
    }

    /* attach to the segment to get a pointer to it: */
    data = (char *) shmat(shmid, 0, 0);
    if (data == (char *)(-1)) {
        perror("shmat");
        exit(1);
    }


    /* read or modify the segment, based on the command line: */
    if (argc == 2) {
        printf("writing to segment: \"%s\"\n", argv[1]);
        strncpy(data, argv[1], SHM_SIZE);
    } else
        printf("segment contains: \"%s\"\n", data);

    /* detach from the segment: */
    if (shmdt(data) == -1) {
        perror("shmdt");
        exit(1);
    }

    return 0;
}
```

The only change we make to the copies startDB2.c and startDB3.c is to specify a different file name in the “make the key” function so that these programs create a segment with a different segment ID. Note that if any of these programs are run a second time and the filename is the same for the “make the key” function, it will not create a new segment but instead it will attach and modify the segment that already exists with the same key.

The startup scripts for the example programs:

```
# more startDB1.sh
```

```
WinId=$(getmemwindow database1)
setmemwindow -i $WinId /home/user/startDB1 "Hello World!"

# more startDB2.sh
WinId=$(getmemwindow database2)
setmemwindow -i $WinId /home/user/startDB2 "DB2 says Bon Jour!"

# more startDB3.sh
WinId=$(getmemwindow database3)
setmemwindow -i $WinId /home/user/startDB3 "DB3 says Ola!"
```