NUMA(3) Linux Programmer's Manual NUMA(3)

NAME top

numa - NUMA policy library

SYNOPSIS top

```
#include <numa.h>

cc ... -lnuma

int numa_available(void);

int numa_max_possible_node(void);
int numa_num_possible_nodes();

int numa_max_node(void);
int numa_num_configured_nodes();
struct bitmask *numa_get_mems_allowed(void);

int numa_num_configured_cpus(void);
struct bitmask *numa_all_nodes_ptr;
struct bitmask *numa_no_nodes_ptr;
struct bitmask *numa_all_cpus_ptr;

int numa_num_task_cpus();
int numa_num_task_nodes();

int numa_parse_bitmap(char *line , struct bitmask *mask);
struct bitmask *numa_parse_nodestring(const char *string);
struct bitmask *numa_parse_nodestring_all(const char *string);
struct bitmask *numa_parse_cpustring(const char *string);
struct bitmask *numa_parse_cpustring_all(const char *string);

long numa_node_size(int node, long *freep);
long long numa_node_size64(int node, long long *freep);

int numa_preferred(void);
void numa_set_preferred(int node);
int numa_get_interleave_node(void);
struct bitmask *numa_get_interleave_mask(void);
void numa_set_interleave_mask(struct bitmask *nodemask);
void numa_interleave_memory(void *start, size_t size, struct bitmask
*nodemask);
void numa_bind(struct bitmask *nodemask);
void numa_set_localalloc(void);
```

```
void numa_set_membind(struct bitmask *nodemask);
struct bitmask *numa_get_membind(void);

void *numa_alloc_onnode(size_t size, int node);
void *numa_alloc_local(size_t size);
void *numa_alloc_interleaved(size_t size);
void *numa_alloc_interleaved_subset(size_t size,  struct bitmask
*nodemask); void *numa_alloc(size_t size);
void *numa_realloc(void *old_addr, size_t old_size, size_t new_size);
void numa_free(void *start, size_t size);

int numa_run_on_node(int node);
int numa_run_on_node_mask(struct bitmask *nodemask);
int numa_run_on_node_mask_all(struct bitmask *nodemask);
struct bitmask *numa_get_run_node_mask(void);

void numa_tonode_memory(void *start, size_t size, int node);
void numa_tonodemask_memory(void *start, size_t size, struct bitmask
*nodemask);
void numa_setlocal_memory(void *start, size_t size);
void numa_police_memory(void *start, size_t size);
void numa_set_bind_policy(int strict);
void numa_set_strict(int strict);

int numa_distance(int node1, int node2);

int numa_sched_getaffinity(pid_t pid, struct bitmask *mask);
int numa_sched_setaffinity(pid_t pid, struct bitmask *mask);
int numa_node_to_cpus(int node, struct bitmask *mask);
int numa_node_of_cpu(int cpu);

struct bitmask *numa_allocate_cpumask();

void numa_free_cpumask();
struct bitmask *numa_allocate_nodemask();

void numa_free_nodemask();
struct bitmask *numa_bitmask_alloc(unsigned int n);
struct bitmask *numa_bitmask_clearall(struct bitmask *bmp);
struct bitmask *numa_bitmask_clearbit(struct bitmask *bmp, unsigned
int n);
int numa_bitmask_equal(const struct bitmask *bmp1, const struct
bitmask *bmp2);
void numa_bitmask_free(struct bitmask *bmp);
int numa_bitmask_isbitset(const struct bitmask *bmp, unsigned int n);
unsigned int numa_bitmask_nbytes(struct bitmask *bmp);
struct bitmask *numa_bitmask_setall(struct bitmask *bmp);
struct bitmask *numa_bitmask_setbit(struct bitmask *bmp, unsigned int
n);
void copy_bitmask_to_nodemask(struct bitmask *bmp, nodemask_t
*nodemask)
void copy_nodemask_to_bitmask(nodemask_t *nodemask, struct bitmask
*bmp)
void copy_bitmask_to_bitmask(struct bitmask *bmpfrom, struct bitmask
*bmpto)
```

unsigned int numa_bitmask_weight(const struct bitmask *bmp)

int numa_move_pages(int *pid***, unsigned long** *count***, void *****pages***, const int ****nodes***, int ****status***, int** *flags***);**
int numa_migrate_pages(int *pid***, struct bitmask ****fromnodes***, struct bitmask ****tonodes***);**

void numa_error(char ****where*);**

extern int *numa_exit_on_error***;**
extern int *numa_exit_on_warn***;**
void numa_warn(int *number***, char ****where***, ...);**

DESCRIPTION top

The *libnuma* library offers a simple programming interface to the NUMA (Non Uniform Memory Access) policy supported by the Linux kernel. On a NUMA architecture some memory areas have different latency or bandwidth than others.

Available policies are page interleaving (i.e., allocate in a round-robin fashion from all, or a subset, of the nodes on the system), preferred node allocation (i.e., preferably allocate on a particular node), local allocation (i.e., allocate on the node on which the task is currently executing), or allocation only on specific nodes (i.e., allocate on some subset of the available nodes). It is also possible to bind tasks to specific nodes.

Numa memory allocation policy may be specified as a per-task attribute, that is inherited by children tasks and processes, or as an attribute of a range of process virtual address space. Numa memory policies specified for a range of virtual address space are shared by all tasks in the process. Further more, memory policies specified for a range of a shared memory attached using shmat(2) or mmap(2) from shmfs/hugetlbfs are shared by all processes that attach to that region. Memory policies for shared disk backed file mappings are currently ignored.

The default memory allocation policy for tasks and all memory range is local allocation. This assumes that no ancestor has installed a non-default policy.

For setting a specific policy globally for all memory allocations in a process and its children it is easiest to start it with the numactl(8) utility. For more finegrained policy inside an application this library can be used.

All numa memory allocation policy only takes effect when a page is actually faulted into the address space of a process by accessing it. The **numa_alloc_*** functions take care of this automatically.

A *node* is defined as an area where all memory has the same speed as seen from a particular CPU. A node can contain multiple CPUs. Caches are ignored for this definition.

Most functions in this library are only concerned about numa nodes and their memory. The exceptions to this are: *numa_node_to_cpus*(), *numa_node_of_cpu*(), *numa_bind*(), *numa_run_on_node*(), *numa_run_on_node_mask*(), *numa_run_on_node_mask_all*(), and *numa_get_run_node_mask*(). These functions deal with the CPUs associated with numa nodes. See the descriptions below for more information.

Some of these functions accept or return a pointer to struct bitmask. A struct bitmask controls a bit map of arbitrary length containing a bit representation of nodes. The predefined variable *numa_all_nodes_ptr* points to a bit mask that has all available nodes set; *numa_no_nodes_ptr* points to the empty set.

Before any other calls in this library can be used **numa_available**() must be called. If it returns -1, all other functions in this library are undefined.

numa_max_possible_node() returns the number of the highest possible node in a system. In other words, the size of a kernel type nodemask_t (in bits) minus 1. This number can be gotten by calling **numa_num_possible_nodes()** and subtracting 1.

numa_num_possible_nodes() returns the size of kernel's node mask (kernel type nodemask_t). In other words, large enough to represent the maximum number of nodes that the kernel can handle. This will match the kernel's MAX_NUMNODES value. This count is derived from /proc/self/status, field Mems_allowed.

numa_max_node() returns the highest node number available on the current system. (See the node numbers in /sys/devices/system/node/). Also see **numa_num_configured_nodes().**

numa_num_configured_nodes() returns the number of memory nodes in the system. This count includes any nodes that are currently disabled. This count is derived from the node numbers in /sys/devices/system/node. (Depends on the kernel being configured with /sys (CONFIG_SYSFS)).

numa_get_mems_allowed() returns the mask of nodes from which the process is allowed to allocate memory in it's current cpuset context. Any nodes that are not included in the returned bitmask will be ignored in any of the following libnuma memory policy calls.

numa_num_configured_cpus() returns the number of cpus in the system. This count includes any cpus that are currently disabled. This count is derived from the cpu numbers in /sys/devices/system/cpu. If the kernel is configured without /sys (CONFIG_SYSFS=n) then it falls back to using the number of online cpus.

numa_all_nodes_ptr points to a bitmask that is allocated by the library with bits representing all nodes on which the calling task may allocate memory. This set may be up to all nodes on the system, or up to the nodes in the current cpuset. The bitmask is allocated by a call to **numa_allocate_nodemask()** using size by a call to **numa_allocate_nodemask()** using size **numa_max_possible_node().** The set of nodes to record is derived from /proc/self/status, field "Mems_allowed". The user should not alter this bitmask.

numa_no_nodes_ptr points to a bitmask that is allocated by the library and left all zeroes. The bitmask is allocated by a call to **numa_allocate_nodemask()** using size **numa_max_possible_node().** The user should not alter this bitmask.

numa_all_cpus_ptr points to a bitmask that is allocated by the library with bits representing all cpus on which the calling task may execute. This set may be up to all cpus on the system, or up to the cpus in the current cpuset. The bitmask is allocated by a call to **numa_allocate_cpumask()** using size **numa_num_possible_cpus().** The set of cpus to record is derived from /proc/self/status, field "Cpus_allowed". The user should not alter this bitmask.

numa_num_task_cpus() returns the number of cpus that the calling task is allowed to use. This count is derived from the map /proc/self/status, field "Cpus_allowed". Also see the bitmask **numa_all_cpus_ptr.**

numa_num_task_nodes() returns the number of nodes on which the calling task is allowed to allocate memory. This count is derived from the map /proc/self/status, field "Mems_allowed". Also see the bitmask **numa_all_nodes_ptr.**

numa_parse_bitmap() parses *line* , which is a character string such as found in /sys/devices/system/node/nodeN/cpumap into a bitmask structure. The string contains the hexadecimal representation of a bit map. The bitmask may be allocated with **numa_allocate_cpumask().** Returns 0 on success. Returns -1 on failure. This function is probably of little use to a user application, but it is used by *libnuma* internally.

numa_parse_nodestring() parses a character string list of nodes into a bit mask. The bit mask is allocated by **numa_allocate_nodemask().** The string is a comma-separated list of node numbers or node ranges. A leading ! can be used to indicate "not" this list (in other words, all nodes except this list), and a leading + can be used to indicate that the node numbers in the list are relative to the task's cpuset. The string can be "all" to specify all (**numa_num_task_nodes()**) nodes. Node numbers are limited by the number in the system. See **numa_max_node()** and **numa_num_configured_nodes().**
Examples: 1-5,7,10 !4-5 +0-3
If the string is of 0 length, bitmask **numa_no_nodes_ptr** is returned. Returns 0 if the string is invalid.

numa_parse_nodestring_all() is similar to **numa_parse_nodestring** , but can parse all possible nodes, not only current nodeset.

numa_parse_cpustring() parses a character string list of cpus into a bit mask. The bit mask is allocated by **numa_allocate_cpumask().** The string is a comma-separated list of cpu numbers or cpu ranges. A leading ! can be used to indicate "not" this list (in other words,

leading ! can be used to indicate "not" this list (in other words, all cpus except this list), and a leading + can be used to indicate that the cpu numbers in the list are relative to the task's cpuset. The string can be "all" to specify all (**numa_num_task_cpus()**) cpus. Cpu numbers are limited by the number in the system. See **numa_num_task_cpus()** and **numa_num_configured_cpus().**
Examples: 1-5,7,10 !4-5 +0-3
Returns 0 if the string is invalid.

numa_parse_cpustring_all() is similar to **numa_parse_cpustring** , but can parse all possible cpus, not only current cpuset.

numa_node_size() returns the memory size of a node. If the argument *freep* is not NULL, it used to return the amount of free memory on the node. On error it returns -1.

numa_node_size64() works the same as **numa_node_size**() except that it returns values as *long long* instead of *long*. This is useful on 32-bit architectures with large nodes.

numa_preferred() returns the preferred node of the current task. This is the node on which the kernel preferably allocates memory, unless some other policy overrides this.

numa_set_preferred() sets the preferred node for the current task to *node*. The system will attempt to allocate memory from the preferred node, but will fall back to other nodes if no memory is available on the the preferred node. Passing a *node* of -1 argument specifies local allocation and is equivalent to calling **numa_set_localalloc**().

numa_get_interleave_mask() returns the current interleave mask if the task's memory allocation policy is page interleaved. Otherwise, this function returns an empty mask.

numa_set_interleave_mask() sets the memory interleave mask for the current task to *nodemask*. All new memory allocations are page interleaved over all nodes in the interleave mask. Interleaving can be turned off again by passing an empty mask (*numa_no_nodes*). The page interleaving only occurs on the actual page fault that puts a new page into the current address space. It is also only a hint: the kernel will fall back to other nodes if no memory is available on the interleave target.

numa_interleave_memory() interleaves *size* bytes of memory page by page from *start* on nodes specified in *nodemask*. The *size* argument will be rounded up to a multiple of the system page size. If *nodemask* contains nodes that are externally denied to this process, this call will fail. This is a lower level function to interleave allocated but not yet faulted in memory. Not yet faulted in means the memory is allocated using mmap(2) or shmat(2), but has not been accessed by the current process yet. The memory is page interleaved to all nodes specified in *nodemask*. Normally **numa_alloc_interleaved**() should be used for private memory instead, but this function is useful to handle shared memory areas. To be useful the memory area should be several megabytes at least (or tens of megabytes of hugetlbfs mappings) If the **numa_set_strict**() flag is true then the operation will cause a numa_error if there were already pages in the mapping that do not follow the policy.

numa_bind() binds the current task and its children to the nodes specified in *nodemask*. They will only run on the CPUs of the specified nodes and only be able to allocate memory from them. This function is equivalent to calling *numa_run_on_node_mask(nodemask)* followed by *numa_set_membind(nodemask)*. If tasks should be bound to individual CPUs inside nodes consider using *numa_node_to_cpus* and the sched_setaffinity(2) syscall.

numa_set_localalloc() sets the memory allocation policy for the calling task to local allocation. In this mode, the preferred node for memory allocation is effectively the node where the task is executing at the time of a page allocation.

numa_set_membind() sets the memory allocation mask. The task will only allocate memory from the nodes set in *nodemask*. Passing an empty *nodemask* or a *nodemask* that contains nodes other than those in the mask returned by *numa_get_mems_allowed*() will result in an error.

numa_get_membind() returns the mask of nodes from which memory can currently be allocated. If the returned mask is equal to *numa_all_nodes*, then memory allocation is allowed from all nodes.

numa_alloc_onnode() allocates memory on a specific node. The *size* argument will be rounded up to a multiple of the system page size. if the specified *node* is externally denied to this process, this call will fail. This function is relatively slow compared to the malloc(3), family of functions. The memory must be freed with **numa_free**(). On errors NULL is returned.

numa_alloc_local() allocates *size* bytes of memory on the local node. The *size* argument will be rounded up to a multiple of the system page size. This function is relatively slow compared to the malloc(3) family of functions. The memory must be freed with **numa_free**(). On errors NULL is returned.

numa_alloc_interleaved() allocates *size* bytes of memory page interleaved on all nodes. This function is relatively slow and should only be used for large areas consisting of multiple pages. The interleaving works at page level and will only show an effect when the area is large. The allocated memory must be freed with **numa_free**(). On error, NULL is returned.

numa_alloc_interleaved_subset() attempts to allocate *size* bytes of memory page interleaved on all nodes. The *size* argument will be rounded up to a multiple of the system page size. The nodes on which a process is allowed to allocate memory may be constrained externally. If this is the case, this function may fail. This function is relatively slow compare to malloc(3), family of functions and should only be used for large areas consisting of multiple pages. The interleaving works at page level and will only show an effect when the area is large. The allocated memory must be freed with **numa_free**(). On error, NULL is returned.

numa_alloc() allocates *size* bytes of memory with the current NUMA policy. The *size* argument will be rounded up to a multiple of the system page size. This function is relatively slow compare to the malloc(3) family of functions. The memory must be freed with **numa_free**(). On errors NULL is returned.

numa_realloc() changes the size of the memory area pointed to by *old_addr* from *old_size* to *new_size.* The memory area pointed to by *old_addr* must have been allocated with one of the **numa_alloc*** functions. The *new_size* will be rounded up to a multiple of the system page size. The contents of the memory area will be unchanged to the minimum of the old and new sizes; newly allocated memory will be uninitialized. The memory policy (and node bindings) associated with the original memory area will be preserved in the resized area. For example, if the initial area was allocated with a call to **numa_alloc_onnode(),** then the new pages (if the area is enlarged) will be allocated on the same node. However, if no memory policy was set for the original area, then **numa_realloc**() cannot guarantee that the new pages will be allocated on the same node. On success, the address of the resized area is returned (which might be different from that of the initial area), otherwise NULL is returned and *errno* is set to indicate the error. The pointer returned by **numa_realloc**() is suitable for passing to **numa_free**().

numa_free() frees *size* bytes of memory starting at *start*, allocated by the **numa_alloc_*** functions above. The *size* argument will be rounded up to a multiple of the system page size.

numa_run_on_node() runs the current task and its children on a specific node. They will not migrate to CPUs of other nodes until the node affinity is reset with a new call to **numa_run_on_node_mask**(). Passing -1 permits the kernel to schedule on all nodes again. On success, 0 is returned; on error -1 is returned, and *errno* is set to indicate the error.

numa_run_on_node_mask() runs the current task and its children only on nodes specified in *nodemask*. They will not migrate to CPUs of other nodes until the node affinity is reset with a new call to **numa_run_on_node_mask**() or **numa_run_on_node**(). Passing *numa_all_nodes* permits the kernel to schedule on all nodes again. On success, 0 is returned; on error -1 is returned, and *errno* is set to indicate the error.

numa_run_on_node_mask_all() runs the current task and its children only on nodes specified in *nodemask* like *numa_run_on_node_mask* but without any cpuset awareness.

numa_get_run_node_mask() returns a mask of CPUs on which the current task is allowed to run.

numa_tonode_memory() put memory on a specific node. The constraints described for **numa_interleave_memory**() apply here too.

numa_tonodemask_memory() put memory on a specific set of nodes. The constraints described for **numa_interleave_memory**() apply here too.

numa_setlocal_memory() locates memory on the current node. The constraints described for **numa_interleave_memory**() apply here too.

numa_police_memory() locates memory with the current NUMA policy. The constraints described for **numa_interleave_memory**() apply here too.

numa_distance() reports the distance in the machine topology between two nodes. The factors are a multiple of 10. It returns 0 when the distance cannot be determined. A node has distance 10 to itself. Reporting the distance requires a Linux kernel version of *2.6.10* or newer.

numa_set_bind_policy() specifies whether calls that bind memory to a specific node should use the preferred policy or a strict policy. The preferred policy allows the kernel to allocate memory on other nodes when there isn't enough free on the target node. strict will fail the allocation in that case. Setting the argument to specifies strict, 0 preferred. Note that specifying more than one node non strict may only use the first node in some kernel versions.

numa_set_strict() sets a flag that says whether the functions allocating on specific nodes should use use a strict policy. Strict means the allocation will fail if the memory cannot be allocated on the target node. Default operation is to fall back to other nodes. This doesn't apply to interleave and default.

numa_get_interleave_node() is used by *libnuma* internally. It is probably not useful for user applications. It uses the MPOL_F_NODE flag of the get_mempolicy system call, which is not intended for application use (its operation may change or be removed altogether in future kernel versions). See get_mempolicy(2).

numa_pagesize() returns the number of bytes in page. This function is simply a fast alternative to repeated calls to the getpagesize system call. See getpagesize(2).

numa_sched_getaffinity() retrieves a bitmask of the cpus on which a task may run. The task is specified by *pid.* Returns the return value of the sched_getaffinity system call. See sched_getaffinity(2). The bitmask must be at least the size of the kernel's cpu mask structure. Use **numa_allocate_cpumask()** to allocate it. Test the bits in the mask by calling **numa_bitmask_isbitset().**

numa_sched_setaffinity() sets a task's allowed cpu's to those cpu's specified in *mask.* The task is specified by *pid.* Returns the return value of the sched_setaffinity system call. See sched_setaffinity(2). You may allocate the bitmask with **numa_allocate_cpumask().** Or the bitmask may be smaller than the kernel's cpu mask structure. For example, call **numa_bitmask_alloc()** using a maximum number of cpus from **numa_num_configured_cpus().** Set the bits in the mask by calling **numa_bitmask_setbit().**

numa_node_to_cpus() converts a node number to a bitmask of CPUs. The user must pass a bitmask structure with a mask buffer long enough to represent all possible cpu's. Use numa_allocate_cpumask() to create it. If the bitmask is not long enough *errno* will be set to *ERANGE* and -1 returned. On success 0 is returned.

numa_node_of_cpu() returns the node that a cpu belongs to. If the user supplies an invalid cpu *errno* will be set to *EINVAL* and -1 will be returned.

numa_allocate_cpumask () returns a bitmask of a size equal to the kernel's cpu mask (kernel type cpumask_t). In other words, large enough to represent NR_CPUS cpus. This number of cpus can be gotten by calling **numa_num_possible_cpus().** The bitmask is zero-filled.

numa_free_cpumask frees a cpumask previously allocate by *numa_allocate_cpumask.*

numa_allocate_nodemask() returns a bitmask of a size equal to the kernel's node mask (kernel type nodemask_t). In other words, large enough to represent MAX_NUMNODES nodes. This number of nodes can be gotten by calling **numa_num_possible_nodes().** The bitmask is zero-filled.

numa_free_nodemask() frees a nodemask previous allocated by *numa_allocate_nodemask().*

numa_bitmask_alloc() allocates a bitmask structure and its associated bit mask. The memory allocated for the bit mask contains enough words (type unsigned long) to contain *n* bits. The bit mask is zero-filled. The bitmask structure points to the bit mask and contains the *n* value.

numa_bitmask_clearall() sets all bits in the bit mask to 0. The bitmask structure points to the bit mask and contains its size (*bmp* ->size). The value of *bmp* is always returned. Note that **numa_bitmask_alloc()** creates a zero-filled bit mask.

numa_bitmask_clearbit() sets a specified bit in a bit mask to 0. Nothing is done if the *n* value is greater than the size of the bitmask (and no error is returned). The value of *bmp* is always returned.

numa_bitmask_equal() returns 1 if two bitmasks are equal. It returns 0 if they are not equal. If the bitmask structures control bit masks of different sizes, the "missing" trailing bits of the smaller bit mask are considered to be 0.

numa_bitmask_free() deallocates the memory of both the bitmask structure pointed to by *bmp* and the bit mask. It is an error to attempt to free this bitmask twice.

numa_bitmask_isbitset() returns the value of a specified bit in a bit mask. If the *n* value is greater than the size of the bit map, 0 is returned.

numa_bitmask_nbytes() returns the size (in bytes) of the bit mask

numa_bitmask_nbytes() returns the size (in bytes) of the bit mask controlled by *bmp*. The bit masks are always full words (type unsigned long), and the returned size is the actual size of all those words.

numa_bitmask_setall() sets all bits in the bit mask to 1. The bitmask structure points to the bit mask and contains its size (*bmp* ->size). The value of *bmp* is always returned.

numa_bitmask_setbit() sets a specified bit in a bit mask to 1. Nothing is done if *n* is greater than the size of the bitmask (and no error is returned). The value of *bmp* is always returned.

copy_bitmask_to_nodemask() copies the body (the bit map itself) of the bitmask structure pointed to by *bmp* to the nodemask_t structure pointed to by the *nodemask* pointer. If the two areas differ in size, the copy is truncated to the size of the receiving field or zero-filled.

copy_nodemask_to_bitmask() copies the nodemask_t structure pointed to by the *nodemask* pointer to the body (the bit map itself) of the bitmask structure pointed to by the *bmp* pointer. If the two areas differ in size, the copy is truncated to the size of the receiving field or zero-filled.

copy_bitmask_to_bitmask() copies the body (the bit map itself) of the bitmask structure pointed to by the *bmpfrom* pointer to the body of the bitmask structure pointed to by the *bmpto* pointer. If the two areas differ in size, the copy is truncated to the size of the receiving field or zero-filled.

numa_bitmask_weight() returns a count of the bits that are set in the body of the bitmask pointed to by the *bmp* argument.

numa_move_pages() moves a list of pages in the address space of the currently executing or current process. It simply uses the move_pages system call.
pid - ID of task. If not valid, use the current task.
count - Number of pages.
pages - List of pages to move.
nodes - List of nodes to which pages can be moved.
status - Field to which status is to be returned.
flags - MPOL_MF_MOVE or MPOL_MF_MOVE_ALL
See move_pages(2).

numa_migrate_pages() simply uses the migrate_pages system call to cause the pages of the calling task, or a specified task, to be migated from one set of nodes to another. See migrate_pages(2). The bit masks representing the nodes should be allocated with **numa_allocate_nodemask()** , or with **numa_bitmask_alloc()** using an *n* value returned from **numa_num_possible_nodes().** A task's current node set can be gotten by calling **numa_get_membind().** Bits in the *tonodes* mask can be set by calls to **numa_bitmask_setbit().**

numa_error() is a *libnuma* internal function that can be overridden by the user program. This function is called with a *char ** argument the user program. This function is called with a *char* * argument when a *libnuma* function fails. Overriding the library internal definition makes it possible to specify a different error handling strategy when a *libnuma* function fails. It does not affect **numa_available**(). The **numa_error**() function defined in *libnuma* prints an error on *stderr* and terminates the program if *numa_exit_on_error* is set to a non-zero value. The default value of *numa_exit_on_error* is zero.

numa_warn() is a *libnuma* internal function that can be also overridden by the user program. It is called to warn the user when a *libnuma* function encounters a non-fatal error. The default implementation prints a warning to *stderr*. The first argument is a unique number identifying each warning. After that there is a printf(3)-style format string and a variable number of arguments. *numa_warn* exits the program when *numa_exit_on_warn* is set to a non-zero value. The default value of *numa_exit_on_warn* is zero.

Compatibility with libnuma version 1 top

Binaries that were compiled for libnuma version 1 need not be re-compiled to run with libnuma version 2.
Source codes written for libnuma version 1 may be re-compiled without change with version 2 installed. To do so, in the code's Makefile add this option to CFLAGS: -DNUMA_VERSION1_COMPATIBILITY

THREAD SAFETY top

numa_set_bind_policy and *numa_exit_on_error* are process global. The other calls are thread safe.

COPYRIGHT top

SEE ALSO top

get_mempolicy(2), set_mempolicy(2), getpagesize(2), mbind(2), mmap(2), shmat(2), numactl(8), sched_getaffinity(2) sched_setaffinity(2) move_pages(2) migrate_pages(2)

COLOPHON top

This page is part of the *numactl* (NUMA commands) project. Information about the project can be found at ⟨http://oss.sgi.com/projects/libnuma/⟩. If you have a bug report for this manual page, send it to linux-numa@vger.kernel.org. This page was obtained from the tarball numactl-2.0.11.tar.gz fetched from

⟨ftp://oss.sgi.com/www/projects/libnuma/download⟩ on 2017-05-03. If you discover any rendering problems in this HTML version of the page,

```
       you discover any rendering problems in this HTML version of the page,
       or you believe there is a better or more up-to-date source for the
       page, or you have corrections or improvements to the information in
       this COLOPHON (which is not part of the original manual page), send a
       mail to man-pages@man7.org
```

SuSE Labs **December 2007** **NUMA(3)**

Pages that refer to this page: get_mempolicy(2), mbind(2), migrate_pages(2), move_pages(2), set_mempolicy(2), numa_maps(5), numa(7), numastat(8)

HTML rendering created 2017-05-03 by Michael Kerrisk, author of *The Linux Programming Interface*, maintainer of the Linux *man-pages* project.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look here.

Hosting by jambit GmbH.