Overview

The Memory Usage on HP-UX Integrity Servers - Version 1.1 (February 15, 2006) paper discusses the reasons why increases in code and data sizes may be observed when migrating applications to HP-UX Integrity servers. That paper includes some guidelines on how to estimate potential increases; however, it does not suggest the use of any specific procedures or utilities. The intent of this paper is to provide some assistance in collecting and interpreting data required to estimate potential increases. As a prerequisite to fully benefit from this paper, it is highly recommended that you be familiar with the content of the Memory Usage on HP-UX Integrity Servers paper.

This paper is divided into the following sections:

- Section 1, Virtual Address Space and Text Memory Usage, outlines the virtual address space of a process in HP-UX and it explains how to use features of the GlancePlus tool to determine the amount of physical memory allocated to text (= code, instructions) of processes.

  The Memory Usage on HP-UX Integrity Servers paper states, “To account for code expansion, first identify the amount of compiled code present in the in-memory working set of a typical workload on the existing system, then allow for an 80% or so increase.”

- Section 2, Size of Initialized Data in Program Files, explains how to use the size command to determine the size of the initialized data portion present in a program file. It also includes an example that demonstrates how the use of different compiler options and versions for Integrity systems may affect the size of initialized data.

  The Memory Usage on HP-UX Integrity Servers paper states, “The increase in memory requirements may be estimated by comparing the data segment size of the PA-RISC application binary with that of the Itanium binary, and multiplying the difference by the expected number of running processes.”

Section 1 of this paper makes the following assumptions:

- The “existing system” is PA-RISC.

- The “existing system” is configured appropriately such that paging or swapping does not prevent the application from allocating an optimal amount of physical memory to its code and data.

- The code of the application running on the “existing system” is shared among processes. Address space models such as EXEC_MAGIC where the text segments are private are not considered. The type TEXT/Priv displayed by the GlancePlus tool differentiates these special cases from the standard type TEXT/Shared.

The examples and the screen output in Section 1 were taken from an HP-UX 11i v1 system running on a 64-bit PA-RISC processor; however, the techniques of using various features of the GlancePlus tool and the methodology to determine the amount of physical memory allocated to code are similar in the following environments:

- HP-UX 11i v1 on 32-bit PA-RISC

- HP-UX 11i v2 on 64-bit PA-RISC (and on Integrity)
1. Virtual Address Space and Text Memory Usage

The virtual address space of a process in HP-UX consists of various types of virtual address segments such as text, data, stack, and so on that are embedded into the total virtual address range of the process as shown in the following figure. The virtual address segments are also known as memory regions. The virtual addresses on the left are examples from a 64-bit process on PA-RISC.

If a process allocates virtual memory by executing the `brk()` system call or by calling a library routine such as `malloc()`, the data segment is extended towards the higher virtual addresses as shown in the figure. A process can create new memory regions in its virtual address space by executing system calls such as `mmap()` and `shmat()`. For example, `mmap()` is used by the Dynamic Loader to map the text and data of shared libraries into the virtual address space of a process. The `shmat()` system call can be used by a process to attach to a shared memory segment.

Memory regions define ranges of virtual addresses that a process may access. However, during its lifetime, a process often does not access all of the virtual addresses in each memory region of its virtual address space. Therefore, the amount of physical memory allocated to a region may be less than its virtual size. The HP-UX kernel may also decide to reclaim physical memory allocated to a region if there is a temporary memory resource shortage. This is another reason why the amount of physical memory allocated to a region may be less than its virtual size.
The **glance** program and **gpm** program let you examine the memory regions of processes, including the virtual sizes of the regions and the amount of physical memory allocated to the regions. **glance** provides a character mode user interface. **gpm** provides a Motif-based user interface. Both programs are components of the GlancePlus tool. Refer to [GlancePlus Overview and Features](#), and to **glance**(1), and **gpm**(1) for more information. The following sections demonstrate how you can use the **glance** command in interactive mode and in “adviser only” mode to examine the memory regions of processes.

### 1.1 Using the **glance** Command in Interactive Mode

If you enter the **glance** command without specifying any command-line options, it comes up with a default Process List screen. You can select a process by typing lowercase “s”. **glance** will prompt you for a process ID (PID). After you enter the PID, **glance** displays the Process Resources screen. At this stage, you can type uppercase “M” to switch to the Process Memory Regions screen. **glance** will display a screen output similar to the one that follows. The number of memory regions that **glance** can display in one screen output depends on the type of the terminal window in which you run the **glance** command. If **glance** cannot display all of the memory regions of a process in one screen output, you can walk through the list of memory regions by entering lowercase “f” (scroll page forward) or “b” (scroll page backward). You can type lowercase “s” again to select another process.

The figure on the previous page only includes a subset of the actual memory regions. For completeness those regions are identified by an asterisk (*) on the left of the following output.

| CPU Util | S | 1% | 1% | 5% |
| Disk Util | F | 1% | 1% | 1% |
| Mem Util | S SU UB B | 51% | 51% | 51% |
| Swap Util | U UR R | 10% | 10% | 10% |

---

**Memory Regions**

<table>
<thead>
<tr>
<th>Type</th>
<th>RefCt</th>
<th>RSS</th>
<th>VSS</th>
<th>Locked</th>
<th>File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULLDR/Shared</td>
<td>95</td>
<td>4kb</td>
<td>4kb</td>
<td>0kb</td>
<td>&lt;nulldef&gt;</td>
</tr>
<tr>
<td>TEXT /Shared</td>
<td>2</td>
<td>12kb</td>
<td>12kb</td>
<td>0kb</td>
<td>/var/tmp/test/example</td>
</tr>
<tr>
<td>UAREA /Priv</td>
<td>1</td>
<td>32kb</td>
<td>32kb</td>
<td>0kb</td>
<td>&lt;uarea&gt;</td>
</tr>
<tr>
<td>DATA /Priv</td>
<td>1</td>
<td>3.2mb</td>
<td>5.1mb</td>
<td>0kb</td>
<td>/var/tmp/test/example</td>
</tr>
<tr>
<td>MEMMAP /Priv</td>
<td>1</td>
<td>4kb</td>
<td>4kb</td>
<td>0kb</td>
<td>/opt/.../libogltls.2</td>
</tr>
<tr>
<td>MEMMAP /Priv</td>
<td>1</td>
<td>8kb</td>
<td>8kb</td>
<td>0kb</td>
<td>&lt;mmmap&gt;</td>
</tr>
<tr>
<td>MEMMAP /Priv</td>
<td>1</td>
<td>4kb</td>
<td>4kb</td>
<td>0kb</td>
<td>/usr/lib/pa20_64/libdl.1</td>
</tr>
<tr>
<td>MEMMAP /Priv</td>
<td>1</td>
<td>60kb</td>
<td>60kb</td>
<td>0kb</td>
<td>/usr/lib/pa20_64/libc2</td>
</tr>
<tr>
<td>MEMMAP /Priv</td>
<td>1</td>
<td>20kb</td>
<td>52kb</td>
<td>0kb</td>
<td>&lt;mmmap&gt;</td>
</tr>
<tr>
<td>MEMMAP /Priv</td>
<td>1</td>
<td>8kb</td>
<td>8kb</td>
<td>0kb</td>
<td>&lt;mmmap&gt;</td>
</tr>
<tr>
<td>MEMMAP /Priv</td>
<td>1</td>
<td>20kb</td>
<td>20kb</td>
<td>0kb</td>
<td>/usr/lib/pa20_64/dld.sl</td>
</tr>
<tr>
<td>MEMMAP /Priv</td>
<td>1</td>
<td>8kb</td>
<td>8kb</td>
<td>0kb</td>
<td>&lt;mmmap&gt;</td>
</tr>
<tr>
<td>STACK /Priv</td>
<td>1</td>
<td>12kb</td>
<td>64kb</td>
<td>0kb</td>
<td>&lt;stack&gt;</td>
</tr>
<tr>
<td>MEMMAP /Shared</td>
<td>2</td>
<td>180kb</td>
<td>252kb</td>
<td>0kb</td>
<td>/usr/lib/pa20_64/dld.sl</td>
</tr>
<tr>
<td>MEMMAP /Shared</td>
<td>2</td>
<td>4kb</td>
<td>8kb</td>
<td>0kb</td>
<td>/usr/lib/pa20_64/libgltls.2</td>
</tr>
<tr>
<td>MEMMAP /Shared</td>
<td>2</td>
<td>4kb</td>
<td>4kb</td>
<td>0kb</td>
<td>/usr/lib/pa20_64/libc2</td>
</tr>
<tr>
<td>SHMEM /Shared</td>
<td>2</td>
<td>1.0mb</td>
<td>1.1mb</td>
<td>0kb</td>
<td>/usr/lib/pa20_64/libc2</td>
</tr>
<tr>
<td>SHMEM /Shared</td>
<td>2</td>
<td>1.0mb</td>
<td>2.0mb</td>
<td>0kb</td>
<td>&lt;shmem&gt;</td>
</tr>
</tbody>
</table>

The **Process Memory Regions** screen shows six columns: Type, RefCt, RSS, VSS, Locked, and File Name. Within the GlancePlus tool, these columns are also known as metrics. The **glance** online help facility lets you view a detailed description by typing lowercase “h” and selecting “Current Screen Metrics”. The following paragraphs summarize these columns and metrics.
The **Type** column combines two metrics: the type of the memory region such as text, data, stack, and others and a tag that specifies if the region is shared by processes or private to a process. The physical memory allocated to a shared region is shared by all processes that have mapped the region into their virtual address spaces. The **RefCt** column specifies the number of processes (plus one) that are sharing the region.

The **TEXT/Shared** type identifies the process text segment. As mentioned earlier, the text and data of shared libraries are mapped into the virtual address space of a process with the `mmap()` system call. Therefore, the **MEMMAP/Shared** type identifies a shared library text region if the content of the **File Name** column refers to a shared library image file. There may be situations where **glance** displays an inode number instead of a file name. If this happens, refer to the **glance** online help text pertaining to the Process Memory Regions screen metrics for advice.

The **VSS** and **RSS** columns specify the virtual size of the region and the amount of physical memory allocated to it. In the example, the shared library text of `libc.2` is mapped into the virtual address range from `0xc00000000100000` to `0xc0000000021c000`. Therefore, the virtual size of the region is `0xc0000000021c000 - 0xc00000000100000 = 0x11c000 bytes = 1.1 MB (VSS)`. As shown in the output of **glance**, the amount of physical memory allocated to this region is 1.1 MB (RSS).

**Methodology**

The interactive mode of the **glance** command is appropriate to determine the amount of physical memory allocated to the code of an application, if the application processes running on a server are instances of a small set of different executable program files.

For example, an application consists of three server programs **Px**, **Py**, and **Pz**. **Px** and **Py** require the shared library **Lxy**. **Py** and **Pz** require the shared library **Lyz**. All of the programs require the shared library **Lxyz**. If the server is running many instances of these programs, it is sufficient to examine the memory regions of any three processes: one instance of **Px**, **Py**, and **Pz** as shown in the following figure.

Therefore, the total amount of physical memory allocated to program and shared library text is as follows:

\[
\text{RSS}(\text{Px}) + \text{RSS}(\text{Py}) + \text{RSS}(\text{Pz}) + \text{RSS}(\text{Lxy}) + \text{RSS}(\text{Lyz}) + \text{RSS}(\text{Lxyz}) = 0.5 \text{ MB} + 1.0 \text{ MB} + 1.5 \text{ MB} + 2.0 \text{ MB} + 2.5 \text{ MB} + 3.0 \text{ MB} = 10.5 \text{ MB}
\]
**Resulting Estimate for Integrity**

The *Memory Usage on HP-UX Integrity Servers* paper states, “Generally, an application’s code size is small relative to its data size, so this increase in code size has only a minor effect on the system's memory usage. In the standard programming model, compiled code is shareable across processes, so that on a server where the same application is running in 1000 processes, only one copy of the code will need to be in memory, while 1000 copies of the data will be needed. For typical server workloads, one expects code to account for only about 5% of all active pages in memory. If we assume a code expansion of 80%, we would expect no more than a 4% increase in total memory usage due to code expansion.”

The rule of thumb is to assume a code expansion of 80 percent. For example:

\[
10.5 \text{ MB} + (0.8 \times 10.5 \text{ MB}) = 18.9 \text{ MB}
\]

1.2 Using the glance Command in Adviser Only Mode

The following adviser syntax script utilizes the `PROCESS LOOP` and `PROC_REGION LOOP` syntax statements to examine the memory regions of a specified process (for example, PID 379). The `PROCESS LOOP` cycles through all of the active processes. When the `IF` statement detects a process ID match, a heading is printed and the `PROC_REGION LOOP` is executed to cycle through the memory regions of that process. These are the metrics that are printed for each region.

- `PROC_REGION_TYPE, PROC_REGION_PRIVATE_SHARED_FLAG`
  The equivalent to the *Type* column of the Process Memory Regions screen.

- `PROC_REGION_RES, PROC_REGION_FILENAME`
  The equivalent to the *RSS* and *File Name* columns of the Process Memory Regions screen.

The `PRINT` statements write their output to *stdout* (standard output).

```
$ cat mregions.syn

PROCESS LOOP
{   IF PROC_PROC_ID == 379 THEN
    #                  ^^^ the PID of the process to be examined
    PRINT ""
    PRINT "Memory Regions PID:", PROC_PROC_ID, ", ", PROC_PROC_NAME
    PRINT "" "Type RSS File Name"
    PRINT "----------------------------------------",
    "----------------------------------------"
    PROC_REGION LOOP
    {   PRINT PROC_REGION_TYPE, "/", PROC_REGION_PRIVATE_SHARED_FLAG, ", ",
        PROC_REGION_RES, ", " ,PROC_REGION_FILENAME
    }
}

```

Refer to the *Adviser Reference Guide* and online help facility of `glance` for more information on the adviser syntax.
The following command line will launch `glance` in “adviser only” mode (no interactive dialog). `glance` will execute the adviser syntax script contained in the `mregions.syn` file for two iterations and then terminate. The interval between the two iterations is 60 seconds.

```
$ glance -adviser_only -syntax mregions.syn -iterations 2 -j 60
```

Refer to `glance(1)` for information on command-line options.

The output produced by the adviser `mregions.syn` syntax script is similar to the Process Memory Regions screen in interactive mode. The `RefCt`, VSS, and Locked columns are not included because they are not relevant to determining the amount of physical memory allocated to the code of an application. Only the first output from the two iterations is shown here. The second one has been omitted (symbolized with `[...]`). For completeness the asterisk (*) on the left identifies the regions shown in the figure on page 3.

```
Memory Regions PID:       379, example
Type                 RSS   File Name
--------------------------------------------------------------------------------
NULLDR/Shared        4kb   <nullrdf>
* TEXT /Shared       12kb   /var/tmp/test/example
 UAREA /Priv         32kb   <uarea>
* DATA /Priv          3.2mb  /var/tmp/test/example
 MEMMAP/Priv         40kb   /opt/graphics/OpenGL/lib/pa20_64/libogltls.2
 MEMMAP/Priv         80kb   <mmap>
 MEMMAP/Priv        200kb   /usr/lib/pa20_64/libdld.1
* MEMMAP/Priv       600kb   /usr/lib/pa20_64/libc.2
 MEMMAP/Priv         20kb   <mmap>
 MEMMAP/Priv         80kb   <mmap>
 MEMMAP/Priv        200kb   /usr/lib/pa20_64/dld.dl
 MEMMAP/Priv         80kb   <mmap>
* STACK /Priv         12kb   <stack>
 MEMMAP/Shared       176kb   /usr/lib/pa20_64/dld.dl
 MEMMAP/Shared        4kb   /usr/lib/pa20_64/libdl.1
* MEMMAP/Shared        4kb   /opt/graphics/OpenGL/lib/pa20_64/libogltls.2
* SHMEM /Shared       1.1mb   /usr/lib/pa20_64/libc.2
* SHMEM /Shared       1.0mb   <shmem>
```

The “adviser only” mode of the `glance` command may be more convenient than the interactive mode, depending on the number of processes that need to be examined. The comment line in the adviser syntax script shown on the previous page indicates that the PID of the process to be examined has to be modified accordingly; however, the script may not be applicable to examine short-lived processes because a process can already have terminated before it can be captured by the script.

The Methodology to determine the amount of physical memory allocated to code is the same, in principle, as that outlined on page 5 under the 1.1 Using the glance command in Interactive Mode section.
2. Size of Initialized Data in Program Files

The *Memory Usage on HP-UX Integrity Servers* paper discusses how the use of constant arrays with pointers can lead to an increase of a program’s data size, and thus to an increase in physical memory usage as applications are migrated to Integrity servers. Versions 9i and 10gR1 of the Oracle server are mentioned as examples of this issue; however, other applications may be similarly affected. Therefore, it may be of interest to compare the initialized data sizes of:

- a 32-bit program file on PA-RISC with its corresponding (migrated) 32-bit program file on Integrity
- a 64-bit program file on PA-RISC with its corresponding (migrated) 64-bit program file on Integrity

The `size` command is very convenient in order to achieve this. You can use the `size` command on PA-RISC (HP-UX 11i v1 and 11i v2) as well as on Integrity (HP-UX 11i v2) with 32-bit and 64-bit object files, and SOM and ELF formats. Refer to `a.out(4)` for information on object file formats.

The `size` command outputs four numbers: the size of the text, initialized data, BSS (uninitialized in the object file and zero-filled at run time), and the sum of these sizes.

```
$ size example
4944 + 1048576 + 2097184 = 3150896
```

The `size` command compares PA-RISC with Integrity.

You can also use the `odump` command (for SOM format) or the `elfdump` command (for ELF format) to examine the content of an object file. Refer to `odump(1)` and `elfdump(1)` for more information.

The *Memory Usage on HP-UX Integrity Servers* paper also states, “Beginning with the A.06.10 release of the HP-UX C and C++ compilers for Itanium systems, any data containing pointers that could be made read-only in a program file will be placed in a conditional read-only data section, which the linker can place with read-only data in a program file or with writable data in a shared library. With older HP-UX compilers for Itanium systems, however, constant data is placed in a read-only section only if the program is compiled with the -exec or -minshared options.”

The following C program demonstrates the differences in placement of a constant pointer array in a program file, depending on the compiler options and versions for Integrity systems. The C program consists of two source files:

- **array_subr.c**
  This source file contains the definition of the `const pointers[256*1024]` array and the `array_addr()` function that returns the virtual address of the array.

- **array_main.c**
  This source file contains the `main()` function, which calls the `array_addr()` function and prints the return value.
$ cat array_subr.c

const char * const pointers[256*1024] = {
    "string of characters 1",
    "string of characters 2",
    "string of characters 3"
};
char **array_addr()
{
    return((char **)pointers);
}

$ cat array_main.c

char **array_addr();

main()
{
    printf("virtual address of constant pointer array : 0x%p\n", array_addr());
}

The `+DDD64` compiler option is specified in the following examples to create 64-bit objects where the size of a pointer is 8 bytes. The pointer array consists of 256 * 1024 entries. Therefore, the size of the array is 256 * 1024 * 8 bytes = 2097152 bytes (0x200000).

The `-v` and `-x` options of the `size` command provide a detailed list of the sections in an object file, with numbers in the output given in hexadecimal. Refer to `size(1)` for information on the command-line options. Only those sections that are relevant to the examples are shown in the following output (the other sections have been omitted and are symbolized with `[...]`).

**Example with Compiler Version A.06.10 on HP-UX 11i v2**

The `array_subr.c` source file is compiled separately. The compiler version A.06.10 places the pointer array into a special `.rodata_cond` section of the resulting `array_subr.o` object file by default. This object file can be linked into a program file as well as into a shared library.

$ cc array_subr.c +DD64 -V -c
cc: HP aC++/ANSI C B3910B A.06.10 [Mar 22 2006]

$ size -v -x array_subr.o

<table>
<thead>
<tr>
<th>Section</th>
<th>Size</th>
<th>Virtual Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0x200000</td>
</tr>
</tbody>
</table>

The `array_main.c` source file is compiled separately. The resulting object file is linked with the `array_subr.o` object file to create the `array` program file. The linker places the pointer array into a special `.rodata_cond` section of the program file. This section becomes part of the process text segment at run time; therefore, it will be shared by all instances of the program. Note that the size of the text is 2100576 bytes, so the pointer array accounts for most of the text portion. The size of the initialized data is only 96 bytes.
The –b linker option is used in the following example to link the same array_subr.o object file into the array.so shared library. Note that the linker places the pointer array into the library’s initialized data portion which will not be shared by processes that are using this library at run time. The size of the initialized data is 2097160 bytes, so the pointer array accounts for most of the initialized data.

Example with Compiler Version A.06.00 on HP-UX 11i v2

The array_subr.c source file is compiled separately. The compiler version A.06.00 places the pointer array into the .data section of the resulting array_subr.o object file by default.
Example with Compiler Version A.06.00 and –exec Option on HP-UX 11i v2

The `array_subr.c` source file is compiled separately. If the `–exec` option is specified, the compiler version A.06.00 places the pointer array into the `.rodata` section of the resulting `array_subr.o` object file. This object file can be linked into a program file only (not into a shared library).

The `array_main.c` source file is compiled separately. The resulting object file is linked with the `array_subr.o` object file to create the `array` program file. The linker places the pointer array into the `.rodata` section of the program file. This section becomes part of the process text segment at run time; therefore, it will be shared by all instances of the program. Note that the size of the text is 2100528 bytes, so the pointer array accounts for most of the text portion. The size of the initialized data is only 88 bytes.
For more information

- Memory Usage on HP-UX Integrity Servers - Version 1.1 (February 15, 2006)
- GlancePlus Overview and Features
  The Adviser Reference Manual is found under “Technical Resources >> Product manuals”.
- HP-UX Reference (Manpages)
- Simplifying Integrity Performance Web Site
- HP-UX Performance Forum
- Please send your comments to: tec-sipt@hp.com