Sizing Memory for Oracle on Solaris

George Schlossnagle
george@omniti.com

0.1 Introduction

Anyone who manages or owns an Oracle database needs to know what the performance limitations on the system are. Especially in this era of explosively growing web sites and rapidly changing business priorities, being able to accurately plan for capacity is a critical skill. Capacity planning is often approached by measuring key performance metrics on the host system over time, correlating them with business metrics, and extrapolating them over time.

This is an excellent approach for planning for CPU and storage upgrades, but the nature of the Solaris Virtual Memory architecture obscures critical information on memory pressure, making it difficult to plan for using traditional techniques. Further, by using application specific knowledge about our Oracle instance, we can predict memory-related performance envelopes with a great degree of accuracy. Oracle installs driving web based applications often require a large number of simultaneous users running queries. Each additional user session consumes memory, a limited and performance-critical resource.

In this document we focus on using Sun and Oracle tools to gauge either the optimal configuration for a statically sized install, or the growth capacity for an install based on a current physical configuration.

0.2 Design Goals

Our first task is to decide what our design goals are. 'Run fast' is a good goal, but is rather unspecific. Since we are talking about memory here, we will set out the goal of eliminating memory pressure. Specifically, we should retain sufficient memory that we avoid unnecessary paging and avoid swapping at all costs. A swapping database system will grind to a halt.

0.3 Oracle Memory Usage

Before we start looking at some real systems, let's look at how and where Oracle uses memory. In Oracle terminology, memory is broken up into the SGA (Shared Global Area), UGA (User Global Area) and PGA (Process Global Area). For the purposes of our discussion, we will take a more systems-oriented view and break about memory usage into shared memory usage and process private memory usage.

- SMON
- PMON
- LGWR
- DBWR
- ARC

  All of these possess very small private areas and are largely irrelevant in analyzing memory usage.

- Shadow Processes

Component	Shared Size	Private Size
Libraries and Binaries	system dependent 32M	very small
Shared Pool	shared_pool_size	0
Reserved Pool	shared_pool_reserved_size	0
Large Pool	large_pool_size	0
Java Pool	java_pool_size	0
Oracle Buffer Cache	db_block_buffers * db_block_size	0
Sort Area	0	bounded by sort_area_size
Hash Join Area	0	bounded by hash_area_size

  An efficient way of getting a good estimate for the actual private heap areas used by sessions (in bytes) is

```
SELECT name, avg(value)
FROM v$session se, v$sesstat ss, v$statname sn
WHERE ss.sid=se.sid
```

```
AND sn.statistic# = ss.statistic#
AND sn.name = 'session pga memory'
GROUP BY name;
```

Doing more advanced statistical analysis on this field may be valuable if you have a fixed set of jobs which perform large sorts or joins. These numbers should be sanity checked using the pmem tool that ships with the RMCMem package for Solaris to measure the heap memory size of your Shadow processes.

It should be noted that a new Oracle shadow process does not have any memory allocated for sort or join operations. As memory is used for this functionality, the heap is grown and lessened, but memory is not usually released to the OS. That having been said, currently unutilized sort space will be paged out under normal memory pressure. This is why we use the 'session pga memory' for estimating active usage.

- Sizing the shared pool

  Sizing your shared pool is a very complicated issue, and we won't even begin to discuss it here. Steve Adams' website http://www.ixora.com.au/ has a number of very good scripts to help arriva at an efficient shared pool size. For the purpose of this article, we will assume we've done this already, and that the full size of the shared pool, db_block_buffers, etc. has all been pre-determined optimally.

- Non-Oracle memory usage

  Use RMC tools also provide good insight into memory usage not directly allocated by Oracle. We start by using the `prtmem` command to get a broad overview of what is going on.

```
15:53:31(root@mysystem)[~]> /opt/RMCmem/bin/prtmem

Total memory:            7969 Megabytes
Kernel Memory:            418 Megabytes
Application:             5687 Megabytes
Executable & libs:         58 Megabytes
File Cache:              1458 Megabytes
Free, file cache:         239 Megabytes
Free, free:               126 Megabytes
```

- Kernel Memory

Kernel memory is allocated at boot time to hold the initial kernel code and grows dynamically at runtime as drivers and kernel modules are loaded, and as dynamically allocated kernel structures (like the process table) expand.

- Application Memory

  This is our process memory, much of it Oracle. The RMC package contains good tools (notably `pmem`) that allow good visibility into the memory usage by individual processes. We've already used it a bit to determine the memory usage for our Oracle processes above. The Memtool documentation is a great source for more information.

- FileSystem Buffer Cache

  If direct IO is not performed on your datafiles, their filesystem will do caching to buffer writes to disk and to improve future reads. For large filesystems, a buffer cache can exert significant memory pressure. The filesystem buffer cache does not consume a fixed amount of memory, but is managed by the virtual memory system and utilizes the same memory as processes. Even on tight-memory systems severe pressure can be exerted. Using `memps -m` we can get good detail on what is in the buffer cache:

```
16:20:29(root@mysystem)[~]> /opt/RMCmem/bin/memps -m
   Size   InUse E/F Filename
 91496k      0k F   /oracle (inode  1045771)
 67736k      0k F   /oracle (inode  1045788)
 66976k  11712k F   /database/redo/log1_g2.dbf
 65560k      0k F   /database (inode     1204)
 46616k      0k F   /database (inode     1206)
 43376k   5232k F   /database (inode     1205)
 40232k  40232k F   /database (inode     3242)
 37216k      0k F   log1_g2.dbf
 30176k  30176k F   /database (inode     3237)
....
```

  We can see active redo logs being written to, as well as datafiles (identified by file system and inode, e.g. /database (inode 3237). Use find *mount* -inum *inode* to identify the file.)

  Note that if you use raw datafiles, Veritas QuickIO files, or if the filesystem can be mounted in 'direct' mode, datafile accesses will bypass the buffer cache.

- Free Memory

  As we stated in our design goals, running out of memory is very bad. Thus, we desire to keep some wiggle room.

0.4 An example

Let's try and figure out the maximum number of processes we can support on a given host. First, to see what we've got to work with.

```
17:58:48(root@mysystem)[~]> /opt/RMCmem/bin/prtmem

Total memory:            7966 Megabytes
Kernel Memory:            881 Megabytes
Application:             5786 Megabytes
Executable & libs:         41 Megabytes
File Cache:              1165 Megabytes
Free, file cache:         124 Megabytes
Free, free:                 0 Megabytes
```

So best case we have 7966 - 881 = 7085M of RAM available to play with. Now let's pull our SGA parameters:

```
  1  SELECT sum(value)
  2  FROM v$parameter
  3  WHERE name IN ('shared_pool_size',
  4             'large_pool_size',
  5             'shared_pool_reserved_size',
  6*            'java_pool_size')
SQL> /

SUM(VALUE)
----------
 202000000

  1  SELECT value
  2  FROM v$parameter
  3* WHERE name IN ('db_block_buffers', 'db_block_size')
SQL> /

VALUE
```

```
-------------------------------------------------------------------------------
190000
8192
```

So the total allocation for shared memory, should be 1758M. This can be confirmed by using `ipcs` or `pmem`. If we reserve 128M for free memory, we now have 7085 - 1758 - 128 = 5199M. The average PGA size for a process is:

```
SELECT name, (avg(value)/1000000) Megs
FROM v$session se, v$sesstat ss, v$statname sn
WHERE ss.sid=se.sid
   AND sn.statistic# = ss.statistic#
   AND sn.name = 'session pga memory'
GROUP BY name
SQL> /

NAME                                                                   MEGS
---------------------------------------------------------------- ----------
session pga memory                                               3.03488271
```

This basic analysis suggests a max_processes value of 5199/3.03 = 1715. Anything past that will violate our memory restrictions.

Before we conclude, let's investigate filesystem buffer cache usage. This database uses QuickIO, so 1165M of fs buffer cache is large.

```
19:33:26(root@mysystem[~]> /opt/RMCmem/bin/memps -m
   Size   InUse E/F Filename
1763992k      0k F
 73480k  73480k F   /database/oraarch (inode      107)
 65704k  65704k F   /database/oraarch (inode      106)
 51304k  13952k F   /database/oraarch (inode      105)
 37848k      0k F   /database/oraarch (inode      104)
 25336k  25336k F   /var/adm/log (inode      12)
 16384k  16264k E   /oracle/product/8.1.6/bin/oracle
....
```

The first entry is not a file at all, it's Oracle's shared memory allocation. Let's look at the others. Inodes 104-107 in /database/oraarch are archive redo logs. These files are written in one big sequential write, and the are read back later in one sequential read when they are backed up to tape. This means that buffering reads or writes to this file are effectively useless and just create artificial memory pressure. We can address this by remounting the filesystem with unbuffered IO. Remounting and rechecking `memps`

```
19:49:49(root@mysystem)[~]> /opt/RMCmem/bin/memps -m
   Size   InUse E/F Filename
1763992k      0k F
144776k      0k F
 25664k  25664k F   /var/adm/log (inode       12)
 16384k  16280k E   /oracle/product/8.1.6/bin/oracle
  3360k   2872k E   /oracle/product/8.1.6/lib/libclntsh.so.8.0
...
```

Much better!