Tru64 UNIX

NUMA Overview

Part Number: AA-NUMAG-DE

January 2001

Operating System and Version: Tru64 UNIX Version 5.1 or higher

This document introduces the operating system features that support Non-Uniform Memory Access (NUMA) and explains when and how they are used.

Compaq Computer Corporation
Houston, Texas

© 2001 Compaq Computer Corporation

COMPAQ, the Compaq logo, and AlphaServer Registered in U.S. Patent and Trademark Office. Tru64 is a trademark of Compaq Information Technologies Group, L.P.

Microsoft Windows and Windows NT are trademarks of Microsoft Corporation. UNIX and The Open Group are registered trademarks of The Open Group. All other product names mentioned herein may be trademarks of their respective companies.

Confidential computer software. Valid license from Compaq required for possession, use, or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

Compaq shall not be liable for technical or editorial errors or omissions contained herein. The information in this document is subject to change without notice.

THE INFORMATION IN THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND. THE ENTIRE RISK ARISING OUT OF THE USE OF THIS INFORMATION REMAINS WITH RECIPIENT. IN NO EVENT SHALL COMPAQ BE LIABLE FOR ANY DIRECT, CONSEQUENTIAL, INCIDENTAL, SPECIAL, PUNITIVE, OR OTHER DAMAGES WHATSOEVER (INCLUDING WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION OR LOSS OF BUSINESS INFORMATION), EVEN IF COMPAQ HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES AND WHETHER IN AN ACTION OF CONTRACT OR TORT, INCLUDING NEGLIGENCE.

The limited warranties for Compaq products are exclusively set forth in the documentation accompanying such products. Nothing herein should be construed as constituting a further or additional warranty.

Contents

Preface

1 NUMA Concepts

1.1 RADs and QBBs 1–2
1.2 RADs and Partitioning 1–6
1.3 RADs, Resource Allocation, and Process Scheduling 1–7

2 NUMA-Aware Applications

2.1 Default NUMA-Aware Behavior of the Operating System 2–1
2.2 NUMA APIs for User Applications 2–2
2.3 NUMA Memory Management Policies 2–11

A The radtool Program

Index

Examples

A–1 Source File for the radtool Program A–2
A–2 Header File for the radtool Program A–7
A–3 Makefile for the radtool Program A–7

Figures

1–1 QBBs 1–3
1–2 RAD/QBB Mapping 1–4
1–3 Partitioned NUMA System 1–6

Tables

2–1 RADs and RAD Sets 2–5
2–2 CPUs and CPU Sets 2–7
2–3 NUMA Scheduling Groups 2–8
2–4 Processes and Threads 2–9

2–5 Memory Management 2–10

Preface

This is a post-release document that is currently available only on line, in HTML and PDF formats, at the Compaq Tru64™ UNIX web site. This document is not orderable in printed form nor is it included on the Tru64 UNIX Version 5.1 documentation CD-ROM.

If you are using a web browser to read the HTML version of this document, you can click on documentation cross-references to display them. Some cross-references are to different sections of this document and some cross-references are to reference pages included in the Tru64 UNIX Version 5.1 documentation set. The reference pages display in a different window from sections in this document. You can therefore navigate among sections of this document in one window and from one reference page to another in the supplementary window.

Audience

This document is aimed at system administrators and programmers who will be using Tru64 UNIX Version 5.1 on NUMA platforms (GS80, GS160, and GS320 AlphaServer systems).

Conventions

This document uses the following conventions:

`%`
`$` — A percent sign represents the C shell system prompt. A dollar sign represents the system prompt for the Bourne, Korn, and POSIX shells.

`#` — A number sign represents the superuser prompt.

`file` — Italic (slanted) type indicates variable values, placeholders, and function argument names.

[|]
{ | } — In syntax definitions, brackets indicate items that are optional and braces indicate items that are required. Vertical bars separating items inside

	brackets or braces indicate that you choose one item from among those listed.
⋮	A vertical ellipsis indicates that a portion of an example that would normally be present is not shown.
...	In syntax definitions, a horizontal ellipsis indicates that the preceding item can be repeated one or more times.
`cat`(1)	A cross-reference to a reference page includes the appropriate section number in parentheses. For example, `cat`(1) indicates that you can find information on the `cat` command in Section 1 of the reference pages.

1

NUMA Concepts

On traditional multiprocessor systems, there is one interconnect, either a bus or a switch, that links all system resources. This means that all CPUs in the system are subject to the same latency and bandwidth restrictions with respect to accessing the system's memory and I/O channels. Uniform Memory Access (UMA) is a term sometimes used to describe the system architecture in which all CPUs access memory and I/O by using the same bus or switch. This document refers to systems that use UMA architecture as traditional symmetric multiprocessor (SMP) systems.

The drawback of the architecture of traditional SMP systems is that scaling the system to large numbers of CPUs causes the system bus to become a performance bottleneck. One way to address this bottleneck is to design a system built from SMP blocks (each with a limited number of CPUs, memory arrays, and I/O ports) and add a second-level bus or switch to connect the blocks. Non-Uniform Memory Access (NUMA) is the term used to describe this type of system architecture because it results in bandwidth and latency differences, depending on whether a particular CPU accesses memory and I/O resources locally (in the same building block where the CPU resides) or remotely (in another building block).

Hardware Requirements for NUMA Support

The first NUMA implementations did not support cache coherency between the system building blocks, only within them. On these early NUMA implementations, software was responsible for ensuring cache coherency when a CPU accessed memory in any building block other than the one in which the CPU was located.

The GS80, GS160, and GS320 AlphaServer systems are the first Alpha implementations of cache-coherent NUMA (CC-NUMA) systems, meaning that the system hardware handles cache coherency between the system building blocks as well as within them. Therefore, software is relieved of this responsibility. The operating system and user applications can treat a CC-NUMA system the same way they treat a traditional SMP system and still be programmatically correct.

Performance Implications of NUMA Support

Although software can treat a CC-NUMA system as a traditional SMP system and still be programmatically correct, obtaining optimal performance from a CC-NUMA system depends on appropriate use of its capabilities. In a network of systems, an application must sometimes run on a remote system rather than one local to the user. However, the application will almost always run faster on a local system that has all the resources needed by the application. This is because the connection between the systems increases response latency. The same principal applies when you consider a CC-NUMA multiprocessor system as a network of building blocks, each of which contains a set of CPUs, memory arrays, and I/O ports. The CPUs in one system building block can access memory that is available locally (in their own block) or remotely (in another block). However, using the local memory is faster because memory access through the interblock switch increases response latency.

Starting with Tru64 UNIX Version 5.1, the operating system includes kernel algorithms, utilities, and programming APIs that are NUMA aware. These algorithms and user interfaces are used to maximize the ratio of local to remote memory accesses and thereby help ensure optimal performance on CC-NUMA hardware.

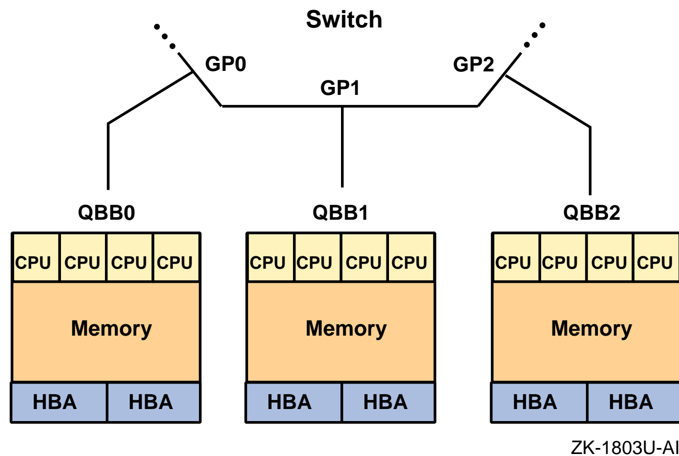
In most of the software product documentation pertaining to NUMA support, references to a NUMA system assume a CC-NUMA hardware implementation. Therefore, this document also refers to CC-NUMA systems as NUMA systems.

1.1 RADs and QBBs

On Tru64 UNIX systems, the building blocks that make up a NUMA system are mapped to structures called Resource Affinity Domains (RADs). A RAD identifies the set of CPUs, memory arrays, and I/O busses that, when used together, allow the system to work most efficiently.

Resource affinity domain, like most abstract concepts, is easier learned in the context of a concrete example. For that reason, it is worthwhile to consider what a RAD corresponds to in current hardware products, such as the GS80, GS160, and GS320 AlphaServer systems. On these systems, each SMP unit is called a Quad Building Block (QBB or QUAD), as shown in Figure 1–1.

Figure 1–1: QBBs



Each QBB can contain up to four CPUs, a set of memory arrays, and an I/O processor (IOP) that, through two host bus adapters (HBAs), accommodates two to eight I/O busses. An internal switch in each QBB allows all CPUs equal access to both local memory and the I/O busses connected to the local I/O processor. An application running on a CPU in one QBB accesses the memory in another QBB by routing through the global port (GP) of the local QBB and the global port (GP) of the other QBB. On larger NUMA systems (GS160 and GS320), access is also routed through a Hierarchical Switch (sometimes called the HSwitch or Global Switch) that connects the global ports of all QBBs. Therefore, the remote/local response latency between QBBs is on the order of 2/1 or 3/1, depending on the type of system.

Although the operating system supports transparent resource access through the global ports and the HSwitch, performance is optimized when process operations use memory and I/O channels in the same QBB as the CPU where the process is running. Therefore, the CPUs, memory arrays, and I/O busses in the same QBB are viewed by the operating system as having affinity for one another and are included in the same RAD. Starting with Tru64 UNIX Version 5.1, the operating system makes a best effort to:

- Schedule all threads of a multithreaded application on CPUs in the same RAD
- Allocate memory for each process or application thread in the same RAD as the CPU where the process or thread is running

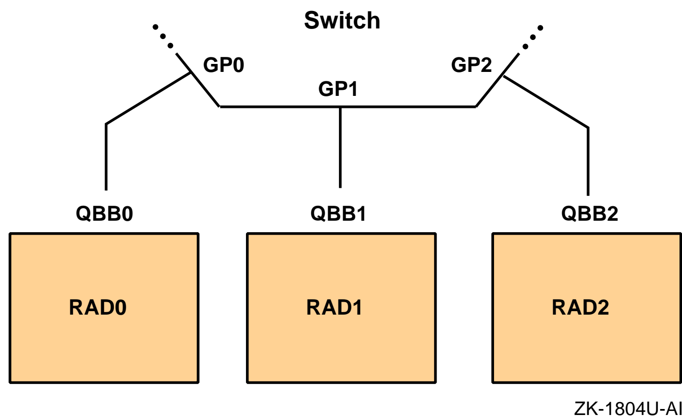
The default NUMA-aware algorithms for scheduling and allocating resources to a process or thread work well when the resources in one RAD can accommodate the number of threads and the memory demands in any one application.

The NUMA application programming interfaces (APIs) allow applications to make scheduling and resource allocation decisions based on advance knowledge of the application's resource needs and behavior. Proper manipulation of system resources and process scheduling through NUMA APIs has the following potential advantages:

- A master application can distribute associated applications among available QBBs in a way that will ensure each the most likelihood of using CPU cycles, memory cache, and I/O channels of the same QBB.
- An application can notify the operating system of relationships between processes and threads that should be scheduled on the same RAD and, if migration to another RAD becomes advantageous, must be moved together.
- A very large and complex application whose resource demands and number of threads exceed the capacity of one QBB can stripe its CPU cycles, I/O load, and the memory that contains program data across QBBs.

Figure 1–2 shows how a RAD maps to a QBB.

Figure 1–2: RAD/QBB Mapping



Applications can assign themselves to a particular RAD. In addition, system administrators can move applications to a RAD by using the `-r` option of the `runon` command. However, to be portable and maintainable, applications and scripts should not bind themselves to hardware topology. In particular, applications and system administration scripts should never depend on the existence of a particular RAD identifier, such as 0, 1, or 2.

It is important to emphasize two points about NUMA platforms:

- A RAD is a more generic concept than a QBB.

- In most cases, it is not necessary for programmers to rewrite existing applications or for system administrators to assign applications to specific RADs to obtain good performance on NUMA platforms running a mix of applications. Use of NUMA APIs and RAD-specific scheduling by a system administrator are recommended only for specific cases.

A RAD is used by software to identify and use optimal combinations of run-time resource combinations, whereas a QBB is a physical building block for a particular implementation of NUMA hardware architecture. RAD structures and NUMA programming functions are designed to be hardware independent so they will support different hardware architectures. On future NUMA AlphaServer platforms, CPUs, memory arrays, and I/O busses might not be grouped into QBBs. Therefore, RADs support application portability among different NUMA platforms.

This application portability also applies to traditional SMP systems, such as those in the ES and DS families of AlphaServer systems. Starting with Tru64 UNIX Version 5.1, NUMA-aware applications can run on these SMP systems by handling them as single-RAD systems. On single-RAD systems, the only RAD that exists is RAD0, which contains all the CPUs, memory arrays, and I/O channels in the system. There is no performance advantage to running NUMA-aware applications on a single-RAD system (all resources are treated as being equidistant from one another). However, application portability is preserved as long as the application is designed to:

- Query the configuration to get information about available RADs
- Use only the RADs that are currently available

The NUMA structures and functions discussed in Chapter 2 are analogous to those used by the operating system software. Although published for use by programmers, NUMA APIs should be used only in specialized layered software, such as databases, transaction processing products, or high performance technical computing applications, for which dedicated use and control of system resources are appropriate. For system administrators, there are also tuning parameters that can be adjusted to customize memory allocation on a per-RAD basis. However, even RAD-specific tunable parameters are best left to be automatically set by the operating system (or, if reset, be the same value for all RADs) unless all user applications being run on the system are NUMA aware. Therefore, RAD-specific system tuning should be used only when the NUMA system (or one of its hardware partitions) is dedicated to running NUMA-aware applications. (See Section 1.2 for more information about partitioning.)

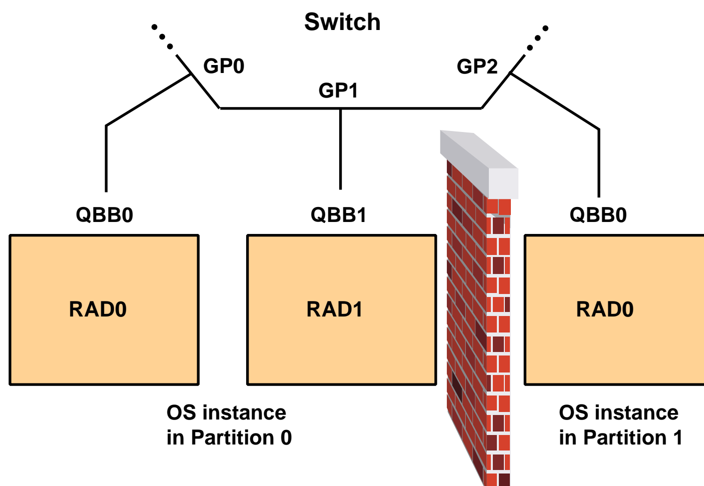
1.2 RADs and Partitioning

Partitioning refers to dividing a system into two or more resource groups, or partitions, each partition containing CPUs, memory, and I/O channels. After a system is partitioned, each partition is used independently of the others.

Theoretically, partitioning can be done through hardware (hard partitioning) or operating system software (soft partitioning). Hard partitioning is currently supported on the GS160 and GS320 AlphaServer systems. The operating system provides resource management features that do not cross partition boundaries.

A partition on a GS160 or GS320 AlphaServer system can contain one or more QBBs and is set up through a hardware console facility. When a system is partitioned, an operating system is installed in each partition. Therefore, a GS320 with two partitions can be running two instances of the same version of Tru64 UNIX software, two different versions of Tru64 UNIX software, or two entirely different operating systems. As shown in Figure 1–3, each operating system instance is essentially firewalled from access to resources in any hard partition except the one in which it is running.

Figure 1–3: Partitioned NUMA System



The instance of Tru64 UNIX software that is running in Partition 0 can access two QBBs whereas the instance that is running in Partition 1 can access 1 QBB. In each partition, QBB numbering and RAD numbering starts at 0 and are unique only within the same partition. An operating system does not have access to information about RADs or their associated QBBs

in any hard partition save the one in which it is installed. Even firmware upgrades must be installed independently on each hard partition.

This point is important when using NUMA user and programming interfaces provided by the operating system. When the operating system is installed in a partition, queries about the number of RADs or CPU slots available in the system return the number of RADs available in the partition.

Recognition that the system contains three QBBs occurs at the hardware level through the HSwitch, which recognizes each QBB through its unique global port identifier. System operators can access platform-wide information through an external System Management Console (SMC). Operating system instances that are installed on any of the partitions do not have access to the SMC.

1.3 RADs, Resource Allocation, and Process Scheduling

Starting with Tru64 UNIX Version 5.1, system administrators can use the `-r` option of the `runon` command to execute an application on a specific RAD.

The commands used to create and use processor sets (psets) on traditional SMP systems can also be used to advantage on NUMA systems. However, system administrators should factor in the RAD locations of CPUs when using pset interfaces on NUMA systems. On a NUMA system, it is important to define a pset to contain processors in the same RAD or (if more than four processors are required) in the fewest number of RADs that are needed to meet the resource requirements of the applications to be run on the processor set. This practice optimizes performance on NUMA systems because it maximizes the ratio of local-to-remote memory accesses. System administrators and programmers must apply this principle both to processor sets being defined for new applications and processor sets being defined for applications that previously ran on traditional SMP systems.

See `runon`(1) for more information about the `runon` command.
See `pset_create`(1) for information about creating a pset and for cross-references to other pset-related reference pages.

Note

Currently, system administrators cannot determine the RAD location of a CPU through a command-line or graphical interface. (For programs, the `rad_get_cpus()` function returns this information.) However, Appendix A contains the source code for a utility that queries the system for RAD and CPU identifiers. Sites can copy, adapt, and build this program for local use.

For the GS80, GS160, and GS320 AlphaServer systems, CPU identifiers and RAD identifiers have a fixed relationship, such

that CPUs 0 to 3 are in RAD0, CPUs 4 to 7 are in RAD1, and so forth. Therefore, system administrators can assume for use in the `pset_assign_cpu` command that this fixed relationship of CPU numbers to RAD numbers is valid. However, the fixed relationship of these numbers is not likely to apply to future NUMA AlphaServer systems. Therefore, users should not write scripts or programs that assume a fixed relationship of CPU numbers to RAD numbers if they want these scripts and programs to be portable to future generations of NUMA AlphaServer systems.

RAD0 always contains the boot CPU on GS80, GS160, and GS320 AlphaServer systems. However, this assumption, too, is not likely to apply to future generations of NUMA AlphaServer systems.

The NUMA APIs are used to:

- Identify and query the number of existing RADS and the availability of resources in these RADs
- Schedule processes and threads to run in RADs that offer the appropriate balance of available CPU cycles and memory for what the processes will be doing

See Section 2.2 for a summary of the library routines associated with NUMA-aware resource allocation and process scheduling.

The NUMA APIs are recommended for new versions of applications that currently create and manipulate psets if those applications will run on NUMA AlphaServer systems as well as traditional SMP systems. Existing applications that use the functions `create_pset()`, `destroy_pset()`, `assign_cpu_to_pset()`, `assign_pid_to_pset()`, and `print_pset_error()` do not require changes to be able to run on NUMA AlphaServer systems. However, the manner in which existing programs assign CPUs to a processor set does not take into account the recommended practice of maximizing the ratio of local-to-remote memory accesses on a NUMA system. If this ratio is not as good as possible (given the amount of local memory that is available), the application does not achieve optimal performance.

For this reason, new applications designed for use on NUMA systems or on both traditional SMP and NUMA systems should use NUMA APIs. Reference pages for these APIs are listed in Section 2.2.

2

NUMA-Aware Applications

Starting with Tru64 UNIX Version 5.1, Tru64 UNIX software is composed of NUMA-aware programs. Therefore, the majority of user applications do not have to use NUMA APIs to achieve reasonable performance on NUMA systems. However, certain user applications might be optimized through direct use of these APIs. This chapter describes the default NUMA-aware behavior in the operating system, and provides an overview of the NUMA APIs that applications can use directly.

2.1 Default NUMA-Aware Behavior of the Operating System

Starting with Tru64 UNIX Version 5.1, the following defaults are in place to increase the likelihood that NUMA system resources are used efficiently for most types of applications:

- The operating system defines a “home RAD” for each process and all its threads. Default process or thread scheduling and memory allocation are done on the assigned home RAD whenever possible.

  In other words, the operating system attempts to schedule a process and all its threads on CPUs in the home RAD. Furthermore, the operating system attempts to allocate memory for application and kernel data on the home RAD. The cache affinity algorithms previously available only for traditional SMP systems are also used. Therefore, if a thread that previously ran on a particular CPU needs to be scheduled, the operating system attempts to schedule that thread on the same CPU.

  The operating system also defines a default overflow set of RADs. When there is insufficient free memory for application and kernel data on the home RAD, the operating system attempts to allocate memory from one or more remote RADs based on the default overflow set.

- For data that is globally accessed, the operating system attempts to replicate the data in or stripe it across all RADs where it might be accessed. More specifically, the operating system attempts to:

  – Replicate kernel code and kernel read-only data on all RADs at boot time

  – Replicate other kinds of read-only data, such as shared program and library code, on all RADs where a running process or thread needs to access it

If there is insufficient free memory on the RAD where the process or thread is running to replicate shared, read-only data, the operating system will utilize a copy on a remote RAD rather than wait for free memory on the local RAD to make the copy.

  – Stripe System V shared memory (which is not read-only) across all RADs

    Striping minimizes the likelihood that certain processes and threads always access System V shared memory locally while others always access it remotely.

- The operating system attempts to balance the load on each RAD so that local CPU cycles and local memory pages are both available to the processes running on the RAD.

  Local availability of memory and CPU cycles influences RAD selection at the time a process is created. The same factors might cause the operating system to migrate a process and associated memory pages from one RAD to another in response to changing resource requirements and access patterns.

2.2 NUMA APIs for User Applications

When a mix of applications with differing resource needs are run on the same system, it is best for user applications to rely on the default behavior of operating system software. However, large and highly specialized user applications might realize additional performance advantages through direct use of NUMA APIs. For example:

- An application for which I/O requests are extremely large might realize significant performance advantages when the CPU cycles and memory pages associated with an I/O request are striped across all available RADs. This optimization strategy works only if the data being read from or written to disk is also striped across controllers that are attached to the I/O ports of different RADs. (If I/O ports on different RADs channel data into the same RAID controllers, device latency will likely offset the bandwidth increase for CPU cycles and memory.)
- Applications with many subprocesses or threads that operate on large but different subsets of the same data might benefit from explicit resource management. In this case, NUMA APIs can help to increase the ratio of local to remote accesses by changing the default algorithms for replicating or striping program data and System V shared memory.

NUMA APIs are included in the following libraries:

- The NUMA Library (`libnuma`)
- The Standard C Library (`libc`)

Certain routines required for NUMA-aware programming are included in the `libc` library because they perform operations that are also useful in more generic types of programs.

- The POSIX Threads Library (`libpthread`)

  NUMA routines that are useful only in multithreaded programs are included in the `libpthread` library.

The NUMA data types, structures, and function prototypes are defined by including the `numa.h` header file. These APIs introduce three new constructs:

- RAD set

  A RAD set is a mechanism for passing information about RADs between an application and the operating system or between two applications. For example, an application can use a RAD set to query the operating system about the number of existing RADs. An application can also specify a RAD set to pass information about the number of RADs needed to meet application resource requirements.

  In traditional applications, the identifiers for system components are typically returned as a bit mask that is stored in a word or longword buffer. A fixed-length buffer limits the number of components that can be identified to the number of bits in the buffer (32 or 64 for a word or longword, respectively). However, a RAD set is represented by an opaque data type so that applications do not include a fixed-length buffer for querying or passing information about RADs.

- CPU set

  A CPU set is also a mechanism for passing information about CPUs between an application and the operating system or between two applications. Like a RAD set, a CPU set is represented by an opaque data type.

  A CPU set is different from the processor set (pset) that is created and manipulated by the APIs and commands already in use on traditional SMP systems. A pset reserves specific CPUs for use only by user-specified applications, while a CPU set is simply an information-passing mechanism.

  A NUMA-aware application that requests allocation of system resources uses RAD sets and CPU sets to ensure that CPUs and memory are evaluated and used in the context of the RADs in which these resources are located. If the application intends to isolate some number of CPUs on the system for exclusive use by one or more key processes, the application first queries the number of RADs on the system and the RAD locations of the available CPUs. In almost all cases, the CPUs selected for a processor set should be from the same RAD or, if any one RAD

has an insufficient number of CPUs for the expected workload, from the fewest number of RADs. If the application runs on a traditional SMP system, all available CPUs are in a single RAD; however, the NUMA-aware logic for evaluating and using information about system processors remains the same.

- NUMA Scheduling Group (NSG)

  A NUMA Scheduling Group is the construct through which a NUMA-aware application ensures that a related set of processes and threads execute on the same RAD.

  An application can attach the identifiers of one RAD and of one or more processes to a NUMA Scheduling Group. By doing this, the application specifies that:

  – All those processes and any of their subprocesses or threads must execute on the same RAD

  – In the event that any of the processes or threads must be moved to a new RAD, all other processes and threads attached to the NUMA Scheduling Group are moved as well

See `numa_types(4)` for a detailed description of the data types, structures, and macros used with the NUMA functions. See `numa_scheduling_groups(4)` for a description of a NUMA scheduling group.

NUMA functions can be grouped into categories according to what is being queried or used. The tables referred to in the following list include the name, purpose, library, and reference page for each routine in the category. Some routines are duplicated in two tables because they query or create a relationship that spans two categories:

- RADs and RAD sets: Table 2–1
- CPUs and CPU sets: Table 2–2
- NUMA Scheduling Groups: Table 2–3
- Processes and threads: Table 2–4
- Memory management: Table 2–5

  See Section 2.3 for a summary of policies for NUMA memory management.

Table 2–1: RADs and RAD Sets

Function	Purpose	Library	Reference Page
nloc()	Returns the RAD set that is local or remote to a resource.	libnuma	nloc(3)
rad_attach_pid()	Attaches a process to a RAD (assigns a home RAD but allows execution on other RADs).	libnuma	rad_attach_pid(3)
rad_bind_pid()	Binds a process to a RAD (assigns a home RAD and restricts execution to the home RAD).	libnuma	rad_attach_pid(3)
rad_foreach()	Scans a RAD set for members and returns the first member found.	libnuma	rad_foreach(3)
rad_get_current_home()	Returns the caller's home RAD.	libnuma	rad_get_current_home(3)
rad_get_cpus()	Returns the set of CPUs that are in a RAD.	libnuma	rad_get_num(3)
rad_get_freemem()	Returns a snapshot of the free memory pages that are in a RAD.	libnuma	rad_get_num(3)
rad_get_info()	Returns information about a RAD, including its state (online or offline) and the number of CPUs and memory pages it contains.	libnuma	rad_get_num(3)
rad_get_max()	Returns the number of RADs in the system. [a]	libnuma	rad_get_num(3)
rad_get_num()	Returns the number of RAD's in the caller's partition. [a]	libnuma	rad_get_num(3)
rad_get_physmem()	Returns the number of memory pages assigned to a RAD.	libnuma	rad_get_num(3)
rad_get_state()	Reserved for future use. (Currently, RAD state is always set to ONLINE.)	libnuma	rad_get_num(3)
radaddset()	Adds a RAD to a RAD set.	libnuma	radsetops(3)

Table 2–1: RADs and RAD Sets (cont.)

Function	Purpose	Library	Reference Page
radandset()	Performs a logical AND operation on two RAD sets, storing the result in a RAD set.	libnuma	radsetops(3)
radcopyset()	Copies the contents of one RAD set to another RAD set.	libnuma	radsetops(3)
radcountset()	Returns the members of a RAD set.	libnuma	radsetops(3)
raddelset()	Removes a RAD from a RAD set.	libnuma	radsetops(3)
raddiffset()	Finds the logical difference between two RAD sets, storing the result in another RAD set.	libnuma	radsetops(3)
rademptyset()	Initializes a RAD set such that no RADs are included.	libnuma	radsetops(3)
radfillset()	Initializes a RAD set such that it includes all RADs.	libnuma	radsetops(3)
radisemptyset()	Tests whether a RAD set is empty.	libnuma	radsetops(3)
radismember()	Tests whether a RAD belongs to a given RAD set.	libnuma	radsetops(3)
radorset()	Performs a logical OR operation on two RAD sets, storing the result in another RAD set.	libnuma	radsetops(3)
radsetcreate()	Allocates a RAD set and sets it to empty.	libnuma	radsetops(3)
radsetdestroy()	Releases the memory allocated for a RAD set.	libnuma	radsetops(3)
radxorset()	Performs a logical XOR operation on two RAD sets, storing the result in another RAD set.	libnuma	radsetops(3)

[a] On a partitioned system, the system and the partition are equivalent. In this case, the operating system returns information only for the partition in which it is installed.

Table 2–2: CPUs and CPU Sets

Function	Purpose	Library	Reference Page
cpu_foreach()	Enumerates the members of a CPU set.	libc	cpu_foreach(3)
cpu_get_current()	Returns the identifier of the current CPU on which the calling process is running.	libc	cpu_get_current(3)
cpu_get_info()	Returns CPU information for the system. [a]	libc	cpu_get_info(3)
cpu_get_max()	Returns the number of CPU slots available in the caller's partition. [a]	libc	cpu_get_info(3)
cpu_get_num()	Returns the number of available CPUs.	libc	cpu_get_info(3)
cpu_get_rad()	Returns the RAD identifier for a CPU.	libnuma	cpu_get_rad(3)
cpuaddset()	Adds a CPU to a CPU set.	libc	cpusetops(3)
cpuandset()	Performs a logical AND operation on the contents of two CPU sets, storing the result in a third CPU set.	libc	cpusetops(3)
cpucopyset()	Copies the contents of one CPU set to another CPU set.	libc	cpusetops(3)
cpucountset()	Returns the number of CPUs in a CPU set.	libc	cpusetops(3)
cpudelset()	Deletes a CPU from a CPU set.	libnuma	cpusetops(3)
cpudiffset()	Finds the logical difference between two CPU sets, storing the result in a third CPU set.	libnuma	cpusetops(3)
cpuemptyset()	Initializes a CPU set such that it includes no CPUs.	libnuma	cpusetops(3)
cpufillset()	Initializes a CPU set such that it includes all CPUs.	libnuma	cpusetops(3)
cpuisemptyset()	Tests whether a CPU set is empty.	libnuma	cpusetops(3)
cpuismember()	Tests whether a CPU is a member of a particular CPU set.	libnuma	cpusetops(3)

Table 2–2: CPUs and CPU Sets (cont.)

Function	Purpose	Library	Reference Page
cpuorset()	Performs a logical OR operation on the contents of two CPU sets, storing the result in a third CPU set.	libnuma	cpusetops(3)
cpusetcreate()	Allocates a CPU set and sets it to empty.	libnuma	cpusetops(3)
cpusetdestroy()	Releases the memory allocated to a CPU set.	libnuma	cpusetops(3)
cpuxorset()	Performs a logical XOR operation on the contents of two CPU sets, storing the result in a third CPU set.	libnuma	cpusetops(3)

[a] On a partitioned system, the system and the partition are equivalent. In this case, the operating system returns information only for the partition in which it is installed.

Table 2–3: NUMA Scheduling Groups

Function	Purpose	Library	Reference Page
nsg_attach_pid()	Attaches a process to a NUMA scheduling group.	libnuma	nsg_attach_pid(3)
nsg_destroy()	Removes a NUMA scheduling group and deallocates its structures.	libnuma	nsg_destroy(3)
nsg_detach_pid()	Detaches a process from a NUMA scheduling group.	libnuma	nsg_attach_pid(3)
pthread_nsg_attach()	Attaches a thread to a NUMA scheduling group.	libpthread	pthread_nsg_attach(3)
pthread_nsg_detach()	Detaches a thread from a NUMA scheduling group.	libpthread	pthread_nsg_detach(3)
nsg_get()	Returns the status of a NUMA scheduling group.	libnuma	nsg_get(3)
nsg_get_nsgs()	Returns a list of NUMA scheduling groups that are active.	libnuma	nsg_get_nsgs(3)

Table 2–3: NUMA Scheduling Groups (cont.)

Function	Purpose	Library	Reference Page
nsg_get_pids()	Returns a list of processes attached to a NUMA scheduling group.	libnuma	nsg_get_pids(3)
nsg_init()	Looks up (and possibly creates) a NUMA scheduling group.	libnuma	nsg_init(3)
nsg_set()	Sets group ID, user ID, and permissions for a NUMA scheduling group.	libnuma	nsg_set(3)
pthread_nsg_get()	Returns a list of threads attached to a NUMA scheduling group.	libpthread	pthread_nsg_get(3)

Table 2–4: Processes and Threads

Function	Purpose	Library	Reference Page
nfork()	Creates a child process that is an exact copy of its parent process. See also the table entry for rad_fork().	libnuma	nfork(3)
nmadvise()	Tells the system what behavior to expect from a process with respect to referencing mapped files and shared memory regions.	libnuma	nmadvise(3)
nsg_attach_pid()	Attaches a process to a NUMA scheduling group.	libnuma	nsg_attach_pid(3)
nsg_detach_pid()	Detaches a process from a NUMA scheduling group.	libnuma	nsg_attach_pid(3)
pthread_nsg_attach()	Attaches a thread to a NUMA scheduling group.	libpthread	pthread_nsg_attach(3)
pthread_nsg_detach()	Detaches a thread from a NUMA scheduling group.	libpthread	pthread_nsg_detach(3)

Table 2–4: Processes and Threads (cont.)

Function	Purpose	Library	Reference Page
pthread_rad_attach()	Attaches a thread to a RAD set.	libpthread	pthread_rad_attach(3)
pthread_rad_bind()	Attaches a thread to a RAD set and restricts its execution to the home RAD.	libpthread	pthread_rad_attach(3)
pthread_rad_detach()	Detaches a thread from a RAD set.	libpthread	pthread_rad_detach(3)
rad_attach_pid()	Attaches a process to a RAD (assigns a home RAD but allows execution on other RADs).	libnuma	rad_attach_pid(3)
rad_bind_pid()	Binds a process to a RAD (assigns a home RAD and restricts execution to the home RAD).	libnuma	rad_attach_pid(3)
rad_fork()	Creates a child process on a RAD that optionally does not inherit the RAD assignment of its parent. See also the table entry for nfork().	libnuma	rad_fork(3)

Table 2–5: Memory Management

Function	Purpose	Library	Reference Page
memalloc_attr()	Returns the memory allocation policy for a RAD set specified by its virtual address.	libnuma	memalloc_attr(3)
nacreate()	Sets up an arena [a] for memory allocation for use with the amalloc() function.	libc	amalloc(3)

Table 2–5: Memory Management (cont.)

Function	Purpose	Library	Reference Page
nmadvise()	Tells the system what behavior to expect from a process with respect to referencing mapped files and shared memory regions.	libnuma	nmadvise(3)
nmmap()	Maps an open file (or anonymous memory) onto the address space for a process by using a specified memory allocation policy.	libnuma	nmmap(3)
nshmget()	Returns or creates the ID for a shared memory region.	libnuma	nshmget(3)

[a] An arena is used in multithreaded programs when there is a need for thread-specific heap memory allocation.

2.3 NUMA Memory Management Policies

Starting with Tru64 UNIX Version 5.1, application programmers can choose among the following policies to control memory allocation on NUMA systems. The policies can be specified for either a specific memory object or a kernel memory allocation request by using one of the following memory allocation attributes that are defined in the `numa_types.h` file:

`MPOL_DIRECTED`
: Allocate memory from a specific RAD (directed allocation)

`MPOL_THREAD`
: Allocate memory from the current thread's home RAD (directed allocation that operates in the context of a multithreaded application)

`MPOL_STRIPED`
: Stripe application data across the memory in a specified RAD set

`MPOL_REPLICATED`
: Replicate application data in the memory of all RADs

These major attributes have several associated attributes for further refinement of the memory allocation policy. For the directed and thread-related memory allocation policies, an application programmer

can define an overflow set of RADs for use when sufficient resources are unavailable in the preferred RAD. For the striped memory allocation policy, a programmer can define the number of pages for the stripe width (stride). To request that the operating system not migrate already allocated pages to another RAD, a programmer can combine the major policy attributes with the `MPOL_NO_MIGRATE` attribute.

When the NUMA memory allocation policy is not set by the application, the operating system applies the following defaults for different parts of an application's address space:

- Memory for private data, such as the heap and stacks for processes and threads, is allocated from the home RADs of the processes and threads.
- Program text and shared libraries are replicated in all RADs
- Shared data, such as System V shared memory, is striped (using a one-page stride) across all RADs

To override these defaults, an application programmer can use the following functions:

- The `nmmap()` function to override the default policy for a new file object or an already mapped range of addresses for an open file
- The `nshmget()` function to override the default policy for an already mapped address range of shared memory
- The `nmadvise()` function to override the default policy used for process access to an already mapped range of address space for an open file or a region of shared memory

The `nmmap()`, `nshmget()`, and `nmadvise()` functions include a parameter of type `memalloc_attr_t` to contain the NUMA memory allocation policy and associated attribute values. When this argument is null, the `nmmap()`, `nshmget()`, and `nmadvise()` functions have the same behavior as their traditional counterparts (`mmap()`, `shmget()`, and `madvise()`, respectively).

Note

A memory allocation policy request on any UNIX platform (NUMA or traditional SMP) is not implemented in an absolute manner. UNIX architecture is designed for efficient sharing of resources among system and user processes rather than dedicated resource assignments, particularly where memory is concerned. This means that the operating system does not allow the policies requested by or for any one application to completely override the minimal memory requirements of other user and system processes that are running at the same time. Therefore, to

ensure consistent implementation of the memory allocation policy requested through NUMA APIs, a NUMA-aware application should be run on a system (or system partition) that contains sufficient memory resources for both the application's processes and any other processes that will be running at the same time.

A

The radtool Program

This appendix contains the source code (Example A–1 and Example A–2) and the Makefile (Example A–3) for the `radtool` utility. This utility queries the system for identifiers of available RADs and for identifiers of CPUs in a specified RAD. The source code for this tool illustrates the use of several NUMA APIs that all NUMA-aware programs will need to use. In addition, the `radtool` utility can be built and installed on a customer system, then used by system administrators and site-specific scripts to avoid dependence on static assignments of RAD numbers and of CPU numbers within RADs.

The command-line synopsis for `radtool` is as follows:

path`/radtool` [`-x`] | [[`-v`] [`-r` | `-c` *rad-id*]]

Where:

`-x`	Displays the utility's usage message.
`-v`	Displays descriptive headings and comma separators for returned values.
`-r`	Returns identifiers of existing RADs. This is also the behavior when the command is entered without any options.
`-c` *rad-id*	Returns identifiers of available CPUs in the specified RAD.

The program header file for this example is a template and is included for possible site enhancements. (The header file does not supply definitions used in this version of the program but is referred to in the Makefile.) For example, the program might be internationalized to support message catalogs for translated messages and also include a header file that is created by the `mkcatdefs` command. In this case, the header file will be renamed `radtool_msg.h` and will define macros for default message strings. See `mkcatdefs`(1) for more information about creating message catalogs and a header file that centralizes maintenance of default message strings.

Example A–1: Source File for the radtool Program

```
/*
 * radtool.c -- NUMA API Example program
 *
 *
 */
#include <sys/types.h>
#include <sys/time.h>
#include <sys/siginfo.h>

#include <errno.h>
#include <stdarg.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#include <numa.h>
#include <cpuset.h>
#include <radset.h>
#include "radtool.h"

/*
 * command-line options:
 *
 * -r         = display existing RADs
 * -c <radid> = display CPUs for specified RAD
 * -x         = display eXplanation.
 */
#define OPTIONS "c:rvx"

/*
 * command-line settable parameters and flags:
 */
bool show_rads    = false;   /* display existing RADs */
bool verbose      = false;   /* annotate output */
radid_t parm_rad = RAD_NONE; /* display CPUs for this RAD */

/*
 * usage/help message
 */
char *USAGE = "\nUsage:  %s {[-r] | [-c <radid>]} [-v] | [-x]\n\n\
Where:\n\
\t-r         = Display existing RADs. Same as no arguments.\n\
\t-c <radid> = Display CPUs for specified RAD.\n\
\t-v         = Include formatting text in display.\n\
\t-x         = Display this explanation.\n\
";
char *cmd;

bool  error = false;

/*
 * die() - Emit error message and exit w/ specified return code.
 *    If exit_code < 0, save current errno, and fetch associated
 *    error string. Print error string after app error message.
 *    Then exit with abs(exit_code).
 */
void
die(int exit_code, char *format, ... )
{
 va_list ap;
 char *errstr;
```

Example A–1: Source File for the radtool Program (cont.)

```
 int saverrno;

 va_start(ap, format);

 if (exit_code < 0) {
  saverrno = errno;
  errstr = strerror(errno);
 }

 (void) vfprintf(stderr, format, ap);
 va_end(ap);

 if (exit_code < 0)
  fprintf(stderr,"Error = (%d) %s\n", saverrno, errstr);

 exit(abs(exit_code));
}

void
usage(){
 fprintf(stderr, USAGE, cmd);
 exit(1);
}


/*
 * ========================================================================
 */
/*
 * rad_get_existing() -- return set of currently existing rads, using nloc()
 *
 * NOTE: It is the caller's responsibility to free the returned radset when it
 *       is no longer needed. See the call to radsetdestroy() at the end
 *       of the program.
 */
radset_t
rad_get_existing()
{
 radset_t allrads;
 numa_attr_t nat;

 if(radsetcreate(&allrads) == -1) {
  die(0-errno, "Unable to create radset for allrads\n");
 }

 /*
  * This works for Tru64 UNIX Version 5.1.
  * It returns the set of RADs that are <= RAD_DIST_REMOTE from an
  * empty RAD set. All existing RADs satisfy this relationship.
  */
 nat.nattr_type         = R_RAD;
 nat.nattr_descr.rd_radset = allrads;
 nat.nattr_distance     = RAD_DIST_REMOTE;
 nat.nattr_flags        = 0;

 if(nloc(&nat, allrads) == -1) {
  die(0-errno, "rad_get_existing: failure to get allrads\n");
 }

 return(allrads);
}
```

Example A–1: Source File for the radtool Program (cont.)

```
/*
 * radshowset():  display a RAD set
 *
 * The "note" parameter is for annotating the display with text
 * to indicate what the returned numbers represent.
 * If "note" is NULL, RAD numbers are printed in a single line,
 * separated by whitespace. The latter case is useful for returning
 * values to commands in a shell script, for example:
 *
 * for rad in `radtool -r`; do whatever; done
 *
 */
void
radshowset(const radset_t set, const char *note)
{
 radid_t id;
 rad_cursor_t cursor = SET_CURSOR_INIT;
 int flags = 0;
 int i;

 if (note != NULL)
  printf("\n%s:\n", note);

 if (radisemptyset(set)) {
  if (note != NULL)
   fprintf(stderr, "\tNone");
  return;
 }

 for(i = 0;
   (id = rad_foreach(set, flags, &cursor)) != RAD_NONE;
  ++i) {
  if (note != NULL) {
   /*
    * "pretty print" - 8 to the bar
    */
   if((i % 8) == 0)
    printf("\n");
   else
     printf(", ");
  }
  printf("%3d", id);
 }
 printf("\n");
}
/*
 * cpushowset():  display the CPU set
 *
 * The "note" parameter is for annotating the display with text to
 * indicate what the returned numbers represent.
 * If "note" is NULL, CPU numbers are printed in a single line and
 * separated by whitespace. The latter case is useful for returning
 * values to commands in a shell script, for example:
 *
 * for cpu in `radtool -c 2`; do whatever; done
 */
void
cpushowset(const cpuset_t set, const char *note)
{
```

Example A–1: Source File for the radtool Program (cont.)

```
 cpuid_t id;
 cpu_cursor_t cursor = SET_CURSOR_INIT;
 int flags = 0;
 int i;

 if (note != NULL)
  printf("\n%s:\n", note);

 if (cpuisemptyset(set)) {
  if (note != NULL)
   fprintf(stderr, "\tNone");
  return;
 }

 for(i = 0;
   (id = cpu_foreach(set, flags, &cursor)) != CPU_NONE;
  ++i) {
  if (note != NULL) {
   /*
    * "pretty print" - 8 to the bar
    */
   if((i % 8) == 0)
    printf("\n");
   else
     printf(", ");
  }
  printf("%3d", id);
 }
 printf("\n");
}


/*
 * ==========================================================================
 */


void
main(int argc, char *argv[])
{
 extern int optind;
 extern char *optarg;
 char c;

 cmd = argv[0];

 /*
  * process command-line options.
  */
 while ((c = getopt(argc, argv, OPTIONS)) != (char)EOF ) {
  char *next;

  switch (c) {
  case 'c':
   parm_rad = strtoul(optarg, &next, 0);
   if (parm_rad < 0 || *next != '\0') {
    fprintf(stderr,
   "Error:  RAD identifier must be a positive integer\n");
    error = true;
   }
   break;
```

Example A–1: Source File for the radtool Program (cont.)

```
  case 'r':
   show_rads = true;
   break;

  case 'v':
   verbose = true;
   break;

  case 'x':
   usage();
   /* NOT REACHED */

  default:
   error = true;
   break;
  }
 }
done:

 if (error) {
  usage();
 }

 /*
  * If a number was specified, it must be the "-c" argument.
     * Display CPUs on the RAD with that number.
  */
 if (parm_rad != RAD_NONE) {
  cpuset_t cpus_in_rad;
  char note[32]; /* big enough for now */

  cpusetcreate(&cpus_in_rad);

  if (rad_get_cpus(parm_rad, cpus_in_rad) == -1)
   die(-2, "Unable to get CPUs in RAD %d\n", parm_rad);

  if (verbose)
   sprintf(note, "CPUs in RAD %d", parm_rad);

  cpushowset(cpus_in_rad, verbose ? note : NULL);

  cpusetdestroy(&cpus_in_rad);

  exit(0);

 } else {
  show_rads = true; /* show something! */
 }

 /*
  * Show all existing RADs, if requested.
  */
 if (show_rads) {
  radset_t allrads = rad_get_existing();

  radshowset(allrads, verbose ? "Existing RADs" : NULL);

  radsetdestroy(&allrads); /* be a good citizen */
 }
```

Example A–1: Source File for the radtool Program (cont.)

```
 exit(0);
}
```

Example A–2: Header File for the radtool Program

```
/*
 * radtool.h -- local header template for NUMA RAD tool program
 */

/*
 * a useful type definition -- for example:
 */
typedef enum {false=0, true} bool;
```

Example A–3: Makefile for the radtool Program

```
# Makefile template for NUMA sample programs
#
SHELL   = /bin/sh

MACH    =

CMODE = -std0
COPT = $(CMODE) -O2 #-non_shared
DEFS    =
INCLS   =
CFLAGS  = $(COPT) $(DEFS) $(INCLS) $(ECFLAGS)

CXX     = cxx
CXXMODE =
CXXFLAGS = $(CXXMODE) $(DEFS) $(INCLS) $(ECXXFLAGS)

# ASFLAGS for alpha assembler:
ASDEFS  = -DLANGUAGE_ASSEMBLY -D_LANGUAGE_ASSEMBLY -D__alpha
ASPIC =
ASCPP =
ASFLAGS = $(ASPIC) -tune generic $(ASCPP) $(ASDEFS) $(EASFLAGS)

LDOPTS = #-dnon_shared
LDLIBS = -lnuma
LDFLAGS = $(CMODE) $(LDOPTS) $(ELDFLAGS)

# export to environment as needed...
ROOTDIR = /
TMPDIR  = /var/tmp
COMP_HOST_ROOT =
COMP_TARGET_ROOT =

HDRS    = radtool.h

OBJS    = radtool.o

PROGS = radtool

#---------------------------------
```

Example A–3: Makefile for the radtool Program (cont.)

```
all:    $(PROGS)

radtool:  radtool.o
 $(CC) -o $@ $(LDFLAGS) $@.o $(LDLIBS)

$(OBJS):    $(HDRS)

install:
 @echo "Nothing to do..."

clean:
 -rm -f *.o core.[0-9]*

clobber: clean
 -rm -f  $(PROGS)
```

Index

A

APIs, NUMA
advantages of, 1–4
appropriate applications for, 1–5
categories of, 2–4
compared to SMP pset interfaces, 1–8
header file for including, 2–3
library locations, 2–2
portability issues, 1–5
purpose, 1–8
system defaults when APIs not used, 2–1
when to use, 2–2

C

cache coherency, 1–1
CPU
getting RAD location of, 1–7
CPU sets
APIs for, 2–7t
purpose, 2–3
cpu_foreach function, 2–7t
use in radtool example, A–2e
cpu_get_current function, 2–7t
cpu_get_info function, 2–7t
cpu_get_max function, 2–7t
cpu_get_num function, 2–7t
cpu_get_rad function, 2–7t
cpuaddset function, 2–7t
cpuandset function, 2–7t
cpucopyset function, 2–7t
cpucountset function, 2–7t
cpudelset function, 2–7t
cpudiffset function, 2–7t
cpuemptyset function, 2–7t
cpufillset function, 2–7t
cpuisemptyset function, 2–7t
use in radtool example, A–2e
cpuismember function, 2–7t
cpuorset function, 2–8t
cpusetcreate function, 2–8t
use in radtool example, A–2e
cpusetdestroy function, 2–8t
use in radtool example, A–2e
cpuxorset function, 2–8t

G

Global Port, 1–3
Global Switch
(*See* Hierarchical Switch (HSwitch))
GS80, GS160, and GS320 systems, 1–1
QBBs in, 1–2
RAD to QBB mapping, 1–7

H

Hierarchical Switch (HSwitch), 1–3

M

memalloc_attr function, 2–10t
memalloc_attr_t structure, 2–12
memory management
default system behavior, 1–3

NUMA APIs for, 2–10t
NUMA policies for, 2–11
response latency issues, 1–3
system tuning issues, 1–5
MPOL_* attributes, 2–11

N

nacreate function, 2–10t
nfork function, 2–9t
nloc function, 2–5t
use in radtool example, A–2e
nmadvise function, 2–11t
nmmap function, 2–12
Non-Uniform Memory Access, 1–1
nsg_attach_pid function, 2–8t
nsg_attrach_pid function, 2–9t
nsg_destroy function, 2–8t
nsg_detach_pid function, 2–8t, 2–9t
nsg_get function, 2–8t
nsg_get_nsgs function, 2–8t
nsg_get_pids function, 2–9t
nsg_init function, 2–9t
nsg_set function, 2–9t
NSGs
(*See* NUMA Scheduling Groups (NSGs))
nshmget function, 2–12
NUMA
(*See* Non-Uniform Memory Access)
NUMA Scheduling Groups (NSGs), 2–4
APIs for, 2–8t
purpose, 2–4

P

partitioning, 1–6
software implications, 1–7
processes
NUMA APIs for, 2–9t
processor sets
(*See* psets)
psets, 1–7
compared to CPU sets, 2–3
pthread_nsg_attach function, 2–8t, 2–9t
pthread_nsg_detach function, 2–8t, 2–9t
pthread_rad_attach function, 2–10t
pthread_rad_bind function, 2–10t
pthread_rad_detach function, 2–10t

Q

QBB
(*See* Quad Building Block)
QUAD
(*See* Quad Building Block)
Quad Building Block, 1–2

R

RAD
(*See* Resource Affinity Domains)
RAD sets
APIs for, 2–5t
purpose, 2–3
rad_attach_pid function, 2–5t, 2–10t
rad_bind_pid function, 2–5t, 2–10t
rad_foreach function, 2–5t
use in radtool example, A–2e
rad_fork function, 2–10t
rad_get_cpus function, 2–5t
use in radtool example, A–2e
rad_get_current_home function, 2–5t
rad_get_freemem function, 2–5t
rad_get_info function, 2–5t
rad_get_max function, 2–5t

rad_get_num function, 2–5t
rad_get_physmem function, 2–5t
rad_get_state function, 2–5t
radaddset function, 2–5t
radandset function, 2–6t
radcopyset function, 2–6t
radcountset function, 2–6t
raddelset function, 2–6t
raddiffset function, 2–6t
rademptyset function, 2–6t
radfillset function, 2–6t
radisemptyset function, 2–6t
  use in radtool example, A–2e
radismember function, 2–6t
radorset function, 2–6t
radsetcreate function, 2–6t
  use in radtool example, A–2e
radsetdestroy function, 2–6t
  use in radtool example, A–2e
radtool utility, A–1
  Makefile, A–7e
  radtool.c, A–2e
  radtool.h, A–7e
radxorset function, 2–6t
Resource Affinity Domains, 1–2
runon command, 1–7

S

scheduling
  (*See* NUMA Scheduling Groups (NSGs))

T

threads
  NUMA APIs for, 2–9t