Memory and Thread Placement Optimization Developer's Guide

Part No: E35301–01
October 2012

U.S. GOVERNMENT END USERS. Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

121010@25097

Contents

Preface

The *Memory and Thread Placement Optimization Developer's Guide* provides information on locality groups and the technologies that are available to optimize the use of computing resources in the Oracle Solaris operating system.

Who Should Use This Book

This book is intended for use by system administrators, performance engineers, systems programmers, and support engineers, and developers who are writing applications in an environment with multiple CPUs and a non-uniform memory architecture. The programming interfaces and tools that are described in this book give the developer control over the system's behavior and resource allocation.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Typographic Conventions

The following table describes the typographic conventions that are used in this book.

TABLE P–1 Typographic Conventions

Typeface	Description	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name%` **`su`** `Password:`

TABLE P–1 Typographic Conventions *(Continued)*

Typeface	Description	Example
aabbcc123	Placeholder: replace with a real name or value	The command to remove a file is `rm` *filename*.
AaBbCc123	Book titles, new terms, and terms to be emphasized	Read Chapter 6 in the *User's Guide*. A *cache* is a copy that is stored locally. Do *not* save the file. **Note:** Some emphasized items appear bold online.

Shell Prompts in Command Examples

The following table shows UNIX system prompts and superuser prompts for shells that are included in the Oracle Solaris OS. In command examples, the shell prompt indicates whether the command should be executed by a regular user or a user with privileges.

TABLE P–2 Shell Prompts

Shell	Prompt
Bash shell, Korn shell, and Bourne shell	`$`
Bash shell, Korn shell, and Bourne shell for superuser	`#`
C shell	`machine_name%`
C shell for superuser	`machine_name#`

CHAPTER 1

Overview of Locality Groups

Locality Groups Overview

Shared memory multiprocessor computers contain multiple CPUs. Each CPU can access all of the memory in the machine. In some shared memory multiprocessors, the memory architecture enables each CPU to access some areas of memory more quickly than other areas.

When a machine with such a memory architecture runs the Oracle Solaris software, providing information to the kernel about the shortest access times between a given CPU and a given area of memory can improve the system's performance. The locality group (lgroup) abstraction has been introduced to handle this information. The lgroup abstraction is part of the Memory Placement Optimization (MPO) feature.

An lgroup is a set of CPU–like and memory–like devices in which each CPU in the set can access any memory in that set within a bounded latency interval. The value of the latency interval represents the least common latency between all the CPUs and all the memory in that lgroup. The latency bound that defines an lgroup does not restrict the maximum latency between members of that lgroup. The value of the latency bound is the shortest latency that is common to all possible CPU-memory pairs in the group.

Lgroups are hierarchical. The lgroup hierarchy is a Directed Acyclic Graph (DAG) and is similar to a tree, except that an lgroup might have more than one parent. The root lgroup contains all the resources in the system and can include child lgroups. Furthermore, the root lgroup can be characterized as having the highest latency value of all the lgroups in the system. All of its child lgroups will have lower latency values. The lgroups closer to the root have a higher latency while lgroups closer to leaves have lower latency.

A computer in which all the CPUs can access all the memory in the same amount of time can be represented with a single lgroup (see Figure 1–1). A computer in which some of the CPUs can access some areas of memory in a shorter time than other areas can be represented by using multiple lgroups (see Figure 1–2).

FIGURE 1–1 Single Locality Group Schematic

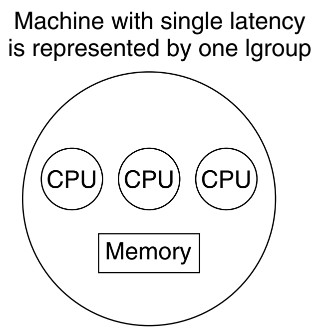
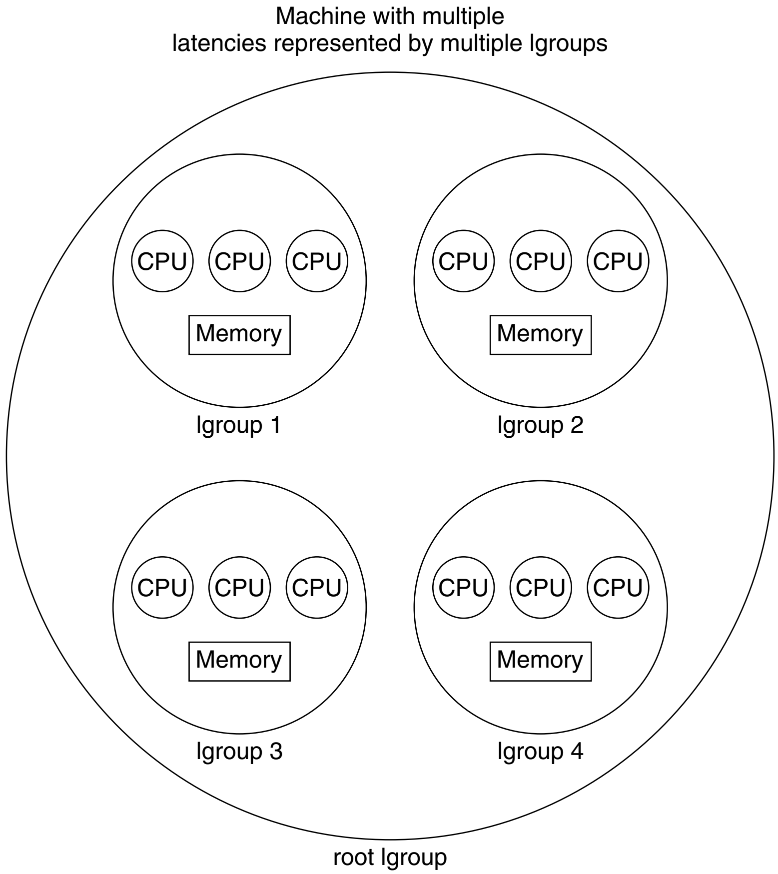


FIGURE 1–2 Multiple Locality Groups Schematic



The organization of the lgroup hierarchy simplifies the task of finding the nearest resources in the system. Each thread is assigned a home lgroup upon creation. The operating system attempts to allocate resources for the thread from the thread's home lgroup by default. For example, the Oracle Solaris kernel attempts to schedule a thread to run on the CPUs in the thread's home lgroup and allocate the thread's memory in the thread's home lgroup by default. If the desired resources are not available from the thread's home lgroup, the kernel can traverse the lgroup hierarchy to find the next nearest resources from parents of the home lgroup. If the desired resources are not available in the home lgroup's parents, the kernel continues to traverse the lgroup hierarchy to the successive ancestor lgroups of the home lgroup. The root lgroup is the ultimate ancestor of all other lgroups in a machine and contains all of the machine's resources.

The Memory Placement Optimization (MPO) tools enable developers to tune the performance of the MPO features in cases where the default MPO behaviors do not yield the desired performance.

The lgroup APIs export the lgroup abstraction for applications to use for observability and performance tuning. A new library, called `liblgrp`, contains the new APIs. Applications can use the APIs to perform the following tasks:

- Traverse the group hierarchy
- Discover the contents and characteristics of a given lgroup
- Affect the thread and memory placement on lgroups

MPO Observability Tools

The MPO tools help developers to answer questions about system configuration and balance or placement. The tools also provide the basic information and mechanisms that developers need in order to determine whether MPO is successful and to diagnose problems related to MPO.

To determine the degree of success that MPO has in providing useful locality assignments and acceptable performance, it is important to know a given thread's affinities for lgroups, including its home lgroup, and where the thread's memory is allocated.

The MPO observability tools provide developers with the ability to determine the actions taken by the system. The MPO thread and memory placement tools enable developers to act on that information. Developers can also use the `dtrace(1M)` tool to gain further insights into the system's behavior.

CHAPTER 2

MPO Observability Tools

This chapter describes the tools that are available to use the MPO functionality that is available in the Oracle Solaris operating system.

This chapter discusses the following topics:

- "The pmadvise utility" on page 11 describes the tool that applies rules that define the memory use of a process.
- "Using the madv.so.1 Shared Object" on page 13 describes the madv.so.1 shared object and how to use it to configure virtual memory advice.
- "The plgrp tool" on page 15 describes the tool that can display and set a thread's affinity for a locality group.
- "The lgrpinfo Tool" on page 17 prints information about the lgroup hierarchy, contents, and characteristics.
- "The Solaris::lgrp Module" on page 19 describes a Perl interface to the locality group API that is described in Chapter 3, "Locality Group APIs."

The pmadvise utility

The `pmadvise` utility applies rules to a process that define how that process uses memory. The `pmadvise` utility applies the rules, called *advice*, to the process with the `madvise`(3C) tool. This tool can apply advice to a specific subrange of locations in memory at a specific time. By contrast, the `madv.so.1`(1) tool applies the advice throughout the execution of the target program to all segments of a specified type.

The `pmadvise` utility has the following options:

`-f` This option takes control of the target process. This option overrides the control of any other process. See the `proc`(1) manual page.

`-o` This option specifies the advice to apply to the target process. Specify the advice in this format:

```
private=advice
shared=advice
heap=advice
stack=advice
address:length=advice
```

The value of the *advice* term can be one of the following:

```
normal
random
sequential
willneed
dontneed
free
access_lwp
access_many
access_default
```

You can specify an address and length to specify the subrange where the advice applies. Specify the address in hexadecimal notation and the length in bytes.

If you do not specify the length and the starting address refers to the start of a segment, the `pmadvise` utility applies the advice to that segment. You can qualify the length by adding the letters K, M, G, T, P, or E to specify kilobytes, megabytes, gigabytes, terabytes, or exabytes, respectively.

`-v` This option prints verbose output in the style of the `pmap(1)` tool that shows the value and locations of the advice rules currently in force.

The `pmadvise` tool attempts to process all legal options. When the pmadvise tool attempts to process an option that specifies an illegal address range, the tool prints an error message and skips that option. When the `pmadvise` tool finds a syntax error, it quits without processing any options and prints a usage message.

When the advice for a specific region conflicts with the advice for a more general region, the advice for the more specific region takes precedence. Advice that specifies a particular address range has precedence over advice for the heap and stack regions, and advice for the heap and stack regions has precedence over advice for private and shared memory.

The advice rules in each of the following groups are mutually exclusive from other advice rules within the same group:

```
MADV_NORMAL, MADV_RANDOM, MADV_SEQUENTIAL
MADV_WILLNEED, MADV_DONTNEED, MADV_FREE
MADV_ACCESS_DEFAULT, MADV_ACCESS_LWP, MADV_ACCESS_MANY
```

Using the madv.so.1 Shared Object

The `madv.so.1` shared object enables the selective configuration of virtual memory advice for launched processes and their descendants. To use the shared object, the following string must be present in the environment:

```
LD_PRELOAD=$LD_PRELOAD:madv.so.1
```

The `madv.so.1` shared object applies memory advice as specified by the value of the `MADV` environment variable. The `MADV` environment variable specifies the virtual memory advice to use for all heap, shared memory, and mmap regions in the process address space. This advice is applied to all created processes. The following values of the `MADV` environment variable affect resource allocation among lgroups:

`access_default` This value resets the kernel's expected access pattern to the default.

`access_lwp` This value advises the kernel that the next LWP to touch an address range is the LWP that will access that range the most. The kernel allocates the memory and other resources for this range and the LWP accordingly.

`access_many` This value advises the kernel that many processes or LWPs will access memory randomly across the system. The kernel allocates the memory and other resources accordingly.

The value of the `MADVCFGFILE` environment variable is the name of a text file that contains one or more memory advice configuration entries in the form *exec-name*:*advice-opts*.

The value of *exec-name* is the name of an application or executable. The value of *exec-name* can be a full pathname, a base name, or a pattern string.

The value of *advice-opts* is of the form *region*=*advice*. The values of *advice* are the same as the values for the `MADV` environment variable. Replace *region* with any of the following legal values:

`madv` Advice applies to all heap, shared memory, and `mmap(2)` regions in the process address space.

`heap` The heap is defined to be the `brk(2)` area. Advice applies to the existing heap and to any additional heap memory allocated in the future.

`shm` Advice applies to shared memory segments. See `shmat(2)` for more information on shared memory operations.

`ism` Advice applies to shared memory segments that are using the `SHM_SHARE_MMU` flag. The `ism` option takes precedence over `shm`.

`dsm` Advice applies to shared memory segments that are using the `SHM_PAGEABLE` flag. The `dsm` option takes precedence over `shm`.

`mapshared` — Advice applies to mappings established by the `mmap()` system call using the `MAP_SHARED` flag.

`mapprivate` — Advice applies to mappings established by the `mmap()` system call using the `MAP_PRIVATE` flag.

`mapanon` — Advice applies to mappings established by the `mmap()` system call using the `MAP_ANON` flag. The `mapanon` option takes precedence when multiple options apply.

The value of the `MADVERRFILE` environment variable is the name of the path where error messages are logged. In the absence of a `MADVERRFILE` location, the `madv.so.1` shared object logs errors by using `syslog`(3C) with a `LOG_ERR` as the severity level and `LOG_USER` as the facility descriptor.

Memory advice is inherited. A child process has the same advice as its parent. The advice is set back to the system default advice after a call to `exec`(2) unless a different level of advice is configured using the `madv.so.1` shared object. Advice is only applied to `mmap()` regions explicitly created by the user program. Regions established by the run-time linker or by system libraries that make direct system calls are not affected.

madv.so.1 Usage Examples

The following examples illustrate specific aspects of the `madv.so.1` shared object.

EXAMPLE 2–1 Setting Advice for a Set of Applications

This configuration applies advice to all ISM segments for applications with exec names that begin with `foo`.

```
$ LD_PRELOAD=$LD_PRELOAD:madv.so.1
$ MADVCFGFILE=madvcfg
$ export LD_PRELOAD MADVCFGFILE
$ cat $MADVCFGFILE
        foo*:ism=access_lwp
```

EXAMPLE 2–2 Excluding a Set of Applications From Advice

This configuration sets advice for all applications with the exception of `ls`.

```
$ LD_PRELOAD=$LD_PRELOAD:madv.so.1
$ MADV=access_many
$ MADVCFGFILE=madvcfg
$ export LD_PRELOAD MADV MADVCFGFILE
$ cat $MADVCFGFILE
        ls:
```

EXAMPLE 2–3 Pattern Matching in a Configuration File

Because the configuration specified in `MADVCFGFILE` takes precedence over the value set in `MADV`, specifying * as the *exec-name* of the last configuration entry is equivalent to setting `MADV`. This example is equivalent to the previous example.

```
$ LD_PRELOAD=$LD_PRELOAD:madv.so.1
$ MADVCFGFILE=madvcfg
$ export LD_PRELOAD MADVCFGFILE
$ cat $MADVCFGFILE
        ls:
        *:madv=access_many
```

EXAMPLE 2–4 Advice for Multiple Regions

This configuration applies one type of advice for `mmap()` regions and different advice for heap and shared memory regions for applications whose `exec()` names begin with `foo`.

```
$ LD_PRELOAD=$LD_PRELOAD:madv.so.1
$ MADVCFGFILE=madvcfg
$ export LD_PRELOAD MADVCFGFILE
$ cat $MADVCFGFILE
        foo*:madv=access_many,heap=sequential,shm=access_lwp
```

The plgrp tool

The `plgrp` utility can display or set the home lgroup and lgroup affinities for one or more processes, threads, or lightweight processes (LWPs). The system assigns a home lgroup to each thread on creation. When the system allocates a CPU or memory resource to a thread, it searches the lgroup hierarchy from the thread's home lgroup for the nearest available resources to the thread's home.

The system chooses a home lgroup for each thread. The thread's affinity for its home lgroup is initially set to none, or no affinity. When a thread sets an affinity for an lgroup in its processor set that is higher than the thread's affinity for its home lgroup, the system moves the thread to that lgroup. The system does not move threads that are bound to a CPU. The system rehomes a thread to the lgroup in its processor set that has the highest affinity when the thread's affinity for its home lgroup is removed (set to none).

For a full description of the different levels of lgroup affinity and their semantics, see the `lgrp_affinity_set`(3LGRP) manual page.

The `plgrp` tool supports the following options:

`-a` *lgroup list*	This option displays the affinities of the processes or threads that you specify for the lgroups in the list.

`-A`*lgroup list*/`none|weak|strong[,...]`	This option sets the affinity of the processes or threads that you specify for the lgroups in the list. You can use a comma separated list of *lgroup*/*affinity* assignments to set several affinities at once.
`-F`	This option takes control of the target process. This option overrides the control of any other process. See the `proc(1)` manual page.
`-h`	This option returns the home lgroup of the processes or threads that you specify. This is the default behavior of the `plgrp` tool when you do not specify any options.
`-H` *lgroup list*	This option sets the home lgroup of the processes or threads that you specify. This option sets a strong affinity for the listed lgroup. If you specify more than one lgroup, the `plgrp` utility will attempt to home the threads to the lgroups in a round robin fashion.

Specifying Lgroups

The value of the *lgroup list* variable is a comma separated list of one or more of the following attributes:

- lgroup ID
- Range of lgroup IDs, specified as *start lgroup ID-end lgroup ID*
- `all`
- `root`
- `leaves`

The `all` keyword represents all of the lgroup IDs in the system. The `root` keyword represents the ID of the root lgroup. The `leaves` keyword represents the IDs of all of the leaf lgroups. A leaf lgroup is an lgroup that does not have any children.

Specifying Process and Thread Arguments

The `plgrp` utility takes one or more space-separated processes or threads as arguments. You can specify processes and threads in a the same syntax that the `proc(1)` tools use. You can specify a process ID as an integer, with the syntax *pid* or `/proc/`*pid*. You can use shell expansions with the `/proc/`*pid* syntax. When you give a process ID alone, the arguments to the `plgrp` utility include all of the threads of that process.

You can specify a thread explicitly by specifying the process ID and thread ID with the syntax *pid*/*lwpid*. You can specify multiple threads of a process by defining ranges with can be selected at once by using the `-` character to define a range, or with a comma-separated list. To specify threads 1, 2, 7, 8, and 9 of a process whose process ID is *pid*, use the syntax *pid*/`1,2,7-9`.

The lgrpinfo Tool

The `lgrpinfo` tool prints information about the lgroup hierarchy, contents, and characteristics. The `lgrpinfo` tool is a Perl script that requires the `Solaris::Lgrp` module. This tool uses the `liblgrp`(3LIB) API to get the information from the system and displays it in the human-readable form.

The `lgrpinfo` tool prints general information about all of the lgroups in the system when you call it without any arguments. When you pass lgroup IDs to the `lgrpinfo` tool at the command line, the tool returns information about the lgroups that you specify. You can specify lgroups with their lgroup IDs or with one of the following keywords:

`all` — This keyword specifies all lgroups and is the default behavior.

`root` — This keyword specifies the root lgroup.

`leaves` — This keyword specifies all of the leaf lgroups. A leaf lgroup is an lgroup that has no children in the lgroup hierarchy.

`intermediate` — This keyword specifies all of the intermediate lgroups. An intermediate lgroup is an lgroup that has a parent and children.

When the `lgrpinfo` tool receives an invalid lgroup ID, the tool prints a message with the invalid ID and continues processing any other lgroups that are passed in the command line. When the `lgrpinfo` tool finds no valid lgroups in the arguments, it exits with a status of `2`.

Options for the lgrpinfo Tool

When you call the `lgrpinfo` tool without any arguments, the tool's behavior is equivalent to using the options `-celmrt all`. The valid options for the `lgrpinfo` tool are:

`-a` — This option prints the topology, CPU, memory, load and latency information for the specified lgroup IDs. This option combines the behaviors of the `-tcemrlL` options, unless you also specify the `-T` option. When you specify the `-T` option, the behavior of the `-a` option does not include the behavior of the `-t` option.

`-c` — This option prints the CPU information.

- `-C` This option replaces each lgroup in the list with its children. You cannot combine this option with the `-P` or `-T` options. When you do not specify any arguments, the tool applies this option to all lgroups.
- `-e` This option prints lgroup load averages for leaf lgroups.
- `-G` This option prints the OS view of the lgroup hierarchy. The tool's default behavior displays the caller's view of the lgroup hierarchy. The caller's view only includes the resources that the caller can use. See the `lgrp_init`(3LGRP) manual page for more details on the OS and caller's view.
- `-h` This option prints the help message for the tool.
- `-I` This option prints only IDs that match the IDs you specify. You can combine this option with the `-c`, `-G`, `-C`, or `-P` options. When you specify the `-c` option, the tool prints the list of CPUs that are in all of the matching lgroups. When you do not specify the `-c` option, the tool displays the IDs for the matching lgroups. When you do not specify any arguments, the tool applies this option to all lgroups.
- `-l` This option prints information about lgroup latencies. The latency value given for each lgroup is defined by the operating system and is platform-specific. It can only be used for relative comparison of lgroups on the running system. It does not necessarily represent the actual latency between hardware devices and may not be applicable across platforms.
- `-L` This option prints the lgroup latency table. This table shows the relative latency from each lgroup to each of the other lgroups.
- `-m` This option prints memory information. The tool reports memory sizes in the units that give a size result in the integer range from 0 to 1023. You can override this behavior by using the `-u` option. The tool will only display fractional results for values smaller than 10.
- `-P` This option replaces each lgroup in the list with its parent or parents. You cannot combine this option with the `-C` or `-T` options. When you do not specify any arguments, the tool applies this option to all lgroups.
- `-r` This option prints information about lgroup resources. When you specify the `-T` option, the tool displays information about the resources of the intermediate lgroups only.
- `-t` This option prints information about lgroup topology.
- `-T` This option prints the lgroup topology of a system graphically, as a tree. You can only use this option with the `-a`, `-c`, `-e`, `-G`, `-l`, `-L`, `-m`, `-r`, and `-u` options. To restrict the output to intermediate lgroups, use the `-r` option. Omit the `-t` option when you combine the `-T` option with the `-a` option. This option does not print information for the root lgroup unless it is the only lgroup.

-u*units* This option specifies memory units. The value of the *units* argument can be b, k, m, g, t, p, or e for bytes, kilobytes, megabytes, gigabytes, terabytes, petabytes, or exabytes, respectively.

The Solaris::lgrp Module

This Perl module provides a Perl interface to the lgroup APIs that are in liblgrp. This interface provides a way to traverse the lgroup hierarchy, discover its contents and characteristics, and set a thread's affinity for an lgroup. The module gives access to various constants and functions defined in the lgrp_user.h header file. The module provides the procedural interface and the object interface to the library.

The default behavior of this module does not export anything. You can use the following tags to selectively import the constants and functions that are defined in this module:

:LGRP_CONSTANTS	LGRP_AFF_NONE, LGRP_AFF_STRONG, LGRP_AFF_WEAK, LGRP_CONTENT_DIRECT, LGRP_CONTENT_HIERARCHY, LGRP_MEM_SZ_FREE, LGRP_MEM_SZ_INSTALLED, LGRP_VER_CURRENT, LGRP_VER_NONE, LGRP_VIEW_CALLER, LGRP_VIEW_OS, LGRP_NONE, LGRP_RSRC_CPU, LGRP_RSRC_MEM, LGRP_CONTENT_ALL, LGRP_LAT_CPU_TO_MEM
:PROC_CONSTANTS	P_PID, P_LWPID, P_MYID
:CONSTANTS	:LGRP_CONSTANTS, :PROC_CONSTANTS
:FUNCTIONS	lgrp_affinity_get(), lgrp_affinity_set(), lgrp_children(), lgrp_cookie_stale(), lgrp_cpus(), lgrp_fini(), lgrp_home(), lgrp_init(), lgrp_latency(), lgrp_latency_cookie(), lgrp_mem_size(), lgrp_nlgrps(), lgrp_parents(), lgrp_root(), lgrp_version(), lgrp_view(), lgrp_resources(), lgrp_lgrps(), lgrp_leaves(), lgrp_isleaf(), lgrp_lgrps(), lgrp_leaves().
:ALL()	:CONSTANTS(), :FUNCTIONS()

The Perl module has the following methods:

- new()
- cookie()
- stale()
- view()
- root()
- children()
- parents()
- nlgrps()
- mem_size()
- cpus()

- `isleaf()`
- `resources()`
- `version()`
- `home()`
- `affinity_get()`
- `affinity_set()`
- `lgrps()`
- `leaves()`
- `latency()`

You can export constants with the `:CONSTANTS` or `:ALL` tags. You can use any of the constants in the following list in Perl programs.

- `LGRP_NONE`
- `LGRP_VER_CURRENT`
- `LGRP_VER_NONE`
- `LGRP_VIEW_CALLER`
- `LGRP_VIEW_OS`
- `LGRP_AFF_NONE`
- `LGRP_AFF_STRONG`
- `LGRP_AFF_WEAK`
- `LGRP_CONTENT_DIRECT`
- `LGRP_CONTENT_HIERARCHY`
- `LGRP_MEM_SZ_FREE`
- `LGRP_MEM_SZ_INSTALLED`
- `LGRP_RSRC_CPU`
- `LGRP_RSRC_MEM`
- `LGRP_CONTENT_ALL`
- `LGRP_LAT_CPU_TO_MEM`
- `P_PID`
- `P_LWPID`
- `P_MYID`

When an underlying library function fails, the functions in this module return either `undef` or an empty list. The module can use the following error codes:

`EINVAL` The value supplied is not valid.

`ENOMEM` There was not enough system memory to complete an operation.

`ESRCH` The specified process or thread was not found.

`EPERM` The effective user of the calling process does not have the appropriate privileges, and its real or effective user ID does not match the real or effective user ID of one of the threads.

Functions in the Solaris::lgrp Module

`lgrp_init([LGRP_VIEW_CALLER | LGRP_VIEW_OS])`
This function initializes the lgroup interface and takes a snapshot of the lgroup hierarchy with the given view. Given the view, the `lgrp_init()` function returns a cookie that represents this snapshot of the lgroup hierarchy. Use this cookie with the other routines in the lgroup interface that require the lgroup hierarchy. Call the `lgrp_fini()` function with this cookie when the system no longer needs the hierarchy snapshot. Unlike the `lgrp_init`(3LGRP) function, this function assumes a value of `LGRP_VIEW_OS` as the default if the system provides no view. This function returns a cookie upon successful completion. If the `lgrp_init` function does not complete successfully, it returns a value of `undef` and sets `$!` to indicate the error. See the man page for the `lgrp_init`(3LGRP) function for more information.

`lgrp_fini($cookie)`
This function takes a cookie, frees the snapshot of the lgroup hierarchy that the `lgrp_init()` function created, and cleans up anything else that the `lgrp_init()` function set up. After calling this function, do not use the cookie that the lgroup interface returns. This function returns `1` upon successful completion. If the `lgrp_fini` function does not complete successfully, it returns a value of `undef` and sets `$!` to indicate the error. See the man page for the `lgrp_fini`(3LGRP) function for more information.

`lgrp_view($cookie)`
This function takes a cookie that represents the snapshot of the lgroup hierarchy and returns the snapshot's view of the lgroup hierarchy. If the given view is `LGRP_VIEW_CALLER`, the snapshot contains only the resources that are available to the caller. When the view is `LGRP_VIEW_OS`, the snapshot contains the resources that are available to the operating system. This function returns the view for the snapshot of the lgroup hierarchy that is represented by the given cookie upon successful completion. If the `lgrp_view` function does not complete successfully, it returns a value of `undef` and sets `$!` to indicate the error. See the man page for the `lgrp_view`(3LGRP) function for more information.

`lgrp_home($idtype, $id)`
This function returns the home lgroup for the given process or thread. To specify a process, give the `$idtype` argument the value `P_PID` and give the `$id` argument the value of the process id. To specify a thread, give the `$idtype` argument the value `P_LWPID` and give the `$id` argument the value of the thread's LWP id. To specify the current process or thread, give the `$id` argument the value `P_MYID`. This function returns the id of the home lgroup of the specified process or thread upon successful completion. If the `lgrp_home` function does not complete successfully, it returns a value of `undef` and sets `$!` to indicate the error. See the man page for the `lgrp_home`(3LGRP) function for more information.

`lgrp_cookie_stale($cookie)`
This function returns the staleness status of the specified cookie upon successful completion. If the `lgrp_cookie_stale` function does not complete successfully, it returns a value of

undef and sets $! to indicate the error. This function fails and returns EINVAL if the cookie is not valid. See the man page for the lgrp_cookie_stale(3LGRP) function for more information.

lgrp_cpus($cookie, $lgrp, $context)
: This function takes a cookie that represents a snapshot of the lgroup hierarchy and returns the list of CPUs that are in the lgroup that is specified by the $lgrp argument. Give the $context argument the value LGRP_CONTENT_HIERARCHY to make the lgrp_cpus() function return the list of all the CPUs that are in the specified lgroup, including child lgroups. Give the $context() argument the value LGRP_CONTENT_DIRECT to make the lgrp_cpus() function return the list of CPUs that are directly contained in the specified lgroup. This function returns the number of CPUs that are in the specified lgroup when you call it in a scalar context. If the lgrp_cpus function does not complete successfully when you call it in a scalar context, it returns a value of undef and sets $! to indicate the error. If the lgrp_cpus function does not complete successfully when you call it in a list context, it returns the empty list and sets $! to indicate the error. See the man page for the lgrp_cpus(3LGRP) function for more information.

lgrp_children($cookie, $lgrp)
: This function takes a cookie that represents a snapshot of the lgroup hierarchy and returns the list of lgroups that are children of the specified lgroup. When called in scalar context, the lgrp_children() function returns the number of children lgroups for the specified lgroup when you call it in a scalar context. If the lgrp_children function does not complete successfully when you call it in a scalar context, it returns a value of undef and sets $! to indicate the error. If the lgrp_children function does not complete successfully when you call it in a list context, it returns the empty list and sets $! to indicate the error. See the man page for the lgrp_children(3LGRP) function for more information.

lgrp_parents($cookie, $lgrp)
: This function takes a cookie that represents a snapshot of the lgroup hierarchy and returns the list of parent groups of the specified lgroup. When called in scalar context, the lgrp_parents() function returns the number of parent lgroups for the specified lgroup when you call it in a scalar context. If the lgrp_parents function does not complete successfully when you call it in a scalar context, it returns a value of undef and sets $! to indicate the error. If the lgrp_parents function does not complete successfully when you call it in a list context, it returns the empty list and sets $! to indicate the error. See the man page for the lgrp_parents(3LGRP) function for more information.

lgrp_nlgrps($cookie)
: This function takes a cookie that represents a snapshot of the lgroup hierarchy. It returns the number of lgroups in the hierarchy. This number is always at least one. If the lgrp_nlgrps function does not complete successfully, it returns a value of undef and sets the value of $! to EINVAL to indicate that the cookie is invalid. See the man page for the lgrp_nlgrps(3LGRP) function for more information.

`lgrp_root($cookie)`
This function returns the ID of the root lgroup. If the `lgrp_root` function does not complete successfully, it returns a value of `undef` and sets the value of `$!` to `EINVAL` to indicate that the cookie is invalid. See the man page for the `lgrp_root`(3LGRP) function for more information.

`lgrp_mem_size($cookie, $lgrp, $type, $content)`
This function takes a cookie that represents a snapshot of the lgroup hierarchy. The function returns the memory size of the given lgroup in bytes. Set the value of the `$type` argument to `LGRP_MEM_SZ_FREE` to have the `lgrp_mem_size()` function return the amount of free memory. Set the value of the `$type` argument to `LGRP_MEM_SZ_INSTALLED` to have the `lgrp_mem_size()` function return the amount of installed memory. Set the value of the `$content` argument to `LGRP_CONTENT_HIERARCHY` to have the `lgrp_mem_size()` function return results for the specified lgroup and each of its child lgroups. Set the value of the `$content` argument to `LGRP_CONTENT_DIRECT` to have the `lgrp_mem_size()` function return results for the specified lgroup only. This function returns the memory size in bytes upon successful completion, the size in bytes is returned. If the `lgrp_mem_size` function does not complete successfully, it returns a value of `undef` and sets `$!` to indicate the error. See the man page for the `lgrp_mem_size`(3LGRP) function for more information.

`lgrp_version([VERSION])`
This function takes an interface version number as the value of the `VERSION` argument and returns an lgroup interface version. To discover the current lgroup interface version, pass the value of `LGRP_VER_CURRENT` or `LGRP_VER_NONE` in the `VERSION` argument. The `lgrp_version()` function returns the requested version if the system supports that version. The `lgrp_version()` function returns `LGRP_VER_NONE` if the system does not supports the request version. The `lgrp_version()` function returns the current version of the library when you call the function with `LGRP_VER_NONE` as the value of the `VERSION` argument. This code fragment tests whether the version of the interface used by the caller is supported:

```
lgrp_version(LGRP_VER_CURRENT) == LGRP_VER_CURRENT or
            die("Built with unsupported lgroup interface");
```

See the man page for the `lgrp_version`(3LGRP) function for more information.

`lgrp_affinity_set($idtype, $id, $lgrp, $affinity)`
This function sets the affinity that the LWPs you specify with the `$idtype` and `$id` arguments have for the given lgroup. You can set the lgroup affinity to `LGRP_AFF_STRONG`, `LGRP_AFF_WEAK`, or `LGRP_AFF_NONE`. When the value of the `$idtype` argument is `P_PID`, this function sets the affinity for all the LWPs of the process with the process id specified in the `$id` argument. The `lgrp_affinity_set()` function sets the affinity for the LWP of the current process with LWP id `$id` when the value of the `$idtype` argument is `P_LWPID`. You can specify the current LWP or process by assigning the `$id` argument a value of `P_MYID`. This function returns `1` upon successful completion. If the `lgrp_affinity_set` function does not complete successfully, it returns a value of `undef` and sets `$!` to indicate the error. See the man page for the `lgrp_affinity_set`(3LGRP) function for more information.

`lgrp_affinity_get($idtype, $id, $lgrp)`
This function retrieves the affinity that the LWPs you specify with the `$idtype` and `$id` arguments have for the given lgroup. When the value of the `$idtype` argument is `P_PID`, this function retrieves the affinity for one of the LWPs in the process. The `lgrp_affinity_get()` function retrieves the affinity for the LWP of the current process with LWP id `$id` when the value of the `$idtype` argument is `P_LWPID`. You can specify the current LWP or process by assigning the `$id` argument a value of `P_MYID`. This function returns `1` upon successful completion. If the `lgrp_affinity_get` function does not complete successfully, it returns a value of `undef` and sets `$!` to indicate the error. See the man page for the `lgrp_affinity_get(3LGRP)` function for more information.

`lgrp_latency_cookie($cookie, $from, $to, [$between=LGRP_LAT_CPU_TO_MEM])()`
This function takes a cookie that represents a snapshot of the lgroup hierarchy and returns the latency value between a hardware resource in the `$from` lgroup to a hardware resource in the `$to` lgroup. This function returns the latency value within a given lgroup when the values of the `$from` and `$to` arguments are identical. Set the value of the optional `$between` argument to `LGRP_LAT_CPU_TO_MEM` to specify the hardware resources to measure the latency between. `LGRP_LAT_CPU_TO_MEM` represents the latency from CPU to memory and is the only valid value for this argument in this release. This function returns `1` upon successful completion. If the `lgrp_latency_cookie` function does not complete successfully, it returns a value of `undef` and sets `$!` to indicate the error. See the man page for the `lgrp_latency_cookie(3LGRP)` function for more information.

`lgrp_latency($from, $to)()`
The function is similar to the `lgrp_latency_cookie()` function, but returns the latency between the given lgroups at the given instant in time. Because the system dynamically reallocates and frees lgroups, this function's results are not always consistent across calls. This function is deprecated. Use the `lgrp_latency_cookie()` function instead. See the man page for the `lgrp_latency(3LGRP)` function for more information.

`lgrp_resources($cookie, $lgrp, $type)()`
This function is only available for version 2 of the API. When you call this function with version 1 of the API, the `lgrp_resources()` function returns `undef` or the empty list and sets the value of `$!` to `EINVAL`. This function returns the list of lgroups that directly contain the specified type of resources. The resources are represented by a set of lgroups in which each lgroup directly contains CPU and/or memory resources. To specify CPU resources, set the value of the `$type` argument to `LGRP_RSRC_CPU`. To specify memory resources, set the value of the `$type` argument to `LGRP_RSRC_MEM`. If the `lgrp_resources` function does not complete successfully, it returns a value of `undef` or the empty list and sets `$!` to indicate the error. See the man page for the `lgrp_resources(3LGRP)` function for more information.

`lgrp_lgrps($cookie, [$lgrp])()`
This function returns the list of all of the lgroups in a hierarchy, starting from the lgroup specified in the `$lgrp` argument. This function uses the value returned by the `lgrp_root($cookie)` function when the `$lgrp` argument has no value. The `lgrp_lgrps()` function returns the empty list on failure. This function returns the total number of lgroups in the system when you call it in a scalar context.

`lgrp_leaves($cookie, [$lgrp])()`
This function returns the list of all leaf lgroups in a hierarchy that starts from the lgroup specified in the `$lgrp` argument. This function uses the value returned by the `lgrp_root($cookie)` function when the `$lgrp` argument has no value. The `lgrp_leaves()` function returns `undef` or the empty list on failure. This function returns the total number of leaf lgroups in the system when you call it in a scalar context.

`lgrp_isleaf($cookie, $lgrp)()`
This function returns `True` if the lgroup specified by the value of the `$lgrp` argument is a leaf lgroup. Leaf lgroups have no children. The `lgrp_isleaf()` function returns `False` if the specified lgroup is not a leaf lgroup.

Object Methods in the Solaris::lgrp Module

`new([$view])`
This method creates a new `Sun::Solaris::Lgrp` object. An optional argument is passed to the `lgrp_init()` function. This method uses a value for the `$view` argument of `LGRP_VIEW_OS` by default.

`cookie()`
This function returns a transparent cookie that is passed to functions that accept a cookie.

`version([$version])`
This method returns the current version of the `liblgrp(3LIB)` library when you call it without an argument. This is a wrapper for the `lgrp_version()` function with `LGRP_VER_NONE` as the default value of the `$version` argument.

`stale()`
This method returns `T` if the lgroup information in the object is stale. This method returns `F` in all other cases. The `stale` method is a wrapper for the `lgrp_cookie_stale()` function.

`view()`
This method returns the snapshot's view of the lgroup hierarchy. The `view()` method is a wrapper for the `lgrp_view()` function.

`root()`
This method returns the root lgroup. The `root()` method is a wrapper for the `lgrp_root()` function.

`children($lgrp)`
This method returns the list of lgroups that are children of the specified lgroup. The `children` method is a wrapper for the `lgrp_children()` function.

`parents($lgrp)`
This method returns the list of lgroups that are parents of the specified lgroup. The `parents` method is a wrapper for the `lgrp_parents()` function.

`nlgrps()`
: This method returns the number of lgroups in the hierarchy. The `nlgrps()` method is a wrapper for the `lgrp_nlgrps()` function.

`mem_size($lgrp, $type, $content)`
: This method returns the memory size of the given lgroup in bytes. The `mem_size` method is a wrapper for the `lgrp_mem_size()` function.

`cpus($lgrp, $context)`
: This method returns the list of CPUs that are in the lgroup specified by the `$lgrp` argument. The `cpus` method is a wrapper for the `lgrp_cpus()` function.

`resources($lgrp, $type)`
: This method returns the list of lgroups that directly contain resources of the specified type. The `resources` method is a wrapper for the `lgrp_resources()` function.

`home($idtype, $id)`
: This method returns the home lgroup for the given process or thread. The `home` method is a wrapper for the `lgrp_home()` function.

`affinity_get($idtype, $id, $lgrp)`
: This method returns the affinity that the LWP has to a given lgroup. The `affinity_get()` method is a wrapper for the `lgrp_affinity_get()` function.

`affinity_set($idtype, $id, $lgrp, $affinity)`
: This method sets the affinities that the LWPs specified by the `$idtype` and `$id` arguments have for the given lgroup. The `affinity_set()` method is a wrapper for the `lgrp_affinity_set()` function.

`lgrps([$lgrp])`
: This method returns the list of all of the lgroups in a hierarchy starting from the lgroup specified by the value of the `$lgrp` argument. The hierarchy starts from the root lgroup when you do not specify a value for the `$lgrp` argument. The `lgrps()` method is a wrapper for the `lgrp_lgrps()` function.

`leaves([$lgrp])`
: This method returns the list of all of the leaf lgroups in a hierarchy starting from the lgroup specified by the value of the `$lgrp` argument. The hierarchy starts from the root lgroup when you do not specify a value for the `$lgrp` argument. The `leaves()` method is a wrapper for the `lgrp_leaves()` function.

`isleaf($lgrp)`
: This method returns `True` if the lgroup specified by the value of the `$lgrp` argument is a leaf lgroup. A leaf lgroup has no children. This method returns `False` in all other cases. The `isleaf` method is a wrapper for the `lgrp_isleaf` function.

`latency($from, $to)`
: This method returns the latency value between a hardware resource in the lgroup specified by the `$from` argument to a hardware resource in the lgroup specified by the `$to` argument. The `latency` method uses the `lgrp_latency()` function in version 1 of `liblgrp`. The latency

method uses the `lgrp_latency_cookie()` function in newer versions of `liblgrp`.

CHAPTER 3

Locality Group APIs

This chapter describes the APIs that applications use to interact with locality groups.

This chapter discusses the following topics:

- "Verifying the Interface Version" on page 29 describes the functions that give information about the interface.
- "Initializing the Locality Group Interface" on page 30 describes function calls that initialize and shut down the portion of the interface that is used to traverse the locality group hierarchy and to discover the contents of a locality group.
- "Locality Group Hierarchy" on page 31 describes function calls that navigate the locality group hierarchy and functions that get characteristics of the locality group hierarchy.
- "Locality Group Contents" on page 33 describes function calls that retrieve information about a locality group's contents.
- "Locality Group Characteristics" on page 35 describes function calls that retrieve information about a locality group's characteristics.
- "Locality Groups and Thread and Memory Placement" on page 36 describes how to affect the locality group placement of a thread and its memory.
- "Examples of API Usage" on page 42 contains code that performs example tasks by using the APIs that are described in this chapter.

Verifying the Interface Version

The `lgrp_version(3LGRP)` function must be used to verify the presence of a supported lgroup interface before using the lgroup API. The `lgrp_version()` function has the following syntax:

```
#include <sys/lgrp_user.h>
int lgrp_version(const int version);
```

The `lgrp_version()` function takes a version number for the lgroup interface as an argument and returns the lgroup interface version that the system supports. When the current implementation of the lgroup API supports the version number in the `version` argument, the `lgrp_version()` function returns that version number. Otherwise, the `lgrp_version()` function returns `LGRP_VER_NONE`.

EXAMPLE 3–1 Example of `lgrp_version()` Use

```
#include <sys/lgrp_user.h>
if (lgrp_version(LGRP_VER_CURRENT) != LGRP_VER_CURRENT) {
    fprintf(stderr, "Built with unsupported lgroup interface %d\n",
        LGRP_VER_CURRENT);
    exit (1);
    }
```

Initializing the Locality Group Interface

Applications must call `lgrp_init(3LGRP)` in order to use the APIs for traversing the lgroup hierarchy and to discover the contents of the lgroup hierarchy. The call to `lgrp_init()` gives the application a consistent snapshot of the lgroup hierarchy. The application developer can specify whether the snapshot contains only the resources that are available to the calling thread specifically or the resources that are available to the operating system in general. The `lgrp_init()` function returns a cookie that is used for the following tasks:

- Navigating the lgroup hierarchy
- Determining the contents of an lgroup
- Determining whether the snapshot is current

Using `lgrp_init()`

The `lgrp_init()` function initializes the lgroup interface and takes a snapshot of the lgroup hierarchy.

```
#include <sys/lgrp_user.h>
lgrp_cookie_t lgrp_init(lgrp_view_t view);
```

When the `lgrp_init()` function is called with `LGRP_VIEW_CALLER` as the view, the function returns a snapshot that contains only the resources that are available to the calling thread. When the `lgrp_init()` function is called with `LGRP_VIEW_OS` as the view, the function returns a snapshot that contains the resources that are available to the operating system. When a thread successfully calls the `lgrp_init()` function, the function returns a cookie that is used by any function that interacts with the lgroup hierarchy. When a thread no longer needs the cookie, call the `lgrp_fini()` function with the cookie as the argument.

The lgroup hierarchy consists of a root lgroup that contains all of the machine's CPU and memory resources. The root lgroup might contain other locality groups bounded by smaller latencies.

The `lgrp_init()` function can return two errors. When a view is invalid, the function returns `EINVAL`. When there is insufficient memory to allocate the snapshot of the lgroup hierarchy, the function returns `ENOMEM`.

Using `lgrp_fini()`

The `lgrp_fini(3LGRP)` function ends the usage of a given cookie and frees the corresponding lgroup hierarchy snapshot.

```
#include <sys/lgrp_user.h>
int lgrp_fini(lgrp_cookie_t cookie);
```

The `lgrp_fini()` function takes a cookie that represents an lgroup hierarchy snapshot created by a previous call to `lgrp_init()`. The `lgrp_fini()` function frees the memory that is allocated to that snapshot. After the call to `lgrp_fini()`, the cookie is invalid. Do not use that cookie again.

When the cookie passed to the `lgrp_fini()` function is invalid, `lgrp_fini()` returns `EINVAL`.

Locality Group Hierarchy

The APIs that are described in this section enable the calling thread to navigate the lgroup hierarchy. The lgroup hierarchy is a directed acyclic graph that is similar to a tree, except that a node might have more than one parent. The root lgroup represents the whole machine and contains all of that machine's resources. The root lgroup is the lgroup with the highest latency value in the system. Each of the child lgroups contains a subset of the hardware that is in the root lgroup. Each child lgroup is bounded by a lower latency value. Locality groups that are closer to the root have more resources and a higher latency. Locality groups that are closer to the leaves have fewer resources and a lower latency. An lgroup can contain resources directly within its latency boundary. An lgroup can also contain leaf lgroups that contain their own sets of resources. The resources of leaf lgroups are available to the lgroup that encapsulates those leaf lgroups.

Using `lgrp_cookie_stale()`

The `lgrp_cookie_stale(3LGRP)` function determines whether the snapshot of the lgroup hierarchy represented by the given cookie is current.

```
#include <sys/lgrp_user.h>
int lgrp_cookie_stale(lgrp_cookie_t cookie);
```

The cookie returned by the `lgrp_init()` function can become stale due to several reasons that depend on the view that the snapshot represents. A cookie that is returned by calling the `lgrp_init()` function with the view set to `LGRP_VIEW_OS` can become stale due to changes in the lgroup hierarchy such as dynamic reconfiguration or a change in a CPU's online status. A cookie that is returned by calling the `lgrp_init()` function with the view set to `LGRP_VIEW_CALLER` can become stale due to changes in the calling thread's processor set or changes in the lgroup hierarchy. A stale cookie is refreshed by calling the `lgrp_fini()` function with the old cookie, followed by calling `lgrp_init()` to generate a new cookie.

The `lgrp_cookie_stale()` function returns `EINVAL` when the given cookie is invalid.

Using `lgrp_view()`

The `lgrp_view(3LGRP)` function determines the view with which a given lgroup hierarchy snapshot was taken.

```
#include <sys/lgrp_user.h>
lgrp_view_t lgrp_view(lgrp_cookie_t cookie);
```

The `lgrp_view()` function takes a cookie that represents a snapshot of the lgroup hierarchy and returns the snapshot's view of the lgroup hierarchy. Snapshots that are taken with the view `LGRP_VIEW_CALLER` contain only the resources that are available to the calling thread. Snapshots that are taken with the view `LGRP_VIEW_OS` contain all the resources that are available to the operating system.

The `lgrp_view()` function returns `EINVAL` when the given cookie is invalid.

Using `lgrp_nlgrps()`

The `lgrp_nlgrps(3LGRP)` function returns the number of locality groups in the system. If a system has only one locality group, memory placement optimizations have no effect.

```
#include <sys/lgrp_user.h>
int lgrp_nlgrps(lgrp_cookie_t cookie);
```

The `lgrp_nlgrps()` function takes a cookie that represents a snapshot of the lgroup hierarchy and returns the number of lgroups available in the hierarchy.

The `lgrp_nlgrps()` function returns `EINVAL` when the cookie is invalid.

Using `lgrp_root()`

The `lgrp_root(3LGRP)` function returns the root lgroup ID.

```
#include <sys/lgrp_user.h>
lgrp_id_t lgrp_root(lgrp_cookie_t cookie);
```

The `lgrp_root()` function takes a cookie that represents a snapshot of the lgroup hierarchy and returns the root lgroup ID.

Using lgrp_parents()

The lgrp_parents(3LGRP) function takes a cookie that represents a snapshot of the lgroup hierarchy and returns the number of parent lgroups for the specified lgroup.

```
#include <sys/lgrp_user.h>
int lgrp_parents(lgrp_cookie_t cookie, lgrp_id_t child,
                 lgrp_id_t *lgrp_array, uint_t lgrp_array_size);
```

If lgrp_array is not NULL and the value of lgrp_array_size is not zero, the lgrp_parents() function fills the array with parent lgroup IDs until the array is full or all parent lgroup IDs are in the array. The root lgroup has zero parents. When the lgrp_parents() function is called for the root lgroup, lgrp_array is not filled in.

The lgrp_parents() function returns EINVAL when the cookie is invalid. The lgrp_parents() function returns ESRCH when the specified lgroup ID is not found.

Using lgrp_children()

The lgrp_children(3LGRP) function takes a cookie that represents the calling thread's snapshot of the lgroup hierarchy and returns the number of child lgroups for the specified lgroup.

```
#include <sys/lgrp_user.h>
int lgrp_children(lgrp_cookie_t cookie, lgrp_id_t parent,
                  lgrp_id_t *lgrp_array, uint_t lgrp_array_size);
```

If lgrp_array is not NULL and the value of lgrp_array_size is not zero, the lgrp_children() function fills the array with child lgroup IDs until the array is full or all child lgroup IDs are in the array.

The lgrp_children() function returns EINVAL when the cookie is invalid. The lgrp_children() function returns ESRCH when the specified lgroup ID is not found.

Locality Group Contents

The following APIs retrieve information about the contents of a given lgroup.

The lgroup hierarchy organizes the domain's resources to simplify the process of locating the nearest resource. Leaf lgroups are defined with resources that have the least latency. Each of the successive ancestor lgroups of a given leaf lgroup contains the next nearest resources to its child lgroup. The root lgroup contains all of the resources that are in the domain.

The resources of a given lgroup are contained directly within that lgroup or indirectly within the leaf lgroups that the given lgroup encapsulates. Leaf lgroups directly contain their resources and do not encapsulate any other lgroups.

Using `lgrp_resources()`

The `lgrp_resources()` function returns the number of resources contained in a specified lgroup.

```
#include <sys/lgrp_user.h>
int lgrp_resources(lgrp_cookie_t cookie, lgrp_id_t lgrp, lgrp_id_t *lgrpids,
                   uint_t count, lgrp_rsrc_t type);
```

The `lgrp_resources()` function takes a cookie that represents a snapshot of the lgroup hierarchy. That cookie is obtained from the `lgrp_init()` function. The `lgrp_resources()` function returns the number of resources that are in the lgroup with the ID that is specified by the value of the `lgrp` argument. The `lgrp_resources()` function represents the resources with a set of lgroups that directly contain CPU or memory resources. The `lgrp_rsrc_t` argument can have the following two values:

`LGRP_RSRC_CPU` The `lgrp_resources()` function returns the number of CPU resources.

`LGRP_RSRC_MEM` The `lgrp_resources()` function returns the number of memory resources.

When the value passed in the `lgrpids[]` argument is not null and the `count` argument is not zero, the `lgrp_resources()` function stores lgroup IDs in the `lgrpids[]` array. The number of lgroup IDs stored in the array can be up to the value of the `count` argument.

The `lgrp_resources()` function returns `EINVAL` when the specified cookie, lgroup ID, or type are not valid. The `lgrp_resources()` function returns `ESRCH` when the function does not find the specified lgroup ID.

Using `lgrp_cpus()`

The `lgrp_cpus(3LGRP)` function takes a cookie that represents a snapshot of the lgroup hierarchy and returns the number of CPUs in a given lgroup.

```
#include <sys/lgrp_user.h>
int lgrp_cpus(lgrp_cookie_t cookie, lgrp_id_t lgrp, processorid_t *cpuids,
              uint_t count, int content);
```

If the *cpuid[]* argument is not `NULL` and the CPU count is not zero, the `lgrp_cpus()` function fills the array with CPU IDs until the array is full or all the CPU IDs are in the array.

The *content* argument can have the following two values:

`LGRP_CONTENT_ALL` The `lgrp_cpus()` function returns IDs for the CPUs in this lgroup and this lgroup's descendants.

`LGRP_CONTENT_DIRECT` The `lgrp_cpus()` function returns IDs for the CPUs in this lgroup only.

The `lgrp_cpus()` function returns `EINVAL` when the cookie, lgroup ID, or one of the flags is not valid. The `lgrp_cpus()` function returns `ESRCH` when the specified lgroup ID is not found.

Using `lgrp_mem_size()`

The `lgrp_mem_size(3LGRP)` function takes a cookie that represents a snapshot of the lgroup hierarchy and returns the size of installed or free memory in the given lgroup. The `lgrp_mem_size()` function reports memory sizes in bytes.

```
#include <sys/lgrp_user.h>
lgrp_mem_size_t lgrp_mem_size(lgrp_cookie_t cookie, lgrp_id_t lgrp,
                              int type, int content)
```

The *type* argument can have the following two values:

`LGRP_MEM_SZ_FREE` The `lgrp_mem_size()` function returns the amount of free memory in bytes.

`LGRP_MEM_SZ_INSTALLED` The `lgrp_mem_size()` function returns the amount of installed memory in bytes.

The *content* argument can have the following two values:

`LGRP_CONTENT_ALL` The `lgrp_mem_size()` function returns the amount of memory in this lgroup and this lgroup's descendants.

`LGRP_CONTENT_DIRECT` The `lgrp_mem_size()` function returns the amount of memory in this lgroup only.

The `lgrp_mem_size()` function returns `EINVAL` when the cookie, lgroup ID, or one of the flags is not valid. The `lgrp_mem_size()` function returns `ESRCH` when the specified lgroup ID is not found.

Locality Group Characteristics

The following API retrieves information about the characteristics of a given lgroup.

Using `lgrp_latency_cookie()`

The `lgrp_latency(3LGRP)` function returns the latency between a CPU in one lgroup to the memory in another lgroup.

```
#include <sys/lgrp_user.h>
int lgrp_latency_cookie(lgrp_cookie_t cookie, lgrp_id_t from, lgrp_id_t to.
```

```
                        lat_between_t between);
```

The `lgrp_latency_cookie()` function takes a cookie that represents a snapshot of the lgroup hierarchy. The `lgrp_init()` function creates this cookie. The `lgrp_latency_cookie()` function returns a value that represents the latency between a hardware resource in the lgroup given by the value of the *from* argument and a hardware resource in the lgroup given by the value of the *to* argument. If both arguments point to the same lgroup, the `lgrp_latency_cookie()` function returns the latency value within that lgroup.

Note – The latency value returned by the `lgrp_latency_cookie()` function is defined by the operating system and is platform-specific. This value does not necessarily represent the actual latency between hardware devices. Use this value only for comparison within one domain.

When the value of the *between* argument is `LGRP_LAT_CPU_TO_MEM`, the `lgrp_latency_cookie()` function measures the latency from a CPU resource to a memory resource.

The `lgrp_latency_cookie()` function returns `EINVAL` when the lgroup ID is not valid. When the `lgrp_latency_cookie()` function does not find the specified lgroup ID, the "from" lgroup does not contain any CPUs, or the "to" lgroup does not have any memory, the `lgrp_latency_cookie()` function returns `ESRCH`.

Locality Groups and Thread and Memory Placement

This section discusses the APIs used to discover and affect thread and memory placement with respect to lgroups.

- The `lgrp_home(3LGRP)` function is used to discover thread placement.
- The `meminfo(2)` system call is used to discover memory placement.
- The `MADV_ACCESS` flags to the `madvise(3C)` function are used to affect memory allocation among lgroups.
- The `lgrp_affinity_set(3LGRP)` function can affect thread and memory placement by setting a thread's affinity for a given lgroup.
- The affinities of an lgroup may specify an order of preference for lgroups from which to allocate resources.
- The kernel needs information about the likely pattern of an application's memory use in order to allocate memory resources efficiently.
- The `madvise()` function and its shared object analogue `madv.so.1` provide this information to the kernel.

- A running process can gather memory usage information about itself by using the `meminfo()` system call.

Using `lgrp_home()`

The `lgrp_home()` function returns the home lgroup for the specified process or thread.

```
#include <sys/lgrp_user.h>
lgrp_id_t lgrp_home(idtype_t idtype, id_t id);
```

The `lgrp_home()` function returns `EINVAL` when the ID type is not valid. The `lgrp_home()` function returns `EPERM` when the effective user of the calling process is not the superuser and the real or effective user ID of the calling process does not match the real or effective user ID of one of the threads. The `lgrp_home()` function returns `ESRCH` when the specified process or thread is not found.

Using `madvise()`

The `madvise()` function advises the kernel that a region of user virtual memory in the range starting at the address specified in *addr* and with length equal to the value of the *len* parameter is expected to follow a particular pattern of use. The kernel uses this information to optimize the procedure for manipulating and maintaining the resources associated with the specified range. Use of the `madvise()` function can increase system performance when used by programs that have specific knowledge of their access patterns over memory.

```
#include <sys/types.h>
#include <sys/mman.h>
int madvise(caddr_t addr, size_t len, int advice);
```

The `madvise()` function provides the following flags to affect how a thread's memory is allocated among lgroups:

`MADV_ACCESS_DEFAULT`	This flag resets the kernel's expected access pattern for the specified range to the default.
`MADV_ACCESS_LWP`	This flag advises the kernel that the next LWP to touch the specified address range is the LWP that will access that range the most. The kernel allocates the memory and other resources for this range and the LWP accordingly.
`MADV_ACCESS_MANY`	This flag advises the kernel that many processes or LWPs will access the specified address range randomly across the system. The kernel allocates the memory and other resources for this range accordingly.

The `madvise()` function can return the following values:

`EAGAIN`	Some or all of the mappings in the specified address range, from *addr* to *addr+len*, are locked for I/O.
`EINVAL`	The value of the *addr* parameter is not a multiple of the page size as returned by `sysconf`(3C), the length of the specified address range is less than or equal to zero, or the advice is invalid.
`EIO`	An I/O error occurs while reading from or writing to the file system.
`ENOMEM`	Addresses in the specified address range are outside the valid range for the address space of a process or the addresses in the specified address range specify one or more pages that are not mapped.
`ESTALE`	The NFS file handle is stale.

Using `meminfo()`

The `meminfo()` function gives the calling process information about the virtual memory and physical memory that the system has allocated to that process.

```
#include <sys/types.h>
#include <sys/mman.h>
int meminfo(const uint64_t inaddr[], int addr_count,
    const uint_t info_req[], int info_count, uint64_t outdata[],
    uint_t validity[]);
```

The `meminfo()` function can return the following types of information:

`MEMINFO_VPHYSICAL`	The physical memory address corresponding to the given virtual address
`MEMINFO_VLGRP`	The lgroup to which the physical page corresponding to the given virtual address belongs
`MEMINFO_VPAGESIZE`	The size of the physical page corresponding to the given virtual address
`MEMINFO_VREPLCNT`	The number of replicated physical pages that correspond to the given virtual address
`MEMINFO_VREPL|n`	The *n*th physical replica of the given virtual address
`MEMINFO_VREPL_LGRP|n`	The lgroup to which the *n*th physical replica of the given virtual address belongs
`MEMINFO_PLGRP`	The lgroup to which the given physical address belongs

The `meminfo()` function takes the following parameters:

inaddr An array of input addresses.

addr_count The number of addresses that are passed to `meminfo()`.

info_req An array that lists the types of information that are being requested.

info_count The number of pieces of information that are requested for each address in the *inaddr* array.

outdata An array where the `meminfo()` function places the results. The array's size is equal to the product of the values of the *info_req* and *addr_count* parameters.

validity An array of size equal to the value of the *addr_count* parameter. The *validity* array contains bitwise result codes. The *0*th bit of the result code evaluates the validity of the corresponding input address. Each successive bit in the result code evaluates the validity of the response to the members of the *info_req* array in turn.

The `meminfo()` function returns `EFAULT` when the area of memory to which the *outdata* or *validity* arrays point cannot be written to. The `meminfo()` function returns `EFAULT` when the area of memory to which the *info_req* or *inaddr* arrays point cannot be read from. The `meminfo()` function returns `EINVAL` when the value of *info_count* exceeds 31 or is less than 1. The `meminfo()` function returns `EINVAL` when the value of *addr_count* is less than zero.

EXAMPLE 3–2 Use of `meminfo()` to Print Out Physical Pages and Page Sizes Corresponding to a Set of Virtual Addresses

```
void
print_info(void **addrvec, int how_many)
{
        static const int info[] = {
                MEMINFO_VPHYSICAL,
                MEMINFO_VPAGESIZE};
        uint64_t * inaddr = alloca(sizeof(uint64_t) * how_many);
        uint64_t * outdata = alloca(sizeof(uint64_t) * how_many * 2;
        uint_t * validity = alloca(sizeof(uint_t) * how_many);

        int i;

        for (i = 0; i < how_many; i++)
                inaddr[i] = (uint64_t *)addr[i];

        if (meminfo(inaddr, how_many,  info,
                    sizeof (info)/ sizeof(info[0]),
                    outdata, validity) < 0)
                ...

        for (i = 0; i < how_many; i++) {
                if (validity[i] & 1 == 0)
                        printf("address 0x%llx not part of address
                                        space\n",
                                inaddr[i]);

                else if (validity[i] & 2 == 0)
```

EXAMPLE 3–2 Use of `meminfo()` to Print Out Physical Pages and Page Sizes Corresponding to a Set of Virtual Addresses *(Continued)*

```
                                printf("address 0x%llx has no physical page
                                                associated with it\n",
                                        inaddr[i]);

                else {
                        char buff[80];
                        if (validity[i] & 4 == 0)
                                strcpy(buff, "<Unknown>");
                        else
                                sprintf(buff, "%lld", outdata[i * 2 +
                                                        1]);
                        printf("address 0x%llx is backed by physical
                                        page 0x%llx of size %s\n",
                                        inaddr[i], outdata[i * 2], buff);
                }
        }
}
```

Locality Group Affinity

The kernel assigns a thread to a locality group when the lightweight process (LWP) for that thread is created. That lgroup is called the thread's *home lgroup*. The kernel runs the thread on the CPUs in the thread's home lgroup and allocates memory from that lgroup whenever possible. If resources from the home lgroup are unavailable, the kernel allocates resources from other lgroups. When a thread has affinity for more than one lgroup, the operating system allocates resources from lgroups chosen in order of affinity strength. Lgroups can have one of three distinct affinity levels:

1. `LGRP_AFF_STRONG` indicates strong affinity. If this lgroup is the thread's home lgroup, the operating system avoids rehoming the thread to another lgroup if possible. Events such as dynamic reconfiguration, processor, offlining, processor binding, and processor set binding and manipulation might still result in thread rehoming.
2. `LGRP_AFF_WEAK` indicates weak affinity. If this lgroup is the thread's home lgroup, the operating system rehomes the thread if necessary for load balancing purposes.
3. `LGRP_AFF_NONE` indicates no affinity. If a thread has no affinity to any lgroup, the operating system assigns a home lgroup to the thread .

The operating system uses lgroup affinities as advice when allocating resources for a given thread. The advice is factored in with the other system constraints. Processor binding and processor sets do not change lgroup affinities, but might restrict the lgroups on which a thread can run.

Using `lgrp_affinity_get()`

The `lgrp_affinity_get(3LGRP)` function returns the affinity that a LWP has for a given lgroup.

```
#include <sys/lgrp_user.h>
lgrp_affinity_t lgrp_affinity_get(idtype_t idtype, id_t id, lgrp_id_t lgrp);
```

The *idtype* and *id* arguments specify the LWP that the `lgrp_affinity_get()` function examines. If the value of *idtype* is `P_PID`, the `lgrp_affinity_get()` function gets the lgroup affinity for one of the LWPs in the process whose process ID matches the value of the *id* argument. If the value of *idtype* is `P_LWPID`, the `lgrp_affinity_get()` function gets the lgroup affinity for the LWP of the current process whose LWP ID matches the value of the *id* argument. If the value of *idtype* is `P_MYID`, the `lgrp_affinity_get()` function gets the lgroup affinity for the current LWP.

The `lgrp_affinity_get()` function returns `EINVAL` when the given lgroup or ID type is not valid. The `lgrp_affinity_get()` function returns `EPERM` when the effective user of the calling process is not the superuser and the ID of the calling process does not match the real or effective user ID of one of the LWPs. The `lgrp_affinity_get()` function returns `ESRCH` when a given lgroup or LWP is not found.

Using `lgrp_affinity_set()`

The `lgrp_affinity_set(3LGRP)` function sets the affinity that a LWP or set of LWPs have for a given lgroup.

```
#include <sys/lgrp_user.h>
int lgrp_affinity_set(idtype_t idtype, id_t id, lgrp_id_t lgrp,
                      lgrp_affinity_t affinity);
```

The *idtype* and *id* arguments specify the LWP or set of LWPs the `lgrp_affinity_set()` function examines. If the value of *idtype* is `P_PID`, the `lgrp_affinity_set()` function sets the lgroup affinity for all of the LWPs in the process whose process ID matches the value of the *id* argument to the affinity level specified in the *affinity* argument. If the value of *idtype* is `P_LWPID`, the `lgrp_affinity_set()` function sets the lgroup affinity for the LWP of the current process whose LWP ID matches the value of the *id* argument to the affinity level specified in the *affinity* argument. If the value of *idtype* is `P_MYID`, the `lgrp_affinity_set()` function sets the lgroup affinity for the current LWP or process to the affinity level specified in the *affinity* argument.

The `lgrp_affinity_set()` function returns `EINVAL` when the given lgroup, affinity, or ID type is not valid. The `lgrp_affinity_set()` function returns `EPERM` when the effective user of the calling process is not the superuser and the ID of the calling process does not match the real or effective user ID of one of the LWPs. The `lgrp_affinity_set()` function returns `ESRCH` when a given lgroup or LWP is not found.

Examples of API Usage

This section contains code for example tasks that use the APIs that are described in this chapter.

EXAMPLE 3–3 Move Memory to a Thread

The following code sample moves the memory in the address range between *addr* and *addr+len* near the next thread to touch that range.

```
#include <stdio.h>
#include <sys/mman.h>
#include <sys/types.h>



/*
 * Move memory to thread
 */
void
mem_to_thread(caddr_t addr, size_t len)
{
    if (madvise(addr, len, MADV_ACCESS_LWP) < 0)
        perror("madvise");
}
```

EXAMPLE 3–4 Move a Thread to Memory

This sample code uses the `meminfo()` function to determine the lgroup of the physical memory backing the virtual page at the given address. The sample code then sets a strong affinity for that lgroup in an attempt to move the current thread near that memory.

```
#include <stdio.h>
#include <sys/lgrp_user.h>
#include <sys/mman.h>
#include <sys/types.h>


/*
 * Move a thread to memory
 */
int
thread_to_memory(caddr_t va)
{
    uint64_t    addr;
    ulong_t     count;
    lgrp_id_t   home;
    uint64_t    lgrp;
    uint_t      request;
    uint_t      valid;

    addr = (uint64_t)va;
    count = 1;
    request = MEMINFO_VLGRP;
    if (meminfo(&addr, 1, &request, 1, &lgrp, &valid) != 0) {
```

EXAMPLE 3–4 Move a Thread to Memory *(Continued)*

```
        perror("meminfo");
        return (1);
    }

    if (lgrp_affinity_set(P_LWPID, P_MYID, lgrp, LGRP_AFF_STRONG) != 0) {
        perror("lgrp_affinity_set");
        return (2);
    }

    home = lgrp_home(P_LWPID, P_MYID);
    if (home == -1) {
        perror ("lgrp_home");
        return (3);
    }

    if (home != lgrp)
        return (-1);

    return (0);
}
```

EXAMPLE 3–5 Walk the lgroup Hierarchy

The following sample code walks through and prints out the lgroup hierarchy.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/lgrp_user.h>
#include <sys/types.h>


/*
 * Walk and print lgroup hierarchy from given lgroup
 * through all its descendants
 */
int
lgrp_walk(lgrp_cookie_t cookie, lgrp_id_t lgrp, lgrp_content_t content)
{
    lgrp_affinity_t    aff;
    lgrp_id_t          *children;
    processorid_t      *cpuids;
    int                i;
    int                ncpus;
    int                nchildren;
    int                nparents;
    lgrp_id_t          *parents;
    lgrp_mem_size_t    size;

    /*
     * Print given lgroup, caller’s affinity for lgroup,
     * and desired content specified
     */
    printf("LGROUP #%d:\n", lgrp);
```

EXAMPLE 3–5 Walk the lgroup Hierarchy *(Continued)*

```
aff = lgrp_affinity_get(P_LWPID, P_MYID, lgrp);
if (aff == -1)
    perror ("lgrp_affinity_get");
printf("\tAFFINITY: %d\n", aff);

printf("CONTENT %d:\n", content);

/*
 * Get CPUs
 */
ncpus = lgrp_cpus(cookie, lgrp, NULL, 0, content);
printf("\t%d CPUS: ", ncpus);
if (ncpus == -1) {
    perror("lgrp_cpus");
    return (-1);
} else if (ncpus > 0) {
    cpuids = malloc(ncpus * sizeof (processorid_t));
    ncpus = lgrp_cpus(cookie, lgrp, cpuids, ncpus, content);
            if (ncpus == -1) {
        free(cpuids);
                       perror("lgrp_cpus");
        return (-1);
    }
    for (i = 0; i < ncpus; i++)
        printf("%d ", cpuids[i]);
    free(cpuids);
}
printf("\n");

/*
 * Get memory size
 */
printf("\tMEMORY: ");
size = lgrp_mem_size(cookie, lgrp, LGRP_MEM_SZ_INSTALLED, content);
if (size == -1) {
    perror("lgrp_mem_size");
    return (-1);
}
printf("installed bytes 0x%llx, ", size);
size = lgrp_mem_size(cookie, lgrp, LGRP_MEM_SZ_FREE, content);
    if (size == -1) {
    perror("lgrp_mem_size");
    return (-1);
}
printf("free bytes 0x%llx\n", size);

/*
 * Get parents
 */
nparents = lgrp_parents(cookie, lgrp, NULL, 0);
printf("\t%d PARENTS: ", nparents);
if (nparents == -1) {
    perror("lgrp_parents");
    return (-1);
} else if (nparents > 0) {
```

EXAMPLE 3–5 Walk the lgroup Hierarchy *(Continued)*

```
        parents = malloc(nparents * sizeof (lgrp_id_t));
        nparents = lgrp_parents(cookie, lgrp, parents, nparents);
                   if (nparents == -1) {
            free(parents);
                        perror("lgrp_parents");
            return (-1);
                   }
        for (i = 0; i < nparents; i++)
            printf("%d ", parents[i]);
        free(parents);
    }
    printf("\n");

    /*
     * Get children
     */
    nchildren = lgrp_children(cookie, lgrp, NULL, 0);
    printf("\t%d CHILDREN: ", nchildren);
    if (nchildren == -1) {
        perror("lgrp_children");
        return (-1);
    } else if (nchildren > 0) {
        children = malloc(nchildren * sizeof (lgrp_id_t));
        nchildren = lgrp_children(cookie, lgrp, children, nchildren);
                   if (nchildren == -1) {
            free(children);
                        perror("lgrp_children");
            return (-1);
                   }
        printf("Children: ");
        for (i = 0; i < nchildren; i++)
            printf("%d ", children[i]);
        printf("\n");

        for (i = 0; i < nchildren; i++)
            lgrp_walk(cookie, children[i], content);

        free(children);
    }
    printf("\n");

    return (0);
}
```

EXAMPLE 3–6 Find the Closest lgroup With Available Memory Outside a Given lgroup

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/lgrp_user.h>
#include <sys/types.h>


#define    INT_MAX    2147483647
```

EXAMPLE 3–6 Find the Closest lgroup With Available Memory Outside a Given lgroup *(Continued)*

```
/*
 * Find next closest lgroup outside given one with available memory
 */
lgrp_id_t
lgrp_next_nearest(lgrp_cookie_t cookie, lgrp_id_t from)
{
    lgrp_id_t           closest;
    int                 i;
    int                 latency;
    int                 lowest;
    int                 nparents;
    lgrp_id_t           *parents;
    lgrp_mem_size_t     size;


    /*
     * Get number of parents
     */
    nparents = lgrp_parents(cookie, from, NULL, 0);
    if (nparents == -1) {
        perror("lgrp_parents");
        return (LGRP_NONE);
    }

    /*
     * No parents, so current lgroup is next nearest
     */
    if (nparents == 0) {
        return (from);
    }

    /*
     * Get parents
     */
    parents = malloc(nparents * sizeof (lgrp_id_t));
    nparents = lgrp_parents(cookie, from, parents, nparents);
    if (nparents == -1) {
        perror("lgrp_parents");
        free(parents);
        return (LGRP_NONE);
        }

    /*
     * Find closest parent (ie. the one with lowest latency)
     */
    closest = LGRP_NONE;
    lowest = INT_MAX;
    for (i = 0; i < nparents; i++) {
        lgrp_id_t    lgrp;

        /*
         * See whether parent has any free memory
         */
        size = lgrp_mem_size(cookie, parents[i], LGRP_MEM_SZ_FREE,
            LGRP_CONTENT_ALL);
```

EXAMPLE 3–6 Find the Closest lgroup With Available Memory Outside a Given lgroup *(Continued)*

```
        if (size > 0)
            lgrp = parents[i];
        else {
            if (size == -1)
                perror("lgrp_mem_size");

            /*
             * Find nearest ancestor if parent doesn't
             * have any memory
             */
            lgrp = lgrp_next_nearest(cookie, parents[i]);
            if (lgrp == LGRP_NONE)
                continue;
        }

        /*
         * Get latency within parent lgroup
         */
        latency = lgrp_latency_cookie(lgrp, lgrp);
        if (latency == -1) {
            perror("lgrp_latency_cookie");
            continue;
        }

        /*
         * Remember lgroup with lowest latency
         */
        if (latency < lowest) {
            closest = lgrp;
            lowest = latency;
        }
    }

    free(parents);
    return (closest);
}


/*
 * Find lgroup with memory nearest home lgroup of current thread
 */
lgrp_id_t
lgrp_nearest(lgrp_cookie_t cookie)
{
    lgrp_id_t    home;
    longlong_t    size;

    /*
     * Get home lgroup
     */
    home = lgrp_home(P_LWPID, P_MYID);

    /*
     * See whether home lgroup has any memory available in its hierarchy
     */
```

EXAMPLE 3–6 Find the Closest lgroup With Available Memory Outside a Given lgroup *(Continued)*

```
    size = lgrp_mem_size(cookie, home, LGRP_MEM_SZ_FREE,
        LGRP_CONTENT_ALL);
    if (size == -1)
        perror("lgrp_mem_size");

    /*
     * It does, so return the home lgroup.
     */
    if (size > 0)
        return (home);

    /*
     * Otherwise, find next nearest lgroup outside of the home.
     */
    return (lgrp_next_nearest(cookie, home));
}
```

EXAMPLE 3–7 Find Nearest lgroup With Free Memory

This example code finds the nearest lgroup with free memory to a given thread's home lgroup.

```
lgrp_id_t
lgrp_nearest(lgrp_cookie_t cookie)
{
        lgrp_id_t         home;
        longlong_t        size;

        /*
         * Get home lgroup
         */

        home = lgrp_home();

        /*
         * See whether home lgroup has any memory available in its hierarchy
         */

        if (lgrp_mem_size(cookie, lgrp, LGRP_MEM_SZ_FREE,
            LGRP_CONTENT_ALL, &size) == -1)
                perror("lgrp_mem_size");

        /*
         * It does, so return the home lgroup.
         */

        if (size > 0)
                return (home);

        /*
         * Otherwise, find next nearest lgroup outside of the home.
         */

        return (lgrp_next_nearest(cookie, home));
}
```