What is Linux Memory Policy?

In the Linux kernel, "memory policy" determines from which node the kernel will allocate memory in a NUMA system or in an emulated NUMA system. Linux has supported platforms with Non-Uniform Memory Access architectures since 2.4.?. The current memory policy support was added to Linux 2.6 around May 2004. This document attempts to describe the concepts and APIs of the 2.6 memory policy support.

Memory policies should not be confused with cpusets
(Documentation/cgroups/cpusets.txt)
which is an administrative mechanism for restricting the nodes from which memory may be allocated by a set of processes. Memory policies are a programming interface that a NUMA-aware application can take advantage of. When both cpusets and policies are applied to a task, the restrictions of the cpuset takes priority. See "MEMORY POLICIES AND CPUSETS" below for more details.

MEMORY POLICY CONCEPTS

Scope of Memory Policies

The Linux kernel supports _scopes_ of memory policy, described here from most general to most specific:

System Default Policy: this policy is "hard coded" into the kernel. It is the policy that governs all page allocations that aren't controlled by one of the more specific policy scopes discussed below. When the system is "up and running", the system default policy will use "local allocation" described below. However, during boot up, the system default policy will be set to interleave allocations across all nodes with "sufficient" memory, so as not to overload the initial boot node with boot-time allocations.

Task/Process Policy: this is an optional, per-task policy. When defined for a specific task, this policy controls all page allocations made by or on behalf of the task that aren't controlled by a more specific scope. If a task does not define a task policy, then all page allocations that would have been controlled by the task policy "fall back" to the System Default Policy.

The task policy applies to the entire address space of a task. Thus, it is inheritable, and indeed is inherited, across both fork() [clone() w/o the CLONE_VM flag] and exec*(). This allows a parent task to establish the task policy for a child task exec()'d from an executable image that has no awareness of memory policy. See the MEMORY POLICY APIS section, below, for an overview of the system call that a task may use to set/change its task/process policy.

In a multi-threaded task, task policies apply only to the thread [Linux kernel task] that installs the policy and any threads subsequently created by that thread. Any sibling threads existing at the time a new task policy is installed retain their current policy.

A task policy applies only to pages allocated after the policy is

installed. Any pages already faulted in by the task when the task changes its task policy remain where they were allocated based on the policy at the time they were allocated.

VMA Policy: A "VMA" or "Virtual Memory Area" refers to a range of a task's virtual address space. A task may define a specific policy for a range of its virtual address space. See the MEMORY POLICIES APIS section, below, for an overview of the mbind() system call used to set a VMA policy.

A VMA policy will govern the allocation of pages that back this region of the address space. Any regions of the task's address space that don't have an explicit VMA policy will fall back to the task policy, which may itself fall back to the System Default Policy.

VMA policies have a few complicating details:

VMA policy applies ONLY to anonymous pages. These include pages allocated for anonymous segments, such as the task stack and heap, and any regions of the address space mmap()ed with the MAP_ANONYMOUS flag. If a VMA policy is applied to a file mapping, it will be ignored if the mapping used the MAP_SHARED flag. If the file mapping used the MAP_PRIVATE flag, the VMA policy will only be applied when an anonymous page is allocated on an attempt to write to the mapping--i.e., at Copy-On-Write.

VMA policies are shared between all tasks that share a virtual address space--a.k.a. threads--independent of when the policy is installed; and they are inherited across fork(). However, because VMA policies refer to a specific region of a task's address space, and because the address space is discarded and recreated on exec*(), VMA policies are NOT inheritable across exec(). Thus, only NUMA-aware applications may use VMA policies.

A task may install a new VMA policy on a sub-range of a previously mmap()ed region. When this happens, Linux splits the existing virtual memory area into 2 or 3 VMAs, each with it's own policy.

By default, VMA policy applies only to pages allocated after the policy is installed. Any pages already faulted into the VMA range remain where they were allocated based on the policy at the time they were allocated. However, since 2.6.16, Linux supports page migration via the mbind() system call, so that page contents can be moved to match a newly installed policy.

Shared Policy: Conceptually, shared policies apply to "memory objects" mapped shared into one or more tasks' distinct address spaces. An application installs a shared policies the same way as VMA policies--using the mbind() system call specifying a range of virtual addresses that map the shared object. However, unlike VMA policies, which can be considered to be an attribute of a range of a task's address space, shared policies apply directly to the shared object. Thus, all tasks that attach to the object share the policy, and all pages allocated for the shared object, by any task, will obey the shared policy.

As of 2.6.22, only shared memory segments, created by shmget() or

mmap(MAP_ANONYMOUS|MAP_SHARED), support shared policy. When shared policy support was added to Linux, the associated data structures were added to hugetlbfs shmem segments. At the time, hugetlbfs did not support allocation at fault time--a.k.a lazy allocation--so hugetlbfs shmem segments were never "hooked up" to the shared policy support. Although hugetlbfs segments now support lazy allocation, their support for shared policy has not been completed.

As mentioned above [re: VMA policies], allocations of page cache pages for regular files mmap()ed with MAP_SHARED ignore any VMA policy installed on the virtual address range backed by the shared file mapping. Rather, shared page cache pages, including pages backing private mappings that have not yet been written by the task, follow task policy, if any, else System Default Policy.

The shared policy infrastructure supports different policies on subset ranges of the shared object. However, Linux still splits the VMA of the task that installs the policy for each range of distinct policy. Thus, different tasks that attach to a shared memory segment can have different VMA configurations mapping that one shared object. This can be seen by examining the /proc/<pid>/numa_maps of tasks sharing a shared memory region, when one task has installed shared policy on one or more ranges of the region.

Components of Memory Policies

A Linux memory policy consists of a "mode", optional mode flags, and an optional set of nodes. The mode determines the behavior of the policy, the optional mode flags determine the behavior of the mode, and the optional set of nodes can be viewed as the arguments to the policy behavior.

Internally, memory policies are implemented by a reference counted structure, struct mempolicy. Details of this structure will be discussed in context, below, as required to explain the behavior.

Linux memory policy supports the following 4 behavioral modes:

Default Mode--MPOL_DEFAULT: This mode is only used in the memory policy APIs. Internally, MPOL_DEFAULT is converted to the NULL memory policy in all policy scopes. Any existing non-default policy will simply be removed when MPOL_DEFAULT is specified. As a result, MPOL_DEFAULT means "fall back to the next most specific policy scope."

For example, a NULL or default task policy will fall back to the system default policy. A NULL or default vma policy will fall back to the task policy.

When specified in one of the memory policy APIs, the Default mode does not use the optional set of nodes.

It is an error for the set of nodes specified for this policy to be non-empty.

MPOL_BIND: This mode specifies that memory must come from the set of nodes specified by the policy. Memory will be allocated from

the node in the set with sufficient free memory that is closest to the node where the allocation takes place.

MPOL_PREFERRED: This mode specifies that the allocation should be attempted from the single node specified in the policy. If that allocation fails, the kernel will search other nodes, in order of increasing distance from the preferred node based on information provided by the platform firmware.
containing the cpu where the allocation takes place.

Internally, the Preferred policy uses a single node--the preferred_node member of struct mempolicy. When the internal mode flag MPOL_F_LOCAL is set, the preferred_node is ignored and the policy is interpreted as local allocation. "Local" allocation policy can be viewed as a Preferred policy that starts at the node containing the cpu where the allocation takes place.

It is possible for the user to specify that local allocation is always preferred by passing an empty nodemask with this mode. If an empty nodemask is passed, the policy cannot use the MPOL_F_STATIC_NODES or MPOL_F_RELATIVE_NODES flags described below.

MPOL_INTERLEAVED: This mode specifies that page allocations be interleaved, on a page granularity, across the nodes specified in the policy. This mode also behaves slightly differently, based on the context where it is used:

For allocation of anonymous pages and shared memory pages, Interleave mode indexes the set of nodes specified by the policy using the page offset of the faulting address into the segment [VMA] containing the address modulo the number of nodes specified by the policy. It then attempts to allocate a page, starting at the selected node, as if the node had been specified by a Preferred policy or had been selected by a local allocation. That is, allocation will follow the per node zonelist.

For allocation of page cache pages, Interleave mode indexes the set of nodes specified by the policy using a node counter maintained per task. This counter wraps around to the lowest specified node after it reaches the highest specified node. This will tend to spread the pages out over the nodes specified by the policy based on the order in which they are allocated, rather than based on any page offset into an address range or file. During system boot up, the temporary interleaved system default policy works in this mode.

Linux memory policy supports the following optional mode flags:

MPOL_F_STATIC_NODES: This flag specifies that the nodemask passed by the user should not be remapped if the task or VMA's set of allowed nodes changes after the memory policy has been defined.

Without this flag, anytime a mempolicy is rebound because of a change in the set of allowed nodes, the node (Preferred) or nodemask (Bind, Interleave) is remapped to the new set of

allowed nodes. This may result in nodes being used that were previously undesired.

With this flag, if the user-specified nodes overlap with the nodes allowed by the task's cpuset, then the memory policy is applied to their intersection. If the two sets of nodes do not overlap, the Default policy is used.

For example, consider a task that is attached to a cpuset with mems 1-3 that sets an Interleave policy over the same set. If the cpuset's mems change to 3-5, the Interleave will now occur over nodes 3, 4, and 5. With this flag, however, since only node 3 is allowed from the user's nodemask, the "interleave" only occurs over that node. If no nodes from the user's nodemask are now allowed, the Default behavior is used.

MPOL_F_STATIC_NODES cannot be combined with the MPOL_F_RELATIVE_NODES flag. It also cannot be used for MPOL_PREFERRED policies that were created with an empty nodemask (local allocation).

MPOL_F_RELATIVE_NODES: This flag specifies that the nodemask passed by the user will be mapped relative to the set of the task or VMA's set of allowed nodes. The kernel stores the user-passed nodemask, and if the allowed nodes changes, then that original nodemask will be remapped relative to the new set of allowed nodes.

Without this flag (and without MPOL_F_STATIC_NODES), anytime a mempolicy is rebound because of a change in the set of allowed nodes, the node (Preferred) or nodemask (Bind, Interleave) is remapped to the new set of allowed nodes. That remap may not preserve the relative nature of the user's passed nodemask to its set of allowed nodes upon successive rebinds: a nodemask of 1,3,5 may be remapped to 7-9 and then to 1-3 if the set of allowed nodes is restored to its original state.

With this flag, the remap is done so that the node numbers from the user's passed nodemask are relative to the set of allowed nodes. In other words, if nodes 0, 2, and 4 are set in the user's nodemask, the policy will be effected over the first (and in the Bind or Interleave case, the third and fifth) nodes in the set of allowed nodes. The nodemask passed by the user represents nodes relative to task or VMA's set of allowed nodes.

If the user's nodemask includes nodes that are outside the range of the new set of allowed nodes (for example, node 5 is set in the user's nodemask when the set of allowed nodes is only 0-3), then the remap wraps around to the beginning of the nodemask and, if not already set, sets the node in the mempolicy nodemask.

For example, consider a task that is attached to a cpuset with mems 2-5 that sets an Interleave policy over the same set with MPOL_F_RELATIVE_NODES. If the cpuset's mems change to 3-7, the interleave now occurs over nodes 3,5-6. If the cpuset's mems then change to 0,2-3,5, then the interleave occurs over nodes 0,3,5.

Thanks to the consistent remapping, applications preparing nodemasks to specify memory policies using this flag should disregard their current, actual cpuset imposed memory placement and prepare the nodemask as if they were always located on memory nodes 0 to N-1, where N is the number of memory nodes the policy is intended to manage. Let the kernel then remap to the set of memory nodes allowed by the task's cpuset, as that may change over time.

MPOL_F_RELATIVE_NODES cannot be combined with the MPOL_F_STATIC_NODES flag. It also cannot be used for MPOL_PREFERRED policies that were created with an empty nodemask (local allocation).

MEMORY POLICY REFERENCE COUNTING

To resolve use/free races, struct mempolicy contains an atomic reference count field. Internal interfaces, mpol_get()/mpol_put() increment and decrement this reference count, respectively. mpol_put() will only free the structure back to the mempolicy kmem cache when the reference count goes to zero.

When a new memory policy is allocated, its reference count is initialized to '1', representing the reference held by the task that is installing the new policy. When a pointer to a memory policy structure is stored in another structure, another reference is added, as the task's reference will be dropped on completion of the policy installation.

During run-time "usage" of the policy, we attempt to minimize atomic operations on the reference count, as this can lead to cache lines bouncing between cpus and NUMA nodes. "Usage" here means one of the following:

1) querying of the policy, either by the task itself [using the get_mempolicy() API discussed below] or by another task using the /proc/<pid>/numa_maps interface.

2) examination of the policy to determine the policy mode and associated node or node lists, if any, for page allocation. This is considered a "hot path". Note that for MPOL_BIND, the "usage" extends across the entire allocation process, which may sleep during page reclaimation, because the BIND policy nodemask is used, by reference, to filter ineligible nodes.

We can avoid taking an extra reference during the usages listed above as follows:

1) we never need to get/free the system default policy as this is never changed nor freed, once the system is up and running.

2) for querying the policy, we do not need to take an extra reference on the target task's task policy nor vma policies because we always acquire the task's mm's mmap_sem for read during the query. The set_mempolicy() and mbind() APIs [see below] always acquire the mmap_sem for write when installing or replacing task or vma policies. Thus, there is no possibility of a task or thread freeing a policy while another task or thread is querying it.

3) Page allocation usage of task or vma policy occurs in the fault path where we hold them mmap_sem for read. Again, because replacing the task or vma policy requires that the mmap_sem be held for write, the policy can't be freed out from under us while we're using it for page allocation.

4) Shared policies require special consideration. One task can replace a shared memory policy while another task, with a distinct mmap_sem, is querying or allocating a page based on the policy. To resolve this potential race, the shared policy infrastructure adds an extra reference to the shared policy during lookup while holding a spin lock on the shared policy management structure. This requires that we drop this extra reference when we're finished "using" the policy. We must drop the extra reference on shared policies in the same query/allocation paths used for non-shared policies. For this reason, shared policies are marked as such, and the extra reference is dropped "conditionally"--i.e., only for shared policies.

   Because of this extra reference counting, and because we must lookup shared policies in a tree structure under spinlock, shared policies are more expensive to use in the page allocation path. This is especially true for shared policies on shared memory regions shared by tasks running on different NUMA nodes. This extra overhead can be avoided by always falling back to task or system default policy for shared memory regions, or by prefaulting the entire shared memory region into memory and locking it down. However, this might not be appropriate for all applications.

MEMORY POLICY APIs

Linux supports 3 system calls for controlling memory policy. These APIS always affect only the calling task, the calling task's address space, or some shared object mapped into the calling task's address space.

> Note: the headers that define these APIs and the parameter data types for user space applications reside in a package that is not part of the Linux kernel. The kernel system call interfaces, with the 'sys_' prefix, are defined in <linux/syscalls.h>; the mode and flag definitions are defined in <linux/mempolicy.h>.

Set [Task] Memory Policy:

```
long set_mempolicy(int mode, const unsigned long *nmask,
                                unsigned long maxnode);
```

Set's the calling task's "task/process memory policy" to mode specified by the 'mode' argument and the set of nodes defined by 'nmask'. 'nmask' points to a bit mask of node ids containing at least 'maxnode' ids. Optional mode flags may be passed by combining the 'mode' argument with the flag (for example: MPOL_INTERLEAVE | MPOL_F_STATIC_NODES).

See the set_mempolicy(2) man page for more details

Get [Task] Memory Policy or Related Information

```
long get_mempolicy(int *mode,
                   const unsigned long *nmask, unsigned long maxnode,
                   void *addr, int flags);
```

Queries the "task/process memory policy" of the calling task, or the policy or location of a specified virtual address, depending on the 'flags' argument.

See the get_mempolicy(2) man page for more details

Install VMA/Shared Policy for a Range of Task's Address Space

```
long mbind(void *start, unsigned long len, int mode,
           const unsigned long *nmask, unsigned long maxnode,
           unsigned flags);
```

mbind() installs the policy specified by (mode, nmask, maxnodes) as a VMA policy for the range of the calling task's address space specified by the 'start' and 'len' arguments. Additional actions may be requested via the 'flags' argument.

See the mbind(2) man page for more details.

MEMORY POLICY COMMAND LINE INTERFACE

Although not strictly part of the Linux implementation of memory policy, a command line tool, numactl(8), exists that allows one to:

+ set the task policy for a specified program via set_mempolicy(2), fork(2) and exec(2)

+ set the shared policy for a shared memory segment via mbind(2)

The numactl(8) tool is packages with the run-time version of the library containing the memory policy system call wrappers. Some distributions package the headers and compile-time libraries in a separate development package.

MEMORY POLICIES AND CPUSETS

Memory policies work within cpusets as described above. For memory policies that require a node or set of nodes, the nodes are restricted to the set of nodes whose memories are allowed by the cpuset constraints. If the nodemask specified for the policy contains nodes that are not allowed by the cpuset and MPOL_F_RELATIVE_NODES is not used, the intersection of the set of nodes specified for the policy and the set of nodes with memory is used. If the result is the empty set, the policy is considered invalid and cannot be installed. If MPOL_F_RELATIVE_NODES is used, the policy's nodes are mapped onto and folded into the task's set of allowed nodes as previously described.

The interaction of memory policies and cpusets can be problematic when tasks in two cpusets share access to a memory region, such as shared memory segments created by shmget() of mmap() with the MAP_ANONYMOUS and MAP_SHARED flags, and any of the tasks install shared policy on the region, only nodes whose

memories are allowed in both cpusets may be used in the policies. Obtaining this information requires "stepping outside" the memory policy APIs to use the cpuset information and requires that one know in what cpusets other task might be attaching to the shared region. Furthermore, if the cpusets' allowed memory sets are disjoint, "local" allocation is the only valid policy.