

A NUMA API for LINUX*

Technical Linux Whitepaper

www.novell.com

April 2005

Disclaimer	Novell, Inc. makes no representations or warranties with respect to the contents or use of this document and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Please note that if you are compiling the original software or changing basic operating-system packages by yourself, the system may not be covered by the original maintenance program.		
Trademarks	Novell is a registered trademark of Novell, Inc. in the United States and other countries. SUSE is a registered trademark of SUSE LINUX Products GmbH, a Novell business. *Linux is a registered trademark of Linus Torvalds. AMD and Opteron are trademarks of Advanced Micro Devices. IBM and Power PC are registered trademarks and Power5 is a trademark of IBM Corporation. SGI and SGI Altix are registered trademarks of Silicon Graphics, Inc. HP is a trademark of Hewlett-Packard Company. Intel and Itanium are registered trademarks of Intel Corporation. All other third-party trademarks are the property of their respective owners.		
Copyright	Copyright 2005 Novell, Inc. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v1.0, 1999 or later.		
Addresses	<table> <tr> <td>Novell, Inc. 404 Wyman Street, Suite 500 Waltham, MA 02451 USA</td> <td>SUSE LINUX Products GmbH A Novell Business Maxfeldstr. 5 D-90409 Nürnberg Germany</td> </tr> </table>	Novell, Inc. 404 Wyman Street, Suite 500 Waltham, MA 02451 USA	SUSE LINUX Products GmbH A Novell Business Maxfeldstr. 5 D-90409 Nürnberg Germany
Novell, Inc. 404 Wyman Street, Suite 500 Waltham, MA 02451 USA	SUSE LINUX Products GmbH A Novell Business Maxfeldstr. 5 D-90409 Nürnberg Germany		
Prepared by	SUSE LINUX Products GmbH, A Novell Business		
Special thanks	Novell and SUSE LINUX would like to thank AMD for their contributions to and their support of the creation of this document		
Author	Andreas Kleen, Software Engineer, SUSE Labs		
Contributors	Richard Brunner, Fellow, AMD Mark Langsdorf, Technical Staff Member, AMD Meike Chabowski, Product Marketing Manager, SUSE LINUX		
Date	April 2005		

Introduction to NUMA.....	4
Bandwidth Optimization.....	6
Overview of the Implementation.....	7
Policies.....	8
Some Simple numactl Examples.....	9
numactl: Important Command Line Switches.....	10
numactl: Shared Memory	11
libnuma Basics: Checking for NUMA.....	12
libnuma: Nodemasks	13
libnuma: Simple Allocation	14
libnuma: Process Policy	15
libnuma: Changing the Policy of Existing Memory Areas	16
libnuma: Binding to CPUs	17
libnuma: Enquiring about the Environment	18
libnuma: Error Handling	19
NUMA Allocation Statistics with numastat.....	20
System Call Overview.....	24
Limitations and Outlook.....	24

INTRODUCTION TO NUMA

In traditional SMP (*Symmetric Multiprocessing*) systems, the computer has a single memory controller that is shared by all CPUs. This single memory connection often becomes a bottleneck when all processors access memory at the same time. It also does not scale very well for larger systems with a higher number of CPUs. For this reason, more and more modern systems are using a CC/NUMA (*Cache Coherent/Nonuniform Memory Access*) architecture. Examples are AMD* Opteron*, IBM* Power5*, HP* Superdome, and SGI* Altix*.

On an SMP system, all CPUs have equal access to the same shared memory controller that connects to all the memory chips (DIMMs). Communication between the CPUs also goes through this shared resource, which can become congested. The number of memory chips that can be managed by the single controller is also limited, which limits how much memory can be supported by the system. In addition, the latency to access memory through this single traffic hub is relatively high.

The NUMA architecture was designed to surpass the scalability limits of the SMP architecture. Instead of having a single memory controller per computer, the system is split into multiple nodes. Each node has processors and its own memory. The processors have very fast access to the local memory in the node. All the nodes in the system are connected using a fast interconnect. Each new node added to the system provides more aggregate memory bandwidth and capacity to the system, which gives excellent scalability.

The processors in a node all have equal access to the memory in the node. On a system where CPUs have integrated memory controllers, like the AMD Opteron, a node typically consists of a single CPU, possibly with multiple cores or virtual threads. On other more traditional NUMA systems, like SGI Altix or HP Superdome, larger nodes (which are like small SMP systems) with two to four CPUs share memory.

In a NUMA system, each CPU can access local and remote memory. Local memory is located in the same node as the CPU and provides very low memory access latency. Remote memory is located in a different node and must be accessed through the interconnect. From the software point of view, this remote memory can be used in the same way as local memory; it is fully cache coherent. Accessing it takes longer because the interconnect adds more latency than the local memory bus of the node.

In theory, a NUMA system can just be treated like an SMP system by ignoring the differences between local and remote memory in the software. In fact, this is often done. But for best performance, these differences should be taken in account.

One great advantage of the NUMA architecture is that even in a big system with many CPUs it is possible to get very low latency on the local memory. Because modern CPUs are much faster than memory chips, the CPU often spends quite some time waiting when reading data from memory. Minimizing the memory latency can therefore improve software performance.

NUMA policy is concerned with putting memory allocations on specific nodes to let programs access them as quickly as possible. The primary way to do this is to allocate memory for a thread on its local node and keep the thread running there (*node affinity*). This gives the best latency for memory and minimizes traffic over the global interconnect.

On SMP systems, there is a common optimization called cache affinity that is a bit similar. Cache affinity tries to keep data in the cache of a CPU instead of frequently “bouncing” it between processors. This is commonly done by a scheduler in the operating system, which tries to keep threads on a CPU for some time before scheduling it on another. However, there is an important difference from node affinity: When a thread on an SMP system moves between CPUs, its cache contents eventually move with it. Once a memory area is committed to a specific node on a NUMA system, it stays there. A thread running on a different node that accesses it always adds traffic to the interconnect and leads to higher latency. This is why NUMA systems need to try harder to archive node affinity than SMP systems. Of course, cache affinity by itself is a worthwhile optimization on a NUMA system, too. It is just not enough for best performance.

The scheduler in the operating system cannot always optimize purely for node affinity, however. The problem is that not using a CPU in the system would be even worse than a process using remote memory and seeing higher memory latency. In cases where memory performance is more important than even use of all CPUs in the system, the application or the system administrator can override the default decisions of the operating system. This allows better optimization for specific workloads.

Linux traditionally had system calls to bind threads to specific CPUs (using the *sched_set_affinity(2)* system call and *schedutils*). NUMA API extends this to allow programs to specify on which node memory should be allocated.

To make it easier for user space programs to optimize for NUMA configurations, APIs export topology information and allow user specification of processor and memory resources to use. There are also internal kernel APIs to provide NUMA topology information for use by kernel subsystems.

The NUMA API described here separates placement of threads to CPUs and placement of memory. It is primarily concerned with the placement of memory. In addition, the application can configure CPU affinity separately. NUMA API is currently available on SUSE® LINUX Enterprise Server 9 for AMD64 and for Intel® Itanium® Processor Family.

BANDWIDTH OPTIMIZATION

Memory access performance of programs can be optimized for latency or for bandwidth. Most programs seem to prefer lower latency, but there are a few exceptions that want bandwidth.

Using node local memory has the best latency. To get more bandwidth, the memory controllers of multiple nodes can be used in parallel. This is similar to how RAID can improve disk I/O performance by spreading I/O operations over multiple hard disks. NUMA API can use the MMU (Memory Management Unit) in the CPU to interleave blocks of memory from different memory controllers. This means that each consecutive page¹ in such a mapping comes from a different node.

When an application does a large streaming memory access to such an interleaved area, the bandwidth of the memory controllers of multiple nodes is combined. How well this works depends on the NUMA architecture, in particular on the performance of the interconnect and the latency difference between local and remote memory. On some systems, it only works effectively on a subset of neighboring nodes.

Some NUMA systems, like AMD Opteron, can be configured by firmware to interleave all memory across all nodes on a page basis. This is called *node interleaving*. Node interleaving is similar to the interleaving mode offered by the NUMA API, but they differ in important ways. Node interleaving applies to all memory. NUMA API interleaving can be configured for each process or thread individually. If node interleaving is enabled by firmware, the NUMA policy is disabled. To use NUMA policy, node interleaving always must be disabled in the BIOS or firmware.

With NUMA API, each application can individually adjust the policy that memory areas use for latency or bandwidth.

¹ A page is a 4K unit on an AMD64 or PPC*64 system. It is normally 16k on Intel Itanium Processor Family (IA64 architecture).

OVERVIEW OF THE IMPLEMENTATION

NUMA policy is provided by the combined effort of several subsystems. The kernel manages the memory policy for processes or specific memory mappings. This kernel can be controlled by three new system calls. There is a user space shared library called *libnuma* that can be called from applications. *libnuma* is the recommended API for NUMA policy in programs. It offers a more user-friendly and abstracted interface than using the system calls directly. This paper only describes this higher level interface. When the application should not be modified, the administrator can set some policy using the *numactl* command line utility. This is less flexible than controlling policy from the application directly.

The user libraries and applications are included in the *numactl* RPM, part of SUSE LINUX Enterprise Server 9². In addition, the package has some utility programs like *numastat* to collect statistics about the memory allocation and *numademo* to show the effect of different policies on the system. The package also contains man pages for all functions and programs.

2 It was already included in SUSE LINUX 9.1, but it is recommended to use the newer version from SUSE LINUX Enterprise Server 9.

POLICIES

The main task of NUMA API is to manage policies. Policies can be applied to processes or to memory areas.

NUMA API currently supports four policies:

Name	Description
<i>default</i>	Allocate on the local node (the node the thread is running on)
<i>bind</i>	Allocate on a specific set of nodes
<i>interleave</i>	Interleave memory allocations on a set of nodes
<i>preferred</i>	Try to allocate on a node first

The difference between *bind* and *preferred* is that *bind* fails when the memory cannot be allocated on the specified nodes; whereas, *preferred* falls back to other nodes. Using *bind* can lead to earlier memory shortages and delays due to swapping. In *libnuma*, *preferred* and *bind* are combined and can be changed per thread with the *numa_set_strict libnuma* function. The default is the more flexible *preferred* allocation.

Policies can be set per process (process policy) or per memory region. Children inherit the process policy of the parent on fork. The process policy is applied to all memory allocations made in the context of the process. This includes internal kernel allocations made in system calls and the file cache. Interrupts always allocate on the current node. The process policy always applies when a page of memory is allocated by the kernel.

Policies set per memory region, also called *VMA policies*³, allow a process to set a policy for a block of memory in its address space. Memory region policies have a higher priority than the process policy. The main advantage of memory region policies is that they can be set up before an allocation happens. Currently they are only supported for anonymous process memory, SYSV shared memory, *shmem* and *tmpfs* mappings, and *hugetlbfs* files. The region policy for shared memory persists until the shared memory segment or file is deleted.

³ VMA stands for *virtual memory area* and is a region of virtual memory in a process.

SOME SIMPLE NUMACTL EXAMPLES

numactl is a command line tool for running processes with a specific NUMA policy. It is useful for setting policies for programs that cannot be modified and recompiled.

Here are some simple examples of how to use *numactl*:

```
numactl --cpubind=0 --membind=0,1 program
```

Run the program on the CPUs of node 0 and only allocate memory from node 0 or 1. *cpubind* uses node numbers, not CPU numbers. On a system with multiple CPUs per node, this can differ.

```
numactl --preferred=1 numactl --show
```

Allocate memory preferable from node 1 and show the resulting state.

```
numactl --interleave=all numbercruncher
```

Run memory bandwidth-intensive number cruncher with memory interleaved over all available nodes.

```
numactl --offset=1G --length=1G --membind=1 --file /dev/shm/A --touch
```

Bind the second gigabyte in the tmpfs file */dev/shm/A* to node 1.

```
numactl --localalloc /dev/shm/file
```

Reset the policy for the shared memory file */dev/shm/file*.

```
numactl --hardware
```

Print an overview of the available nodes.

NUMACTL: IMPORTANT COMMAND LINE SWITCHES

Here is a quick overview of the important command line switches of *numactl*. Many of these switches need a *nodemask* as argument. Each node in the system has a unique number. A *nodemask* can be a comma-separated list of node numbers, a range of nodes (*node1-node2*), or *all*. See *numactl --hardware* for the nodes available on the current system.

The most common usage is to set the policy for a process. The policy is passed as the first argument and is followed by the program name and its argument. The available policy switches are:

`--membind=nodemask:`

Allocate memory only on the nodes in *nodemask*.

`--interleave=nodemask:`

Interleave all memory allocations over nodes in *nodemask*.

`--cpubind=nodemask:`

Execute the process only on the CPUs of the nodes specified in *nodemask*. It can be specified in addition to other policies because it affects the scheduler separately.

`--preferred=node:`

Allocate memory preferably on the node *node*.

Two other options are useful:

`--show:`

Print the current process NUMA state as inherited from the parent shell.

`--hardware:`

Give an overview of the available NUMA resources on the system.

For additional details, see the *numactl(8)* man page.

NUMACTL: SHARED MEMORY

numactl is also able to change policies in shared memory segments. This is useful for changing the policy of an application.

An example would be a program consisting of multiple processes that uses a common shared memory segment. This is a common setup for database servers. For the individual processes, it is best to use the default policy of allocating memory on their current nodes. This way, they get the best memory latency for their local data structure. The shared memory segment, however, is shared by multiple processes that run on different nodes. To avoid a hot spot on the node that allocated the memory originally, it may be advantageous to set an interleaved policy for the shared memory segment. This way, all access to it are spread evenly over all nodes.

More complex policies are possible. When parts of the shared memory are mostly used by specific processes and are only accessed rarely by others, they could be bound to specific nodes or only interleaved to a subset of nodes near each other.

Shared memory here can be SYSV shared memory (from the *shmat* system call), *mmap*ped files in *tmpfs* or *shmfs* (normally in */dev/shm* on a SUSE LINUX system), or a *hugetlbfs* file. The shared memory policy can be set up before the application starts⁴. The policy stays assigned to areas in the shared memory segment until it is deleted.

The set policy only applies to new pages that are allocated. Already existing pages in the shared memory segment are not moved to conform to the policy.

Set up a 1GB *tmpfs* file to interleave its memory over all nodes.

```
numactl --length=1G --file=/dev/shm/interleaved --interleave=all
```

A *hugetlbfs* file can be set up in the same way, but all lengths must be multiples of the huge page size of the system⁵.

An offset into the shared memory segment or file can be specified with *--offset=number*. All numeric arguments can have unit prefixes: *G* for gigabytes, *M* for megabytes, or *K* for kilobytes. The mode of the new file can be specified with *--mode=mode*.

Alternatively, this can be enforced with the *--strict* option. When *--strict* is set and an already allocated page does not conform to the new policy, *numactl* reports an error.

numactl has several more options for controlling the type of the shared memory segment. For details, see the *numactl(8)* man page.

⁴ This assumes the application does not insist on creating the shared memory segment itself.

⁵ *grep Hugepagesize/proc/meminfo* gives the huge page size of the current system.

LIBNUMA BASICS: CHECKING FOR NUMA

So far, we have described *numactl*, which controls the policy of whole processes. The disadvantage is that the policy always applies to the whole program, not to individual memory areas (except for shared memory) or threads. For some programs, more fine-grained policy control is needed.

This can be done with *libnuma*. *libnuma* is a shared library that can be linked to programs and offers a stable API for NUMA policy. It provides a higher level interface than using the NUMA API system calls directly and is the recommended interface for programs. *libnuma* is part of the *numactl* RPM.

Applications link with *libnuma* as follows:

```
cc ... -lnuma
```

The NUMA API functions and macros are declared in the *numa.h* include file.

```
#include <numa.h>
...
if (numa_available() < 0) {
    printf("Your system does not support NUMA API\n");
    ...
}
...
```

Before any NUMA API functions can be used, the program must call *numa_available()*. When this function returns a negative value, there is no NUMA policy support on the system. In this case, the behavior of all other NUMA API functions is undefined and they should not be called.

The next step is usually to call *numa_max_node()*. This function discovers and returns the number of nodes in the system. The number of nodes is typically needed to set up and verify the memory policy in the program. All programs should discover this dynamically instead of hardcoding a specific system's topology.

All *libnuma* states are kept locally per thread. Changing a policy in one thread does not affect the other threads in the process.

The following sections give an overview of the various *libnuma* functions with some examples. Some uncommon functions are not mentioned. For a more detailed reference, see the *numa(3)* man page.

LIBNUMA: NODEMASKS

libnuma manages sets of nodes in abstract data types called *nodemask_t* defined in *numa.h*. A *nodemask_t* is a fixed size bit set of node numbers. Each node in the system has a unique number. The highest number is the number returned by *numa_max_node()*. The highest node is defined for the implementation in the *NUMA_NUM_NODES* constant. *nodemasks* are passed by reference to many NUMA API functions.

A *nodemask* is initialized empty with *nodemask_zero()*.

```
nodemask_t mask;
nodemask_zero(&mask);
```

A single node can be set with *nodemask_set* and cleared with *nodemask_clr*. *nodemask_equal* compares two *nodemasks*. *nodemask_isset* tests if a bit is set in the *nodemask*.

```
nodemask_set(&mask, maxnode);          /* set node highest */
if (nodemask_isset(&mask, 1)) {      /* is node 1 set? */
    ...
}
nodemask_clr(&mask, maxnode);         /* clear highest node again */
```

There are two predefined *nodemasks*: *numa_all_nodes* stands for all nodes in the system and *numa_no_nodes* is the empty set.

LIBNUMA: SIMPLE ALLOCATION

libnuma offers functions to allocate memory with a specified policy. These allocation functions round all allocations to pages (4 KB on AMD64 systems) and are relatively slow. They should be used only for allocating large memory objects that exceed the cache sizes of the CPU and where NUMA policy is likely to help. When no memory can be allocated, they return NULL. All memory allocated with the *numa_alloc* family of functions should be freed with *numa_free*.

numa_alloc_onnode allocates memory on a specific node:

```
void *mem = numa_alloc_onnode(MEMSIZE\_IN\_BYTES, 1);
if (mem == NULL)
/* report out of memory error */
... pass mem to a thread bound to node 1 ...
```

memsize should be smaller than the node size. Remember that other programs may also have memory allocated on that node and allocating the full node may lead to swapping. The *numa_node_size()* function described below can be used to discover the node size limits of the current system automatically. It is recommended to give administrators a way to override automatic choices of the program and limit the memory consumption.

The thread must eventually free the memory with *numa_free*:

```
numa_free(mem, memsize);
```

By default, *numa_alloc_onnode* tries to allocate memory on the specified node first, but falls back to other nodes when there is not enough memory. When *numa_set_strict(1)* was executed first, it does not fall back and fails the allocation when there is not enough memory on the intended node. Before that, the kernel tries to swap out memory on the node and clear other caches, which can lead to delays.

numa_alloc_interleaved allocates memory interleaved on all nodes in the system.

```
void *mem = numa_alloc_interleaved(MEMSIZE\_IN\_BYTES);
if (mem == NULL)
/* report out of memory error */
... run memory bandwidth intensive algorithm on mem ...
numa\_free(mem, MEMSIZE\_IN\_BYTES);
```

Using memory interleaved over all nodes is not always a performance win. Depending on the NUMA architecture of the machine, sometimes interleaving the program on only a subset of neighboring nodes gives better bandwidth. The *numa_alloc_interleaved_subset* function can be used to interleave on a specific set of nodes only.

Another function is *numa_alloc_local*, which allocates memory on the local node. This is normally the default for all allocations, but useful to specify explicitly when the process has a different process policy. *numa_alloc* allocates memory with the current process policy.

LIBNUMA: PROCESS POLICY

Each thread has a default memory policy inherited from its parent. Unless changed with *numactl*, this policy is normally used to allocate memory preferably on the current node. When existing code in a program cannot be modified to use the *numa_alloc* functions described in the previous section directly, it is sometimes useful to change the process policy in a program. This way, specific subfunctions can be run with a nondefault policy without actually modifying their code. Process policy can also be used to set a policy for a child process before starting it.

numa_set_interleave_mask enables interleaving for the current thread. All future memory allocations allocate memory round robin interleaved over the nodemask specified. Passing *numa_all_nodes* interleaves memory to all nodes. Passing *numa_no_nodes* turns off interleaving again. *numa_get_interleave_mask* returns the current interleave mask. This can be useful to save a state before changing it in a library so it can be restored later.

```
numamask_t oldmask = numa_get_interleave_mask();
numa_set_interleave_mask(&numa_all_nodes);
/* run memory bandwidth intensive legacy library that allocates memory */
numa_set_interleave_mask(&oldmask);
```

numa_set_preferred sets the preferred node of the current thread. The memory allocator tries to allocate memory on that node first. If there is not enough free memory, it reverts to other nodes.

numa_set_membind sets a strict memory-binding mask to a nodemask. “Strict” means that the memory must be allocated on the specified nodes. When there is not enough memory free after swapping, the allocation fails.

numa_get_membind returns the current memory binding mask.

numa_set_localalloc sets the process policy to the standard local allocation policy.

LIBNUMA: CHANGING THE POLICY OF EXISTING MEMORY AREAS

When working with shared memory, it is often not possible to use the *numa_alloc* family of functions to allocate memory. The memory must be gotten from *shmat()* or *mmap* instead. To allow *libnuma* programs to set policy on such areas, there are additional functions for setting memory policy for already existing memory areas.

These functions only affect future allocations in the specified area. Linux uses demand paging and only allocates memory when a page is first touched by the CPU.

numa_interleave_memory sets an interleaving policy with an interleaving mask. Passing *numa_all_nodes* interleaves to all nodes in the system.

```
void *mem = shmat( ... ); /* get shared memory */
numa_interleave_mask(mem, size, numa_all_nodes);
```

numa_tonode_memory allocates the memory on a specific node. *numa_tonodemask_memory* puts the memory onto a mask of nodes.

numa_setlocal_memory gives the memory area a policy to allocate on the current node. *numa_police_memory* uses the current policy to allocate memory. This can be useful when the memory policy is changed later.

When *numa_set_strict(1)* was executed previously, these calls call *numa_error* when any of the already existing pages in the memory area do not conform to the new policy. Otherwise, existing pages are ignored.

LIBNUMA: BINDING TO CPUS

The functions discussed so far allocate memory on specific nodes. Another part of NUMA policy is to run the thread on the CPUs of the correct node. This is done by the `numa_run_on_node` function, which binds the current thread to all CPUs in the node. `numa_run_on_node_mask` binds the thread to all the CPUs in a *nodemask*.

Run current thread on node 1 and allocate memory there:

```
numa_run_on_node(1);
numa_set_preferred(1);
```

A simple way to use *libnuma* is the `numa_bind` function. It binds both the CPU and the memory of the process allocated in the future to a specific *nodemask*. It is equivalent to the previous example.

Bind process CPU and memory allocation to node 1 using `numa_bind`:

```
nodemask_t mask;
nodemask_zero(&mask);
nodemask_set(&mask 1);
numa_bind(&mask);
```

The thread can be allowed to execute on all nodes again by binding it to `numa_all_nodes`:

```
numa_run_on_node_mask(&numa_all_nodes);
```

The `numa_get_run_node_mask` function returns the *nodemask* of nodes on which the current thread is allowed to run. This can be used to save and restore the scheduler affinity state before running a child process or starting a thread.

LIBNUMA: ENQUIRING ABOUT THE ENVIRONMENT

numa_node_size returns the memory size of a node. The return argument is the total size of its memory, which is not necessarily all available to the program. The second argument is a pointer that can be filled with the free memory on the node. The program can allocate node memory somewhere between the free memory (which is normally low because Linux uses free memory for caches) and the maximum memory size. When memory is allocated, Linux frees the cached file data, but allocating too much memory may lead to swapping. This function gives a hint of how much memory is available for allocation on each node, but it should be only taken as a hint, preferably with some way for the administrator to overwrite it. In general, it is recommended never to allocate more than half the total memory of a node by default unless the administrator specified more.

numa_node_to_cpus returns the CPU numbers of all CPUs in a node. This can be used to find out how many CPUs there are in a node. As arguments, it gets the node number and a pointer to an array. The last argument is the length of the array in bytes. The array is filled with a bit mask of CPU numbers. These CPU numbers can later be, for example, passed to the *sched_set_affinity* system call. When the array is not long enough to contain all CPUs, the function returns *-1* and sets *errno* to *ERANGE*. It is recommended for applications to handle this error or pass a very big buffer, such as 512 bytes. Otherwise, there may be failures on very big machines. Linux already runs on 1024 CPU machines and is expected to be moved to even bigger machines.

LIBNUMA: ERROR HANDLING

Error handling in *libnuma* is relatively simple. The main reason for this is that errors in setting NUMA policy can usually be ignored. The worst result of a wrong NUMA policy is that the program runs more slowly than it could.

When an error occurs while setting a policy, the *numa_error* function is called. By default, it prints an error to *stderr*. When the *numa_exit_on_error* global variable is set, it exits the program. The function is declared weak and can be overwritten by defining a replacement function in the main program. For example, a C++ program could throw a C++ exception there.

Memory allocation functions always return *NULL* when no memory is available.

NUMA ALLOCATION STATISTICS WITH NUMASTAT

For each node in the system, the kernel maintains some statistics pertaining to NUMA allocation status as each page is allocated. This information may be useful for testing the effectiveness of a NUMA policy.

The statistics are retrieved with the *numastat* command. The statistics are collected on a per-node basis. On systems with multiple CPU cores per node, *numastat* aggregates the results from all cores on a node to form a single result for the entire node. The *numastat* command reports the following statistics for each node:

When a process requests a page from a particular node and receives a page from the requested node, *numa_hit* is incremented for that particular node. The process may be running on any node in the system.

When a process requests a page from a particular node and instead receives a page from some other node, *numa_miss* is incremented at the node where the page was actually allocated. *numa_foreign* is incremented at the original node from which the page was requested. The process may be running on any node in the system.

interleave_hit is incremented on the node on which a page is allocated when the allocation obeys the interleave policy for the address range. In addition, *numa_hit* and either *local_node* or *other_node* are incremented on the node on which the page is allocated. No statistics are kept for pages allocated according to the interleave policy but not on the requested node because of its lack of free pages.

When a process requests a page and the resulting page is located on the same node as where the process is running, *local_node* is incremented on that particular node.

When a process requests a page and the resulting page is located on a different node than where the process is running, *other_node* is incremented for the node on which the page is actually allocated.

The difference between *numa_miss* and *numa_hit* and *local_node* and *foreign_node* is that the first two count hit or miss for the NUMA policy. The latter count if the allocation was on the same node as the requesting thread.

To better understand the *numastat* values, consider the following examples.

The following example shows which counters are incremented when a process running on node 0 requests a page on node 0 and it is allocated on node 0.

	node 3	node 2	node 1	node 0
<i>numa_hit</i>				+1
<i>numa_miss</i>				
<i>numa_foreign</i>				
<i>interleave_hit</i>				
<i>local_node</i>				+1
<i>other_node</i>				

The following example shows which counters are incremented when a process running on node 0 requests a page on node 0 but it is allocated on node 1 due to a shortage of free pages on node 0.

	node 3	node 2	node 1	node 0
numa_hit				
numa_miss			+1	
numa_foreign				+1
interleave_hit				
local_node				
other_node			+1	

The following example shows which counters are incremented when a process running on node 0 requests and receives a page on node 1. Note the difference between this and the first example.

	node 3	node 2	node 1	node 0
numa_hit			+1	
numa_miss				
numa_foreign				
interleave_hit				
local_node				
other_node			+1	

The following example shows which counters are incremented when a process running on node 0 requests a page on node 1 but it is allocated on node 0 due to a shortage of free pages on node 1.

	node 3	node 2	node 1	node 0
numa_hit				
numa_miss				+1
numa_foreign			+1	
interleave_hit				
local_node				+1
other_node				

As a further example, consider a four-node machine with 4 GB of RAM per node. Initially, *numastat* reports the following statistics for this machine:

	node 3	node 2	node 1	node 0
numa_hit	58956	142758	424386	319127
numa_miss	0	0	0	0
numa_foreign	0	0	0	0
interleave_hit	19204	20238	19675	20576
local_node	43013	126715	409434	305254
other_node	15943	16043	14952	13873

Now suppose that a program called *memhog* runs on node 1, allocating 8 GB of RAM during execution. After *memhog* completes, *numastat* reports the following statistics:

	node 3	node 2	node 1	node 0
numa_hit	58956	142758	424386	320893
numa_miss	48365	1026046	0	0
numa_foreign	0	0	1074411	0
interleave_hit	19204	20238	19675	20577
local_node	43013	126856	1436403	307019
other_node	64308	1042089	14952	13873

Each column represents a node where an allocation event took place. The *memhog* program tried to allocate 1,074,411 pages from node 1 but was unable to do so. Instead, the process ended with 1,026,046 pages from node 2 and 48,365 pages from node 3.

SYSTEM CALL OVERVIEW

NUMA API adds three new system calls: *mbind*, *set_mempolicy*, and *get_mempolicy*. Normally user applications should use the higher level *libnuma* interface and not call the system calls directly. The system call interface is declared in *numaif.h* include file. The system calls are currently not defined in *glibc*, applications that use them should link to *libnuma*. They may return *-1* and *ENOSYS* in *errno* when the kernel does not support NUMA policy.

All of these system calls get *nodemasks* as arguments, similar to the *nodemask_t* type of *libnuma*. In the system call interface, they are defined as arrays of longs, each long containing a string of bits, together with an additional argument that gives the highest node number in the bitmap.

set_mempolicy sets the memory policy of the current thread. The first argument is the policy. Valid policies are *MPOL_BIND*, *MPOL_INTERLEAVE*, *MPOL_DEFAULT*, and *MPOL_PREFERRED*. These map directly to the *numactl* policies described earlier. How the nodemask argument is used depends on the policy. For *MPOL_INTERLEAVE*, it specifies the interleave mask. For *MPOL_BIND* and *MPOL_PREFERRED*, it contains the *membind* mask.

get_mempolicy retrieves the process memory policy of the current thread. In addition to output arguments for policy, nodemask and nodemask size, it has an address and flags arguments. When the *MPOL_MF_ADDR* bit is set in flags, the VMA policy of the address is returned in the other arguments. When *MPOL_F_NODE* is additionally set, the current node to which the address pointed is returned.

mbind sets the memory policy for a memory area. The first argument is a memory address and a length. The rest is a policy with mask and length similar to *set_mempolicy*. In addition, it has a flags argument. When *MPOL_MF_STRICT* is passed for flags, the call fails when any existing page in the mapping violates the specified policy.

For more detailed information, refer to the man pages *get_mempolicy(2)*, *set_mempolicy(2)*, and *mbind(2)*.

LIMITATIONS AND OUTLOOK

On a bigger system, there is normally a hierarchy in the interconnect. This means that neighboring nodes can be accessed more quickly than remote nodes. NUMA API currently represents the system as a flat set of nodes. Future versions will allow the application to query node distances. Future versions of NUMA API may also allow setting a policy on the file cache.