The HP-UX 11i v3 Memory Management Subsystem

Technical white paper

Table of contents

Introduction

The purpose of this paper is to explain the aspects of the HP-UX 11i v3 memory management subsystem relevant to obtaining the best possible computing experience. The first section "Tools to ", describes the tools and utilities that can be used to gain insight into the operation of the subsystem. The second section, "Configuration and tuning opportunities", explores the configuration and tuning opportunities available to the system administrator. The final section, "Considerations for application developers", gives a brief introduction to the attributes of memory management relevant to application developers.

Basic memory management technology is covered in the main body of this document. Advanced technical information appears in the appendices, for additional reference. All of the commands and interfaces illustrated in this paper have manpages that can be consulted for more detailed information about usage and for additional options.

The memory management subsystem has been enhanced with each successive release of HP-UX. This paper describes HP-UX 11i v3. Features introduced since the initial release of HP-UX 11i v3 are noted in appendix 1.

Most aspects of memory management are independent of the hardware platform architecture. Where there are variations with platform architecture, this paper will describe the behavior on Integrity platforms. Differences in memory management on HP Integrity platforms versus HP9000 PA-RISC platforms are described in appendix 2.

The other appendices feature operation of the pager, management of swap space, and the source code for the locinfo program.

The memory management subsystem

The purpose of the memory management subsystem is to control all of the physical memory in a computer system and make that memory available to other parts of the operating system and application programs. Physical memory, also called core or RAM, is the fast, non-persistent storage residing on the computer motherboard in packages called DIMMs. For application programs, the memory is made available only through a set of abstractions called virtual memory. From the viewpoint of application programs, virtual memory is

- laid out according to the memory model of the source programming language, concealing the fact that the physical memory may be of limited size and/or non-contiguous.
- private to the program and protected from access by other programs, unless explicitly shared through the interfaces provided for that purpose.

To handle the case where the memory usage of all application programs exceeds the capacity of the physical memory in the computer, the memory management subsystem must also control some storage, nearly always on disk, called swap space. Any data in memory not in active use at that instant can be copied into the swap space to make the memory space available for some other purpose. When the data in swap is needed again, it can be copied from the storage back into memory.

HP-UX is no longer limited to swapping out entire processes – instead, paging is used to relieve memory pressure. The historical name has been retained, but it is more accurate to think of the storage as "paging space".

Because the access times for disks are much longer than memory access times, the behavior of the memory management subsystem exerts a profound influence over total system performance. This paper provides information to help make that performance as good as possible.

Tools to examine the memory management subsystem

The main tools and utilities used to gain insight into the behavior of the memory management subsystem are top, swapinfo, glance, and sar. For NUMA systems configured with local memory, the pstat_getlocality() system call shows how the memory is distributed. An example program, locinfo, shows how this call can be useful. The command ipcs displays information about shared memory usage. Beyond these tools, HP technical support teams have specialized tools to probe into the internals of memory management subsystem operation. Those are not described in this paper – the support teams can install those tools to analyze the more complex situations.

Top

The **top(1)** command gives statistics about the top processes on the system, as well as system memory size and utilization. For the example below, the command was invoked as **top -h**, to hide the information about individual CPUs.

```
System: vmkona1                                      Wed Jan  6 10:42:36 2010
Load averages: 0.29, 0.31, 0.26
353 processes: 275 sleeping, 78 running
Cpu states: (avg)
 LOAD   USER   NICE    SYS   IDLE  BLOCK  SWAIT   INTR   SSYS
 0.29   9.3%   0.0%   7.8%  82.9%   0.0%   0.0%   0.0%   0.0%

System Page Size: 4Kbytes
Memory: 100362036K (15489832K) real, 102756728K (16389916K) virtual, 127850076K free
Page# 1/26

CPU TTY     PID USERNAME PRI NI   SIZE    RES STATE    TIME %WCPU  %CPU COMMAND
20   ?      114 root     152 20    72K    64K run      1:44  6.84  6.83 swapdaem
55   ?     2253 root     -16 20   283M   259M run      6:02  6.56  6.54 midaemon
 8   ?      134 root     152 20  2880K  2560K run     34:07  0.99  0.99 vxfsd
54   ?     1868 root     154 20  5592K   532K sleep    0:19  0.91  0.90 pwgrd
17 pts/th 22940 root     154 10 45616K  6080K sleep    0:07  0.83  0.83 glance
 2   ?        1 root     152 20  2480K   656K run      0:10  0.66  0.66 init
36   ?     2315 root     152  0 47372K  7848K run      6:56  0.57  0.57 perfd
23 pts/ta 21072 oracle   152 20 63448K  8328K run      0:00 11.01  0.54 orabm
18   ?     2626 root     168 20 15424K  1180K sleep   10:15  0.48  0.47 utild
52 pts/tc 27689 root     178 20 10860K  1656K run      0:05  0.41  0.41 top
 7 pts/ta 22343 oracle   158 20  4848K   564K sleep    0:04  0.28  0.28 sh
34   ?        0 root     127 20    72K    64K sleep    0:18  0.21  0.21 swapper
```

The data relevant to memory management are on the row labeled **Memory** and the columns of process information labeled **SIZE** and **RES**.

The **Memory** row shows three key pieces of information:

- how much real (physical) memory is being consumed by all processes in the system and how much of that is considered "active" (in parentheses). In this example, all processes are consuming a total of 100362036 KB of physical memory, of which 15489832 KB is active, that is, belonging to a process that is running or has run in the last 20 seconds. This is equivalent to 95.7 GB of memory used, with 14.8 GB active.
- the corresponding information about the consumption of virtual memory. The processes are using 102756728 KB of virtual memory, of which 16389916 KB is active (98 GB of virtual memory, 15.6 GB active). .
- the amount of free physical memory in the system. The amount of free memory is 127850076 KB, equivalent to 121.9 GB, which is plenty.

In addition:

- The **SIZE** column shows the total virtual size of the process. This includes the virtual sizes of text, data, stack, mmap regions, shared memory regions, and I/O mapped regions.
- The **RES** column gives an indication of the resident size of the process, that is, the portion of the process's memory that is currently occupying physical memory. Paging activity can occur at any time, so this figure is just an approximate value. It is not meaningful to take a sum of these columns: each memory object that is shared would be counted multiple times, once for each sharing process.

The total virtual size is always at least as large as the resident size, because the former accounts for all memory that has been defined by the process, even that which has not been accessed recently, or ever, and so is not in core.

Swapinfo

The **swapinfo(1M)** command reveals information about the size and utilization of swap space. For the example below, the command was invoked as **swapinfo -mt** to display values in megabytes instead of kilobytes, and to provide a row showing the totals.

```
# /usr/sbin/swapinfo -mt
             Mb      Mb      Mb   PCT  START/      Mb
TYPE      AVAIL    USED    FREE  USED   LIMIT RESERVE  PRI  NAME
dev        8192       0    8192    0%       0       -    1  /dev/vg00/lvol2
localfs      64       0      64    0%      64     125    2  /tmp/paging
reserve       -    8256   -8256
memory   249096  116070  133026   47%
total    257352  124326  133026   48%       -     125    -
```

The first row under the headings shows swap space of the type **dev**; that is, an entire physical disk device or logical volume, in this case **/dev/vg00/lvol2**. The second row shows swap space of the type **localfs**; that is, on a device or logical volume that holds a local file system. In this case, the directory called **paging** was created to provide swap space on the file system **/tmp**. Paging to a physical device is faster, but it requires dedicating the entire device to swap. Paging to a file system incurs more overhead than a device, but the disk space not actually used for swap is available to store files.

The default policy of HP-UX is to reserve sufficient swap space for every process at the time it is created. This guarantees that there is always a place to put the data for every process in the system -- it is never necessary to kill a process because paging out is necessary but swap space is full. The row marked **reserve** indicates how much swap space has been reserved for processes but not actually allocated. To compute the total accurately, the amount of space so reserved appears with a negative sign under the **FREE** column, as it represents a claim upon space that might have to be allocated at any time.

The calculation as to how much space is needed for all of the system processes includes the portion of memory that will hold process data. Space accounted for in this way is referred to as "pseudo-swap" and is indicated on the row marked **memory**. Pseudo-swap is used for all memory locked by the kernel or by applications, and also when all disk swap is exhausted. The operating system decides how much memory should be used for pseudo-swap. It is never the full amount of physical memory: the amount in the **FREE** column is the free pseudo-swap; it is not intended to match the true amount of free memory reported by the other tools.

This particular system has plenty of swap space. Note that in the **total** row, only 48% of the available swap space is actually used. At any utilization less than 80%, the system is operating quite comfortably and has sufficient swap space. When the swap utilization exceeds 90%, it might be appropriate to add more swap space.

The system also has plenty of memory -- enough so that no memory has actually been swapped out to either of the swap devices: both the device and the paging file system report 0 space used. Note that the amount of reserved space matches the sum of the space available on the two swap devices. The rest of the process virtual memory has pseudo-swap allocated for it.

HP GlancePlus

glance(1M), part of the HP GlancePlus product, is a valuable utility that gives insight into many aspects of system performance, to include memory utilization and paging activity. Although **glance** has a graphical version for Motif, the examples in this paper are for the text-based version. There is a wealth of information available through **glance**; the four screens most relevant to memory management are illustrated below. For the examples, the **glance** command was invoked with no arguments, and a single keystroke used to select the screen of interest. To examine the memory characteristics of a specific process, its process identifier (PID) must be specified.

The following screen is for the Memory Report, selected by the keystroke **m**:

```
Glance C.05.00.000            10:45:40  vmkona1     ia64    Current  Avg  High
--------------------------------------------------------------------------------
CPU  Util   S  SU   U                                       | 18%    21%   27%
Disk Util   F                                               |  2%    43%  100%
Mem  Util   S   SU                 UF F                     | 52%    52%   52%
Swap Util   U                     URR                       | 48%    48%   48%
--------------------------------------------------------------------------------
                                 MEMORY REPORT                        Users=   10
Event         Current   Cumulative   Current Rate   Cum Rate   High Rate
--------------------------------------------------------------------------------
Page Faults    112823     22025420      18495.5    18293.5     23200.6
Page In             1          100          0.1        0.0      9657.7
Page Out            0            0          0.0        0.0         0.0
KB Paged In       4kb        400kb          0.6        0.3       101.2
KB Paged Out      0kb          0kb          0.0        0.0         0.0
Reactivations       0            0          0.0        0.0         0.0
Deactivations       0            0          0.0        0.0         0.0
KB Deactivated    0kb          0kb          0.0        0.0         0.0
VM Reads            0            0          0.0        0.0         0.0
VM Writes           0            0          0.0        0.0         0.0
Total VM :  98.0gb   Sys Mem  :  26.0gb   User Mem:  95.7gb   Phys Mem : 255.7gb
Active VM:  18.5gb   Buf Cache:     0mb   Free Mem: 121.9gb   FileCache:  12.2gb
MemFS Blk Cnt:                0   MemFS Swp Cnt:                0   Page 1 of 1
```

The statistics shown here are fully consistent with those given by the top command. In addition to the 95.7 GB of user memory, the Memory Report also shows that the operating system is using 26 GB (**Sys Mem**). The overall memory utilization (**Mem Util**) is then calculated at 52%, which is comfortably low.

The information also agrees with the swapinfo command: the overall swap utilization is 48%. The current rate of paging in is also low. The peak amount of page in activity, shown in the **High Rate** column of the **KB Paged In** row, corresponds to the demand paging generated by the startup of an application. **glance** reports absolutely no page out activity, evidenced by the zeros in the **KB Paged Out** row. This row reports the sum of the page outs to swap space plus the page outs from the filecache. That filecache is using 12.2 GB of memory. (**glance** reports the value of the **filecache_min** kernel tunable parameter if the actual filecache consumption is less than the parameter value.)

To summarize the memory profile: of the 255.7 GB of physical memory, 37% is occupied by user processes, 10% by the kernel, 5% by the filecache, and 48% is free.

Note that there are no process deactivations and reactivations. Nonzero entries on the **Reactivations** or **Deactivations** lines would be an alarming signal, indicating that more physical memory is needed to support the imposed workload.

The following screen is for Swap Space, selected by the keystroke **w**:

```
Glance C.05.00.000            10:46:38  vmkona1     ia64    Current  Avg  High
--------------------------------------------------------------------------------
CPU  Util   S   SU   U                                     | 18%    21%   27%
Disk Util   F                                              |  2%    42%  100%
Mem  Util   S   SU               UF F                      | 52%    52%   52%
Swap Util   U                   URR                        | 48%    48%   48%
--------------------------------------------------------------------------------
                                   SWAP SPACE                      Users=   10

Swap Device                     Type        Avail       Used     Priority
--------------------------------------------------------------------------------
/dev/vg00/lvol2                device       8.0gb        0mb        1
/tmp/paging                    filesys       64mb        0mb        2
pseudo-swap                    memory     243.3gb    113.4gb       -1

Swap Available: 257352  Swap Used: 116159  Swap Util (%):  48  Reserved: 124415
```

This is another way to see the same information about swap space reported by **`swapinfo`**. In the column labeled **`Priority`**, the smaller numerical value indicates higher priority, except for pseudo-swap, which is used only when all device swap is exhausted.

The following screen is for the System Tables Report, selected by the keystroke **t**:

```
Glance C.05.00.000            10:47:19  vmkona1     ia64    Current  Avg  High
--------------------------------------------------------------------------------
CPU  Util   S  SU   U                                      | 18%    21%   27%
Disk Util   FF                                             |  3%    41%  100%
Mem  Util   S   SU               UF F                      | 52%    52%   52%
Swap Util   U                   URR                        | 48%    48%   48%
--------------------------------------------------------------------------------
                              SYSTEM TABLES REPORT                 Users=   10

System Table                 Available        Used      Utilization      High(%)
--------------------------------------------------------------------------------
Proc Table (nproc)                4200         353            8             9
File Table (nfile)          2147483647        1618            0             0
Shared Mem Table (shmmni)          400         111           28            28
Message Table (msgmni)             512           2            0             0
Semaphore Table (semmni)          2048          37            2             2
File Locks (nflocks)              4096          52            1             1
Pseudo Terminals (npty)             60           0            0             0
Buffer Headers (nbuf)               na       47826           na            na
```

The operating system must allocate a certain amount of kernel memory for each of the objects actually in use. From the example, there are 111 shared memory segments in use by applications, each of which requires a modest amount of kernel memory for its management.

The following screen is for the Process Memory Regions, chosen by the keystroke **M** and then the PID **8121** to examine an sqlplus process:

```
Glance C.05.00.000            10:49:36  vmkona1     ia64    Current  Avg  High
--------------------------------------------------------------------------------
CPU  Util   S   SU   U                                          | 19%   21%   27%
Disk Util   F                                                   |  2%   38%  100%
Mem  Util   S   SU                 UF F                         | 52%   52%   52%
Swap Util   U                     URR                           | 48%   48%   48%
--------------------------------------------------------------------------------
Regions PID:      8121, sqlplus      PPID:      3153 euid:  4000 User:oracle

Type          RefCt    RSS     VSS   Locked  File Name
--------------------------------------------------------------------------------
NULLDR/Shared  166     4kb     4kb      0kb <nulldref>
MEMMAP/Shared   85     4kb     4kb      0kb /var/spool/pwgr/status
TEXT  /Shared    3     4kb     4kb      0kb /opt/.../db/bin/sqlplus
DATA  /Priv      1   1.8mb   2.1mb      0kb /opt/.../db/bin/sqlplus
UAREA /Priv      1    64kb    72kb      0kb <uarea>
MEMMAP/Priv      1   392kb   400kb      0kb /opt/.../timezlrg.dat
MEMMAP/Priv      1    16kb    24kb      0kb /usr/lib/tztab
MEMMAP/Priv      1     8kb     8kb      0kb <mmap>

Text  RSS/VSS:  4kb/  4kb  Data  RSS/VSS:1.8mb/2.1mb  Stack RSS/VSS: 80kb/ 80kb
Shmem RSS/VSS:  0kb/  0kb  Other RSS/VSS: 21mb/ 54mb
                                                                   Page 1 of 12
```

This example shows just the first page describing each of the memory objects used by the process. The **RefCt** column is the reference count, the total number of processes that use the object (and the count includes the number of references added by the filecache). The shared process **TEXT** has a total of three references, while the private objects have just one reference each. The **RSS** column is the resident set size (that is, the approximate amount of physical memory currently consumed by the process) and **VSS** is the corresponding virtual size.

Sar

The system activity reporter, **sar(1M)**, shows information about operating system activity. Of the many options, the ones relevant to memory management are for processor run queue lengths and for swapping and switching activity.

The example for processor run queue lengths was obtained by invoking the command as **sar -q 60 1**:

```
# sar -q 60 1

HP-UX vmkona1 B.11.31 U ia64    01/06/10

10:53:33 runq-sz %runocc swpq-sz %swpocc
10:54:33     1.6       7     0.0       0
```

The information about processor run queue lengths and occupancy is another way to view processor utilization, and is consistent with what the other tools show.

The example for swapping and switching activity was obtained by invoking the command as **`sar -w 60 1`**:

```
# sar -w 60 1

HP-UX vmkona1 B.11.31 U ia64    01/06/10

10:51:29 swpin/s bswin/s swpot/s bswot/s pswch/s
10:52:29    0.00     0.0    0.00     0.0   37547
```

In this example, the **`0.0`** in the **`bswin/s`** column indicates that there are no 512-byte units per second being transferred from swap space into memory and the **`0.0`** in the **`bswot/s`** column indicates that no 512-byte units per second are being transferred from memory out to swap space. It is not unusual to see a moderate amount of paging in: some objects are brought into memory only upon demand, that is, when they are actually referenced. However, it is generally undesirable to see any significant amount of paging out, because it is likely that some high-performance processors will become stalled waiting for slow disk accesses to complete. Paging activity is an indication that the total memory footprint of all processes executing on the system exceeds the amount of physical memory available. The appropriate remedy is to add physical memory, if such is available through Instant Capacity or there is unassigned memory in a vPars environment. The other way to reduce memory pressure is to trim the workload imposed on the system, perhaps by redirecting users to some other server or reconfiguring the memory requirements of the application (for example, by reducing the size of the Oracle SGA).

Locinfo to illustrate pstat_getlocality() call

The **`pstat_getlocality(2)`** system call can be invoked by application programs to gain information about memory distribution in NUMA platforms. The **`locinfo`** program gives an example of usage. The full source code for **`locinfo`** is given in appendix 5. Information about the attributes of cells in NUMA platforms is available in the HP Superdome 2 Partitioning Administrator Guide.

```
  --- System wide locality info: ---
 index   ldom physid   type  spus     total      free     float      used
     0      0      0 C{S}LM     6    55.94G    24.68G         0    31.26G
     1      1      1 C{S}LM     6    55.93G    28.98G         0    26.95G
     2      2      2 C{S}LM     6    55.94G    28.95G         0    26.99G
     3      3      3 C{S}LM     6    63.94G    38.02G    38.02G    25.92G
     4     -1     -1    ILM     0    24.00G     7.49G         0    16.51G
                                      -----     -----     -----     -----
                                    255.75G   128.12G    38.02G   127.63G
```

The type of the memory locality is either interleaved memory (ILM) or one of three types of local memory: cell local memory (CLM) for hardware platforms having a cellular architecture, or socket local memory (SLM) or blade local memory (BLM) for hardware platforms composed of blades. From the operating system's perspective, CLM and SLM are equivalent -- both permit the fastest possible memory accesses by local processors -- and so are marked "**`C{S}LM`**" in the output. Blade local memory is interleaved between the two sockets on the blade, but not across multiple blades.

This system has four CLM localities corresponding to the four cells numbered 0, 1, 2, and 3. The locality domain identified as -1 is ILM, composed of memory drawn from the three base cells. Cell 3 is a floating cell, and so contains only local memory; it does not contribute to the memory interleave.

This particular system has 64 GB of physical memory on each of the four cells. Cells 0, 1, and 2 are base cells and were configured to have $^{7}/_{8}$ of their memory (that is, 56 GB) as CLM leaving $^{1}/_{8}$ (8 GB) as ILM. (Some memory is always used by firmware, and memory pages showing errors may be deconfigured, so the amount of memory available to the operating system is somewhat less than the nominal capacity.)

Cell 3 was designated as a floating cell to allow for the possibility of online deactivation. The kernel never places certain data structures in floating memory, but the floating memory is well utilized by applications.

Interleaved memory is used for objects that are expected to be referenced globally, such as program text and shared kernel data. The example system needs just over 16 GB of interleaved memory; configuring it with less than that amount would lead to suboptimal performance.

Here is the memory distribution for the sqlplus process examined previously, obtained by using the **`-p`** option to **`locinfo`**:

```
--- Per-process locality info for pid 8121: ---
   idx     total    shared   private  weighted
     0     2.14M       20K     2.12M     2.12M
     1      884K         0      884K      884K
     2      832K       16K      816K      816K
     3         0         0         0         0
     4    19.15M    18.78M      376K     3.92M
           -----     -----     -----     -----
          22.96M    18.82M     4.15M     7.70M
```

This process has a total memory footprint of almost 23 MB. The biggest part of this, almost 19 MB, is shared memory. Note how most of the shared memory is allocated from the locality at index 4, the interleaved memory locality, which is favorable for global access. The process also has slightly more than 2 MB of private data. This is concentrated in the locality at index 0, which is on cell 0. The scheduler in the kernel will try to schedule the process to run on a processor that is in cell 0, so that the memory access latency time is minimized.

The column marked **`weighted`** indicates the amount of resident memory for this process, weighted by the number of processes sharing the memory. A private page counts as one page; each shared page counts as a page divided by the number of processes sharing that page. For the example sqlplus process, most of the local memory was private. The shared memory was mostly drawn from the interleaved locality, and was shared with three or four other processes.

Ipcs

The **`ipcs(1)`** command reveals information about interprocess communication facilities, such as shared memory. In the example below, the command was invoked as **`ipcs -ma`**, to show all information about shared memory.

```
IPC status from /dev/kmem as of Wed Jan 6 13:54:36 2010
T         ID     KEY        MODE        OWNER     GROUP   CREATOR    CGROUP NAT
Shared Memory:
m          0 0x411c01a8 --rw-rw-rw-      root      root      root      root    6
m          1 0x4e0c0002 --rw-rw-rw-      root      root      root      root    6
m          2 0x4120de56 --rw-rw-rw-      root      root      root      root    6
m          3 0x411c7910 --rw-------      root      root      root      root    5
m          4 0x06347849 --rw-rw-rw-      root      root      root      root    6
m     131077 0x0c6629c9 --rw-r-----      root      root      root      root    8
m      65542 0x49189ef5 --rw-r--r--      root      root      root      root    9
m    9371655 0x00000000 --rw-r-----    oracle       dba    oracle       dba    4
m     196616 0x00000000 --rw-r-----    oracle       dba    oracle       dba    4
m     163849 0x00000000 --rw-r-----    oracle       dba    oracle       dba    4
m     229386 0x00000000 --rw-r-----    oracle       dba    oracle       dba    4
m     229387 0x00000000 --rw-r-----    oracle       dba    oracle       dba    4
m     229388 0x9e466d34 --rw-r-----    oracle       dba    oracle       dba    4
m     917517 0x00a5c581 --rw-------     sfmdb     users     sfmdb     users    1
```

Interpreting the output from the tools

Does my system have enough memory?

The tools described above can be used to determine if a system has enough memory for the workload it is running. The simplest way is to check the memory utilization reported by `glance` at the top of each screen. A good target value for memory utilization is around 85% to 90%. A value lower than 50% indicates that the system is over-provisioned and you're not getting the full value from all the memory that you paid for. It's nice to have a little bit of headroom in memory space, so operating at memory utilizations above 95% is not preferred. With utilization so high, it is likely that system performance will be throttled by memory availability.

As long as memory utilization is below 95%, the amount of physical memory is probably not limiting system performance in any way. Memory utilizations above 95% still might not indicate a memory bottleneck, as HP-UX typically caches file data in memory quite aggressively. But if the memory utilization is high and there is any significant amount of page out activity, it is likely that memory availability is throttling system performance. (Page out activity at lower memory utilizations is probably just the removal of unneeded pages from the file cache, which is normal and healthy.) Other evidence of insufficient system memory is any nonzero value in the **`USED`** column for any disk device in the output from the **`swapinfo`** command.

If the amount of physical memory if the system bottleneck, it may be possible to relieve the pressure by reducing the size of the filecache or by trimming the application workload. If those remedies are not desirable, then it may be necessary to increase the amount of RAM.

Does my system have enough swap space?

If the system has sufficient memory, then it will not resort to swapping very often. Still, it is essential to provide enough swap to support the reservations needed to launch processes.

If your system does not have enough swap space, you might not be able to launch as many processes as can be comfortably sustained by the memory and processor resources. Worse yet, it is possible that a running process might not be able to allocate additional memory and so may crash. The best way to check if the system has enough swap is to see how much space **`swapinfo`** reports as free. When the total amount of free swap space is less than a few percent of the size of physical memory, consider adding more swap. It is possible to form a more precise estimate if you know the virtual memory demands of processes that will be launched in the future, for example, the SGA size of a new instance of Oracle.

The example system

The example system vmkona1 was running the Oracle Database Management System with an SGA of approximately 100 GB. You should always provision an Oracle system such that there is sufficient physical memory to accommodate the entire SGA, so that the operating system will not have to perform any paging of the SGA, allowing the database to run with peak performance. The Oracle application itself will manage transfers of data between main memory and disk, but this is not system paging activity.

The example system was over-provisioned with memory, as evidenced by the fact that slightly over half of the physical memory was free. Following is example output for a system, vmkona2, running a similar Oracle workload, but with only 32 GB of physical memory, and so it shows signs of memory pressure.

The output from the **top** command shows a similar processor load to vmkona1, but there is much less physical memory in the system and much less memory is free:

```
System: vmkona2                                      Fri Feb 26 13:52:27 2010
Load averages: 0.91, 0.94, 0.94
313 processes: 231 sleeping, 82 running
Cpu states: (avg)
 LOAD   USER   NICE    SYS   IDLE  BLOCK  SWAIT   INTR   SSYS
 0.91  86.3%   0.0%   1.0%  12.7%   0.0%   0.0%   0.0%   0.0%

System Page Size: 4Kbytes
Memory: 18974472K (14635652K) real, 43317892K (33395812K) virtual, 123284K free

CPU TTY  PID USERNAME PRI NI   SIZE    RES STATE    TIME %WCPU  %CPU COMMAND
21   ?     3989 oracle   128 20 40427M  6296K sleep    0:00  4.09  4.06 oracleor
17   ?      125 root     152 20 32760K 29120K run    147:46  1.57  1.56 vxfsd
53   ?     2372 root     152 20 54388K  9064K run     15:54  0.27  0.27 cimserve
20   ?     2639 root     -16 20   294M   263M run     52:19  0.26  0.26 midaemon
 2   ?    22129 oracle   154 20 40436M 13608K sleep    0:27  0.22  0.21 ora_j000
50 pts/ta  3912 oracle   178 20  8144K   908K run      0:00  0.20  0.20 top
36   ?     2375 root     152 20 40256K   540K run     36:54  0.09  0.09 cimprova
39   ?      127 root     152 20  2376K  2112K run     26:31  0.06  0.06 pm_sched
39   ?    21539 oracle   152 20   570M 69136K run      9:48  0.06  0.06 java
 0   ?     2261 root     154 20 10852K   540K sleep    9:26  0.05  0.05 sendmail
22   ?     2644 root     127 20 50044K  4368K sleep    5:19  0.03  0.03 scopeux
 0   ?    16700 oracle   154 20 40424M  3008K sleep    6:33  0.02  0.02 ora_mmn
```

The output from the **swapinfo** command shows that there has been sufficient memory pressure to use swap space on both of the available disk devices.

```
             Mb      Mb      Mb   PCT  START/      Mb
TYPE      AVAIL    USED    FREE  USED   LIMIT RESERVE  PRI  NAME
dev        8192     329    7863    4%       0       -    1  /dev/vg00/lvol2
dev      133120     329  132791    0%       0       -    1  /dev/swap/swaplv1
reserve       -   40316  -40316
memory    30915   12266   18649   40%
total    172227   53240  118987   31%       -       0    -
```

Note that the two swap devices have the same priority, and so have been used equally; each reporting 329 MB of space used. Another 40316 MB of swap space is reserved, but the smaller swap device isn't large enough to take half of it. This would force unequal distribution across the two devices. It is better to use devices of the same size in order to maintain equal balance.

The output from **`glance`** shows that memory utilization is 99%.

```
Glance C.04.60.000             13:50:38  vmkona2     ia64    Current  Avg  High
--------------------------------------------------------------------------------
CPU  Util   SU                                    U             | 77%   77%   77%
Disk Util   F             FVV                                   | 34%   26%   34%
Mem  Util   S                  SU                            U  | 99%   99%   99%
Swap Util   U  UR           R                                   | 31%   31%   31%
--------------------------------------------------------------------------------
                                 MEMORY REPORT                      Users=    4
Event          Current   Cumulative   Current Rate   Cum Rate   High Rate
--------------------------------------------------------------------------------
Page Faults        97          410         12.9       50.0       447.1
Page In           113          243         15.0       29.6        61.4
Page Out            0            0          0.0        0.0         0.0
KB Paged In     452kb        972kb         60.2      118.5      2971.2
KB Paged Out      0kb          0kb          0.0        0.0         0.0
Reactivations       0            0          0.0        0.0         0.0
Deactivations       0            0          0.0        0.0         0.0
KB Deactivated    0kb          0kb          0.0        0.0         0.0
VM Reads           31          101          4.1       12.3         9.3
VM Writes           0            0          0.0        0.0         0.0
Total VM :  41.2gb   Sys Mem  :  12.6gb   User Mem:  18.8gb   Phys Mem :  31.7gb
Active VM:  29.7gb   Buf Cache:     0mb   Free Mem:   319mb   FileCache:    1.5gb
```

Note that the amount of free memory is 319 MB, which is approximately 1% of the 31.7 GB of physical memory available to the system. This is typical behavior for HP-UX: it will aggressively use memory to cache objects residing in files, and will push memory objects out to disk only when the amount of free memory falls below the 1% threshold. (Depending on the patch level of HP-UX, the threshold value may be tunable by the system administrator.)

The size of the filecache is 1.5 GB, approximately 5% of physical memory.

The output from the **`sar`** command completes the picture:

```
$ sar -q 60 1

HP-UX vmkona2 B.11.31 U ia64    02/26/10

13:45:04 runq-sz %runocc swpq-sz %swpocc
13:46:04     1.1      26     0.0       0

$ sar -w 60 1

HP-UX vmkona2 B.11.31 U ia64    02/26/10

13:46:04 swpin/s bswin/s swpot/s bswot/s pswch/s
13:47:04    0.00     0.0    0.00     0.0     587
```

During the measurement interval, there was no paging activity reported.

Configuration and tuning opportunities

HP-UX aims to be simple to use. Its behavior straight out of the box should be completely satisfactory for most uses, with no need to perform manual configuration or tuning. However, both the size of the host servers and the workloads imposed upon them vary widely across the range of scenarios in which HP-UX is deployed. For example, system memory size can be as small as 1.5 GB or as large as 8 TB. Applications may create thousands of transient processes that manipulate small messages, or they may have a modest number of long-lived processes that access huge data structures. The HP-UX solution is to offer solid out-of-the-box defaults, but to provide the means to perform fine tuning. This section gives some guidance if tuning is desired.

Local memory versus interleaved memory

HP servers with NUMA (non-uniform memory access) are structured as a complex of modular building blocks each containing memory local to the physical component. When an nPartition (a hardware partition featuring both electrical and operating system isolation) is formed from multiple components, any or all of the memory in the partition can be interleaved. Interleaving, which is effected through the **`parcreate(1M)`** or **`parmodify(1M)`** command, distributes memory references evenly across all involved localities. Interleaving is established at the granularity of a cacheline, which is the quantum of data for communication between processors and memory. Memory not chosen to be interleaved remains as local memory, which can be accessed much more quickly by processor cores that reside in the same locality.

For most commercial applications, the sweet spot is a memory configuration with the ratio $^7/_8$ local memory to $^1/_8$ interleaved memory. Workloads exhibiting no locality of memory reference, such as certain technical applications, might do better with 100% interleaved memory, but the recommended $^7/_8$ to $^1/_8$ ratio is a good starting point for commercial applications.

A program such as **`locinfo`** reveals the relative utilization of the various memory localities. If the interleaved locality is significantly over-utilized or under-utilized, it might be appropriate to change the interleaved memory ratio through the **`parmodify`** command. This requires a reboot to take effect, so should be performed only if it is acceptable to reboot the nPartition.

In a vPars environment, the vPars instances can be built from local memory if the underlying nPartition was so configured. Any virtual partition that contains memory from more than one locality should also include interleaved memory, with $^1/_8$ of the total memory in the instance being the suggested amount. A virtual partition that contains resources from only one locality need not have any interleaved memory at all.

Ejectable (floating) memory

HP-UX supports dynamic platform reconfiguration: physical resources such as entire cell boards or memory ranges can be online activated or deactivated without interrupting the application workload. A cell board that can be online deactivated is called a floating cell; one that cannot be deactivated without a reboot is called a base cell. In a vPars context, a memory range that can be online deactivated is called ejectable or floating; one that cannot be deactivated without a reboot is called non-ejectable or base.

The designation of a cell as base or floating is accomplished via the **`parcreate`** or **`parmodify`** command. The designation of vPars memory as base or floating is accomplished via the **`vparcreate`** or **`vparmodify`** command. On HP Superdome 2 platforms, these operations can also be performed through the Onboard Administrator.

The HP-UX memory management subsystem never places certain kernel data structures in floating memory. This important optimization allows the kernel to perform smoothly during dynamic platform reconfiguration events. The consequence is that the kernel can be starved for memory on a platform configured with too little base memory.

The minimum requirement is to keep at least 25% of memory as base memory. Smaller systems (around 2 GB of memory) may require more than the minimum; larger systems can operate at the minimum comfortably. The **`locinfo`** program can be used to determine the relative utilization of base and floating memory. If a system is starved for base memory but has excess floating memory, it is possible to deactivate some of the floating memory and then reactivate it as base memory, avoiding the need to reboot the system.

The HP-UX Virtual Partitions Administrator's Guide contains more detailed recommendations for configuring base and floating memory in vPars scenarios.

System base page size

A base page is the unit of physical memory management in the HP-UX kernel. It is the smallest block of physical memory that can be allocated for storing data and code and it is the smallest unit of memory protection.

The size of a base page is a tunable parameter for Integrity platforms starting with the September 2008 Update to HP-UX 11i v3. The parameter is tunable, but any change requires a reboot to take effect.

The **`getconf(1)`** command can be used to obtain the base page size of the system. The base page size is tuned by invoking the **`kctune(1M)`** command to change the kernel tunable parameter **`base_pagesize`**. The possible values for this page size are 4 KB [the default], 8 KB, and 16 KB. (The page size 64 KB is also supported, but it is intended solely for the use of the Integrity Virtual Machines host, and is too large for other uses.)

A larger base page size always reduces the kernel memory footprint, and some memory management operations become more efficient. In general, memory allocations are rounded up to a multiple of the base page size, which may be wasteful if many small memory objects are created by applications. Most application workloads measured at HP show a modest (1% to 4%) performance increase when the page size is 16 KB instead of the default 4 KB. Unfortunately, none of the analysis tools mentioned in this paper can predict what workloads will benefit from the larger page size. Experimentation in a controlled environment is the best way to determine the optimal page size.

In cases where the kernel memory footprint is of primary importance, the larger base page sizes incur less memory overhead. The fixed per-page overhead consumes just over 7% of memory when the page size is at its 4 KB default; this is reduced to approximately 3.6% for 8 KB pages and 1.8% for 16 KB pages.

Configuring swap space

The primary command for configuring swap space is **`swapon(1M)`**, which directs that storage be made available as swap space to support paging activity. HP-UX always reserves sufficient swap space for each process that is created. If the system does not have enough swap space to sustain a new process, the attempt to create it will fail. Similarly, if an existing process attempts to allocate more memory such that swap space would be exhausted, the allocation attempt will fail. For those reasons, you should consider adding swap space when the **`swapinfo`** command indicates that the amount of free swap space is small. If the additional swap space is needed only to support additional reservations for a larger process address space, it may be more convenient to add swap on a part of a file system, as the slower performance is not a consideration unless the swap space is actually used.

If there is an abundance of unused swap space, the counterpart command **`swapoff(1M)`** can discontinue the use of a device for swap, making it available for some other purpose.

Sizing the filecache

The minimum and maximum amounts of memory devoted to caching file data are controlled by the tunable kernel parameters **`filecache_min(5)`** and **`filecache_max(5)`**. Memory in the fraction specified by the parameter **`filecache_min`** is dedicated exclusively to accelerating file I/O activity. The memory is not available for any other purpose, even if it is not needed to cache file data. The parameter **`filecache_max`** specifies the maximum size of the filecache. Memory space beyond **`filecache_max`** is used for other purposes, such as caching process dynamic data and kernel data.

As long as the filecache is below its maximum size, it will grow in response to accesses to additional file data. Once the limit established by **`filecache_max`** is reached, cached file data will be pushed from memory to the file system to make room for newly referenced file data.

The filecache parameters trade-off accelerated performance for file I/O activity against all other competing uses for memory. The default settings of the parameters reserve approximately 5% of physical memory for the filecache, but do not allow it to grow past 50% of physical memory. An extensive survey of the HP-UX customer base showed that almost 60% of the systems retained the default minimum filecache size of 5%. About 34% of the systems tuned the **`filecache_min`** parameter downward into the range of 1% to 3%. Only 6% of the systems adjusted the parameter upwards. The customer survey showed that almost one third of the systems retained the default maximum filecache size of 50%. Two thirds of the systems tuned the **filecache_max** parameter downward into the range of 5% to 10% of physical memory. A small minority of systems sized the filecache to be more than half of memory.

Tuning JFS (VxFS)

The Journaled File system, also known as HP OnlineJFS, JFS, or VxFS, is often a major consumer of system memory, in particular for its inode cache. The amount of memory used for the JFS cache is controlled by tunable parameters **vx_ninode** and **vxfs_ifree_timelag**.

The JFS inode cache is a holding location for inodes from disk. Each inode in memory is a superset of the inode from disk. The disk inode stores information for each file such as the file type, permissions, timestamps, size of file, number of blocks, and extent map. The in-memory inode stores the on-disk inode information along with information such as pointers to other structures, pointers used to maintain linked lists, and lock primitives used to manage the inode in memory. The inode does not store the file data itself: that goes in the filecache.

Once the inode is brought into memory, subsequent access to the inode can be done through memory without having to read or write it to disk.

For JFS 3.5 and above, you can use the **vxfsstat** command to display the current number of inodes in the inode cache:

```
# /opt/VRTS/bin/vxfsstat / | grep inodes
    93775 inodes current     93874 peak             1331354 maximum
    96910 inodes alloced      3135 freed
# /opt/VRTS/bin/vxfsstat -v / | grep curino
vxi_icache_curino             93566    vxi_iaccess               1244468
```

This shows that the current number of inodes in the cache is 93566, and this count includes both the inodes actively in use and the inactive inodes still stored in the cache.

For JFS 3.5 and above, you can use **vxfsstat** to determine the actual number of JFS inodes actively in use:

```
# /opt/VRTS/bin/vxfsstat -v / | grep inuse
vxi_icache_inuseino            1096    vxi_icache_maxino         1331354
```

The inode cache is filled with 93566 inodes but only 1096 are in use. The remaining inodes are inactive, and if they remain inactive one of the **vxfsd** daemon threads will start freeing the inodes after a certain period of time.

Each inode consumes approximately 2 KB of memory. Having an inode cache larger than is needed by the system workload is wasteful of memory resources. In such a case, it would be appropriate to reduce the size of the cache. You can tune the maximum size of the JFS inode cache using the **vx_ninode** tunable. With JFS 4.1, **vx_ninode** can be tuned dynamically using **kctune**.

At a minimum, you must have at least one JFS inode cache entry for each file that is opened at any given time on your system. If you are concerned about the amount of memory that JFS can potentially take, then tune **vx_ninode** down so that the cache only takes around 1% or 2% of overall memory. Most systems will work fine with **vx_ninode** tuned in the range of 20,000 to 50,000. However, you need to consider how many processes are running on the system and how many files each process will have open on average. Systems used as file servers and Web servers may have performance benefits from using a large JFS inode cache and the defaults are sufficient.

The default value of **vx_ninode** is 0, which means that the system will size the JFS inode cache automatically. The automatic values are computed based on physical memory size according to a sliding scale shown on the **vx_ninode(5)** manpage. For example, an 8 GB system will devote approximately 6% of physical memory to the inode cache. Systems with more memory will have a larger inode cache, but the fraction of physical memory decreases to around 1.5% for systems with memory sizes of 128 GB and above.

Tuning **ninode** does not affect the JFS inode cache, which is maintained separately from the HFS inode cache. If your only HFS file system is **/stand**, then **ninode** can usually be tuned to a low value, for example, 400.

By default, the JFS inode cache is dynamic in size: it grows and shrinks as needed. However, since the inodes are freed to the kernel memory allocator's free object chain, the memory may not be available for use for other reasons (except for other same-sized memory allocations). The freed inodes on the object free-lists are still considered "used" system memory. Also, the kernel memory allocations and subsequent frees add additional overhead to the kernel.

The dynamic nature of the JFS inode cache does not provide much benefit with the current kernel memory allocator. Using a statically sized JFS inode cache has the advantage of keeping inodes in the cache longer and reducing the overhead of continued kernel memory allocations and frees. Instead, the unused inodes are retained on the JFS inode cache free-list chains. If you need to bring a new JFS inode in from disk, JFS uses the oldest inactive inode. Using a static JFS inode cache also avoids long kernel memory object free chains for each processor. Another benefit to a static JFS inode cache is that the **vxfsd** daemon will not use as much processor time trimming the size of the JFS inode cache, which can be considerable on large-memory systems.

Beginning with HP-UX 11i v2 (with JFS 3.5 and above) a new system-wide tunable **vxfs_ifree_timelag** was introduced to vary the length of time an inode stayed in the cache before it is considered for removal. Setting **vxfs_ifree_timelag** to -1 effectively makes the JFS inode cache a static cache. Setting **vxfs_ifree_timelag** is especially useful on large-memory systems so the **vxfsd** daemon does not use too much processor time. The following example uses **kctune** to change **vxfs_ifree_timelag** without a reboot:

```
# kctune vxfs_ifree_timelag=-1
Tunable                        Value  Expression  Changes
vxfs_ifree_timelag  (before)       0  Default     Immed
                    (now)         -1  -1
```

Considerations for application developers

This section contains information about the ways in which an application program is constructed can interact with the properties of the memory management subsystem.

Addressing model

The virtual address space presented to applications is flat and contiguous – 4 gigabytes in size for 32-bit applications and 16 exabytes for 64-bit applications. The virtual memory subsystem, in conjunction with the compiler and linker, impose an addressing model to govern usage of certain parts of the address space. In particular, the addressing model may restrict the range of virtual addresses that can be used for memory objects that are shared among processes.

The Mostly Private Address Space (MPAS) model gives the most flexibility in the usage of the address space, which can be crucial for 32-bit applications accessing large memory objects. The added flexibility came at the expense of significant performance cost, which was remedied to a great extent in the March 2010 Update.

An application can be built to use the MPAS model by specifying the **+as mpas** option to the linker, **ld(1)**. After being built, an application can be converted to MPAS by giving the same option to the **chatr(1)** command.

Local memory

Hardware platforms that have a NUMA structure can be configured with a mixture of local memory and interleaved memory. Application programs can provide hints to request that their memory objects be placed in the interleaved locality, the current locality, or the locality of the first processor to touch the memory. For shared memory objects, the **shmget(2)** call allows the memory hint to be specified in the flags field. For memory mapped files, the corresponding interface is **mmap(2)**.

Application programs can use the memory hints, possibly in conjunction with the processor binding facility offered through the `mpctl(2)` interface, to attempt to gain favorable processor to memory alignment on NUMA platforms. Such attempts are complicated by the following considerations:

- dynamic platform reconfiguration events can cause processors, memory ranges, or entire localities to change composition or disappear entirely
- other applications running on the same server may also attempt to do memory placement and/or processor binding, which can cause imbalances in resource utilization
- in LORA mode, HP-UX automatically strives to give each application the best possible processor to memory alignment on a continuing basis

For those reasons, applications might do better to forego memory placement hints altogether, and allow the operating system to do its own placement. Alternatively, the system administrator can use the `numa_policy(5)` tunable parameter to govern application memory placement on a system-wide basis.

Large Virtual Pages

At the basic level, the HP-UX memory management subsystem always manages memory in units of the system base page size. Applications may work with objects that are much larger than that base page size. In those cases, it may be more efficient to aggregate a range of base pages into a "superpage", also called a "large page". This efficiency is most important in the utilization of the hardware resource called the translation lookaside buffer (TLB).

HP-UX automatically recognizes situations in which large pages are beneficial and manipulates them accordingly. The system administrator can influence the kernel's behavior in this regard through the `vps_ceiling(5)`, `vps_chatr_ceiling(5)`, and `vps_pagesize(5)` tunable parameters.

In some cases, use of variable pages that are too large can cause inefficiencies, for example, if a large memory object is accessed in a sparse manner. Each individual application can provide hints to the kernel as to the best page size through the `chatr(1)` command. In particular, the `+pi` and `+pd` options can be used to suggest the page sizes for code and data, respectively.

For more information

vPars Administrator's Guide	http://docs.hp.com/en/T1335-90104/index.html
Tunable Base Page Size whitepaper	http://docs.hp.com/en/14670/ENW-TBPS-TW.pdf
LORA whitepaper	http://www.docs.hp.com/en/14655/ENW-LORA-TW.pdf
HP GlancePlus product information	http://www.hp.com/go/glance

Adaptive Address Space whitepaper
http://h21007.www2.hp.com/portal/download/files/unprot/Itanium/aas_white_paper.pdf

HP-UX Performance Cookbook
http://h21007.www2.hp.com/portal/download/files/unprot/devresource/Docs/TechPapers/UXPerfCookBook.pdf

HP Superdome 2 Partitioning Administrator Guide

Appendix 1 -- Variations with OS release

Memory management enhancements since the initial HP-UX 11i v3 release

- Floating cells were first supported in the September 2007 Update.
- The Kernel Access Infrastructure feature to allow the mapping of user virtual addresses into the kernel was introduced in the March 2008 Update.
- The `numa_policy` kernel tunable parameter to control the default placement of application memory objects was introduced in the March 2008 Update.
- The system base page size was made tunable by the system administrator in the September 2008 Update. For earlier releases, the system base page size was hardwired to 4 KB.
- The `swapoff` command was provided in the September 2008 Update to allow swap devices to be removed immediately without requiring a system reboot.
- The September 2009 Update reintroduced kernel tunable parameters to control the behavior of the pager, which parameters had been eliminated from the initial release. Specifically, the parameter `lotsfree_pct(5)` governs the point at which the pager begins to move unused pages from memory to the swap devices. When the fraction of free memory in the system is less than the threshold established by `lotsfree_pct(5)`, the pager will eject pages to swap, otherwise it will leave as many pages as possible in memory. The parameter `desfree_pct(5)` establishes the point at which the pager becomes more aggressive in freeing memory by ejecting pages to swap. That is, when the fraction of free memory falls below `desfree_pct(5)`, the pager will take more aggressive measures to free memory, to include putting low priority processes to sleep.
- The March 2010 Update provides enhancements to the implementation of the Mostly Private Address Space (MPAS) model to increase performance significantly, and also promotes the use of larger superpages in vPars instances with ejectable memory.

Differences between HP-UX 11i v2 and HP-UX 11i v3

Unified Filecache

HP-UX versions prior to HP-UX 11i v3 maintained two separate memory caches to speed up repeated accesses to data residing on disk files: the page cache was used for memory mapped files while the buffer cache was used for files accessed through the read and write interfaces. This required extra administrative overhead to size the two caches and used memory space inefficiently. Moreover, since the operating system did not maintain coherency between the two caches, applications that used both the `mmap(2)` and `read(2)/write(2)` interfaces to access the same files had to perform explicit synchronization in order to behave properly.

In HP-UX 11i v3, there is one single unified filecache to hold the memory images of file data, regardless of the interface applications use for access. This is simpler to manage, more efficient in terms of memory usage, and usually gives an increase in performance.

Unconditional use of pseudo-swap

Prior to HP-UX 11i v3, the kernel tunable parameter `swapmem_on(5)` controlled whether or not memory could be used as pseudo-swap space. Since there is no downside and considerable benefit to using pseudo-swap, the feature is enabled unconditionally in HP-UX 11i v3.

Appendix 2 -- Variations with platform architecture

- The system base page size was made tunable by the system administrator on Integrity platforms. On HP9000 PA-RISC, the system base page size is hardwired to 4 KB.
- The Mostly Private Address Space (MPAS) model is provided on Integrity platforms only. PA-RISC platforms implement only the Mostly Global Address Space (MGAS) model.

Appendix 3 – Operation of the pager (vhand)

Examination of the output from the top command illustrates two key points:

- The total amount of virtual memory used by all of the processes running on the system can greatly exceed the total amount of physical memory available to the system.
- Any given process runs happily with only a fraction of its total virtual address space loaded into memory. The virtual address space contains all the code and data defined by the program; real memory is used only for objects actively used by the program. But any portion of the virtual address space can be accessed at any time as the program executes.

The virtual memory subsystem makes judicious use of the physical memory and the swap space to keep all processes running efficiently by moving objects between the two resources. The process that performs page out activity is called **`vhand`**.

`vhand`'s function is to keep memory available by freeing up the least recently referenced pages. It also performs other functions related to maintaining memory availability, such as garbage collection of the kernel memory allocator free lists.

`vhand` uses a two-handed clock algorithm to decide which pages to free. Conceptually, it has two hands (called the "age hand" and the "steal hand") passing through all of memory. The age hand marks each page as "not recently referenced". The steal hand follows after a delay, and checks each page to see whether it's been accessed (and so marked as recently referenced) since the first hand cleared its referenced bit. Those which have not been accessed may be stolen (paged out and the memory made available to other users).

In actual implementation, **`vhand`** steps through memory by following a doubly linked list of **`pregions`**. A **`pregion`** represents a range of virtual memory in an address space. It doesn't step through all **`pregions`** each time it is woken, and normally looks at only a portion of the pages in each **`pregion`**.

Using **`pregions`** rather than simply scanning all pages (for example, using the physical page database) has the advantage of automatically skipping kernel memory and memory that's already free. However, it has the disadvantage of putting all the memory belonging to a single process together. When the steal hand reached that process's **`pregions`**, all the pages it stole would come from that one process, leaving it frantically paging back in its working set. This would cause noticeable lack of responsiveness in interactive programs. Therefore, only a portion of each **`pregion`** is aged or stolen on each pass, and **`vhand`** thus needs multiple passes through the active **`pregion`** list to visit all of pageable memory.

It's important to keep an appropriate distance between the hands. Too close, and pages are stolen that are in fact in regular use. Too far, and the hands have to move faster to keep the same steal rate; this means that **`vhand`** will consume more processor time. The kernel automatically keeps an appropriate distance between the hands, based on the available paging bandwidth, the number of pages that need to be stolen, the number of pages already scheduled to be freed, and the frequency at which **`vhand`** runs.

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

The **`vhand`** daemon decides when to start paging by determining how much free memory is available. Once free memory drops below an established threshold, paging occurs. **`vhand`** attempts to free enough pages to bring the supply of memory back up to that threshold. The page daemon continues to age pages (that is, clear their reference bits) when woken even if there's enough memory that it doesn't need to steal pages; of course, it won't be woken very often in that situation.

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

How much to age and steal depends on several factors:

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

`vhand` is biased against threads that have nice priorities: the nicer a thread, the more likely **`vhand`** will steal its pages.

There are ways that an application can prevent **`vhand`** from ejecting its pages out of memory onto the swap space: the application can lock the pages in memory via the **`mlock(2)`**or **`plock(2)`** calls, or the application can be designated as a real-time process. These facilities should be used judiciously: the net effect of locking memory is to reduce the amount of pageable memory, increasing contention for that resource and possibly causing memory pressure.

Appendix 4 – Management of swap space

Swap space is an area on a high-speed storage device (almost always a disk drive), reserved for use by the virtual memory subsystem for paging and deactivation of processes. At least one swap device (primary swap) must be present on the system. You can add swap as needed (that is, dynamically) while the system is running, without having to reboot the kernel.

The swapper reserves swap space at process creation time, but does not allocate swap space from the disk until pages need to go out to disk. Reserving swap at process creation protects the swapper from running out of swap space.

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

Pseudo-Swap Space

System memory used for swap space is called pseudo-swap space. It allows users to execute processes in memory without allocating physical swap. Typically, when the system executes a process, swap space is reserved for the entire process, in case it must be paged out. According to this model, to run one gigabyte of processes, the system would have to have one gigabyte of configured swap space. Although this protects the system from running out of swap space, disk space reserved for swap is under-utilized if minimal or no swapping occurs.

To avoid such waste of resources, HP-UX is configured to access up to $^{7}/_{8}$ of system memory capacity as pseudo-swap. This means that system memory serves two functions: as process-execution space and as swap space. By using pseudo-swap space, a 2 GB memory system with 2 GB of swap can run up to 3.75 GB of processes (2GB physical swap space + 1.75GB pseudo-swap). As before, if a process attempts to grow or be created beyond this extended threshold, it will fail.

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

For systems such as factory-floor controllers, which perform best when the entire application is resident in memory, pseudo-swap space can be used to enhance performance: you can either lock the application in memory or ensure that all processes don't use more than $^7/_8$ of system memory.

When the number of processes created approaches capacity, the system might exhibit thrashing and a decrease in system response time.

Physical Swap Space

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

Device Swap Space

Device swap space resides in its own reserved area (an entire disk or logical volume of a disk) and is generally faster than file system swap.

File system Swap Space

File system swap space is located on a mounted file system and can vary in size with the system's swapping activity. However, its throughput is slower than device swap, because free file system blocks may not always be contiguous, leading to extra read/write requests; and because of the extra overhead of an additional layer of code.

Reservation of Physical Swap Space

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

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

The exception to this general rule is called "lazy swap". An entire program may be designated for lazy swap through use of the `chatr(1) +z` option, and so can any private memory mapped object through the use of the `mmap(2) MAP_NORESERVE` flag. Swap space is not reserved in advance for any object designated for lazy swap. If page out for such an object is required and there is no swap space available, then the responsible process is given a `SIGBUS` signal, which usually results in process termination.

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

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

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

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

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

Allocation of Pseudo-Swap Space

Approximately $^7/_8$ of available system memory is available as pseudo-swap space. Pseudo-swap is tracked in the global pseudo-swap reservation counters `swapmem_max` (enabled pseudo-swap) and `swapmem_cnt` (currently available pseudo-swap). If physical swap space is exhausted and no additional file system swap can be acquired, pseudo-swap space is reserved for the process by decrementing `swapmem_cnt`.

For example, on a 4 GB system, `swapmem_max` and `swapmem_cnt` track approximately 3.5 GB of pseudo-swap space, the remainder tracked by the global `sys_mem`, which represents the number of pages reserved for system use only.

Once both device swap and pseudo-swap are exhausted (that is, `swapspc_cnt==0` and `swapmem_cnt==0`), attempts at process creation or growth will fail.

Processes track the number of pseudo-swap pages allocated to them by incrementing a per-region counter `r_swapmem`. Once a process no longer needs its allocated pseudo-swap space, `swapmem_cnt` is incremented by the amount released and `r_swapmem` is updated.

Pseudo-swap consumes memory that could otherwise be used for other purposes, so it is used sparingly. The operating system periodically checks to see if physical swap space has been recently freed. If it has, the system attempts to migrate processes using pseudo-swap only to use the available physical swap. `swapspc_cnt` is decremented by the `r_swapmem` value for each region on the list until either `swapspc_cnt` drops to zero or no other regions utilize pseudo-swap. `swapmem_cnt` is then incremented by the amount of pseudo-swap successfully migrated.

Pseudo-Swap and Kernel Memory

Pseudo-Swap competes with the kernel for the use of system memory. Initially, $^1/_8$ of available memory (`sys_mem` pages) is made unavailable for pseudo-swap use; however, this is usually not enough to handle both kernel dynamic memory and buffer cache space. Instead, the kernel borrows memory from pseudo-swap for these purposes, decrementing `swapmem_cnt` when it borrows a page; once `swapmem_cnt` reaches zero, it starts taking pages from `sys_mem` until that too reaches zero.

When borrowed pseudo-swap is returned, the amount being released is first added to `sys_mem`. Once `sys_mem` grows to its maximum value (`maxmem - swapmem_max`), any additional pages returned are used to increase `swapmem_cnt`.

Pseudo-Swap and Lockable Memory

Because pseudo-swap is related to system memory usage, the swap reservation scheme reflects lockable memory policies. Although the system is not necessarily allocating additional memory when a process locks itself into memory, locked pages are no longer available for general use. This causes `swapmem_cnt` to be decremented to account for the pages. `swapmem_cnt` is also decremented by the size of the entire process if that process gets `plock`ed in memory.

How Swap Space is Prioritized

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

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

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

Appendix 5 – source code for locinfo program

The source code for the locinfo program follows, in the hope that it will prove useful in analyzing the memory distribution on NUMA servers. The code is based on the example that appears in the **`pstat_getlocality(2)`** manpage.

```
/*
 * This program returns system-wide and per-process memory locality information.
 * To compile the 32-bit version, use -D_PSTAT64.
 * The 64-bit version does not need any special compiler flags.
 */

#include <unistd.h>
#include <stdio.h>
#include <sys/param.h>
#include <sys/pstat.h>
#include <sys/errno.h>

#define BURST ((size_t)3)
#define STRSZ 80

unsigned long pgsize;

void pid_locinfo ( pid_t pid );
void sys_locinfo ( void );
void pages_to_str ( uint64_t pages, char *str );

void
usage ( int argc, char **argv )
{
      fprintf ( stderr, "Usage: %s [-p pid]\n", argv[0] );
      fprintf ( stderr, "This program prints out per locality " );
      fprintf ( stderr, "memory usage.\nIf 'pid' is supplied, " );
      fprintf ( stderr, "information on that process is\n" );
      fprintf ( stderr, "returned in addition to system-wide " );
      fprintf ( stderr, "information.\n\n" );
      exit(1);
}

/*
 * Verify arguments, call sys_locinfo(), and call pid_locinfo()
 * if desired.
 */
int
main ( int argc, char **argv )
{
      pid_t pid = (pid_t) 0;

      if ( (argc == 2) || (argc > 3) ||
           ((argc == 3) && (strncmp(argv[1], "-p", 2))) ) {
            usage(argc, argv);
      }

      if ( argc == 3 ) {
            pid = atoi(argv[2]);
            if (pid < 0) {
                /* note that pid 0 is "this process" */
                usage(argc, argv);
            }
      }

      /* Get the size of a page for later calculations */
      pgsize = sysconf ( _SC_PAGE_SIZE );

      sys_locinfo();

      if ( argc == 3 ) {
            pid_locinfo ( pid );
      }

      return 0;
}

/*
 * Display the system-wide memory usage per locality.
 */
void
sys_locinfo ( void )
{
      int i;          /* index within pstl[] */
      int count;      /* the actual number of pstl structures */
      int idx = 0;    /* index within the context of localities */
      struct pst_locality pstl[BURST];
      char total_str[STRSZ], free_str[STRSZ], used_str[STRSZ], ffree_str[STRSZ];
      uint64_t total=0, free=0, floating_free=0;

      printf ( " --- System wide locality info: --- \n" );
      printf ( "%6s%6s%7s%7s%6s%10s%10s%10s%10s\n",
             "index", "ldom", "physid", "type", "spus",
             "total", "free", "float", "used" );
      /* Get a maximum of BURST pst_locality structures */
      count = pstat_getlocality ( pstl, sizeof(struct pst_locality),
                  BURST, idx );

      while ( count > 0 ) {
            for ( i=0 ; i<count ; i++ ) {
                  /* Keep running totals for later */
                  total += pstl[i].psl_total_pages;
                  free  += pstl[i].psl_free_pages;
                  floating_free += pstl[i].psl_floating_free_pages;

                  /* Convert integers into strings */
                  pages_to_str ( pstl[i].psl_total_pages, total_str );
                  pages_to_str ( pstl[i].psl_free_pages, free_str );
                  pages_to_str ( pstl[i].psl_floating_free_pages, ffree_str );
                  pages_to_str ( (pstl[i].psl_total_pages -
                                  pstl[i].psl_free_pages), used_str );

                  printf ( "%6d%6lld%7lld%7s%6u%10s%10s%10s%10s\n",
                           (idx+i),
                           pstl[i].psl_ldom_id,
                           pstl[i].psl_physical_id,
                           ((pstl[i].psl_flags & PSL_INTERLEAVED) ?
                                   "ILM": (pstl[i].psl_cpus ? "C{S}LM":"BLM")),
                           (int)(pstl[i].psl_cpus),
                           total_str, free_str, ffree_str, used_str );
            }
            idx += count;

            /*
             * Get (at most) the next BURST pst_locality
             * structures, starting at idx
             */
            count = pstat_getlocality ( pstl,
                                        sizeof(struct pst_locality),
                                        BURST, idx );
        }

        if ( count < 0 ) {
                perror ( "pstat_getlocality" );
                exit(1);
        }

        if ( idx == 1 ) {
                /* Don't print totals if there's one locality */
                printf ( "\n" );
                return;
        }

        /* Convert integer totals into strings */
        pages_to_str ( total, total_str );
        pages_to_str ( free, free_str );
        pages_to_str ( floating_free, ffree_str );
        pages_to_str ( total-free, used_str );

        /* Print totals */
        printf ( "%6s%6s%7s%7s%6s%10s%10s%10s%10s\n",
                 "", "", "", "", "", "-----", "-----", "-----", "-----" );
        printf ( "%6s%6s%7s%7s%6s%10s%10s%10s%10s\n\n",
                 "", "", "", "", "", total_str, free_str, ffree_str, used_str );
}

/*
 * Given a pid, display its per-locality physical memory usage.
 */
void
pid_locinfo ( pid_t pid )
{
      int count, i=0;
      struct pst_proc_locality ppl;
      char total_str[STRSZ], shared_str[STRSZ];
      char private_str[STRSZ], weighted_str[STRSZ];
      uint64_t total=0, shared=0, private=0, weighted=0;

      /*
       * With this interface, information on only one locality
       * can be returned at a time.  This will get the first:
       */
      count = pstat_getproclocality ( &ppl,
                    sizeof(struct pst_proc_locality), pid, i );

      printf ( " --- Per-process locality info for pid %d: ---\n",
               pid );
      printf ( "%6s%10s%10s%10s%10s\n",
               "idx", "total", "shared", "private", "weighted" );

      while ( count == 1 ) {

              total    += ppl.ppl_rss_total;
              shared   += ppl.ppl_rss_shared;
              private  += ppl.ppl_rss_private;
              weighted += ppl.ppl_rss_weighted;

              pages_to_str ( ppl.ppl_rss_total, total_str );
              pages_to_str ( ppl.ppl_rss_shared, shared_str );
              pages_to_str ( ppl.ppl_rss_private, private_str );
              pages_to_str ( ppl.ppl_rss_weighted, weighted_str );
 printf ( "%6d%10s%10s%10s%10s\n",
                       i, total_str, shared_str,
                       private_str, weighted_str );
              i++;
              count = pstat_getproclocality ( &ppl,
                              sizeof(struct pst_proc_locality),
                              pid, i );
      }

      if ( count < 0 ) {
              if ( errno == ESRCH ) {
                      fprintf ( stderr, "Process %d not found\n", pid );
                      exit(1);
              }
              perror ( "pstat_getproclocality" );
              exit(1);
      }

      if ( i == 1 ) {
              /* Don't print totals if there's one locality */
              printf ( "\n" );
              return;
      }

      pages_to_str ( total, total_str );
      pages_to_str ( shared, shared_str );
      pages_to_str ( private, private_str );
      pages_to_str ( weighted, weighted_str );

      printf ( "%6s%10s%10s%10s%10s\n",
               "", "-----", "-----", "-----", "-----" );
      printf ( "%6s%10s%10s%10s%10s\n\n",
               "", total_str, shared_str,
               private_str, weighted_str );
}

/*
 * Given a quantity of memory in pages, fill str with a
 * human-readable string representing that amount.
 */
void
pages_to_str ( uint64_t pages, char *str )
{
      uint64_t kpg = pages*(pgsize/1024L);
      uint64_t mpg = kpg/1024L;
      uint64_t gpg = mpg/1024L;

      if ( gpg > 0 ) {
              sprintf ( str, "%.2lfG", (((double)kpg/1024.0)/1024.0));
      } else if ( mpg > 0 ) {
              sprintf ( str, "%.2lfM", ((double)kpg/1024.0));
      } else if ( kpg > 1 ) {
              sprintf ( str, "%lluK", kpg );
      } else {
              sprintf ( str, "%llu", pages );
      }
}
```

5900-0908, August 2010