

The Solaris Memory System

Sizing, Tools and Architecture



THE NETWORK IS THE COMPUTER™

Sun Microsystems Computer Company

A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043 USA
415 960-1300 fax 415 969-9131

Part No.: 8xx-xxxx-xx
Revision C, October 1997

Copyright 1997 Sun Microsystems, Inc. 2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A. All rights reserved.

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any. Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Parts of the product may be derived from Berkeley BSD systems, licensed from the University of California. UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

Sun, Sun Microsystems, the Sun logo, SunSoft, SunDocs, SunExpress, and Solaris [\[ATtribution of All Other Sun Trademarks Mentioned Significantly Throughout Product or Documentation. Do Not Leave This Text in Your Document!\]](#) are trademarks, registered trademarks, or service marks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc. [\[Third-Party Trademarks That Require Attribution Appear in 'TMARK.' If You Believe a Third-Party Mark Not Appearing in 'TMARK' Should Be Attributed, Consult Your Editor or the Sun Trademark Group for Guidance.\]](#)

The OPEN LOOK and Sun™ Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

RESTRICTED RIGHTS: Use, duplication, or disclosure by the U.S. Government is subject to restrictions of FAR 52.227-14(g)(2)(6/87) and FAR 52.227-19(6/87), or DFAR 252.227-7015(b)(6/95) and DFAR 227.7202-3(a).

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright 1997 Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, California 94043-1100 Etats-Unis. Tous droits réservés.

Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y en a. Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Des parties de ce produit pourront être dérivées des systèmes Berkeley BSD licenciés par l'Université de Californie. UNIX est une marque déposée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company, Ltd.

Sun, Sun Microsystems, le logo Sun, SunSoft, SunDocs, SunExpress, et Solaris [\[ATtribution of All Other Sun Trademarks Mentioned Significantly Throughout Product or Documentation. Do Not Leave This Text in Your Document!\]](#) sont des marques de fabrique ou des marques déposées, ou marques de service, de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays. Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc. [\[Third-Party Trademarks That Require Attribution Appear in 'TMARK.' If You Believe a Third-Party Mark Not Appearing in 'TMARK' Should Be Attributed, Consult Your Editor or the Sun Trademark Group for Guidance.\]](#)

L'interface d'utilisation graphique OPEN LOOK et Sun™ a été développée par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox pour la recherche et le développement du concept des interfaces d'utilisation visuelle ou graphique pour l'industrie de l'informatique. Sun détient une licence non exclusive de Xerox sur l'interface d'utilisation graphique Xerox, cette licence couvrant également les licenciés de Sun qui mettent en place l'interface d'utilisation graphique OPEN LOOK et qui en outre se conforment aux licences écrites de Sun.

CETTE PUBLICATION EST FOURNIE "EN L'ETAT" ET AUCUNE GARANTIE, EXPRESSE OU IMPLICITE, N'EST ACCORDEE, Y COMPRIS DES GARANTIES CONCERNANT LA VALEUR MARCHANDE, L'APTITUDE DE LA PUBLICATION A REpondre A UNE UTILISATION PARTICULIERE, OU LE FAIT QU'ELLE NE SOIT PAS CONTREFAISANTE DE PRODUIT DE TIERS. CE DENI DE GARANTIE NE S'APPLIQUERAIT PAS, DANS LA MESURE OU IL SERAIT TENU JURIDIQUEMENT NUL ET NON AVENU.



Please
Recycle



Adobe PostScript

Contents

1. Solaris Memory Quickstart	15
Unix Memory Sizing	15
How is my memory being used?	16
Total Physical Memory	16
File Buffering Memory	16
Kernel Memory	17
Free Memory	17
Do I have a memory shortage?	18
How much memory is my Application using?	19
2. Sizing and Capacity Planning	21
Sizing a desktop system	21
The performance trade-off for desktops	22
System Daemon Memory Requirements	22
Memory used by system libraries	24
Operating System (Kernel) Memory	26

CDE Memory Requirements	26
Total desktop memory requirements	28
Sizing a server application	30
Process Memory Requirements	30
System Processes	30
Background Processes	31
Per-User processes	32
Memory used by system libraries	34
Buffer Cache Memory	36
System V Shared Memory	36
Operating System (Kernel) Memory	37
Summary - Sizing our DB2 example	38
3. Memory Analysis & Tools	39
The vmstat and	
swap commands	40
Free Memory	41
Swap Space Utilization	41
Paging Counters	43
Systems with little or no filesystem I/O	43
Systems with a lot of filesystem I/O	44
Kernel Memory	45
Using Sar to look at Kernel Memory Allocation	45
Using crash for detailed kernel memory allocation	46
Kernel Allocation Methods	46

The kmastat Command	46
Using ipcs to display shared memory	48
MemTool - Unbundled Memory Tools	49
Tools provided with MemTool	49
Basis of operation.	49
Process memory usage and the pmem command . . .	49
Buffer cache memory	51
Using the MemTool GUI	54
Buffer Cache Memory	54
Process Memory	55
Process Matrix	57
GUI Options	58
The Workspace Monitor Utility - WSM.	60
Finding Memory Leaks with DBX	61
The Run-time Leak Checker	61
Compiling the program	61
Running the Leak Test.	62
4. Solaris Memory Architecture	63
Why Have A Virtual Memory System?	63
Demand Paging	64
Combined I/O and Memory Managment	65
Design Goals of the Solaris Virtual Memory	65
PAGES - The basic unit of Solaris memory	66
Hardware Memory Management Units	66

VNODE's - The Virtual File Abstraction	67
The HAT Layer	69
Pages as Vnode and Offset	70
Virtual Address Spaces	72
Memory Segments	72
Segment Protection	74
Process Address Spaces as Mapped Segments	75
The Pageout Process	77
Basis of Operation	78
Pageout Scanner	78
Pageout Scanner Parameters	79
The Memory Scheduler	82
Soft Swapping	83
Hard Swapping	83
5. I/O via the VM System	85
Filesystem I/O	88
Directory Lookups	89
UFS Inode Cache	90
The Old Buffer Cache	90
Free Behind and Read Ahead	90
The fsflush process	92
Direct I/O	93
RAW Devices	93
Asynchronous I/O	94

Figures

Figure 3-2	vmstat output.	40
Figure 3-1	Stages of Swap space reservation	42
Figure 3-2	MemTool GUI - Buffer Cache Memory	55
Figure 3-7	MemTool GUI - Process Memory	56
Figure 3-9	MemTool GUI - Process/File Matrix	58
Figure 3-3	MemTool GUI - Sort Options.	59
Figure 4-2	VNODE Interface.	68
Figure 4-4	VM Layers.	70
Figure 4-5	PAGE Level Interface	71
Figure 4-7	Segment Interface	73
Figure 4-9	Solaris 2.6 Virtual Address Space for a Process	75
Figure 4-1	Pageout scanner	79
Figure 4-2	Pageout Scanner Parameters	80
Figure 5-1	Solaris I/O Framework.	87

Tables

Table 2-1	System Process Memory Requirements	23
Table 2-2	Kernel memory required for desktop systems	26
Table 2-3	CDE Desktop memory requirements	27
Table 2-4	CDE Desktop Memory Requirements	28
Table 2-5	CDE Desktop with Netscape Memory Requirements	28
Table 2-6	System Process Memory Requirements	30
Table 2-7	Kenel Memory Required by Tuneables	37
Table 2-8	Memory Sizing Rules and Example	38
Table 3-1	Memory related tools	39
Table 3-3	Detailed description of vmstat Paging Counters	44
Table 3-4	Sar command fields	45
Table 3-5	MemTool Utilities	49
Table 3-6	Buffer Cache Fields	54
Table 3-8	Process Memory Fields	57
Table 4-1	Sun MMU Page Sizes	66
Table 4-8	Solaris 2.6 Segment Drivers	74

Table 4-10	Pageout Parameters.....	81
Table 4-11	Memory Scheduler Parameters.....	84
Table 5-1	I/O Memory Buffers and Caches.....	85
Table 5-2	Filesystems in Solaris.....	88
Table 5-3	Parameters that affect fsflush.....	92

Preface

The question “do I have enough memory in my system” or “how much memory do I need to run my application” often arises from customers and Systems Engineers. Questions like these are typically followed by a short period of silence, mostly due to a lack of information about how memory is used in Solaris.

This paper is aimed at providing the necessary information to answer these type of questions.

The first chapter is a summary of the most commonly asked questions about memory utilization in Solaris, and serves as a quick introduction to the tools and techniques that can be used.

The rest of the paper covers memory topics in more detail. The first two chapters “Sizing and Capacity Planning” and “Memory Analysis & Tools” provide a step by step process for measuring memory utilization and sizing and applications memory requirements. The last two chapters “Solaris Memory Architecture” and “I/O via the VM System” provide a detailed technical description of the architecture of the Solaris memory system.

Please send comments and suggestions to Richard.McDougall@Eng.Sun.COM

Who Should Read This White Paper

This paper is written for customers and partners of Sun, including Solaris System Administrators, Vendors and Developers.

It is not intended to be a all-encompassing document on the architecture of Solaris, rather a means to understand how to measure, predict and influence the behavior of the memory system.

Related Material

Books

- *Sun Performance & Tuning* - Adrian Cockcroft, 1995
- *Configuration and Capacity Planning* - Brian Wong, 1997
- *The Magic Garden* - Goodheart & Cox, 1993

Papers

- *Solaris Virtual Memory Implementation* - Rob Gingel, 1987
- *The Bunyip Memory Tools Documentation* - 1997

How This Paper Is Organized

Chapter 1, “Solaris Memory Quickstart” is an introduction to Solaris memory behaviour, measurement and sizing.

Chapter 2, “Sizing and Capacity Planning” presents a methodology for sizing applications and predicting the memory requirements of a system.

Chapter 3, “Memory Analysis & Tools” explains the various tools that are available to measure memory behavior in Solaris. Both Solaris commands and unbundled tools are covered.

Chapter 4, “Solaris Memory Architecture” is a detailed technical description of the Solaris Virtual Memory system.

Chapter 5, “I/O via the Virtual Memory System” details how I/O is performed in Solaris, and how it interacts with the Virtual Memory system.

What Typographic Changes Mean

The following table describes the typographic changes used in this book.

TABLE P-1 Typographic Conventions

Typeface or Symbol	Meaning	Example
AaBbCc123	The names of commands, files, and directories; on-screen computer output	Edit your <code>.login</code> file. Use <code>ls -a</code> to list all files. <code>machine_name% You have mail.</code>
AaBbCc123	What you type, contrasted with on-screen computer output	<div style="border: 1px solid black; padding: 5px;"><code>machine_name% su</code> <code>Password:</code></div>
<i>AaBbCc123</i>	Command-line placeholder: replace with a real name or value	To delete a file, type <code>rm filename</code> .
<i>AaBbCc123</i>	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in <i>User's Guide</i> . These are called <i>class</i> options. You <i>must</i> be root to do this.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P-2 Shell Prompts

Shell	Prompt
C shell prompt	<code>machine_name%</code>
C shell superuser prompt	<code>machine_name#</code>
Bourne shell and Korn shell prompt	<code>\$</code>
Bourne shell and Korn shell superuser prompt	<code>#</code>

This chapter is a quick overview some of the most frequently asked questions about how applications use memory in a Solaris system.

Unix Memory Sizing

Accurate memory sizing and measurement tools are rarely found on Unix platforms, which often leads to confusion about the real memory requirements of an application.

Often, comparisons are made between applications running on Solaris and other Unix implementations, and the application appears to use significantly more memory on Solaris.

What is really happening is that Solaris provides a wider range of operating system features, and hence has significantly larger shared system libraries. Without the ability to distinguish between the shared library and application memory, the Solaris application appears to use more memory.

In reality the actual memory usage of each application is very similar to other platforms.

Solaris now has the ability to look at the shared and non-shared portions of memory, which allows more accurate sizing of applications, without guessing at the shared component. This functionality was introduced at Solaris 2.6 with the `pmap` command as discussed later in this chapter.

How is my memory being used?

The first thing to observe in a system is where the memory has been allocated. In a broad perspective, we are interested in knowing-

- The total amount of physical memory available
- How much memory is being used for file buffering
- How much memory is being used for the kernel
- How much memory my applications are using
- How much memory is free

To answer all of the above, we need to use MemTool. MemTool is discussed in detail in Chapter 2. The latest version of MemTool can be obtained by sending a request to memtool-request@chessie.eng.sun.com. These tools are provided free, but are not covered by normal Sun support.

MemTool is provided in pkgadd format. Simply log in as root, untar the package and use the pkgadd command to install.

The tools are installed into the `/opt/RMCmem/bin` directory.

The MemTool version at time of writing was 3.5.

Total Physical Memory

The amount of total physical memory can be ascertained by looking at the output of the Solaris `prtconf` command.

```
# prtconf
System Configuration: Sun Microsystems sun4u
Memory size: 384 Megabytes
System Peripherals (Software Nodes):
```

File Buffering Memory

The buffer cache uses available free memory to buffer files on the filesystem. On most systems, the amount of free memory is almost zero as a direct result of this.

To look at the amount of file buffer cache, you will need to use the MemTool package. The MemTool `memp`s command can be used to dump the contents of the buffer cache.

A summary of the buffer cache memory is available with the MemTool `prtmem` command.

```
# prtmem
Total Physical Memory: 384 Megabytes
Buffer Cache Memory: 112 Megabytes
Kernel Memory: 63 Megabytes
Free Memory: 17 Megabytes
```

Kernel Memory

The amount of kernel memory can be found by using the Solaris `sar` command, and summing all of the `alloc` columns. The output is in bytes.

```
# sar -k 1 1
SunOS williams 5.5 Generic sun4m 08/22/97
12:12:49 sml_mem alloc fail lg_mem alloc fail ovsz_alloc fail
12:12:52 3158016 18032129 0 39298671 18023923 3 27898314 0
```

Free Memory

Free memory is almost always zero, because the buffer cache grows to consume free memory. Free memory can be measured with the `vmstat` command.

The first line of output from `vmstat` is an average since boot, so the real free memory figure is available on the 2nd line. The output is in kilobytes.

```
# vmstat 3
procs          memory
r  b  w  swap  free  re  mf  pi  po  fr  de  sr  f0  s2  s3  s4  in  sy  cs  us  sy  id
0  0  0  81832 74792  0  12 75  4 93  0 36  0  1  1  1 265 1940 303 5  1 93
0  0  0 560248 12920  0  0  0  0  0  0  0  0  0  0  0 217  872 296 0  0 100
0  0  0 560248 12920  0  0  0  0  0  0  0  0  0  0 205  870 296 0  0 99
```

Do I have a memory shortage ?

A critical component of performance analysis is ascertaining where the bottlenecks are. Detecting memory bottlenecks is not quite as straight forward as measuring processor and disk, and requires a few more steps to arrive at a conclusion.

To determine if there is a memory shortage, we need to-

- Determine if the applications are paging excessively because of a memory shortage
- Determine if the system could benefit by making more memory available for file buffering

The Solaris memory system counts paging activity generated by both file I/O and application paging with the same counters. This means that the paging activity we can observe with the Solaris `vmstat` is not a fail-safe method of identifying memory shortages.

We can however use `vmstat` to rule out any question of a memory shortage in some circumstances.

The steps I recommend taking to ascertain if there is a memory shortage are:-

- Use `vmstat` to see if the system is paging. If not, then there is no chance of a memory shortage. Excessive paging activity is evident by activity in the scan-rate (`sr`) and page-out (`po`) columns, where values are constantly over about 20.
- Look at the swap device for activity. If there is application paging, then the swap device will have I/O's queued to it. Any significant I/O to the swap device means that there is application paging, and is a sure sign of memory shortage.

- Use the MemTool to measure the distribution of memory in the system. If there is a application memory shortage, then the filesystem buffer cache size will be very small (i.e. less than 10 percent of the total memory available).

How much memory is my Application using?

Knowing how much memory your application is using is vital to predicting how much memory your application will need when running more users.

Using MemTool, it is possible to look at how much memory a process is using, including how much is shared between each copy of the program.

If you have Solaris 2.6, then you can use the pmap command, now that the MemTool's pmem functionality has been integrated into Solaris.

Lets take a look at the ksh command, running on a desktop system.

```
# /usr/proc/bin/pmap -x 1069
-- or --
# /opt/RMCmem/bin/pmem 1069

1069:  /bin/ksh
Address  Kbytes Resident Shared Private Permissions      Mapped File
00010000    184    184    184      - read/exec      ksh
0004C000     8      8      8      - read/write/exec ksh
0004E000    40     40     -    40 read/write/exec [ heap ]
EF5E0000    16     16     8      8 read/exec      en_AU.so.1
EF5F2000     8      8     -     8 read/write/exec en_AU.so.1
EF600000   592    576    576     - 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    376    368     8 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   1632    1520    1368    152
```

The output of the `pmap` command shows us that the `ksh` process is using 1520k of real memory. Of this, 1368k is shared with other processes on the system, via shared libraries and executables.

The `pmap` command also shows us that the `ksh` process is using 152k of private memory. This is the amount of memory that this process is using which is not shared. Another instance of `ksh` will only consume 152k of memory (assuming it's private memory requirements are similar).

An important component in systems administration is knowing and understanding how to size applications, so that a capacity planning methodology can be developed.

Developing a sizing methodology for memory is relatively straight forward once the characteristics of an application are known.

The key to understanding the memory requirements of a particular system is to break the resources into a system wide category, and a per-process category.

The goal of sizing the memory requirements of a system or application is to minimise paging. On a server system, it is usually possible to eliminate almost all paging by configuring enough memory to run the required applications.

On a desktop system this is a little harder because of the number of different applications, whose sum total memory requirements is often larger than economically practical for a desktop.

In this chapter we will first look at a simple example, a desktop system, followed by a more complex server sizing exercise.

Sizing a desktop system

There are several aspects that need to be considered when sizing a desktop system:-

- How much memory will the system daemons and libraries use?
- How much Operating system kernel memory is going to be used?
- Which desktop applications do I want to run, and what are their memory requirements?

The performance trade-off for desktops

Without sufficient memory, an application will stall while page faults wait for memory to be paged in/out from disk, and will not make full use of the processor in the system. Because of this, there is a natural trade-off between memory size and system efficiency.

Valued judgements need to be made about how important system performance is, i.e. if you purchase the latest 300Mhz system at premium to provide better performance, you need to ensure that you have sufficient memory so that the application uses as much CPU as available. On the other hand, if cost is the most important factor, then a sensible medium must be found between the ideal memory configuration and an affordable one.

This is a different approach to sizing a server system, where the goal is to configure more memory than the application requires, and use the excess for file buffering.

The best memory size for a desktop system can be calculated using the following methodology, and in most cases will exceed the budget of the average desktop user. A cost conscious desktop should be sized so that the most frequently used applications fit into memory without causing paging, rather than catering for everything at once.

Less frequently used applications will incur paging, but only when switching between applications.

Of course if you have a big enough budget, then you can purchase enough memory to run all of your applications at once, avoiding paging all together!

System Daemon Memory Requirements

The system processes are started at boot time, and provide operating system services to the applications. There are typically 20-30 system processes on a given system, depending on the types of network services configured.

A desktop system will typically have the following system processes started at boot time:

Table 2-1 System Process Memory Requirements

Process	Description	Typical Memory
cron	Commands run over night daemon	296k
nscd	Name service cache daemon	912k
nis	NIS or Nisplus deamons	384k
sendmail	Mailer daemon	568k
sac	Terminal controller	136k
inetd	TCP port listener	512k
powerd	Power Management daemon	61k
in.rdisc	Route discovery daemon	144k
rpcbind	RPC registry	416k
syslog	System logger	440k
keyserv	Kerberos Daemon	72k
vold	Volume Manager	616k
lockd	NFS Lock Daemon	120k
statd	NFS Lock Status Daemon	264k
snmpd	SMNP daemon	212k
lpsched	Print Spooler	272k
automountd	Automounter Daemon	992k
mountd	Mount Daemon	208k
utmpd	Utmp Deamon	96k
ttymon	Console ttymon	192k

On a desktop system, the system processes typically occupy about 6.7MB. If more printers are configured, more memory may be required for the lpsched and lpNet processes.

Memory used by system libraries

The system libraries are dynamically linked libraries which reside in `/usr/lib`. Although these libraries are mapped into every executable, their memory requirements are shared between the many processes on the system.

Generally speaking, the Solaris 2.6 shared libraries will take about 12MB.

The amount of memory used by these files can be summarised by using the MemTool `memp`s command. It can also be approximated by summing the size of the `/usr/lib/lib*.so` files.

```
# memps -m |grep "/usr.*\.so"
1744k /usr/lib/libc.so.1
848k /usr/lib/fn/libfnsp.so.1
808k /usr/openwin/lib/libdps.so.5
784k /usr/dt/lib/libDtSvc.so.1
744k /usr/lib/libfn_spf.so.1
688k /usr/lib/libnsl.so.1
632k /usr/lib/libC.so.5
568k /usr/lib/fn/fn_ctx_onc_fn_nis.so.1
488k /usr/lib/fn/fn_ctx_initial.so.1
456k /usr/lib/libthread.so.1
352k /usr/lib/libxfn.so.2
304k /usr/lib/libelf.so.1
304k /usr/openwin/lib/X11/fonts/F3/Palatino-Roman.f3b
288k /usr/lib/uucp/uuxqt
288k /usr/openwin/lib/X11/fonts/F3/Palatino-Italic.f3b
272k /usr/openwin/lib/libXext.so.0
248k /usr/lib/libresolv.so.2
248k /usr/lib/nss_nis.so.1
240k /usr/lib/libsocket.so.1
224k /usr/lib/libm.so.1
168k /usr/lib/ld.so.1
168k /usr/lib/libprint.so.2
168k /usr/openwin/lib/libdeskset.so.0
160k /usr/lib/libbsm.so.1
128k /usr/dt/lib/libSDtFwa.so.1
120k /usr/openwin/lib/X11/DPS13Fonts.upr
112k /usr/lib/libmp.so.2
112k /usr/openwin/lib/libSM.so.6
112k /usr/lib/libmapallocc.so.1
112k /usr/dt/lib/nls/msg/C
112k /usr/lib/libposix4.so.1
104k /usr/lib/uucp/uusched
88k /usr/lib/nss_xfn.so.1
88k /usr/openwin/lib/libdstt.so.0
88k /usr/share/lib/terminfo/x/xterm
72k /usr/lib/libpthread.so.1
64k /usr/openwin/lib/libolgx.so.3
64k /usr/lib/libintl.so.1
56k /usr/lib/librpcsvc.so.1
56k /usr/share/lib/zoneinfo/Australia/South
56k /usr/lib/nss_files.so.1
48k /usr/openwin/lib/X11/DPSF3Bitmaps.upr
48k /usr/lib/locale/en_AU/en_AU.so.1
40k /usr/lib/fn
40k /usr/lib/libkvm.so.1
40k /usr/lib/libaio.so.1
40k /usr/openwin/lib/locale/common/xlibi18n.so.2
```

Of course, if you don't want to add all of these up you can use a regular expression to make the job easier :-)

```
# prtlibs
Library (.so) Memory:      11856 K-Bytes
```

The unattractive looking regular expression shown above takes the output from `mempfs` and formats each file size into an expression that the `bc` calculator can use.

The later versions of MemTool (3.5 onwards) include the `prtlibs` command to do the same.

Note that all of the library names must be able to be resolved to provide an accurate summary. If not all of the library names are shown, either reboot the system and use the `bunyip` module loader in `/etc/rc2.d`, or run a `"find . -print"` over the directory where the libraries are stored.

Operating System (Kernel) Memory

The amount of memory that the kernel uses on a desktop system is fairly consistent, and scales with the amount of total physical memory installed.

Memory Size	Kernel Size
16MB	8MB
32MB	10MB
48MB	11MB
64MB	12MB
96MB	14MB
128MB	15MB
256MB	17MB
384MB	24MB

Table 2-2 Kernel memory required for desktop systems

CDE Memory Requirements

Since Solaris 2.3, the new Common Desktop Environment (CDE) has been available as alternative desktop for Solaris. At Solaris 2.6, the default desktop is CDE.

CDE uses significantly more memory than OpenWindows. The absolute minimum memory required to run CDE is 32MB, and 48MB is more realistic minimum for a CDE desktop system.

CDE consists of:

- The OpenWindows X Server (Xsun)
- A set of shared libraries which reside in /usr/dt/lib
- The dtlogin (xdm replacement) daemon
- The dtwm window manager
- Various desktop applications

The memory requirements of the CDE processes are shown in the following table.

Process	Description	Typical Memory
Xsun	X Windows Display Server	8120k
dtlogin	CDE Login Banner	464k
Xsession	Users login session	212k
dtwm	CDE Window Manager	2420k
rpc.ttdbserverd	Tooltalk Server	264k
dsdm	CDE display manager	168k
dtpad -server	CDE Textedit server	912k
speckeyd	Keyboard Daemon	176k
sdtvolcheck	Daemon for cdrom and floppy pop-ups	160k
fbconsole	Frame buffer console	104k
dtsession	CDE Session manager	1440k
ttsession	Tooltalk session	1880k
clock	Openwindows Clock	960k
dtfile	CDE File Manager	1240k
dtterm	CDE Terminal	920k
dtmail	CDE Mail Tool	2632k
Total		22072k

Table 2-3 CDE Desktop memory requirements

Total desktop memory requirements

The total memory requirements of a CDE desktop system can be found by adding the system processes, system libraries, kernel memory and CDE memory requirements together.

Our example 64MB system is sized as follows:

Function	Typical Memory
Operating System Kernel	12MB
Operating System Libraries	12MB
CDE Windowing System	22MB
Total	46MB

Table 2-4 CDE Desktop Memory Requirements

If you plan to run other applications, then you will need to add in the memory requirements of that application.

Use the `pmap` command to collect the private memory total for the application in question. If your application executes more than one copy of the same process, then you will also need to include the resident portion of the binary text and data segments. These are the first two segments listed with `pmap`.

For example, a process dump of netscape for Solaris shows that with the default settings, an additional 11MB of memory is required.

Function	Typical Memory
Operating System Kernel	12MB
Operating System Libraries	12MB
CDE Windowing System	22MB
Netscape Browser	11MB
Total	57MB

Table 2-5 CDE Desktop with Netscape Memory Requirements

The process dump from Netscape is as follows:

```

# /usr/proc/bin/pmap -x 1069
-- or --
# /opt/RMCmem/bin/pmem 1069

/usr/proc/bin/pmap -x 10943
10943:(netscape)
Address      Kbytes Resident Shared Private Permissions Mapped File
00010000     9992      9992   1184    8808 read/exec      netscape
009E0000      896       880     8      872 read/write/exec netscape
00AC0000      312        96     -      96 read/write/exec [ heap ]
EF220000       16         -     -      - - [ anon ]
EF224000      128        16     -      16 read/write/exec [ anon ]
EF244000       16         -     -      - - [ anon ]
EF250000       72         24     16     8 read/exec      libICE.so.6
EF270000       8          8     -      8 read/write/exec libICE.so.6
EF272000       8          -     -      - read/write/exec [ anon ]
EF280000      592        576    576     - read/exec      libc.so.1
EF322000       24         24     8      16 read/write/exec libc.so.1
EF328000       8          8     -      8 read/write/exec [ anon ]
EF330000       8          8     -      8 read/write/exec [ anon ]
EF340000       16         16     16     - read/exec      libmp.so.2
EF352000       8          8     8      8 - read/write/exec libmp.so.2
EF360000       32         24     16     8 read/exec      libSM.so.6
EF376000       16         16     -      16 read/write/exec libSM.so.6
EF380000      448        400    368    32 read/exec      libnsl.so.1
EF3FE000       32         32     8      24 read/write/exec libnsl.so.1
EF406000       24         8     -      8 read/write/exec [ anon ]
EF420000       16         16     16     - read/exec      libc_psr.so.1
EF430000       88         88     88     - read/exec      libm.so.1
EF454000       8          8     8      - read/write/exec libm.so.1
EF460000       24         16     8      8 read/exec      libresolv.so.1
EF474000       16         16     -      16 read/write/exec libresolv.so.1
EF480000      432        432    432     - read/exec      libX11.so.4
EF4FA000       24         24     8      16 read/write/exec libX11.so.4
EF510000       32         32     32     - read/exec      libsocket.so.1
EF526000       8          8     8      - read/write/exec libsocket.so.1
EF528000       8          -     -      - read/write/exec [ anon ]
EF530000       72         64     64     - read/exec      libXext.so.0
EF550000       8          8     8      - read/write/exec libXext.so.0
EF560000       80         80     72     8 read/exec      libXmu.so.4
EF582000       8          8     -      8 read/write/exec libXmu.so.4
EF584000       8          -     -      - read/write/exec [ anon ]
EF590000      328        328    328     - read/exec      libXt.so.4
EF5F0000       24         24     8      16 read/write/exec libXt.so.4
EF5F6000       8          -     -      - read/write/exec [ anon ]
EF600000      1440       1424   1416    8 read/exec      libXm.so.3
EF776000       72         64     8      56 read/write/exec libXm.so.3
EF788000       8          -     -      - read/write/exec [ anon ]
EF7A0000       8          8     -      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
E7FFA000       24         24     -      24 read/write/exec [ stack ]
-----
total Kb     15536     14944     4840     10104

```

Sizing a server application

There are several aspects that need to be considered:-

- What are the per-process memory requirements for this application?
- How many processes will be running?
- How much memory will the binaries and libraries use?
- How much buffer cache should I allow for?
- How much System V shared memory do I need?
- How much Operating system kernel memory is going to be used?

Process Memory Requirements

The most significant portion of memory is usually consumed by the application processes. Typically, there are a static number of system processes, and a set of similar processes which are proportional with the number of users.

System Processes

There are several Unix system processes included in this list, such as:-

Table 2-6 System Process Memory Requirements

Process	Description	Typical Memory
cron	Commands run over night daemon	296k
nscd	Name service cache daemon	912k
nis	NIS or Nisplus deamons	384k
sendmail	Mailer daemon	568k
sac	Terminal controller	136k
inetd	TCP port listener	512k
powerd	Power Management daemon	61k
in.rdisc	Route discovery daemon	144k
rpcbind	RPC registry	416k
syslog	System logger	440k
keyserv	Kerberos Daemon	72k
vold	Volume Manager	616k

Table 2-6 System Process Memory Requirements

Process	Description	Typical Memory
lockd	NFS Lock Daemon	120k
statd	NFS Lock Status Daemon	264k
snmpd	SMNP daemon	212k
lpsched	Print Spooler	272k
automountd	Automounter Daemon	992k
mountd	Mount Daemon	208k
utmpd	Utmp Deamon	96k
ttymon	Console ttymon	192k

On a desktop system, the system processes occupy about 3.7MB. Larger server systems could use more memory, particularly if the name service cache daemon is configured larger, and/or lpsched has more printers configured.

Desktop systems usually have an Xserver and dtlogin process running, which occupy significantly more memory.

Background Processes

There is often a large component of memory used by background processes. These are typically associated with RDBMS engines, queue managers and other application specific tasks.

The amount of memory used by the background tasks is often independent of the number of users on the system, so they can be sized separately.

Use `pmem` or `pmap` to get the total resident size for the background tasks, and sum all of the memory used. (The `pmap` and `pmem` commands are described on page 49).

Per-User processes

The most variable part of the workload is likely to be the per-user application processes. Most workloads, including database servers, timeshare systems and middleware clients all have a few processes per client. If you are sizing a machine without a per-user process load (e.g. NFS Server, Threaded web server, etc) then this section is not applicable.

The objective is to establish the relationship between the number of users and the amount of memory required by calculating the private and shared portions of a sample process.

Using MemTool (or `/usr/proc/bin/pmap -x` in Solaris 2.6), it is possible to determine the system-wide and private portions of a process.

The DB2 process following shows a per-process memory dump which contains both SysV shared memory and large application specific shared libraries.

System-wide Portion

To calculate the shared portion, use the executable segments from the output of the `pmap` command. Don't bother with common shared libraries found in `/usr/lib`, because we will count them elsewhere. If there are shared libraries specific to the application, then do count them.

In our example we will count the `db2sysc` executable (which in this case is very small), and the shared db2 libraries.

We can calculate the system-wide portion as:-

$$56k \text{ (executable)} + 15.5MB \text{ (libdb2e)} = 15.6MB.$$

Address	Kbytes	Resident	Shared	Private	Permissions	Mapped File
00010000	40	40	40	-	read/exec	db2sysc
00028000	16	16	8	8	read/write/exec	db2sysc
0002C000	344	168	8	160	read/write/exec	[heap]
10000000	4096	4096	-	-	read/write/exec/shared	[shmid=0xc401]
4AC00000	553712	553712	-	-	read/write/exec/shared	[shmid=0x11c04]
6DF82000	8	8	-	8	read/write/exec	[anon]
6E180000	520	8	8	-	read/write/exec/shared	[shmid=0xa400]
6ED00000	592	592	592	-	read/exec	libc.so.1
6EDA2000	24	24	8	16	read/write/exec	libc.so.1
6EDA8000	8	8	-	8	read/write/exec	[anon]
6EDC0000	8	8	-	8	read/write/exec/shared	[anon]
6EDD0000	16	16	16	-	read/exec	libc_psr.so.1
6EDE0000	16	16	16	-	read/exec	libbmp.so.2
6EDF2000	8	8	8	-	read/write/exec	libbmp.so.2
6EE00000	8	8	8	-	read/write/exec/shared	[anon]
6EE10000	160	128	128	-	read/exec	libC.so.5
6EE46000	32	32	8	24	read/write/exec	libC.so.5
6EE4E000	32	16	-	16	read/write/exec	[anon]
6EE60000	24	16	16	-	read/exec	libresolv.so.1
6EE74000	16	16	8	8	read/write/exec	libresolv.so.1
6EE80000	24	24	24	-	read/exec	libposix4.so.1
6EE94000	8	8	8	-	read/write/exec	libposix4.so.1
6EEA0000	88	88	88	-	read/exec	libthread.so.1
6EEC4000	16	16	8	8	read/write/exec	libthread.so.1
6EEC8000	32	24	-	24	read/write/exec	[anon]
6EEE0000	24	24	24	-	read/exec	libaio.so.1
6EEF4000	8	8	16	-8	read/write/exec	libaio.so.1
6EEF6000	8	8	-	8	read/write/exec	[anon]
6EF00000	448	392	392	-	read/exec	libnsl.so.1
6EF7E000	32	32	24	8	read/write/exec	libnsl.so.1
6EF86000	24	8	-	8	read/write/exec	[anon]
6EFA0000	8	8	-	8	read/write/exec	[anon]
6EFB0000	32	32	32	-	read/exec	libsocket.so.1
6EFC6000	8	8	8	-	read/write/exec	libsocket.so.1
6EFC8000	8	-	-	-	read/write/exec	[anon]
6EFD0000	88	64	56	8	read/exec	libm.so.1
6EFF4000	8	8	16	-8	read/write/exec	libm.so.1
6F000000	11552	11480	4768	6712	read/exec	libdb2e.0721_threadfix
6FB56000	4216	4152	1576	2576	read/write/exec	libdb2e.0721_threadfix
6FF74000	112	48	24	24	read/write/exec	[anon]
6FFA0000	8	8	8	-	read/exec/shared	libw.so.1
6FFB0000	8	8	8	-	read/exec/shared	libdl.so.1
6FFC0000	112	112	112	-	read/exec	ld.so.1
6FFEA000	16	16	8	8	read/write/exec	ld.so.1
EFF80000	32	32	-	32	read/write/exec	[stack]
-----	-----	-----	-----	-----		
total Kb	576688	575632	8080	9744		

Per-process Portion

The per-process portion is the private portion of the process. In our example, the private portion of the process is 9.7MB.

In our DB2 example, there is just one process per user. Take care to include other per-user processes such as `in.telnetd`, `/bin/sh`, etc.

The total amount of private memory per user can later be multiplied by the number of users to arrive at the total private memory. It can also be extrapolated to perform what-if's. For example, if another 100 DB2 clients were connected to our system, we know that 970MB of memory will be required.

Memory used by system libraries

The amount of memory used by system libraries is fairly static. This is because we have already taken into account the private portion of the libraries in the per-process section.

The portion of the libraries that we have not accounted for is the shared library files, which mostly live in `/usr/lib`.

Generally speaking, the shared libraries will take about 15MB on a server, and 25MB on a desktop system running CDE.

The amount of memory used by these files can be summarised by using the MemTool `memp`s command. It can also be approximated by summing the size of the `/usr/lib/lib*.so` files.

```
# memps -m |grep "/usr.*\.so"
1744k /usr/lib/libc.so.1
848k /usr/lib/fn/libfnsp.so.1
808k /usr/openwin/lib/libdps.so.5
784k /usr/dt/lib/libDtSvc.so.1
744k /usr/lib/libfn_spf.so.1
688k /usr/lib/libnsl.so.1
632k /usr/lib/libC.so.5
568k /usr/lib/fn/fn_ctx_onc_fn_nis.so.1
488k /usr/lib/fn/fn_ctx_initial.so.1
456k /usr/lib/libthread.so.1
352k /usr/lib/libxfn.so.2
304k /usr/lib/libelf.so.1
304k /usr/openwin/lib/X11/fonts/F3/Palatino-Roman.f3b
288k /usr/lib/uucp/uuxqt
288k /usr/openwin/lib/X11/fonts/F3/Palatino-Italic.f3b
272k /usr/openwin/lib/libXext.so.0
248k /usr/lib/libresolv.so.2
248k /usr/lib/nss_nis.so.1
240k /usr/lib/libsocket.so.1
224k /usr/lib/libm.so.1
168k /usr/lib/ld.so.1
168k /usr/lib/libprint.so.2
168k /usr/openwin/lib/libdeskset.so.0
160k /usr/lib/libbsm.so.1
128k /usr/dt/lib/libSDtFwa.so.1
120k /usr/openwin/lib/X11/DPS13Fonts.upr
112k /usr/lib/libmp.so.2
112k /usr/openwin/lib/libSM.so.6
112k /usr/lib/libmapallocc.so.1
112k /usr/dt/lib/nls/msg/C
112k /usr/lib/libposix4.so.1
104k /usr/lib/uucp/uusched
88k /usr/lib/nss_xfn.so.1
88k /usr/openwin/lib/libdstt.so.0
88k /usr/share/lib/terminfo/x/xterm
72k /usr/lib/libpthread.so.1
64k /usr/openwin/lib/libolgx.so.3
64k /usr/lib/libintl.so.1
56k /usr/lib/librpcsvc.so.1
56k /usr/share/lib/zoneinfo/Australia/South
56k /usr/lib/nss_files.so.1
48k /usr/openwin/lib/X11/DPSF3Bitmaps.upr
48k /usr/lib/locale/en_AU/en_AU.so.1
40k /usr/lib/fn
40k /usr/lib/libkvm.so.1
40k /usr/lib/libaio.so.1
40k /usr/openwin/lib/locale/common/xlibi18n.so.2
```

Of course, if you don't want to add all of these up you can use a regular expression to make the job easier :-)

```
# prtlibs
Library (.so) Memory:      11856 K-Bytes
```

The unattractive looking regular expression shown above takes the output from `memps` and formats each file size into an expression that the `bc` calculator can use.

The later versions of MemTool (3.5 onwards) include the `prtlibs` command to do the same.

Note that all of the library names must be able to be resolved to provide an accurate summary. If not all of the library names are shown, either reboot the system and use the `bunyip` module loader in `/etc/rc2.d`, or run a “`find . -print`” over the directory where the libraries are stored.

Buffer Cache Memory

Sizing the buffer cache is somewhat more difficult. There is no fixed size of memory required for a buffer, it's really sized by the payback between cost of additional memory and the performance gained by having a larger buffer.

A general rule of thumb is to use about 2% of the size of the dataset. For example, a 10GB database should have about 200MB of buffer cache.

If the database is on RAW filesystems, then buffer cache is not required for this, however the same amount of memory should be allocated for the shared-memory buffer used by the database. If your database is on RAW, then you should still plan to leave about 10% of the total memory free for UFS buffer cache, which will be used by system processes, logfiles, and any other components that are on filesystems.

You can look at the amount of memory currently being used by the buffer cache by using the MemTool `memps` command, and excluding any shared libraries and/or binaries.

On our example system, `memps` shows that there is about 100MB of miscellaneous UFS files in the buffer cache.

System V Shared Memory

Most RDBMS systems use some form of shared memory, either for synchronisation between the various processes, or for a private buffer cache. In the cases where shared memory is used for a private buffer cache, the shared segment can be quite large.

The size of the shared memory segment for database systems is usually decided by the database administrator, because it closely reflects the database tuning parameters.

If you are not sure how much System V memory you are using, use `ipcs` as discussed on page 36 to look at the size of the System V Shared memory segments.

In our example, we have 560MB of System V Shared Memory.

Operating System (Kernel) Memory

The amount of memory that the kernel uses varies significantly, based on the size of the tunable parameters.

A lot of the tuneable parameters are set at boot in proportion with the amount of physical RAM in the system.

As a general rule of thumb, if all of the parameters are standard, you can allow about 15% of physical RAM for the kernel.

In addition to this, you may need to allow for increased tunables. The following table provides a list of some common tunables, and the amount of memory that will be required for each.

Table 2-7 Kernel Memory Required by Tuneables

Tuneable	Description	Memory Required
<code>ncsize</code>	Directory name lookup cache	<code>ncsize*60</code> bytes
<code>ufs_ninode</code>	UFS Inode cache	<code>ufs_ninode*336</code> bytes

In addition to tuneables, some facilities use more memory if they are worked harder. The streams facility follows this behavior. If your system is handling a lot of TCP connections, with high transfer rates, then it is likely that streams could use significantly more memory. By default, streams uses about 2MB, but this could easily expand to 10MB with heavy usage.

Summary - Sizing our DB2 example

In our examples, we looked at a 2GB system running DB2. We saw that this system had a combination of processes, application shared libraries and shared System V memory.

We can summarise the memory requirements for this system running 100 users by collating all of the information so far.

The example is shown in the table.

Table 2-8 Memory Sizing Rules and Example

Item	Rule	Example	Comment
System Processes	+system	10MB	We have allowed a little more than the usual 3.7MB
System Libraries	+system_libraries	15MB	This is fairly consistent with most servers
User-Processes	+proc_user * nusers +proc_systemwide	9.7MB *100=970MB 15.6MB	This is a abnormally large application.
Background Processes	+background_private +background_shared		
Buffer Cache Memory	+buffer_cache	100MB	The database is running on RAW, so we just reserve some memory for other UFS specific requirements.
System V Shared Memory	+shared_memory	560MB	The DB2 DBA set parameters that generated a 560MB shared memory segment.
Kernel Memory	+kernel_memory	150MB	Because this is a RAW database system, there has been no tuning of the DNLC or UFS inode caches, hence we use the default 15%.
Total		1820MB	

To understand the memory behavior and requirements of a particular system, we need to be able to measure the activity and operation of the Virtual Memory system.

In this chapter, we will look at the current tools bundled with Solaris, and some other unbundled tools that allow us to look a little deeper into the Virtual Memory system.

There are two basic objectives when looking at memory in Solaris, one is to find out where all of the memory is allocated, and the other is to look at memory (or paging) activity.

Following is a list of the tools discussed, and the capabilities of each.

Table 3-1 Memory related tools

Tool	Origin	Memory Utilization	Paging Activity
<i>vmstat</i>	/usr/bin	Basic	Fair
<i>ps</i>	/usr/bin	Process Size	-
<i>swap</i>	/usr/bin	Swap allocation	-
<i>wsm</i>	Engineering/free	Working Set Size	Read/Writes per page
<i>ipcs</i>	/usr/bin	SysV Shared Memory	-
MemTool	Engineering/free	Process/Buffer Cache and System	File paging stats

Table 3-1 Memory related tools

Tool	Origin	Memory Utilization	Paging Activity
<i>pmap</i>	/usr/proc/bin	Process Address Map	-
<i>pmap -x</i>	/usr/proc/bin (2.6)	Process Memory Util.	-
<i>crash</i>	/usr/bin/crash	Kernel Memory Util.	-
<i>dbx</i>	SPARCworks	Memory Leaks	-

The *vmstat* and *swap* commands

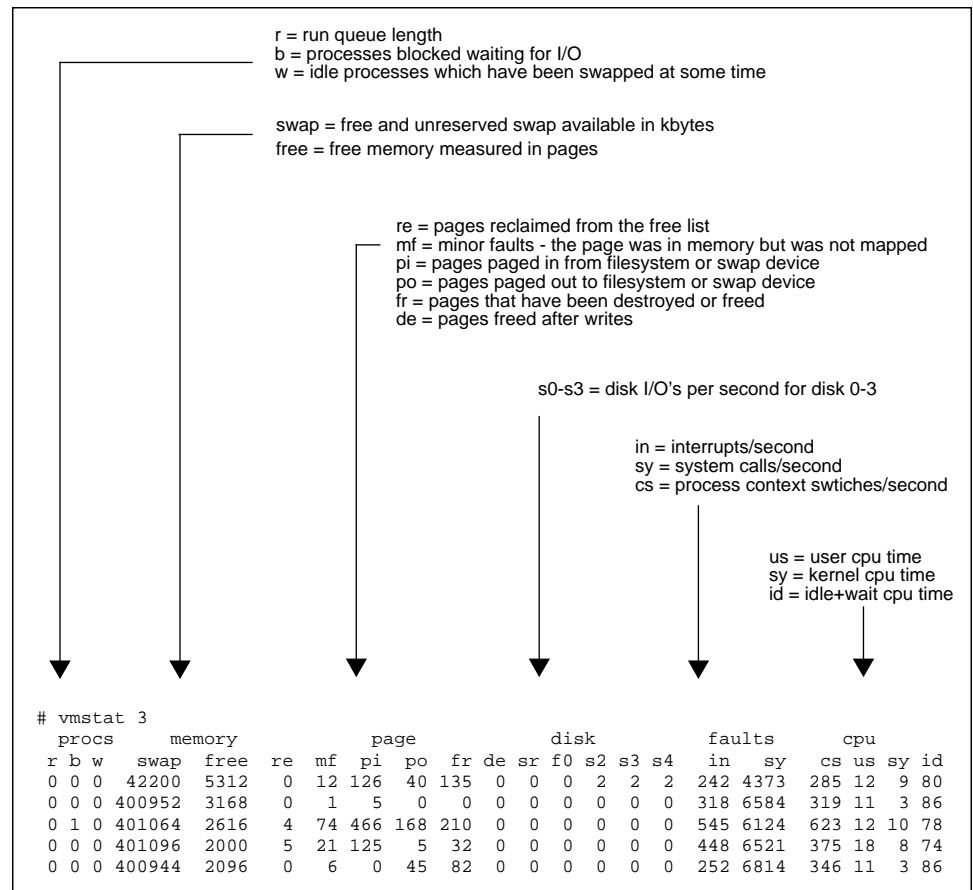


Figure 3-2 *vmstat* output

The `vmstat` utility in Solaris is very similar to the utility that shipped with early versions of BSD Unix. It provides a summary of various functions within the system, including system wide free memory, paging counters, summarized disk activity, system calls and cpu utilization.

The output of `vmstat` is shown above with explanations of the various fields. Lets take a look at how we can use `vmstat` to get a quick summary of what is happening on our system.

Note that the first line of output from `vmstat` shows a summary since boot, followed by the output over the last 3 seconds for each additional line.

The first stop is to look at system wide resources, such as free memory and swap. Systems should have ample swap space available, and in this case we can see that our system has 400MB of swap space free. If it gets down to a few megabytes, process startup will fail.

Free Memory

Our `vmstat` example shows that we have 2096KB of memory free, which seems awful low for a system with 128MB. As discussed in the introduction, this is because the VM system has used all of the free memory for UFS caching, which means that free memory has fallen to approximately the value of `lotsfree`.

Whilst free memory is almost zero, there may be plenty of memory available for applications.

We will look at how to observe how much of our memory is being used for UFS caching later when we discuss MemTool.

Swap Space Utilization

The `vmstat` command reports the amount of swap space that is free (not reserved or allocated). This is the most useful measure.

Swap space is reserved first, then may be allocated. When a process requests memory via `malloc()` for example, the address space is created, but real pages are not allocated to it. At this point, swap space is reserved, but not allocated.

The first time each page is accessed, a real page of memory is allocated to it and swap space is also allocated.

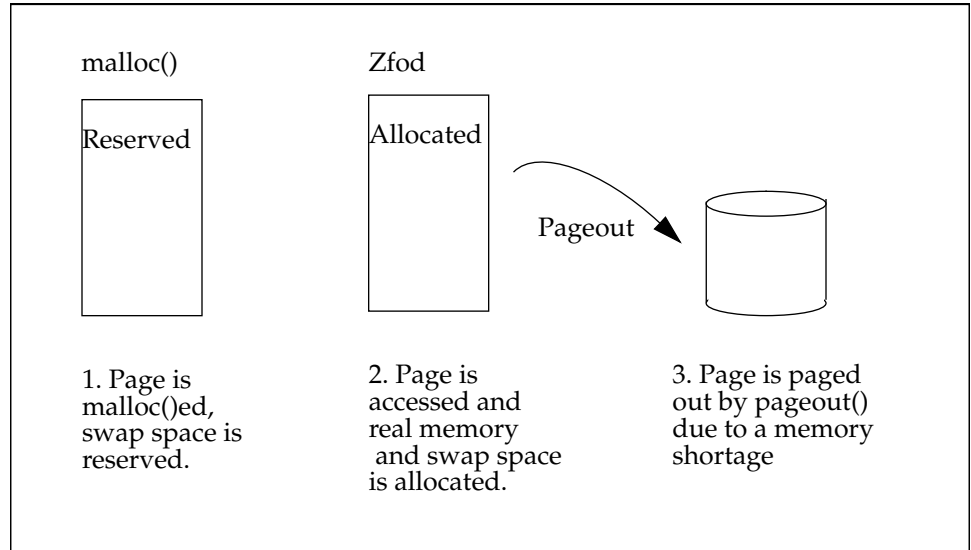


Figure 3-1 Stages of Swap space reservation

In our example we can see that 191MB of swap has been allocated, and 20MB is reserved but not used. This particular system is a desktop with about 20 applications running, hence the large amount of allocated swap space.

```
# swap -s
total: 191504k bytes allocated + 20392k reserved = 211896k used, 400088k available

# swap -l
swapfile      dev  swaplo blocks  free
/dev/dsk/c0t0d0s1  32,121      16 1048784 690672
```

The second swap command shows us a list of the swap devices. These can be either partitions on storage devices or files in a regular filesystem. Note that free space on the swap device does not equal free swap available. This is because Solaris uses an extra layer in-between the swap vnodes and the physical swap device.

The extra layer is the `swapfs` filesystem, which uses a combination of physical swap space and free memory to create a larger than life swap device. This is particularly useful for diskless boot, where there are no physical swap devices during the boot process.

The `/tmp` partition is also uses the `swapfs` filesystem, which means that the size of `/tmp` varies accordingly with the amount of free swap space. An easy check on free swap space is `df -k /tmp`. Note that filling the `/tmp` filesystem will also cause the system to run low on swap space, which will could adversely affect applications and/or performance.

Paging Counters

The `vmstat` paging counters provide us with some insight as to how busy VM system is, and if there are any memory resource issues.

The first thing to look for in the paging counters is the scan rate. The scan rate is the number of pages per second that the pageout scanner is scanning. If the scan rate is consistently zero, then the pageout scanner is not running, and there must be greater than `lotsfree` free memory. If the scan rate is zero, then there is no memory shortage.

A non-zero scan rate does not always mean there is cause for concern. Remember that as reads and writes occur, pages are taken from the free list and eventually the amount of free memory will fall below `lotsfree`. In this case, the pageout scanner will be invoked to free up memory, hence a non-zero scan rate.

Systems with little or no filesystem I/O

On a system with only a small amount of UFS I/O, it is possible to use the page counters to ascertain if there is a memory shortage. A system with a memory shortage will cause excessive page faults can be noted by excessive amount of pageout's(`po`) and a high scan rate (`sr`).

Systems with a lot of filesystem I/O

A system with data on filesystems (as opposed to raw) and a large working set size where the data frequently accessed is larger than the amount of physical memory will also cause excessive paging, and high scan rate numbers. This makes it much more difficult to determine if there is a memory shortage.

If there is a true memory shortage, then the majority of these page faults will incur I/O to the swap device.

Monitoring the swap device I/O is an accurate method of identifying memory shortages. It is strongly recommended that the swap partition be placed on separate partition to make this clearly visible. (On pre 2.6 systems it is necessary to put swap on a separate device, because there are no per-partition statistics available).

If your system does have a lot of filesystem I/O, then we may also use a different method to verify if there is sufficient memory in the system. This will be discussed when MemTool is covered.

Table 3-3 Detailed description of *vmstat* Paging Counters

Counter	Description
re	Page Reclaims - If a page was on the free-cache list, but still contained data that was needed from a new request that it can be taken from the free list and remapped.
mf	Minor Faults - If the page is already in memory, then a minor fault simply re-establishes the mapping to it
pi	Page In's - A page in will occur whenever a page is brought back in from the swap device, or into the new buffer cache. A page-in will cause a process to stop execution until the page is read from disk, and will adversely affect the processes performance.
po	Page Out's - A page out will be counted whenever a page is written and freed. Often this is as a result of the pageout scanner, fsflush or file close.
fr	Page Free's - The number of pages that the pagescanner has freed.
de	The number of pages freed as a result of a pageout.
sr	The number of pages scanned by the page scanner.

Kernel Memory

The amount of memory allocated to the kernel can be found by using the `sar` and `crash` commands.

Using Sar to look at Kernel Memory Allocation

```
# sar -k 3 3

SunOS williams 5.5 Generic sun4m    07/22/97

13:02:14 sml_mem  alloc  fail  lg_mem  alloc  fail  ovsz_alloc  fail
13:02:17 1437696 1163452    0 4571136 3685544    0    2297856    0
13:02:20 1437696 1163452    0 4571136 3685544    0    2297856    0
13:02:23 1437696 1163452    0 4571136 3685544    0    2297856    0
```

The `sar` command can be used to get a coarse grained view of kernel memory allocation.

Table 3-4 Sar command fields

Fields	Description
sml_mem	The amount of virtual memory in bytes KMA has for the small pool.
alloc	The amount of memory in bytes allocated to this pool.
fail	The number of requests for small amounts of memory that were not satisfied (failed)
lg_mem	The amount of virtual memory in bytes KMA has for the large pool.
alloc	The amount of memory in bytes allocated to this pool.
fail	The number of requests for small amounts of memory that were not satisfied (failed)

Fields	Description
ovsz_alloc	The amount of virtual memory in bytes KMA has oversize requests
alloc	The amount of memory in bytes allocated to this pool.
fail	The number of requests for small amounts of memory that were not satisfied (failed)

Using crash for detailed kernel memory allocation

The `crash` command is used to look at data structures in the kernel, either on a live system or a crash dump. Crash also provides a detailed display of the kernel memory allocation status, and is useful for looking at the kernel memory usage on a live system.

Kernel Allocation Methods

The kernel has two methods of allocating memory; either as pageable heap or wired-down permanent memory. The later is known in the kernel as cache memory.

Cache memory is used when multiple occurrences of identical sized memory are required for the same data structure. The kernel allocates physical memory for these data structures and handles the management associated with the holes that are left when structures are deallocated.

The kmastat Command

The `kmastat` command in `crash` provides a detailed summary of kernel memory allocation. It shows each type of `kmem` cache memory, and some statistics for each.

The amount of memory allocated to each type of `kmem` cache is indicated in the `mem-in-use` column.

At the end of the summary is the oversize, or paged heap memory allocation.

```

# crash
dumpfile = /dev/mem, namelist = /dev/ksyms, outfile = stdout
> kmastat

```

cache name	buf size	buf avail	buf total	memory in use	#allocations succeed	fail
-----	-----	-----	-----	-----	-----	-----
kmem_magazine_1	8	1006	1020	8192	7607	0
kmem_magazine_3	16	465	510	8192	94018	0
kmem_magazine_7	32	207	255	8192	70361	0
kmem_magazine_15	64	272	381	24576	80862	0
kmem_magazine_31	128	0	0	0	0	0
kmem_magazine_47	192	0	0	0	0	0
kmem_magazine_63	256	0	0	0	0	0
kmem_magazine_95	384	0	0	0	0	0
kmem_magazine_143	576	0	0	0	0	0
kmem_slab_cache	32	230	765	24576	33621	0
kmem_bufctl_cache	16	680	2550	40960	117420	0
kmem_bufctl_audit_cache	96	0	0	0	0	0
kmem_pagectl_cache	16	463	510	8192	14142	0
kmem_alloc_8	8	218	5100	40960	15486199	0
kmem_alloc_16	16	577	3060	49152	3926197	0
kmem_alloc_24	24	210	1020	24576	10241904	0
kmem_alloc_32	32	248	2295	73728	217807360	0
kmem_alloc_40	40	177	816	32768	218329495	0
kmem_alloc_48	48	180	680	32768	3965748	0
kmem_alloc_56	56	39	290	16384	4476833	0
kmem_alloc_64	64	250	508	32768	392008	0
kmem_alloc_80	80	266	612	49152	9532776	0
kmem_alloc_96	96	45	85	8192	232507209	0
kmem_alloc_112	112	26	72	8192	1055741	0
kmem_alloc_128	128	536	1071	139264	245290152	0
kmem_alloc_144	144	7	56	8192	953967	0
kmem_alloc_160	160	12	306	49152	224277	0
kmem_alloc_176	176	10	184	32768	364206	0
kmem_alloc_192	192	250	504	98304	569160	0
kmem_alloc_208	208	108	351	73728	7292	0
kmem_alloc_224	224	30	36	8192	2611022	0
.
authloopback_cache	40	204	204	8192	18490	0
rnode_cache	432	212	360	163840	4206	0
nfs_access_cache	20	336	340	8192	8596	0
cachefs_cnode_cache	1088	53	195	212992	3441	0
cachefs_async_request	48	170	170	8192	138730	0
cachefs_fscache	592	12	13	8192	1	0
cachefs_filegrp	80	149	204	16384	666	0
cachefs_cache_t	368	21	22	8192	1	0
exi_cache_handle	28	0	0	0	11	0
vfsname_cache	72	56	8249	598016	8193	0
vfsmem_cache	40	1585	1836	73728	2780	0
-----	-----	-----	-----	-----	-----	-----
permanent	-	-	-	65536	371	0
oversize	-	-	-	3891200	12384	0
-----	-----	-----	-----	-----	-----	-----
Total	-	-	-	20176896	1236762467	0

```

>

```

Using *ipcs* to display shared memory

System V Shared memory can be displayed using the `ipcs` command. This shows all of the shared memory segments in the system, and the size of each.

System V Shared memory is typically normal paged memory. In the case where ISM is invoked, (usually this means an RDBMS is being used), then the entire shared memory segment is permanent physical memory.

```
# ipcs -mb
IPC status from <running system> as of Tue Jul 22 01:14:37 1997
T      ID      KEY      MODE      OWNER      GROUP      SEGSZ
Shared Memory:
m      41984    0x74080a1f --rw-rw-rw- dbbench      dba      527096
m      50177    0x61080a1f --rw----- dbbench      dba      4194304
m      62466    0x62080a1f --rw----- dbbench      dba      2195456
m      75779    00000000 --rw----- dbbench      dba      65536
m      72708    00000000 --rw----- dbbench      dba      567001088
m      45061    00000000 --rw----- dbbench      dba      1622016
```

MemTool - Unbundled Memory Tools

MemTool was developed with the intent of providing a more in-depth look at where memory has been allocated on a Solaris system. Using these tools it is possible to find out where every page of memory is, and in what proportions.

MemTool is available as a free, unsupported package from Engineering. Note that these tools are not supported by the normal Sun support channels.

The latest version of MemTool can be obtained by sending a request to mentool-request@chessie.eng.sun.com.

Tools provided with MemTool

There are both command line , character, and GUI tools provided with the MemTool package.

Table 3-5 MemTool Utilities

Tool	Interface	Description
<i>pmem</i>	CL	Command line process memory map and usage
<i>memp</i>	CL	Utility to dump process summary and UFS (-m)
<i>mentool</i>	GUI	Comprehensive GUI for UFS and process memory
<i>mem</i>	CUI	Curses Interface for UFS and process memory

Basis of operation

The basis for the operation of MemTool is a loadable kernel module which uses the /proc interface to look at the memory allocation of processes and the UFS buffer cache.

Process memory usage and the pmem command

Traditionally, the only information about process memory utilization was the virtual memory size and RSS figure available from the `ps` command and `top`.

The virtual address size of a process often bears no resemblance to the amount of memory a process is using because it contains all of the unallocated memory, libraries, shared memory and sometimes hardware devices (in the case of XSun).

The RSS figure is a measure of the amount of physical memory mapped into a process, but often there is more than one copy of the process running, and a large proportion of a process is shared with another.

MemTool provides a mechanism for getting a detailed look at a processes memory utilization. MemTool can show how much memory is in-core, how much of that is shared, and hence how much private memory a process has.

The `pmem` command (or `/usr/proc/bin/pmap -x` in Solaris 2.6) can be used to show the memory utilization of a single process.

```
# pmem 25888
or
# /usr/proc/bin/pmap -x 25888

25888: ksh
```

Addr	Size	Res	Shared	Priv	Prot	Segment-Name
00010000	184K	184k	184k	0k	read/exec	/bin/ksh
0004C000	8K	8k	8k	0k	read/write/exec	/bin/ksh
0004E000	40K	40k	0k	40k	read/write/exec	[heap]
EF5E0000	16K	16k	8k	8k	read/exec	/usr/lib/locale/en_AU.so.1
EF5F2000	8K	8k	0k	8k	read/write/exec	/usr/lib/locale/en_AU.so.1
EF600000	592K	568k	560k	8k	read/exec	/usr/lib/libc.so.1
EF6A2000	24K	24k	8k	16k	read/write/exec	/usr/lib/libc.so.1
EF6A8000	8K	8k	0k	8k	read/write/exec	
EF6B0000	8K	0k	0k	0k	read/write/exec	
EF6C0000	16K	16k	16k	0k	read/exec	/usr/lib/libc_psr.so.1
EF6D0000	16K	16k	16k	0k	read/exec	/usr/lib/libmp.so.2
EF6E2000	8K	8k	8k	0k	read/write/exec	/usr/lib/libmp.so.2
EF700000	448K	400k	400k	0k	read/exec	/usr/lib/libnsl.so.1
EF77E000	32K	32k	8k	24k	read/write/exec	/usr/lib/libnsl.so.1
EF786000	24K	8k	0k	8k	read/write/exec	
EF790000	32K	32k	32k	0k	read/exec	/usr/lib/libsocket.so.1
EF7A6000	8K	8k	8k	0k	read/write/exec	/usr/lib/libsocket.so.1
EF7A8000	8K	0k	0k	0k	read/write/exec	
EF7B0000	8K	8k	8k	0k	read/exec/shared	/usr/lib/libdl.so.1
EF7C0000	112K	112k	112k	0k	read/exec	/usr/lib/ld.so.1
EF7EA000	16K	16k	8k	8k	read/write/exec	/usr/lib/ld.so.1
EFFFC000	16K	16k	0k	16k	read/write/exec	
EFFFC000	16K					[stack]

	1632K	1528k	1384k	144k		

The example output from `pmem` shows the memory map of the `/bin/ksh` command. At the top of the output is the executable text and data segments. All of the executable binary is shared with other processes because it is mapped read only into each process. A small portion of the data segment is shared, whilst some is private because of copy-on-write operations (COW).

The next segment in the address space is the heap space, or user application data. This segment is typically 100% private to a process.

Following the heap space is the shared libraries. Each shared library has a text, and data segment, which are partially shared.

At the bottom of the process dump is the stack, which like the heap is 100% private.

A summary of the total Virtual size, resident portion and private memory are printed at the bottom.

Buffer cache memory

Traditionally there has been no method of showing where the pool of buffer cache memory has been allocated. MemTool makes this possible by providing a list of all of the vnode's in the buffer cache.

The list summarizes the size of each vnode in the buffer cache, and where possible the real filename. If the real filename cannot be determined, then the device and inode number are printed for that vnode.

The MemTool kernel module collects filenames as each file is opened or referenced. If the kernel module has recently been loaded, then not all of the filenames will be available. The best way to cure this is to use the `/etc/rc2.d` script to load the `bunyip` module at boot, which will capture the first 8192 filenames referenced.

If you have a system with many files, you might like to put the following statement into `/etc/system` so that MemTool can store more pathnames. Note that this uses extra kernel memory, and should be avoided on large sun4d (SPARCcenter 1000,2000 machines).

```
set bunyipmod:vfsname_maxitems = 32768
```

The list of vnode's in the UFS buffer cache can be displayed with the memps command, and with the MemTool GUI.

```
# memps -m
SunOS devnull 5.6 SunOS_Development sun4u    07/21/97

11:27:03
  Size  Filename
12152k  /export/home/scott/file1
10680k  /export/home/scott/file20
 8032k  /2b40001: 370743
 6576k  /15c0007: 709619
 5152k  /export/home/scott/file18
 5056k  /export/home/scott/file11
 3744k  /15c0008: 166191
 3288k  /usr/dt/lib/libXm.so.3
 2456k  /15c0007: 709592
 2376k  /export/home/file8
 2272k  /15c0007: 586146
 2264k  /15c0008: 196636
 2016k  /800078: 5970
 1912k  /usr/openwin/lib/libxview.so.3
 1744k  /export/home/scott/file16
 1720k  /15c0007: 709594
 1696k  /15c0007: 132642
 1504k  /2b40001: 1206281
 1504k  /800078: 106190
 1496k  /2b40001: 1204243
 1448k  /15c0007: 709611
 1392k  /export/home/scott/file19
 1264k  /usr/lib/libc.so.1
 1256k  /80007b: 182313
 1200k  /15c0007: 132666
 1096k  /800078: 100213
 1096k  /usr/openwin/lib/libX11.so.4
 1088k  /15c0007: 586141
 1080k  /usr/openwin/lib/libtt.so.2
 1072k  /15c0007: 709632
 1056k  /15c0007: 8844
 1032k  /2b40001: 929861
 1000k  /800078: 200260
  952k  /export/local/bin/perl
  880k  /usr/dt/lib/libDtSvc.so.1
  880k  /15c0007: 709610
  856k  /6167clac: 0
  856k  /usr/openwin/lib/libXt.so.4
  800k  /15c0008: 7231
  752k  /80007b: 113922
  720k  /800078: 82526
.
.
.
```

Note that in the example, not all filenames were visible. This was because the MemTool kernel module was loaded on a live system, and had only captured filenames since the module was loaded.

Using the MemTool GUI

The MemTool GUI interface provides an easy method of invoking most of the functionality of the MemTool kernel interfaces.

Invoke the GUI as the root user to see all of the process and file information.

```
# /opt/RMCmem/bin/memtool &
```

There are three basic modes on the MemTool GUI, Buffer cache memory, Process memory, and a Process/Buffer cache mapping matrix.

Buffer Cache Memory

The initial screen shows the contents of the Buffer Cache memory.

The Buffer Cache Memory display shows each entry in the UFS Buffer cache. The fields shown are as follows:-

Table 3-6 Buffer Cache Fields

Field	Description
Resident	The amount of physical memory that this file has associated with it.
Used	The amount of physical memory that this file has mapped into a process segment or SEGMAP. Generally the difference between this and the resident figure is what is on the free list associated with this file.
Shared	The amount of memory that this file has in memory that is shared with more than one process
Pageins	The amount of minor and major pagein's for this file
Pageouts	The amount of pageouts for this file
Filename	The filename of the VNODE or if not known the device and inode number in the format 0x0000123:456

Resident	Inuse	Shared	Pageins	Pageouts	File Name
11296k	11272k	0k	0	0	/2b40001:370743
4920k	4328k	0k	0	0	/15c0007:709619
4816k	4672k	0k	0	0	/export/home/webarchives/mail/netw
4320k	4272k	56k	0	0	/15c0008:166191
4264k	2576k	8k	0	0	/export/home/webarchives/mail/sun-r
3936k	3840k	1328k	0	0	/usr/dt/lib/libXm.so.3
3304k	3224k	0k	0	0	/800078:5970
3216k	3000k	0k	0	0	/export/home/webarchives/index/wo
2928k	1840k	0k	0	0	/80007b:5701
2288k	1992k	0k	0	0	/15c0008:196636
2152k	2136k	0k	0	0	/2b40001:1204243
2088k	1312k	0k	0	0	/800078:35827
2056k	1808k	456k	0	0	/usr/openwin/lib/libxview.so.3
1992k	1280k	0k	0	0	/export/home/webarchives/mail/unigr
1840k	1040k	0k	0	0	/80007b:182313
1824k	1032k	0k	0	0	/800078:35831
1712k	1656k	0k	0	0	/800078:106190
1560k	1520k	432k	0	0	/usr/openwin/lib/libX11.so.4
1544k	1448k	0k	0	0	/800078:200260
1480k	1456k	0k	0	0	/2b40001:1206281
1368k	824k	0k	0	0	/800078:129575
1304k	1208k	560k	0	0	/usr/lib/libc.so.1
1272k	1256k	328k	0	0	/usr/openwin/lib/libXt.so.4
1240k	792k	0k	0	0	/export/local/bin/perl
1192k	1096k	352k	0	0	/usr/openwin/lib/libtt.so.2
1184k	1168k	0k	0	0	/2b40001:929861
1120k	1032k	0k	0	0	/800078:100213

Figure 3-2 MemTool GUI - Buffer Cache Memory

The GUI will only display the largest 250 files. A status panel at the top of the display shows the total amount of files and the number that have been displayed.

Process Memory

The second mode of the MemTool GUI is the process memory display. Click on the “Process Memory” checkbox at the left of the GUI to select this mode.

The process memory display shows the process table with a memory summary for each process. Each line of the process table is the same as the per-process summary from the `pmem` command.

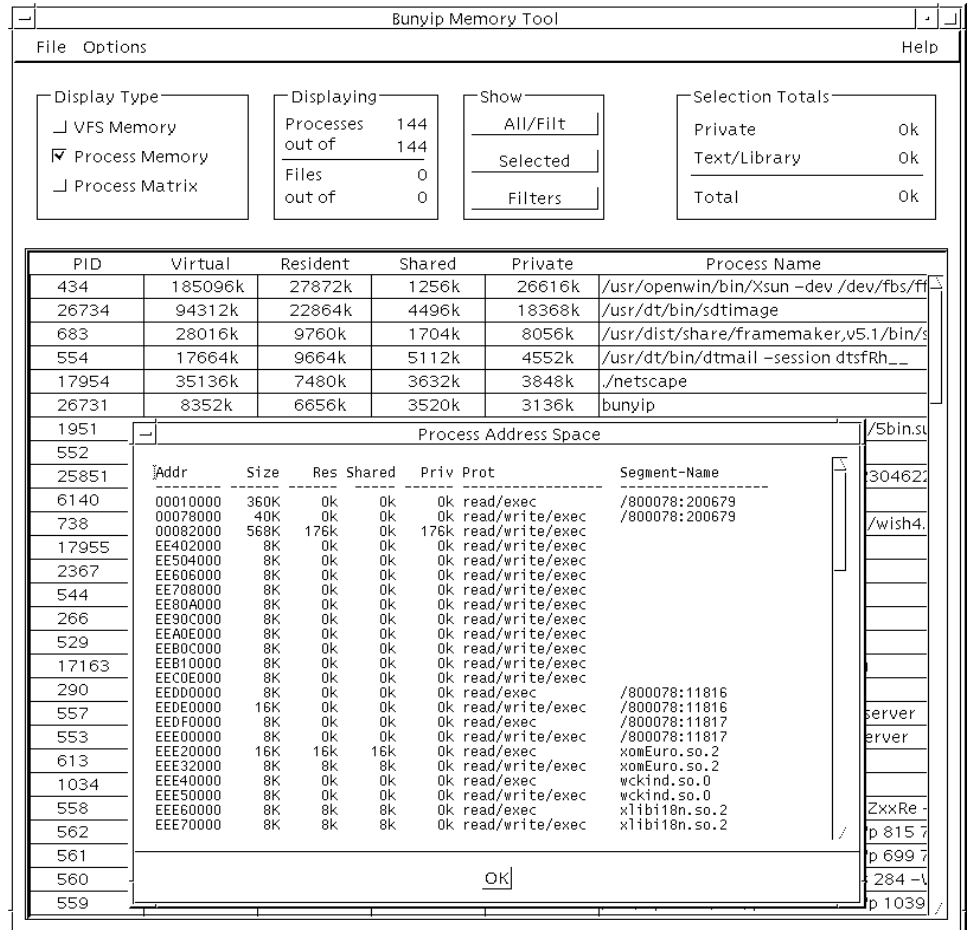


Figure 3-7 MemTool GUI - Process Memory

The fields for the Process Memory display are as follows:-

Table 3-8 Process Memory Fields

Field	Description
PID	Process ID of process
Virtual	The virtual size of the process, including swapped out and unallocated memory
Resident	The amount of physical memory that this process has, including shared binaries, libraries etc
Shared	The amount of memory that this process is sharing with another process, i.e. shared libraries, shared memory etc.
Private	The amount of resident memory that this process has which is not shared with other processes. This figure is essentially Resident - Shared and does not include the application binaries.
Process	The full process name and arguments

The individual process map for a process can be selected by clicking on one of the process entries.

Process Matrix

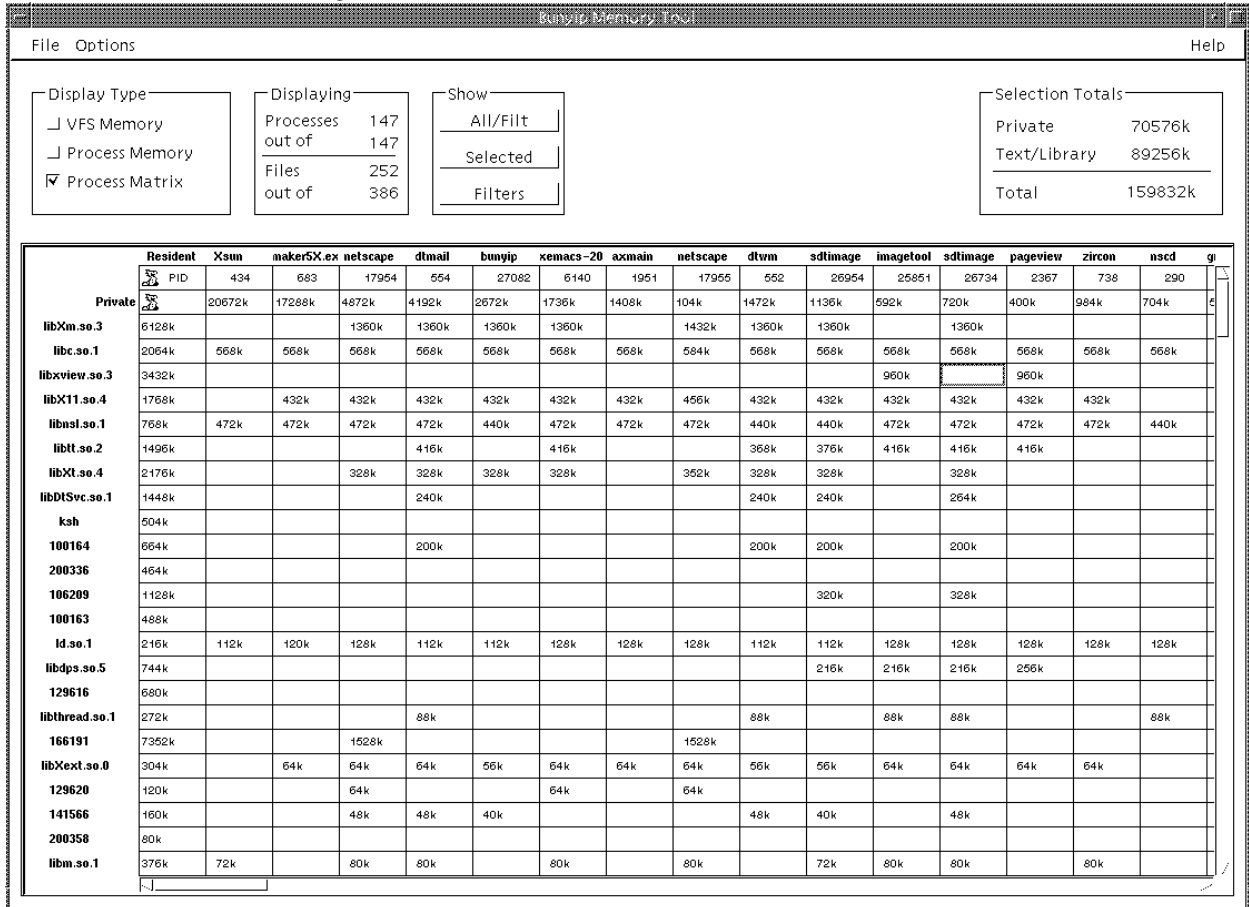
The process matrix shows the relationship between processes and mapped files. Across the top of the display is the list of processes that we viewed in the process memory display, and down the side is a list of the files which are mapped into these processes.

Each column of the matrix shows the amount of memory mapped into that process for each file, with an extra row for the private memory associated with that process.

The matrix can be used to show the total memory usage of a group of processes. By default, the summary box at the top right hand corner shows the memory used by all of the processes displayed.

A group of processes can be selected with the left mouse button, and then summarized by hitting the *selection* button at the top-middle of the display. The full display can be returned by selecting the *all/filt* button.

Figure 3-9 MemTool GUI - Process/File Matrix



GUI Options

There are also some options to configure the order of the rows of files or processes displayed. By default, they are sorted in reverse memory size order. The Options menu can be used to select the sort options dialog.

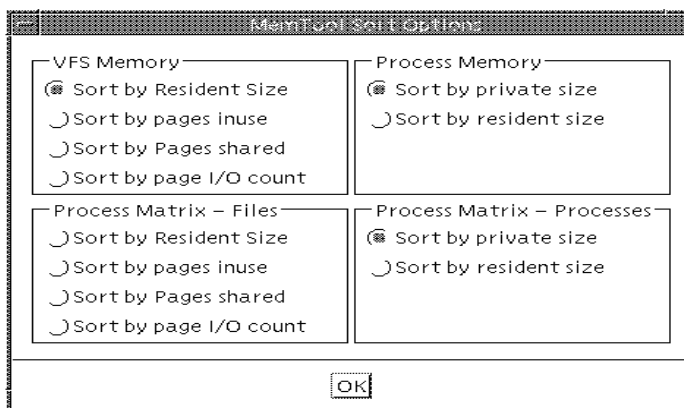


Figure 3-3 MemTool GUI - Sort Options

The Workspace Monitor Utility - WSM

Another good utility for monitor memory usage is the workspace monitor. It shows a live status of a processes memory map, and the amount of memory that has been read and/or written to in the sampled interval.

This is particuly useful for determining how much memory a process is using at any given instant.

The `wsm` command is invoked against a single process.

```
# wsm -p 732 -t 10
Read  Write  Mapped  PROT Segment  maker5X.exe(pid 683)  Mon Jul 21 15:44:10 1997
235   0      782    (R-X) maker
10    11      36     (RWX) maker
207   384    2690   (RWX) Bss & Heap
14    0       74     (R-X) /usr/lib/libc.so.1
2     1       3      (RWX) /usr/lib/libc.so.1
0     1       1      (RWX) /dev/zero <or other device>
0     0       1      (R-X) /usr/lib/straddr.so
0     0       1      (RWX) /usr/lib/straddr.so
1     0       2      (R-X) /usr/platform/SUNW,Ultra-2/lib/libc_psr.so.1
1     0       1      (RWX) /dev/zero <or other device>
0     0       56     (R-X) /usr/lib/libnsl.so.1
0     0       4      (RWX) /usr/lib/libnsl.so.1
0     0       3      (RWX) /dev/zero <or other device>
0     0       2      (R-X) /usr/lib/libmp.so.2
0     0       1      (RWX) /usr/lib/libmp.so.2
0     0       9      (R-X) /usr/openwin/lib/libXext.so.0
0     0       1      (RWX) /usr/openwin/lib/libXext.so.0
26    0       54     (R-X) /usr/openwin/lib/libX11.so.4
2     1       3      (RWX) /usr/openwin/lib/libX11.so.4
0     0       4      (R-X) /usr/lib/libsocket.so.1
0     0       1      (RWX) /usr/lib/libsocket.so.1
0     0       1      (RWX) /dev/zero <or other device>
0     0       1      (R-X) /usr/lib/libdl.so.1
0     0       14     (R-X) /usr/lib/ld.so.1
2     0       2      (RWX) /usr/lib/ld.so.1
0     3       6      (RWX) Stack
500   401    3753   Totals
```

The counters in the `wsm` utility are in units of pages.

Finding Memory Leaks with DBX

A memory leak occurs when an application allocates memory, and then never frees it.

A application with a memory leak can be confirmed by using MemTool (or pmap) to look at the private portion of resident memory. If the private portion continuously grows, then it is likely there is a memory leak.

The Run-time Leak Checker

The SunPro tools provide a great mechanism for tracking down memory leaks in applications. The memory leak feature was made available in SPARCworks version 3 onwards.

The example test program, memleak.c shows a typical leak.

```
#include <stdio.h>
#include <stdlib.h>

main( int argc, char **argv)
{
    void *p;

    /* Allocate 50 bytes of memory */
    p=malloc(50);

    /* Loose the pointer to the original 50 bytes and
       allocate another 50 bytes */
    p=malloc(50);

    /* Free the second 50 bytes */
    free(p);

    /* Exit */
}

```

Compiling the program

To use the SunPro memory leak checker, we must have access to the source of the application, and compile with the -g flag

```
$ cc -g -o memleak memleak.c
```

Running the Leak Test

The next step is to start the program under control of dbx, after enabling the memory leak checker.

```
$ dbx memleak
Reading symbolic information for memleak
Reading symbolic information for rtld /usr/lib/ld.so.1
Reading symbolic information for libc.so.1
Reading symbolic information for libdl.so.1
Reading symbolic information for libc_psr.so.1
(dbx) check -leaks
leaks checking - ON
(dbx) run
Running: memleak
(process id 9554)
Reading symbolic information for librttc.so
Skipping libc.so.1, already read
Skipping libdl.so.1, already read
Skipping libc_psr.so.1, already read
Enabling Error Checking... done
Checking for memory leaks...

Actual leaks report (actual leaks:      1 total size:      50 bytes)

Total Num of Leaked Allocation call stack
Size  Blocks  Block
      Address
=====
      50      1  0x20a70  main

Possible leaks report (possible leaks:      0 total size:      0 bytes)

execution completed, exit code is 1
(dbx)
```

The Solaris Virtual Memory (VM) system used in Solaris 2.x today is a complete rewrite of the SunOs 3.x VM system. The new VM system first appeared in SunOs 4.x. This new VM system was written from the ground up as an object-oriented extensible framework which allows new technology (including filesystems) to be easily integrated into the operating system.

Together with the *vnode* architecture (*vnode*'s are discussed in the technical VM description) already adopted in SunOs, it formed the core of AT&T's Unix System V Release 4.0 which was a joint development between Sun Microsystems and AT&T.

Why Have A Virtual Memory System?

One of the objectives of a VM System is to allow memory objects to exist which are larger than the available physical memory. This allows processes to have a larger memory than available primary storage (e.g. RAM), and use slower but larger secondary storage (e.g. disk) as a backing store.

A virtual view of memory storage known as an address space is presented to the application, while the VM system transparently manages the virtual storage between RAM and secondary storage.

Because RAM is significantly faster than disk, (100ns vs. 10ms, or approx. 100,000 times faster), the job of the VM system is to keep the most frequently referenced portions of memory in the faster primary storage.

In the event of a RAM shortage, the VM system is required to free RAM by transferring infrequently used memory out to the backing store.

The VM system is also required to cater for the needs of multiple users, tasks and workloads. In these environments, program binaries and application data may be shared between users, and shared memory management is required so that memory is not unnecessarily wasted when multiple instances of applications are executed.

A recap of the major functions performed by a VM system are to manage the:-

- virtual to physical mapping of memory
- swapping of memory between primary and secondary storage to optimize performance
- requirements of shared images between multiple users and processes

Demand Paging

There are two basic types of VM systems used in most operating systems, they are *swapping* or *demand paged*.

The swapping memory systems use a user process as the granularity for managing memory. If there is a shortage of memory then the least active process is swapped out, freeing memory for other processes. This method is easy to implement, but performance suffers badly when there is a memory shortage because a process cannot resume execution until all of its pages have been brought back in from secondary storage.

The demand paged model uses a small chunk of memory known as a *page* as the granularity for memory management. Rather than swapping out a whole process, the memory system just swaps out small least used chunks, which allows processes to continue while an inactive part of them is swapped out.

Solaris uses a combined demand paged and swapping model. Demand paging is used under normal circumstances, and swapping is only used as a last resort method when desperate for memory.

Combined I/O and Memory Management

The Solaris VMVM system implements many more functions than just managing application memory. In fact under Solaris, the VM system is responsible for managing objects related to I/O and memory, including the kernel, user applications, shared libraries and filesystems.

This differs significantly from other operating systems like earlier versions of System V Unix, where there was a separate buffer cache for filesystem I/O.

One of the major advantages of using the VM system to manage filesystem buffering is that all free memory in the system is used for file buffering, providing significant performance improvements and removes the need for tuning the size of the buffer cache.

The VM system gobbles up all free memory for filesystem buffers, which means that on a typical system with filesystem I/O, the amount of free memory available is almost zero. This can often be misleading, and has resulted in numerous bogus memory leak bugs being logged over the years. Don't worry, it's normal.

Design Goals of the Solaris Virtual Memory

The new VM system was built with the following goals in mind:-

- A new object-oriented memory management framework
- A virtual file concept (known as the *vnode*)
- Address spaces that are mapped vnode objects
- Support for shared and private memory (copy-on-write)
- Page based VM management

The VM system which resulted from these design goals provides an open framework which now supports many different memory objects. The most important objects of the memory system are segments, vnode's and pages, which are discussed in more detail later in the text. For example, all of the following have been implemented as abstractions of the new memory objects:-

- Physical memory, in chunks called Pages
- Files, as vnode in a filesystem
- Filesystems, as a hierarchy of vnode's

- Mapped hardware devices, such as framebuffer as a segments of hardware mapped pages
- Process address spaces, as segments of mapped vnode's
- Kernel address space, as segments of mapped vnode's

PAGES - The basic unit of Solaris memory

Hardware Memory Management Units

Modern hardware architectures deal with physical memory in large chunks, rather than individual bytes. These chunks are referred to as *pages*, and the size of the chunk is governed by the hardware memory management unit. The SPARC hardware offered by Sun over the past few years has had several different types of memory management unit, which support a variety of page sizes:

Table 4-1 Sun MMU Page Sizes

System Type	System Type	Solaris 2.x Page Size
SuperSPARC I & II (SC2000, SS20 etc.)	sun4m,d	4k
HyperSPARC	sun4m	4k
MicroSPARC I,II	sun4m	4k
UltraSPARC I,II	sun4u	8k

Each of the MMU's support a wide range of page sizes; however Solaris is mostly implemented using a fixed page size for each architecture. The Solaris sun4c, sun4m and sun4d architectures all use a 4K page size. The new sun4u UltraSPARC machines all use an 8K page size.

The optimal MMU page size is a trade-off between performance and memory size efficiency. A larger page size has less memory management overhead, and hence better performance, while a smaller page size wastes less memory due to it's smaller page size (memory is wasted when a page is not completely filled).

When UltraSPARC was introduced, the cost of memory had greatly reduced, and the average size of memory on a system had grown to the point where a larger PAGE size provided better price/performance.

Solaris 2.6 actually breaks the fixed page size rule by implementing a large kernel PAGE to reduce the kernel's memory management overhead.

VNODE's - *The Virtual File Abstraction*

The basis for all file objects in Solaris is the *vnode*, which also plays a very important role in memory management.

The *vnode* was introduced as a new object to describe a virtual file, which provides a filesystem and device independent interfaces to the kernel. The *vnode* interface allows the 'virtual file' to describe many different logical and physical devices, including disks, tty's, network streams and sockets.

The *vnode* interface provides a information and pointers to the device-specific functions about that file. All file operations (e.g. read, write, open, close) can be performed on the *vnode*, without having to know what the underlying device and filesystem are.

For example, to open a file without knowing that it resides on a UFS filesystem the code fragement would be:

```
vnode_t *vp;    /* Vnode pointer */
cred_t *cred;  /* Credentials, eg userid etc */

VOP_OPEN( vp , FREAD, cred )
```

The *vnode* open macro would call the `open()` function of the underlying filesystem for that *vnode*.

The *vnode* interface is shown in the following diagram:

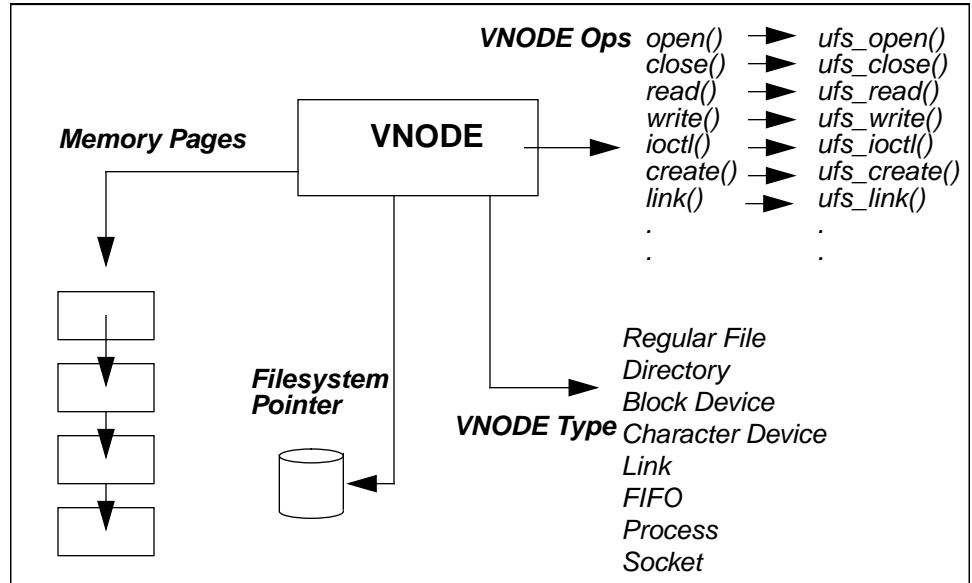


Figure 4-2 VNODE Interface

This diagram shows a vnode on a UFS filesystem. In this example, the vnode has pointers for its functions (open, close etc.) which reference UFS specific functions. A call to `open()` on this file simply calls the `open` function in the vnode, which in turn calls the `ufs_open()` function. The UFS vnode, has a pointer to the associated pages of memory which are currently in physical RAM for this file (this is the buffer cache for this file).

vnode represent many other different types of files with different filesystems. For example, a vnode which represents a raw disk device does not have memory pages associated with it, and rather than pointing to the UFS filesystem, it points to a virtual filesystem for special devices (specfs), which contains functions for operating on character and block devices.

Another example of a vnode pointing to a special disk device is the SWAP device. As we will see later, the SWAP vnode is used with the page structure to represent application memory.

The structure a vnode in Solaris 2.6 shows the basic interface elements, along with the other information contained in the vnode:

```

typedef struct vnode {

    kmutex_t      v_lock;           /* protects VNODE fields */
    u_short       v_flag;           /* VNODE flags (see below) */
    u_long        v_count;          /* reference count */
    struct vfs     *v_vfsmountedhere; /* ptr to vfs mounted here */
    struct vnodeops *v_op;           /* VNODE operations */
    struct vfs     *v_vfsp;          /* ptr to containing VFS */
    struct stdata  *v_stream;        /* associated stream */
    struct page    *v_pages;         /* VNODE pages list */
    enum vtype     v_type;           /* VNODE type */
    dev_t         v_rdev;            /* device (VCHR, VBLK) */
    caddr_t       v_data;            /* private data for fs */
    struct filock  *v_filocks;       /* ptr to filock list */
    struct shrlocklist *v_shrlocks; /* ptr to shrlock list */
    kcondvar_t    v_cv;             /* synchronize locking */

} vnode_t;

```

Data Structure 4-3 Solaris 2.6 VNODE Structure

The HAT Layer

The relationship between physical RAM and the page structure is managed by the Hardware Address Translation layer (HAT layer). The HAT layer is machine specific set of routines that manage the mappings and address translation between the PAGE structures and the MMU Hardware pages. The HAT layer routines are called to set up and pull down the address translations each time a page is created or destroyed (or paged in and out from backing store).

The HAT layer also handles *traps*, so that when a reference is made to a VM location that does not currently have a physical PAGE in core a *fault* routine is invoked to bring the page in from the backing store.

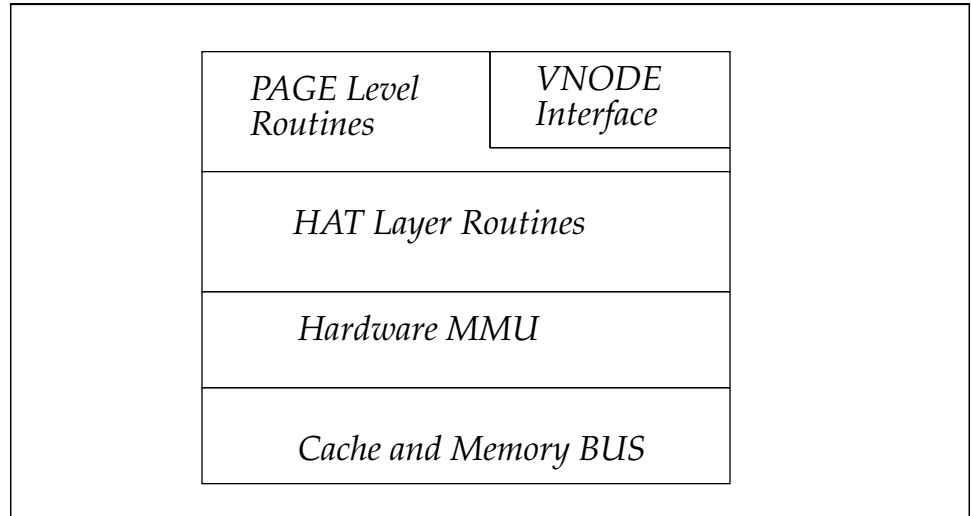


Figure 4-4 VM Layers

Pages as Vnode and Offset

In Solaris there is always a vnode associated with an allocated page of physical memory. Each page of memory is described by a vnode and an offset within that vnode. The vnode is used to describe the backing store for that page of memory.

If the page is application memory, then the vnode for that page is the SWAP device vnode. If the page is a buffer cache entry for a file, then the vnode is that of the file being buffered.

Each page is a member of a hashed list of pages in the system. To find a particular page of memory, the VM system uses the vnode and offset as a hash key to find a pointer to the page. The VM system uses the `page_find()` function to locate pages by searching the hash list.

As well as the page hash list, there are two other lists of pages. These are the free list, and the cache list. The free list is a hashed list of pages that do not have any mappings to VM. The cache list is a list of pages that are free, but are still mapped to a particular vnode and offset. The total amount of free memory = free list pages + cache list pages.

Cache list pages may be reused if the VM system needs to create a new mapping for a page which was already in memory, but freed by the last user. The cache list reuse scheme stops the system from paging in and out the same pages over and over, or *thrashing*.

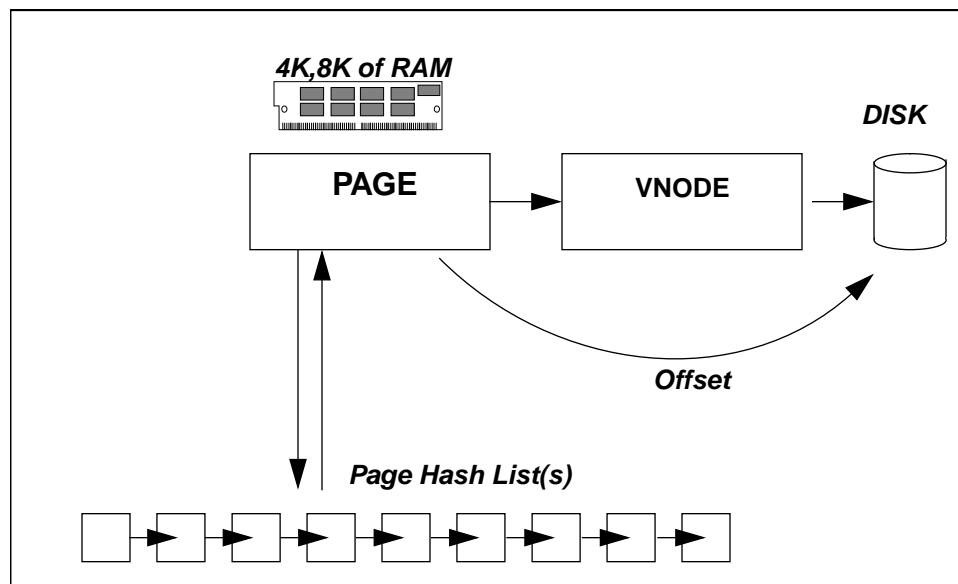


Figure 4-5 PAGE Level Interface

Each page has a state flag which indicates if the page is free, has been referenced or modified. The state flag information is synchronized from the registers in the MMU by the HAT layer each time the page structure is called, or when the HAT layer function `hat_sync()` is called.

The page structure has many more elements than described in the pictorial interface, most of which are locks and condition variables which are used to signal processes which may be waiting for an I/O operation on the page. The Solaris 2.6 page structure can be found in `/usr/include/vm/page.h`.

```

typedef struct page {

    struct vnode*p_vnode;    /* logical vnode this page is from */
    struct page  *p_hash;    /* hash by [vnode, offset] */
    struct page  *p_vpnext;  /* next page in vnode list */
    struct page  *p_vpprev;  /* prev page in vnode list */
    struct page  *p_next;    /* next page in free/intrans lists */
    struct page  *p_prev;    /* prev page in free/intrans lists */
    u_offset_t   p_offset;   /* offset into vnode for this page */
    selock_t     p_selock;   /* shared/exclusive lock on the page */
    u_short      p_lckcnt;   /* number of locks on page data */
    u_short      p_cowcnt;   /* number of copy on write lock */
    kcondvar_t   p_cv;      /* page struct's condition var */
    kcondvar_t   p_io_cv;   /* for iolock */
    u_char       p_iolock_state; /* replaces p_iolock */
    u_char       p_filler;   /* unused at this time */
    u_char       p_fsdata;   /* file system dependent byte */
    u_char       p_state;    /* p_free, p_created */

} page_t;

```

Data Structure 4-6 Solaris 2.6 PAGE Structure

Virtual Address Spaces

Memory Segments

We know that VM pages are mapped to physical pages through the MMU and HAT layer, and that each page has some form of backing store. The missing link is how pages relate to a linear address space, which is what applications expect to see.

The relationship between pages and linear address space is managed by memory segments. A segment is a mapping of a particular memory address and length to a device. There is also an object oriented segment interface, which provides a device independent view of the device from which the segment is mapped. These are called segment drivers.

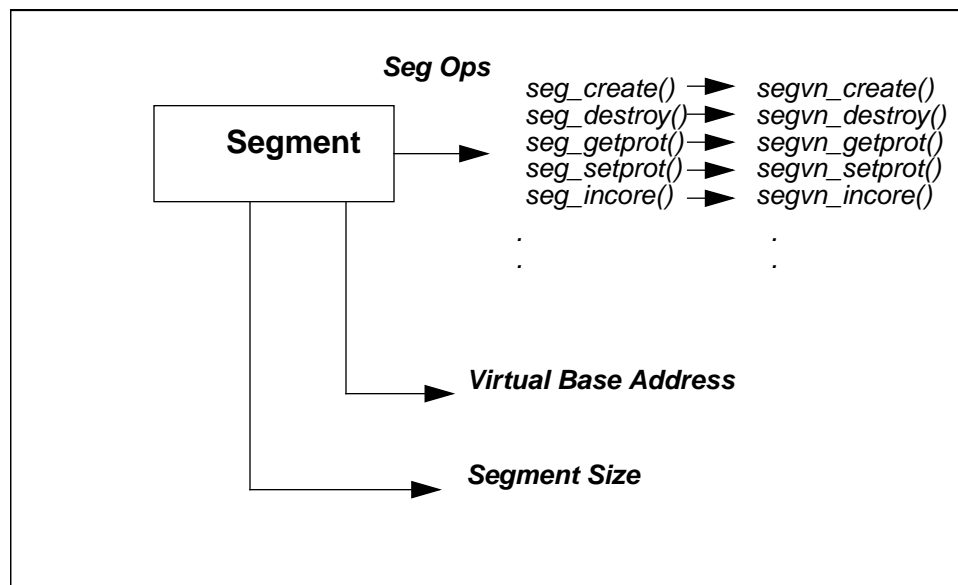


Figure 4-7 Segment Interface

The most commonly used segment driver in Solaris is the vnode segment, or *segvn*, which is used to map a vnode at a particular virtual address and offset. The vnode segment is used for:-

- Anonymous application memory (e.g. malloc()'ed heap memory, kernel heap, System V shared segments, program stacks) which has the SWAP device as it's vnode
- Executable binaries and shared libraries which have the program file in the filesystem (e.g. /bin/sh or /usr/lib/libc.so)
- Regular files, where a file has pages in memory (filesystem cache)

There are other types of memory which don't have pages or vnode's associated with them. These other types of segments are typically associated with hardware devices, such as graphics adapters.

Table 4-8 Solaris 2.6 Segment Drivers

Segment	Function
<i>seg_vn</i>	Mapped files, SWAP etc.
<i>seg_map</i>	Optimized version of <i>seg_vn</i> for I/O
<i>seg_dev</i>	Mapped hardware devices
<i>seg_mapdev</i>	Mapped character devices
<i>seg_mdi</i>	Mapped multimedia devices (graphics)
<i>seg_vpix</i>	For VP/ix V86 DOS emulation
<i>seg_sx</i>	SX Memory Driver for SS20-SX

Segment Protection

Each segment is mapped with a specific protection, which is a combination of:-

- EXEC - The mapping is allowed to have machine codes executed within it's address range, typically shared with other processes.
- READ - The mapping is allowed to be read from, writes will generated a SIGSEGV if write protection is not also enabled.
- WRITE - The mapping is allowed to be written to, reads will generate a SEGSEGV if read protection is not also enabled.
- SHARED - All writes to this segment are shared with other segments, including other processes.
- PRIVATE - Writes to this segment will cause the VM system to fault and allocate a private PAGE of anonymous memory at the write address. This is called Copy On Write (COW).

Segment protection mapping can be read about in the man page for the `mmap()` system call, and in `/usr/include/sys/mman.h`

Process Address Spaces as Mapped Segments

The virtual address space of a process on Solaris 2.6 is 4GB, with the binaries at the bottom, and the stack at the top. Shared libraries appear close to the top of the mapping.

The Address Space of a process is simply a mapping of different segments in to a virtual address space.

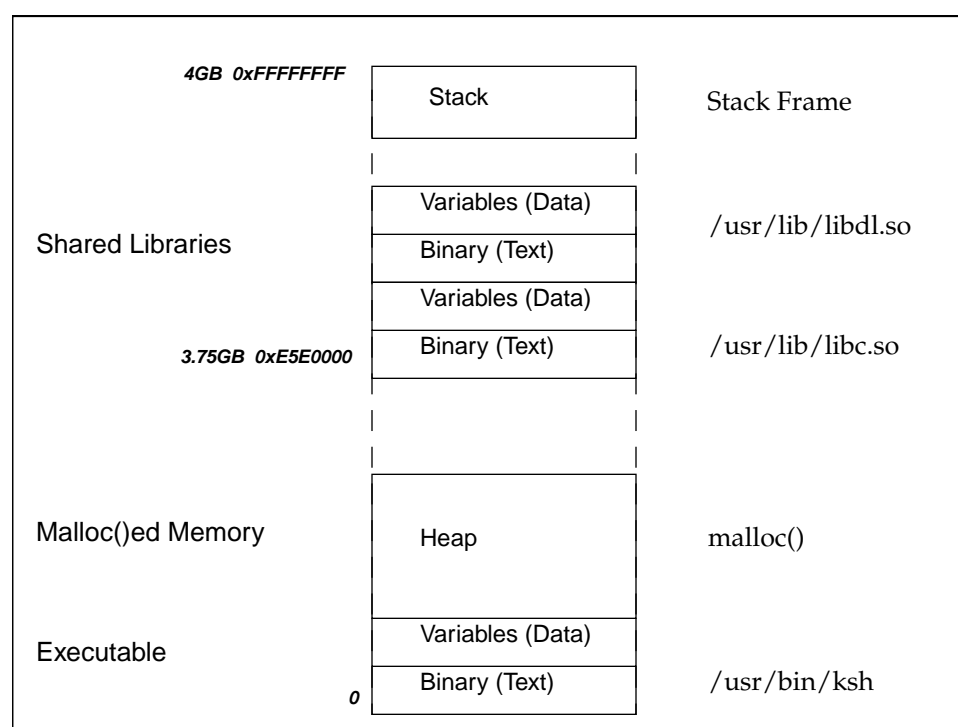


Figure 4-9 Solaris 2.6 Virtual Address Space for a Process

Note that the address space is not contiguous. There is space between the shared libraries and the stack, and there is also space between the heap and the shared libraries. These empty addresses are often used for other segments, such as System V shared memory and mmap()ed files.

The segments are typically:-

Executable Text

A mapping of the pure executable part of the binary, mapped READ only. Executables smaller than 280k (set by the `smallfile` parameter) are pre-faulted in, rather than relying on demand paging.

Executable Data

A mapping of the Data segment of the binary, which contains initialized variables from the binary. The Data segment is mapped READ, WRITE and PRIVATE so that changes to the binaries Data segment are not reflected into other processes (COW).

This happens when a program changes the value of one of its initialized variables. For example, if `prog.c` sets `i=0`, then the value 0 is stored in the Data segment; if the value of `i` is changed, then a COW page is created and mapped over the original page in the Data segment.

Heap

The program Heap contains all of the programs anonymous memory, which is usually allocated via `malloc()` or `brk()`. Anonymous memory is mapped READ, WRITE and PRIVATE.

Shared Library Text and Data

The Shared libraries Text and Data segments are mapped with the same protections as the executable.

Optional System V Shared Memory

Mapped SHARED, and is mapped into the address space of other processes so that changes are reflected.

Optional mmap()ed files.

Files can be mapped into the address space with the `mmap()` system call, They can be mapped with any protection except EXEC.

Stack

The program stack is a separate mapping of anonymous memory which is mapped READ and WRITE.

An example process address space can be seen using the pmap command.

```
# pmem 25888
or
# /usr/proc/bin/pmap -x 25888

25888: ksh
```

Addr	Size	Res	Shared	Priv	Prot	Segment-Name
00010000	184K	184k	184k	0k	read/exec	/bin/ksh
0004C000	8K	8k	8k	0k	read/write/exec	/bin/ksh
0004E000	40K	40k	0k	40k	read/write/exec	[heap]
EF5E0000	16K	16k	8k	8k	read/exec	/usr/lib/locale/en_AU.so.1
EF5F2000	8K	8k	0k	8k	read/write/exec	/usr/lib/locale/en_AU.so.1
EF600000	592K	568k	560k	8k	read/exec	/usr/lib/libc.so.1
EF6A2000	24K	24k	8k	16k	read/write/exec	/usr/lib/libc.so.1
EF6A8000	8K	8k	0k	8k	read/write/exec	
EF6B0000	8K	0k	0k	0k	read/write/exec	
EF6C0000	16K	16k	16k	0k	read/exec	/usr/lib/libc_psr.so.1
EF6D0000	16K	16k	16k	0k	read/exec	/usr/lib/libmp.so.2
EF6E2000	8K	8k	8k	0k	read/write/exec	/usr/lib/libmp.so.2
EF700000	448K	400k	400k	0k	read/exec	/usr/lib/libnsl.so.1
EF77E000	32K	32k	8k	24k	read/write/exec	/usr/lib/libnsl.so.1
EF786000	24K	8k	0k	8k	read/write/exec	
EF790000	32K	32k	32k	0k	read/exec	/usr/lib/libsocket.so.1
EF7A6000	8K	8k	8k	0k	read/write/exec	/usr/lib/libsocket.so.1
EF7A8000	8K	0k	0k	0k	read/write/exec	
EF7B0000	8K	8k	8k	0k	read/exec/shared	/usr/lib/libdl.so.1
EF7C0000	112K	112k	112k	0k	read/exec	/usr/lib/ld.so.1
EF7EA000	16K	16k	8k	8k	read/write/exec	/usr/lib/ld.so.1
EFFFC000	16K	16k	0k	16k	read/write/exec	
EFFFC000	16K					[stack]

	1632K	1528k	1384k	144k		

The Pageout Process

An additional component of the VM system is the pageout scanner. It is installed at boot-time as a kernel process. Its task is to free up memory when the amount of free memory falls below a preset threshold.

Because Solaris uses the VM system to buffer files, a system with I/O activity will very quickly use any free memory available for buffering, which brings the amount of free memory down to the threshold. Of course, when that threshold is met, the pageout scanner is invoked.

This may seem a little strange, because the pageout scanner is being invoked even when there is ample memory in the system. Don't worry, it's normal, but it means that the pageout scanner plays a very important role in every system, even when there is no memory shortage. Please refer to "I/O via the VM System" on page 88 for more information.

Basis of Operation

The pageout scanner is based on the generic code which is present in Unix System V Release 4, and many other platforms. It uses a Not Recently Used (NRU) model which scans through the available pages looking for pages that have not been referenced since the last check.

The pageout daemon checks 4 times per second to see if free memory drops below `lotsfree`, a preset parameter that controls the pageout scanner. The scanner is also woken up when a memory request is made and the free list is below the threshold.

If memory is lower than this threshold, the scanner is invoked. The scanner is responsible for doing the real work in deciding which pages of memory to free.

Pageout Scanner

The scanner uses a two handed clock analogy, where the entire physical RAM is represented by the 12 hours on the clock face. There are two hands rotating around the clock at the same speed, one slightly ahead of the other. As the hands rotate, the front hand clears the referenced flag in the page. The backhand then checks the page as it cycles past some time later to see if the page has either been referenced since the front hand cleared the flags. If the page has not been referenced or modified, then it is a candidate for freeing, subject to one more check.

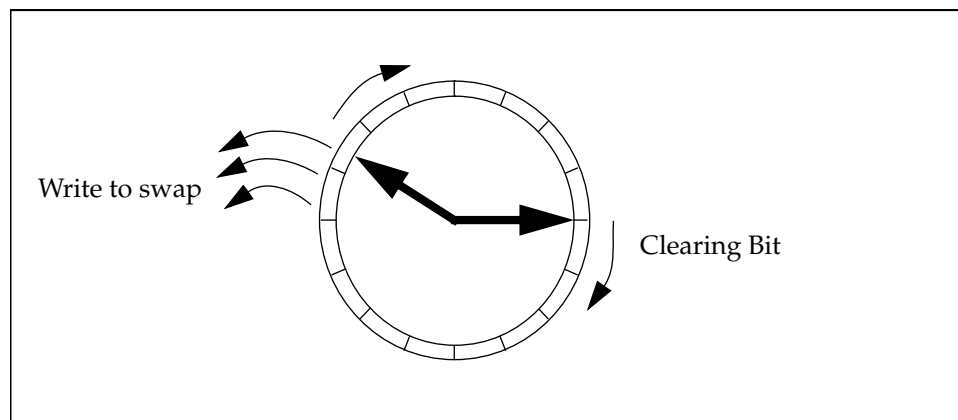


Figure 4-1 Pageout scanner

If the page has more than `po_share` mappings (i.e. it's shared by more than `po_share` processes), then it will be skipped. The variable `po_share` starts at 8, and each time round the scanner is decremented, unless the scan around the clock does not find any page to free, in which case `po_share` will be incremented. This whole process biases the scanner to pick on pages which don't look like shared library or executable pages.

Pageout Scanner Parameters

The parameters which control the clock hands do two things: they control the rate that the scanner scans through pages, and they control the time (or distance) between the front hand and the backhand. The distance between the backhand and the front hand is `handspreadpages`, and is in units of pages.

The scanner starts scanning when there are `lotsfree - deficit` pages free at a rate of `slowscan` pages per second. `deficit` is an internal parameter dynamically set by the kernel to indicate to the VM system how many pages are needed from recent activity.

The rate at which the scanner scans increases linearly between `lotsfree` and a minimum threshold, `minfree`.

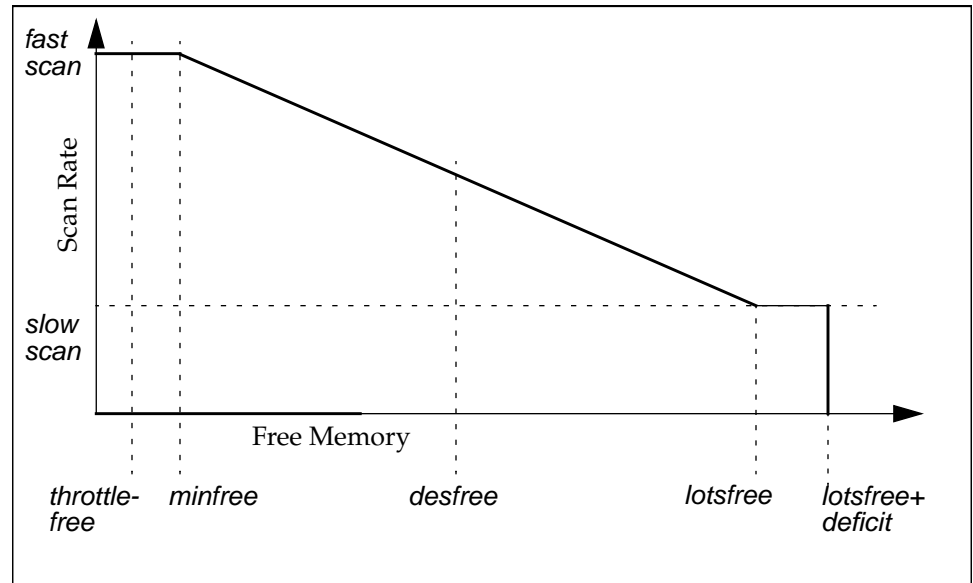


Figure 4-2 Pageout Scanner Parameters

If the amount of free memory falls below `desfree`, the scanner is run at every clock cycle, or by default 100 times a second. This helps the scanner try to keep at least `desfree` pages on the free list.

Once the scanner has started, it will remain running for `desscan` pages. The `desscan` parameter is normally set so to the number of pages the scanner needs to scan to accomplish the rate between `slowscan` and `fastscan` required.

There is another hook in other parts of the system so that if a large amount of memory is needed, `needfree` is set to reflect the amount required and `desscan` will run to scan `fastscan` pages.

Another parameter, `maxpgio`, limits the rate at which I/O is queued to the swap devices. It is set low to prevent saturation of the swap devices. The parameter defaults to 40 I/O's per second on sun4c architectures, and 60 I/O's per second on sun4d and sun4u architectures. The default setting is often inadequate for modern systems, and should be set to 100 times the number of swap spindles.

Because the pageout daemon also pages out I/O requests, this parameter also limits the rate at which pageout can write I/O's.

I/O requests are normally queued and written by user processes, and hence not subject to maxpgio; however when there is a memory shortage, the pageout scanner carries out a lot of the I/O writes, and maxpgio can sometimes be a limiting factor.

The following table describes the parameters which control the pageout process in the current Solaris and patch releases.:

Table 4-10 Pageout Parameters

Parameter	Description	Min	2.6 Default
lotsfree	If free memory falls below lotsfree then the pageout scanner starts 4 times/second, at a rate of slowscan pages/second	512K	1.5% of mem
desfree	If free memory falls below desfree, then the pageout scanner is started 100 times/second	min-free	lotsfree/2
minfree	The point at which scan rate is set to fastscan. The scan rate is a linear interpolation between lotsfree (scan rate=slowscan) and minfree.		desfree/2
throttlefree	The number at which point the page_create routines make the caller wait until free pages are available.	-	minfree
fastscan	The rate of pages scanned per second when free memory = minfree. Measured in pages.		Minimum of 64MB/s or 1/4 Mem. Size.
slowscan	The rate of pages scanned per second when free memory = lotsfree	-	fastscan/10
desscan	The number of pages that the scanner calculates it needs to scan each time the scanner wakes up to achieve the desired scan rate.		Dynamic

Table 4-10 Pageout Parameters

Parameter	Description	Min	2.6 Default
maxpgio	A throttle for the maximum number of pages per second that the swap device can handle	~60	60pgs/s
handspreadpages	The number of pages between the front hand clearing the reference bit and the backhand checking the reference bit.	1	fastscan
phbysmem	Total page count		
deficit	Added to boost lotsfree	0	lotsfree

There is also a CPU utilization clamp on the scan rate, to prevent the pageout daemon from using too much processor time. There are two internal limits that govern the desired and maximum CPU time that the scanner should use. In ideal conditions the scanner will try to use 4% of CPU to scan pages. If there is a critical memory shortage and the scan rate increases, it is capped so that it will occupy no more than 80% of a single CPU.

The Memory Scheduler

In addition to the pageout process, the CPU scheduler/dispatcher can swap out entire processes to conserve memory. This is a separate operation from pageout.

Swapping out a process involves removing all of a process's thread structures and private pages from memory, and setting flags in the process table to indicate that this process has been swapped out. This is an inexpensive way to conserve memory, but dramatically effects a processes performance, and hence is only used when paging fails to consistently free enough memory.

The memory scheduler is launched at boot time, and does nothing unless there is consistently less than `desfree` memory (30 second average). At this point the memory scheduler starts looking for processes which it can completely swap out. The memory scheduler will soft-swap out processes if there is a minimal shortage, or hard-swap (soft-swap and hard-swap are referenced in the following paragraphs) processes if there is a larger memory shortage.

Soft Swapping

Soft swapping occurs when the 30 second average for free memory is below `desfree`. At this point the memory scheduler will look for processes that have been inactive for at least `maxslp` seconds.

When the memory scheduler find a process that has been sleeping for `maxslp` seconds, it swaps out the thread structures for each thread, then pages out all of the private pages of memory for that process.

Hard Swapping

Hard swapping occurs when:

- There are at least two processes on the run queue waiting for CPU
- The average free memory over 30 seconds is consistently less than `desfree`
- There is excessive paging (determined to be true if `pageout+pagein > maxpgio`)

When hard swapping is invoked, a much more aggressive approach is used to find memory. The first step is that the kernel is requested to unload all modules and cache memory that is not currently active, followed by a sequential swapping out of processes until the desired amount of free memory is returned.

Parameters that affect the Memory Scheduler

Table 4-11 Memory Scheduler Parameters

Parameter	Affect on Memory Scheduler
desfree	If the average amount of free memory falls below desfree for 30 seconds, then the memory scheduler is invoked
maxslp	When soft-swapping, the memory scheduler starts swapping processes that have slept for at least maxslp seconds. The default for maxslp is 20 seconds and is tunable.
maxpgio	When the run queue is greater than two, free memory is below desfree and the paging rate is greater than maxpgio then hard swapping occurs, unloading kernel modules and process memory.

Traditional implementations of Unix use a separate memory and I/O system, each with their own behavior and functionality. As we have seen from the overview, the VM system in Solaris is implemented in a manner which provides a framework for both memory management and paged I/O.

Each component of the I/O system uses memory in some shape or form to complete I/O transactions. Memory is used to accelerate the operation by keeping recently used copies in memory for later use. This is often referred to as caching or buffering. Caching refers to storing data structures in memory, whilst buffering refers to storing complete buffers or pages of data in memory.

The major components of the I/O system are shown below, together with their memory association:

Table 5-1 I/O Memory Buffers and Caches

I/O Component	Type	Description
New Buffer Cache	Buffer	Used to buffer filesystem I/O so that repeat reads can often be satisfied from memory, and so that write clustering can occur. Buffer unit size is pages.
Directory Name Cache	Cache	Used by the filesystem infrastructure to lookup inode numbers based on their filesystem name.

Table 5-1 I/O Memory Buffers and Caches

I/O Component	Type	Description
Inode Cache	Cache	Used to keep attribute information about files in memory (e.g. size, access time etc..)
Old Buffer Cache	Buffer	Used to store blocks from the filesystem. Acts as a buffer between the Inode cache and the disk devices.
Stdio Buffer	Buffer	Used to buffer the fread/fwrite calls in the users process, before read() and write().

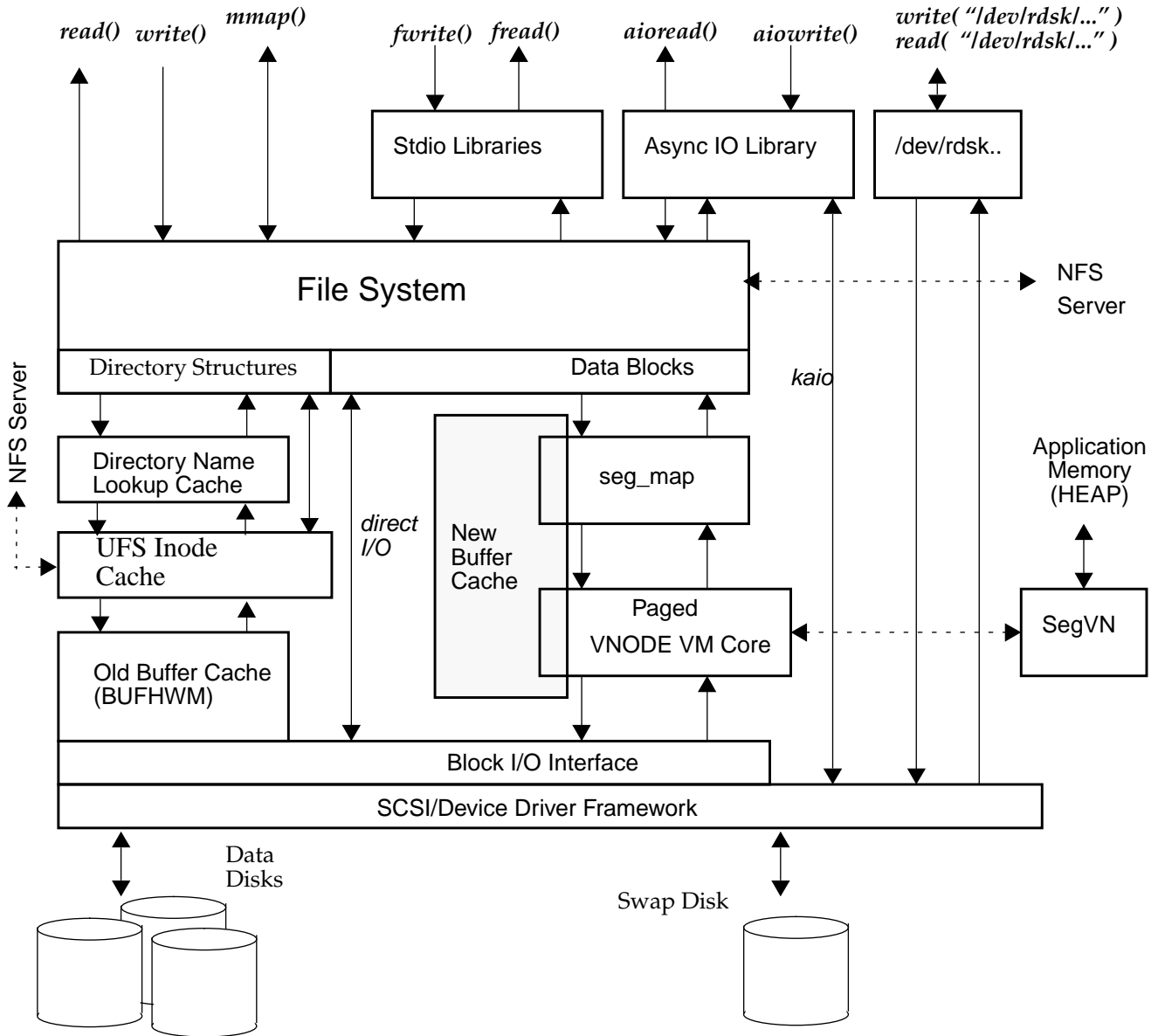


Figure 5-1 Solaris I/O Framework

Filesystem I/O

The filesystem interface provides a user and application view of the underlying storage. Solaris provides a filesystem independent interface, which allows many different types of filesystems to be plugged into the framework.

There are both regular filesystems and special filesystems. Regular filesystems provide an interface to buffered storage devices, whilst special filesystems provide access to pseudo devices. A good example of a special filesystem is the process file system, `/proc`.

Table 5-2 Filesystems in Solaris

Filesystem	Type	Device	Description
<i>ufs</i>	Regular	Disk	Unix Fast Filesystem, default in Solaris
<i>nfs</i>	Regular	Network	Network filesystem
<i>specfs</i>	Special	Device Drivers	Filesystem for the <code>/dev</code> devices
<i>procfs</i>	Special	Kernel	<code>/proc</code> filesystem representing processes
<i>pcfs</i>	Regular	Disk	MSDOS filesystem
<i>sockfs</i>	Special	Network	Filesystem of socket connections
<i>cachefs</i>	Special	Filesystem	Uses a local disk as cache for another fs
<i>tmpfs</i>	Special	Memory	Uses unused memory and swap
<i>autofs</i>	Special	Filesystem	Uses a dynamic layout to mount other fs
<i>vxfs</i>	Regular	Disk	Veritas File System, similar to <i>ufs</i>

The New Buffer Cache

I/O to the filesystem is typically generated by user application I/O, such as `read()` and `write()` system calls. Filesystem I/O is also generated from mapped files, which include executables, shared libraries and `mmap()`ed files.

Regular files in the filesystem are cached in the new buffer cache. Each time a file is read, an entire page size chunk is read from the disk and stored in a page of memory in exactly the same way as a page of user application memory. The page remains in memory until a memory shortage occurs, at which time the pageout scanner may remove this page and place it on the free list.

Writes to the filesystem are similar. The page of data containing the information which is written is updated, and then a page out operation is scheduled for that particular page, which eventually writes the modified page of data out to the filesystem. Synchronous writes are completed in the same manner, although the caller waits for the pageout operation to complete before returning.

It should be noted that the vmstat counters will show pagein's and pageout's for normal file I/O.

Directory Lookups

The meta data for files in filesystems are stored in the filesystem directory structure. The UFS filesystem (and VxFS) use Inodes to store the meta information about each file. All files in these filesystems have Inode numbers, which are linked in the filesystem to each files pages.

When a user wants to open a particular file, a filename is used to reference the file, rather than an Inode number. The filesystem must then look through the files in the current directory until it finds the desired filename to get the Inode number for the required file.

Because this operation is expensive, and can often involve reading many disk blocks, a cache is used to store the name/Inode pair once it is retrieved. This saves the caller from repeating the process, the next time the same file is opened.

This cache is known as the directory name lookup cache (DNLC), and is the first tier in the meta data caches in regular filesystems. The DNLC is a statically sized cache that stores the inode number, plus 31 characters of directory entry or pathname component. Entries longer than 31 bytes are not stored in the DNLC and hence cause additional directory scans.

The size of the DNLC is set at boot time via the `ncsize` parameter.

UFS Inode Cache

The UFS Inode Cache is used to store Inode information about each file. Because regular information such as size, access time, modification time all need to be accessed frequently, storing all of this data directly on disk without buffering would cause significant I/O. For example, each time a file is written to, its modification time must be updated.

All of this meta data is stored in a dynamically sized cache. The number of inactive entries in the inode cache are limited by the kernel parameter `ufsninode`. The data in the UFS Inode cache is obtained via the block I/O system, and uses the Old Buffer Cache to buffer the physical disk blocks on which the Inode data resides.

The Old Buffer Cache

Other Implementations of Unix use a separate buffer cache for the I/O system, which was statically sized at boot up, and needed to be continuously tuned to maintain an acceptable buffer hit rate. Added to this is the added complexity of ensuring that the buffer cache did not use too much memory and adversely affect application performance.

The new dynamic buffer cache is much easier to manage, and is largely self tuning.

The Old Buffer Cache is still implemented in Solaris, but is used to buffer block I/O for meta data. It has been enhanced so that it is semi-dynamic, which means that it can grow itself in size when needed, but cannot shrink. To stop the buffer from growing too large, a high water mark (`BUFHWM`) is used as a limit, which is preset at boot-time.

Free Behind and Read Ahead

To prevent saturation of the VM system, the UFS filesystem implements a free-behind policy when reading large sequential files.

File I/O is deemed to be sequential if the reads to the file follow consecutive pages, and the file is larger than 32k.

A simple example of free behind, is a small C program which reads sequentially through a file using the `read()` system call.

In the example, you can see that as the file is read, the number of page in's (pi) jump up to reflect the file I/O. Because this file is being read sequential, the amount of free memory never goes down far enough to invoke the scanner, which is indicated by zeros in the scan rate (sr) column.

```
# ls -l testfile
total 87760
-rwxr-xr-x  1 root    other    44933120 Jul 15 15:12 testfile

# ./readtest testfile&

# vmstat 3
procs          memory          page          disk          faults          cpu
r  b  w  swap  free  re  mf  pi  po  fr  de  sr  s0  --  --  --  in  sy  cs  us  sy  id
0  0  0  50404 3536  0  0  0  0  0  0  0  0  0  0  0  36  2  13  0  1  99
0  0  0  50404 3528  0  0  4  0  0  0  0  3  0  0  0  66  29  42  2  8  90
0  0  0  50404 3516  0  0  0  0  0  0  0  1  0  0  0  58  29  42  1  7  91
0  0  0  50404 3516  0  0  66  0  0  0  0  3  0  0  0  73  23  46  1  10 89
0  0  0  50512 2884  0  0  321 0  0  0  0  28  0  0  0  215 66 127 1  77 22
0  0  0  50512 1272  0  0  341 0  0  0  0  25  0  0  0  139 58 115 0  82 18
0  0  0  50512 1236  0  0  317 0  0  0  0  19  0  0  0  119 57 120 0  86 14
0  0  0  50512 1276  0  0  322 0  0  0  0  14  0  0  0  100 55 117 0  87 13
0  0  0  50440 1356  0  0  82  0  0  0  0  4  0  0  0  56  22  42  0  23 77
```

The same test program can be rerun, but with a random seek before each read to simulate random I/O. This disables the free-behind algorithm and continues to consume pages of virtual memory.

The test program has been renamed `readtest`, for random read in this case.

```
# ./rreadtest testfile&

# vmstat 3
procs          memory          page          disk          faults          cpu
r  b  w  swap  free  re  mf  pi  po  fr  de  sr  s0  --  --  --  in  sy  cs  us  sy  id
0  0  0  50436 2064  5  0  81  0  0  0  0  15  0  0  0  168 361  69  1  25 74
0  0  0  50508 1336 14  0 222  0  0  0  0  35  0  0  0  210 902 130  2  51 47
0  0  0  50508  648 10  0 177  0  0  0  0  27  0  0  0  168 850 121  1  60 39
0  0  0  50508  584 29  57 88 109 0  0  0  14  0  0  0  108 5284 120  7  72 20
0  0  0  50508  484  0  50 249 96 0  0  0  18 93  0  0  0  199 542 124  0  50 50
0  0  0  50508  492  0  41 260 70 0  0  0  56 34  0  0  0  209 649 128  1  49 50
0  0  0  50508  472  0  58 253 116 0  0  0  45 33  0  0  0  198 566 122  1  46 53
```

In this example, pages are paged in and free memory drops to the point where the system starts scanning looking for pages that it can free. Note that at no time is the system actually short of memory, it's just that all of the free pages have been used by the buffer cache and the scanner is invoked to free some memory.

Readahead is a similar concept. Read ahead will launch a read for the next block when reading sequential data.

The fsflush process

The `fsflush` process has a similar goal to the pageout daemon, in that it scans through the page list looking for suitable pages. It however does not free pages, it merely writes dirty pages out to disk.

The `fsflush` process is launched every by default every 5 seconds, and looks for pages that have been modified (the modified bit is set in the PAGE structure) more than 30 seconds ago. If a page has been modified, then a pageout is scheduled for that page, but without the free flag so the PAGE remains in memory.

The `fsflush` daemon will flush both data and inode pages by default. There are several parameters that affect the behavior of `fsflush`.

Table 5-3 Parameters that affect `fsflush`

Parameter	Description	Min	2.6 Default
<code>tune.fsflushr</code>	The number of seconds between <code>fsflush</code> scans.	1	5
<code>autoup</code>	Pages older than <code>autoup</code> in seconds are written to disk.	1	30
<code>doiflush</code>	By default <code>fsflush</code> will flush both inode and data pages. Set to zero to suppress inode updates.	0	1
<code>dopageflush</code>	Set to zero to suppress page flushes.	0	1

The `fsflush` process will also write all pages that have been scheduled for delayed write.

Direct I/O

A new feature added to Solaris 2.6 is direct I/O. This allows reads and writes to files in a regular filesystem to bypass the paged vnode buffer cache.

If buffers are used to accelerate I/O speed, then you might ask what the benefit of this is. In many cases direct I/O would mean a dramatic drop in performance, because each read must read from the disk, even if read two or three times.

Direct I/O is beneficial when large amounts of data which far exceed the size of the memory in the system are being read, or the data is already being buffered elsewhere.

A good example of this is Oracle with decision support databases. Oracle uses a large shared memory segment to cache database table data. Putting Oracle's cache on top of Solaris's buffer cache just means additional overhead, so often Oracle is installed with raw partitions to avoid this double caching effect.

Direct I/O allows applications like Oracle to use regular filesystems, but without the additional overhead of double caching.

Direct I/O is implemented by mounting the filesystem with a special flag, or using `fadvise()` in the file to disable caching:

```
# mount -o forcedirectio /dev/dsk/c0t0d0s6 /u1
```

Direct I/O will only bypass the buffer cache if all of the following are true:-

- The file is not `mmap()`ed
- The file is not on a SDS logging filesystem
- The file does not have holes
- The read/write is sector aligned (512byte)

RAW Devices

The raw disk devices in `/dev/rdisk` are sometimes used for direct access to storage devices for the same reason as direct I/O.

All reads and writes to raw devices are completely unbuffered.

Asynchronous I/O

Databases often use a modern method of queuing I/O requests to the devices, known as Asynchronous I/O. Using this method, multiple I/O's can be requested at once, with an asynchronous notification via a signal when the I/O has completed.

The I/O calls are made via libaio functions aioread, aiowrite etc.

Solaris has an additional feature - Kernel Asynchronous I/O, which allows libaio to pass the I/O requests directly to the kernel and device drivers of the storage device.

Kernel Asynchronous I/O is scheduled at run-time if the device driver that the I/O is scheduled for supports the asynchronous entry points, and the data is on a non-buffered (e.g. raw) device.

If the device does not support asynchronous entry points then the I/O requests are handled by a user-level thread.

When kernel Asynchronous I/O is used there is no buffering in the VM system of any data.



THE NETWORK IS THE COMPUTER™

Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043
415 960-1300

For U.S. Sales Office Locations, call: 800 821-4643

In other countries, call:
Corporate Headquarters: 415-960-1300
Intercontinental Sales: 415 688-9000