

UNIX MEMORY USAGE, INSTRUMENTATION AND ANALYSIS

PAPER 2173 – CMG98

Unix has traditionally had good monitoring capabilities for CPU, I/O and Networks, but there is comparably little information about memory usage. To address this a new methodology for measuring memory usage in Unix is required. This paper introduces a new methodology to better understand how and where memory is being allocated and used by the operating system and applications, which leads the way for proper capacity planning.

1. INTRODUCTION

Unlike MVS, Unix typically provides a very primitive set of statistics for memory usage.

For example, our site administrator Scott, has 450 users running Oracle financials on a system with 2GB of memory. Scott wants to know if he has enough memory in the system to run another 50 users. Scott logs into the system and looks at the memory statistics; according to the system there is 25MB of free memory, and according to the process listing each user is using 87MB of memory, totaling 3.5GB. At this point, Scott is astonished that his system runs at all, and is convinced he does not have enough memory.

In reality, Scott's system has plenty of memory. The system reporting mechanisms didn't take into account that each user shares the Oracle shared memory segments, nor that the system is using the available free memory as a disk cache.

Scott's problem is not unique, in fact it's the problem that most Unix administrators face every time they attempt to look at memory utilization and sizing on their servers.

2. EXISTING STATISTICS

The statistics provided are often based on the *vmstat* command output, which provide a set of paging counters based on

the original BSD Unix memory implementation.

The BSD Unix memory implementation used in most Unix implementations today is a demand paged model that attempts to manage memory by freeing least recently used pages when there is a memory shortage.

The *vmstat* and *sar* output on System V Release 4 Unix provides the following paging statistics:

- Pages Paged Out Per Second
- Pages Paged In Per Second
- Average Free Memory
- Pages Scanned Per Second
- Pages Freed Per Second
- Pages Reclaimed Per Second

Without intimate knowledge of how the memory subsystem works, it is very hard for the average user to determine if their system has ample memory, or is desperately short. Further more, capacity planning based on the statistics provided is next to impossible, even by the most experienced Unix personnel.

In most modern implementations of Unix, the memory system is also used for caching file systems, which renders these statistics even less meaningful. For example, consider the following:

- The average free memory available is often almost zero, because any free memory will

eventually be used by the file system cache, given enough file activity.

- Regular file I/O generates activity in the memory system, which is reported as page ins and page outs.

With memory statistics that are clouded by regular file I/O, how is a Unix administrator supposed to know how much memory is free, or if the system is paging excessively?

2.1 APPLICATION AND PROCESS MEMORY USAGE

The Unix *ps* command provides information about each process (job) running on the system. Many statistics are provided, and vary accordingly with each different Unix implementation.

Common to most implementations is the memory usage indicator, represented as the resident set size (RSS). The RSS figure is a count of the amount of memory in each process that is resident at the time that the *ps* command is executed.

Unfortunately, since the introduction of shared memory and shared libraries, this RSS figure cannot be used to accurately determine the memory usage of a given application. This is because large portions of an application's executable code and some portions of the application's main memory is shared between other applications, and the proportion of this sharing is unknown.

3. TOWARDS A BETTER MODEL

To provide a more accurate model for accounting for Unix memory usage, new statistics and a new methodology are required.

At a minimum, the operating system should provide information about global memory utilization and more accurate information about per process memory usage that is capable of dealing with the

implications of shared libraries and shared memory.

To implement the new memory instrumentation, we constructed a prototype using loadable kernel modules under Solaris.

This loadable module provides the necessary code to lock and gather statistics from the memory subsystem, and provide new information about memory utilization. Our prototype included statistics for global memory utilization, per process memory utilization, and file system cache information.

3.1 GLOBAL MEMORY STATISTICS

It is important to understand the high-level breakdown of where memory is being used in the system. To facilitate this, a memory summary utility was developed. This utility provides a system wide figure in Megabytes for the different memory allocations:

Let's revert back to Scott's Oracle example. The output from the *prtmem* command shows where the memory is allocated:

```
# prtmem
Total Memory          2048MB
Kernel Memory:       197MB
Application Memory:  1380MB
Buffer cache Memory: 446MB
Free Memory:         25MB
```

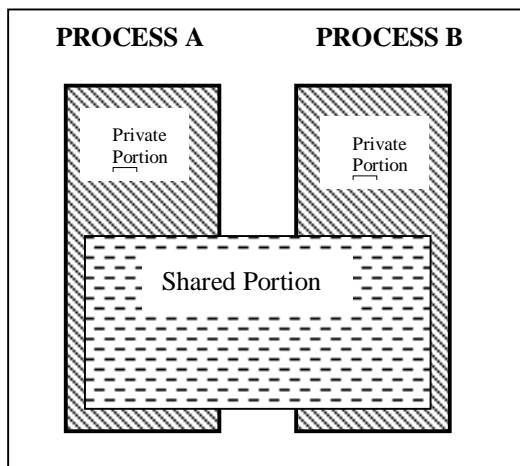
At a glance, we can quickly see that the 450 users are only using 1.3GB of memory. The operating system kernel is using 197MB, and the free memory that is not directly being used by the application is being used by the 446MB file system cache. The operating system always tries to keep a small pool of real free memory, in this case 25MB. It does look like Scott's additional 50 users would quite easily fit on the system without causing an application memory shortage.

3.2 EXAMINING APPLICATION USAGE

By using the system wide memory statistics, we were able to determine that Scott's 450 users of Oracle financials were using approximately 1.3GB of memory, and hence estimate that each additional user would use somewhere in the realm of 2.9MB of memory. This is only an approximation, because not all users and processes on the system will be using the same amount of memory. For example, our 1.3GB includes 450 users, plus the Oracle financials batch processes, which skew our results.

A more accurate way of examining application memory is to look at the application's individual processes. By looking at each process, we can determine how much memory that process is using, and then calculate how much memory additional processes will consume. This is made possible by taking the shared portion of the process's memory into account.

We can assume that each process has a resident memory size, part of which is shared with other similar processes and part which is private to each process.



The types of memory we can account for in each process are the virtual size (SZ), the resident set size (RSS), the shared portion (Shared), and the private portion (Private). They are defined as follows:

SZ – the virtual size of the process's memory address space. Not all of this address space is resident, and hence not all of this space occupies real memory. The virtual size is always greater or equal to the resident set size.

RSS – the amount of real memory allocated to the process. Note that part of this can be shared with other processes and accounted for more than once.

Shared – the amount of the resident size that is shared with at least one other process. This portion should only be accounted for once.

Private – the amount of the resident size that is totally private to this process, and should be accounted for each process.

Using the *memp*s command from the prototype, we can see the different memory types for a group of processes:

```
# memps
PID    Size  RSS  Shared  Priv  Process
8004   1792k 1528k 1360k   168k /bin/ksh
 466   1816k 1528k 1360k   168k /bin/ksh
 603   1784k 1528k 1360k   168k /bin/ksh
 624   1784k 1528k 1360k   168k /bin/ksh
7971   1792k 1528k 1360k   168k /bin/ksh
 824   1792k 1528k 1360k   168k /bin/ksh
 612   1792k 1528k 1360k   168k /bin/ksh
```

It can be seen from the *memp*s output that each shell invocation has 1.8MB of virtual memory, with 1.5MB of real memory allocated. Of this, 1.3MB is shared between the other invocations of the shell, and only 168k is private to each. This means that each new invocation of the shell only uses 168k of additional memory.

The memory used by the entire group of processes can be calculated as $1.5\text{MB} + 7 \times 168\text{k} = 2.7\text{MB}$.

3.3 DRILLING DOWN – LOOKING AT EACH PROCESS

Further information can be discovered about each process. In a Unix process, the memory allocation is typically divided into separate components known as segments.

Each segment of memory has its own characteristics, and may be private memory, shared memory, or a combination of both.

The address space of a typical Unix process is seen in the following example using Unix shell.

Address	Kbytes	Resident	Shared	Private	Permissions	Mapped File
00010000	184	184	176	8	read/exec	ksh
0004C000	8	8	-	8	read/write/exec	ksh
0004E000	32	32	-	32	read/write/exec	[heap]
EF5E0000	16	16	16	-	read/exec	en_US.so.1
EF5F2000	8	8	8	-	read/write/exec	en_US.so.1
EF600000	592	544	512	32	read/exec	libc.so.1
EF6A2000	24	24	8	16	read/write/exec	libc.so.1
EF6A8000	8	8	-	8	read/write/exec	[anon]
EF6C0000	16	16	16	-	read/exec	libc_psr.so.1
EF6D0000	16	16	16	-	read/exec	libmp.so.2
EF6E2000	8	8	8	-	read/write/exec	libmp.so.2
EF6F0000	8	8	-	8	read/write/exec	[anon]
EF700000	448	424	424	-	read/exec	libnsl.so.1
EF77E000	32	32	8	24	read/write/exec	libnsl.so.1
EF786000	24	8	-	8	read/write/exec	[anon]
EF790000	32	32	32	-	read/exec	libsocket.so.1
EF7A6000	8	8	8	-	read/write/exec	libsocket.so.1
EF7A8000	8	-	-	-	read/write/exec	[anon]
EF7B0000	8	8	8	-	read/exec/shared	libdl.so.1
EF7C0000	112	112	112	-	read/exec	ld.so.1
EF7EA000	16	16	8	8	read/write/exec	ld.so.1
EFFFC000	16	16	-	16	read/write/exec	[stack]
-----	-----	-----	-----	-----		
total Kb	1624	1528	1360	168		

The example shows the many memory segments apparent in the shell process. The first two segments are the shell executable binary, and hence are largely shared with the other shell processes. The heap space is the shell's application memory, which is private to each shell invocation. Following the heap are the multiple shared libraries that are mapped into the shell's address space. Of course, most of these are shared with other processes. Finally, there is the stack, which is memory private to each invocation.

4.0 THE FILE SYSTEM CACHE

In our original example, there was 446MB of memory being used as a file system cache. A breakdown of this file system cache would help us understand what files and database tables are using this cache, and help us make decisions about if more file system cache memory would be of benefit.

A function of the prototype loadable kernel module allows us to display the contents of the file system cache on a file by file basis.

```
# memps -m
```

Size	Filename
84504k	/tables/data01.dbf
76408k	/tables/data03.dbf
50176k	/tables/data02.dbf
6104k	/tables/data05.dbf
5808k	/tables/data04.dbf
5672k	/index/index01.dbf
4144k	/index/index02.dbf
4136k	/temp/temp.dbf
4048k	/rbs/rbs01.dbf
3136k	/rbs/rbs02.dbf
2528k	/redo/redo01.dbf
2104k	/redo/redo02.dbf

We can see that in this example, the database tables mainly occupy the file system cache.

5.0 SUMMARY

Using this methodology, it is possible to develop an accurate picture of memory usage for both a system and specific

applications. Gaining an accurate understanding of an applications memory usage allows a concise capacity planning model to be developed, and hence accurate prediction of future memory needs.