

NAME

mpctl() - multiprocessor control

SYNOPSIS

```
#include <sys/mpctl.h>

int mpctl(
    mpc_request_t request,
    spu_t spu,
    pid_t pid
);

int mpctl(
    mpc_request_t request,
    spu_t spu,
    lwpid_t lwpid
);

int mpctl(
    mpc_request_t request,
    ldom_t ldom,
    pid_t pid
);

int mpctl(
    mpc_request_t request,
    ldom_t ldom,
    lwpid_t lwpid
);
```

Remarks

Much of the functionality of this capability is highly dependent on the underlying hardware. An application that uses this system call should not be expected to be portable across architectures or implementations. | m |

Some hardware platforms support online addition and deletion of processors. Due to this capability, processors and locality domains may be added or deleted while the system is running. Applications should be written to handle processor IDs and locality domain IDs that dynamically appear or disappear (for example, sometime after obtaining the IDs of all the processors in the system an application may try to bind an LWP to one of those processors - this system call will return an error if that processor had been deleted).

Processor sets restrict application execution to a designated group of processors. Some applications may query information about processors and locality domains available to them, while other applications may require system-wide information. The **mpctl()** interface supports two unique sets of command requests for these purposes.

Applications using the *pthread* interfaces should not use this system call. A special set of **pthread_*()** routines has been developed for use by pthread applications. See the *pthread_processor_bind_np(3T)* manual page for information on these interfaces.

DESCRIPTION

mpctl() provides a means of determining how many processors and locality domains are available in the system, and assigning processes or lightweight processes to execute on specific processors or within a specific locality domain.

A locality domain consists of a related collection of processors, memory, and peripheral resources that comprise a fundamental building block of the system. All processors and peripheral devices in a given locality domain have equal latency to the memory contained within that locality domain. Use **sysconf()** with the **_SC_CCNUMA_SUPPORT** name to see if the ccNUMA functionality is enabled and available on the system.

Processor sets provide an alternative application scheduling allocation domain. A processor set comprises an isolated group of processors for exclusive use by applications assigned to the processor set. Applications may use **mpctl()** to query about processors and locality domains available for them to scale and optimize accordingly. Use **sysconf()** with **_SC_PSET_SUPPORT** name to see if the processor set functionality is enabled and available on the system.

The `mpctl()` call is expected to be used to increase performance in certain applications, but should not be used to ensure correctness of an application. Specifically, cooperating processes/lightweight processes should not rely on processor or locality domain assignment in lieu of a synchronization mechanism (such as semaphores).

Machine Topology Information

Warning: Processor and locality domain IDs are not guaranteed to exist in numerical order. There may be holes in a sequential list of IDs. Due to the capability of online addition and deletion of processors on some platforms, IDs obtained via these interfaces may be invalid at a later time. Likewise, the number of processors and locality domains in the system may also change due to processors being added or deleted. See the *Processor Set Information* section to query machine topology within the application's processor set.

For processor topology use:

```
int mpctl(mpc_request_t request, spu_t spu, pid_t pid);
```

The *request* argument determines the precise action to be taken by `mpctl()` and is one of the following:

MPC_GETNUMSPUS_SYS

This request returns the number of enabled spus (processors) in the system. It will always be greater than or equal to 1. The *spu* and *pid* arguments are ignored.

MPC_GETFIRSTSPU_SYS

This request returns the ID of the first enabled processor in the system. The *spu* and *pid* arguments are ignored.

MPC_GETNEXTSPU_SYS

This request returns the ID of the next enabled processor in the system after *spu*. The *pid* argument is ignored.

Typically, **MPC_GETFIRSTSPU_SYS** is called to determine the first spu. **MPC_GETNEXTSPU_SYS** is then called in a loop (until the call returns -1) to determine the IDs of the remaining spus.

MPC_GETCURRENTSPU

This request returns the ID of the processor the caller is currently running on (NOT the processor assignment of the caller). The *spu* and *pid* arguments are ignored.

Warning: The information returned by this system call may be out-of-date arbitrarily soon after the call completes due to the scheduler context switching the caller onto a different processor.

For locality domain topology use:

```
int mpctl(mpc_request_t request, ldom_t ldom, pid_t pid);
```

The *request* argument determines the precise action to be taken by `mpctl()` and is one of the following:

MPC_GETNUMLDMOS_SYS

This request returns the number of active locality domains in the system. An active locality domain has at least one enabled processor in it. The number of active locality domains in the system will always be greater than or equal to 1. The *ldom* and *pid* arguments are ignored.

MPC_GETFIRSTLDMOS_SYS

This request returns the ID of the first active locality domain in the system. The *ldom* and *pid* arguments are ignored.

MPC_GETNEXTLDMOS_SYS

This request returns the ID of the next active locality domain in the system after *ldom*. The *pid* argument is ignored.

Typically, **MPC_GETFIRSTLDMOS_SYS** is called to determine the first locality domain. **MPC_GETNEXTLDMOS_SYS** is then called in a loop (until the call returns -1) to determine the IDs of the remaining locality domains.

MPC_GETCURRENTLDMOS

This request returns the ID of the ldom that the caller is currently running on

(NOT the ldom assignment of the caller). The *ldom* and *pid* arguments are ignored.

Warning: The information returned by this system call may be out-of-date arbitrarily soon after the call completes due to the scheduler context switching the caller onto a different ldom.

MPC_LDOMSPUS_SYS

This request returns the number of enabled processors in the locality domain *ldom*. The *pid* argument is ignored.

MPC_SPUTOLDOM This request returns the ID of the locality domain containing processor *spu*. The *pid* argument is ignored.

Proximity Topology Information

All processors in a given locality domain have equal latency to the memory contained within that locality domain. However, a processor may have different cache-to-cache access latency to different processors within its locality domain. The processors with the same cache-to-cache access latency are said to be proximate to one another and form a proximity set. A processor's cache-to-cache access latency to a processor within its proximity set is lower compared to a processor not in its proximity set even within the same locality domain. By definition, a processor is said to be proximate to itself. The topology of the processors in a proximity set is called as Proximity Topology.

Proximity Topology is highly dependent on the underlying architecture of the system. An example of a proximity set and the architecture supporting it is a set of processors on the same Front Side Bus (FSB) on systems that use FSBs. Depending on the architecture:

- each processor by itself may be shown in its proximity set
- a subset of processors belonging to a locality domain may be shown in one proximity set
- all processors in a locality domain may be shown in one proximity set

Note that there may or may not be more than one proximity set in a given locality domain.

Some applications that require only a subset of processors in the system may see performance benefit by running on processors in the same proximity set. This can be achieved by creating a processor set with processors from the same proximity set and running the application in this processor set.

For proximity topology use:

```
int mpctl(mpc_request_t request, spu_t spu, pid_t pid);
```

The *request* argument determines the precise action to be taken by `mpctl()` and is one of the following:

MPC_GETNUMPROXIMATESPUS_SYS

This request returns the number of enabled spus (processors) in the system that are in the same proximity set as that of *spu*. If *spu* is enabled, the value returned will be greater than or equal to 1. Otherwise -1 is returned. The *pid* argument is ignored.

MPC_GETFIRSTPROXIMATESPU_SYS

This request returns the ID of the first enabled processor in the system that is proximate to *spu*. If *spu* is enabled, it will return a valid processor ID. Otherwise -1 is returned. The *pid* argument is ignored.

MPC_GETNEXTPROXIMATESPU_SYS

This request returns the ID of the next enabled processor in the system that is proximate to *spu*. The *pid* argument is ignored.

Typically, `MPC_GETFIRSTPROXIMATESPU_SYS` is called to determine the first proximate spu. `MPC_GETNEXTPROXIMATESPU_SYS` is then called in a loop (until the call returns -1) to determine the IDs of the remaining proximate spus.

MPC_GETNUMPROXIMATESPUS

This request returns the number of enabled spus (processors) in the processor set of the calling thread and that are in the same proximity set as that of *spu*. Even when *spu* is enabled, the return value will be 0 if none of the proximate processors contribute to the processor set of the calling thread. If *spu* is not enabled, -1 is returned. The *pid* argument is ignored.



MPC_GETFIRSTPROXIMATESPU

This request returns the ID of the first enabled processor which is in the processor set of the calling thread and is proximate to *spu*. Even when *spu* is enabled, the return value will be -1 if none of the proximate processors contribute to the processor set of the calling thread. If *spu* is not enabled, -1 is returned. The *pid* argument is ignored.

MPC_GETNEXTPROXIMATESPU

This request returns the ID of the next enabled processor which is in the processor set of the calling thread and is proximate to *spu*. The *pid* argument is ignored.

Typically, **MPC_GETFIRSTPROXIMATESPU** is called to determine the first proximate *spu*. **MPC_GETNEXTPROXIMATESPU** is then called in a loop (until the call returns -1) to determine the IDs of the remaining proximate *spus*.

Processor Set Information

Warning: Dynamic creation and deletion of processor sets, and dynamic reassignment of a processor from one processor set to another may occur. All processors in the system comprise one processor set by default at boot time until new processor sets are created and configured by users.

The following command requests return topology information on processors and locality domains in the processor set of the calling thread. Only an enabled processor can be in a processor set. A locality domain is said to be in a processor set, if it contributes at least one processor to that processor set.

For processor topology use:

```
int mpctl(mpc_request_t request, spu_t spu, pid_t pid);
```

The *request* argument determines the precise action to be taken by **mpctl()** and is one of the following:

MPC_GETNUMSPUS

This request returns the number of *spus* (processors) in the processor set of the calling thread. The *spu* and *pid* arguments are ignored.

MPC_GETFIRSTSPU

This request returns the ID of the first processor in the processor set of the calling thread. The *spu* and *pid* arguments are ignored.

MPC_GETNEXTSPU

This request returns the ID of the next processor in the processor set of the calling thread after *spu*. The *pid* argument is ignored.

Typically, **MPC_GETFIRSTSPU** is called to determine the first *spu*. **MPC_GETNEXTSPU** is then called in a loop (until the call returns -1) to determine the IDs of the remaining *spus*.

For locality domain topology use:

```
int mpctl(mpc_request_t request, ldom_t ldom, pid_t pid);
```

The *request* argument determines the precise action to be taken by **mpctl()** and is one of the following:

MPC_GETNUMLDOMS

This request returns the number of locality domains in the processor set of the calling thread. The *ldom* and *pid* arguments are ignored.

MPC_GETFIRSTLDM

This request returns the ID of the first locality domain in the processor set of the calling thread. The *ldom* and *pid* arguments are ignored.

MPC_GETNEXTLDM

This request returns the ID of the next locality domain in the processor set of the calling thread after *ldom*. The *pid* argument is ignored.

Typically, **MPC_GETFIRSTLDM** is called to determine the first locality domain. **MPC_GETNEXTLDM** is then called in a loop (until the call returns -1) to determine the IDs of the remaining locality domains.

MPC_LDOMSPUS

This request returns the number of processors contributed by the locality domain *ldom* to the processor set of the calling thread. It may be less than the total number of processors in the *ldom*. The *pid* argument is ignored.

Processor Socket Information

For processor socket topology use:

```
int mpctl(mpc_request_t request, spu_t spu, pid_t pid);
```

The *request* argument determines the precise action to be taken by `mpctl()` and is one of the following:

MPC_GETNUMSOCKETS_SYS

This request returns the number of enabled sockets (physical processors) in the system. An enabled socket has at least one core enabled. The value will be greater than or equal to 1. If the call is not implemented the value will be -1. The *spu* and *pid* arguments are ignored.

Logical Processor and Processor Core Information

On systems with Hyper-Threading (HT) feature enabled, each processor core may have more than one hyper-thread per physical processor core. When hyper-threading is enabled at the firmware level, each hyper-thread is represented to the operating system and applications as a logical processor (LCPU). Hence the basic unit of any topology information is a logical processor. However, some applications may want to get the system topology information at the physical processor core level.

For processor core topology use:

```
int mpctl(mpc_request_t request, spu_t spu, pid_t pid);
```

The *request* argument determines the precise action to be taken by `mpctl()` and is one of the following:

MPC_GETNUMCORES_SYS

Returns the number of enabled processor cores in the system; this value will always be greater than or equal to 1. The *spu* and *pid* arguments are ignored.

MPC_GETFIRSTCORE_SYS

Returns the processor core ID of the first enabled processor core in the system. The *spu* and *pid* arguments are ignored.

MPC_GETNEXTCORE_SYS

Returns the processor core ID of the next enabled processor core in the system after the specified processor core ID. The *pid* argument is ignored. Typically **MPC_GETFIRSTCORE_SYS** is called to determine the first processor core. **MPC_GETNEXTCORE_SYS** is then called in a loop (until the call returns -1) to determine the IDs of the remaining processor cores.

MPC_GETCURRENTCORE

Returns the ID of the processor core the calling thread is currently running on (not the processor core assignment of the caller). The *spu* and *pid* arguments are ignored.

MPC_SPUTOCORE Returns the ID of the physical processor core containing the *spu*. The *pid* argument is ignored.

MPC_GETNUMCORES

Returns the number of processor cores in the processor set of the calling thread. The *spu* and *pid* arguments are ignored.

MPC_GETFIRSTCORE

Returns the ID of the first processor core in the processor set of the calling thread. The *spu* and *pid* arguments are ignored.

MPC_GETNEXTCORE

Returns the ID of the processor core in the processor set of the calling thread after the processor core specified in *spu*. The *pid* argument is ignored.

For processor core and locality domain topology use:

```
int mpctl(mpc_request_t request, ldom_t ldom, pid_t pid);
```

The *request* argument determines the precise action to be taken by `mpctl()` and is one of the following:

MPC_LDOMCORES_SYS

Returns the number of enabled processor cores in the locality domain; this value will always be greater than or equal to 0. The *pid* argument is ignored.

| m |

MPC_LDOMCORES Returns the number of enabled processor cores assigned to the current processor set in the locality domain; this value will always be greater than or equal to 0. The *pid* argument is ignored.

Processor and Locality Domain Binding

Each process shall have a processor and locality domain binding. Each LWP (lightweight process) shall have a processor and locality domain binding. The binding assignments for a lightweight process do not have to match the binding assignments for the process.

Setting the processor or locality domain binding on the process of a multithreaded process, causes all LWPs (lightweight processes) in the target process to have their binding assignments changed to what is specified. However, if any LWP belongs to a different processor set such that the specified processor or locality domain does not contribute to that processor set, the binding assignment for such an LWP is not changed.

When a process creates another process (via **fork()** or **vfork()**), the child process will inherit the parent process's binding assignments (NOT the binding assignments of the creating LWP). The initial LWP in the child process shall inherit its binding assignments from the child process. LWPs other than the initial LWP shall inherit their binding assignments from the creating LWP (unless specified otherwise in the LWP create attributes).

Processor binding and locality domain binding are mutually exclusive -- only one can be in effect at any time. If locality domain binding is in effect, the target is allowed to execute on any processor within that locality domain in its processor set.

Setting the processor or locality domain binding will fail if the target processor or locality domain is not in the processor set of the specified process or LWP.

WARNING: Due to the capability of online addition and deletion of processors on some platforms, processors may go away. If this occurs, any processes or LWPs bound to a departing processor will be rebound to a different processor with the same binding type. If the last processor in a locality domain is removed, any processes or LWPs bound to a departing locality domain will be rebound to a different locality domain.

| m |

For processor binding use:

```
int mpctl(mpc_request_t request, spu_t spu, pid_t pid);
int mpctl(mpc_request_t request, spu_t spu, lwpid_t lwpid);
```

The *request* argument determines the precise action to be taken by **mpctl()** and is one of the following:

MPC_SETPROCESS

This call is *advisory*. This request asynchronously assigns process *pid* to processor *spu*. The new processor assignment is returned.

The *pid* **MPC_SELFPID** may be used to refer to the calling process.

The *spu* **MPC_SPUNOCHANGE** may be passed to read the current assignment. The *spu* **MPC_SPUFLOAT** may be used to break any specific-processor assignment. This allows the process to float to any processor.

NOTE: This call is *advisory*. If the scheduling policy for a process conflicts with this processor assignment, the scheduling policy takes precedence. For example, when a processor is ready to choose another process to execute, and the highest priority **SCHED_FIFO** process is bound to a different processor, that process will execute on the selecting processor rather than waiting for the specified processor to which it was bound.

If the process specified by *pid* is a multithreaded process, all LWPs (lightweight processes) in the target process with the same processor set binding as the target process will have their processor assignment changed to what is specified. The processor set binding takes precedence over processor or locality domain binding.

MPC_SETPROCESS_FORCE

This call is identical to **MPC_SETPROCESS** except that the processor binding will take precedence over the scheduling policy. This call is synchronous. For example, when a processor is ready to choose another process to execute, and the highest priority **SCHED_FIFO** process is bound to a different processor, that

process will not be selected to execute on the selecting processor, but instead wait for the specified processor to which it was bound. The selecting processor will then choose a lower priority process to execute on the processor.

NOTE: This option will not guarantee compliance with POSIX real-time scheduling algorithms.

If the process specified by *pid* is a multithreaded process, all LWPs (lightweight processes) in the target process with the same processor set binding as the target process will have their processor assignment changed to what is specified. The processor set binding takes precedence over processor or locality domain binding.

MPC_SETLWP

This call is *advisory*. This request asynchronously assigns LWP (lightweight process) *lwpid* to processor *spu*. The new processor assignment is returned. This option can be used to change the processor assignment of LWPs in any process.

The *lwpid* **MPC_SELFLWPID** may be used to refer to the calling LWP.

The *spu* **MPC_SPUNOCHANGE** may be passed to read the current assignment. The *spu* **MPC_SPUFLOAT** may be used to break any specific-processor assignment. This allows the LWP to float to any processor.

NOTE: This call is *advisory*. If the scheduling policy for a LWP conflicts with this processor assignment, the scheduling policy takes precedence. For example, when a processor is ready to choose another LWP to execute, and the highest priority **SCHED_FIFO** LWP is bound to a different processor, then the LWP will execute on the selecting processor rather than waiting for the specified processor to which it was bound.

MPC_SETLWP_FORCE

This call is identical to **MPC_SETLWP** except that the processor binding will take precedence over the scheduling policy. This call is synchronous. For example, when a processor is ready to choose another LWP to execute, and the highest priority **SCHED_FIFO** LWP is bound to a different processor, that LWP will not be selected to execute on the selecting processor, but instead will wait for the specified processor to which it was bound. The selecting processor will then choose a lower priority LWP to execute on the processor.

NOTE: This option will not guarantee compliance with POSIX real-time scheduling algorithms.

For locality domain binding use:

```
int mpctl(mpc_request_t request, ldom_t ldom, pid_t pid);
int mpctl(mpc_request_t request, ldom_t ldom, lwpid_t lwpid);
```

The *request* argument determines the precise action to be taken by **mpctl()** and is one of the following:

MPC_SETLDM

This request synchronously assigns process *pid* to locality domain *ldom*. The process may now run on any processor within the locality domain in its processor set. The new locality domain assignment is returned.

The *pid* **MPC_SELFPID** may be used to refer to the calling process.

The *ldom* **MPC_LDMNOCHANGE** may be passed to read the current assignment. The *ldom* **MPC_LDMFLOAT** may be used to break any specific-locality domain assignment. This allows the process to float to any locality domain.

When a processor in one locality domain is ready to choose another process to execute, and the highest priority **SCHED_FIFO** process is bound to a different locality domain, that process will not be selected to execute on the selecting processor, but instead wait for a processor in the specified locality domain to which it was bound. The selecting processor will then choose a lower priority process to execute on the processor.

NOTE: This option will not guarantee compliance with POSIX real-time scheduling algorithms.

If the process specified by *pid* is a multithreaded process, all LWPs (lightweight processes) in the target process will have their locality domain assignment changed to what is specified. However, if any LWP belongs to a processor set different from the target process, and if the specified locality domain does not contribute any processor to that locality domain, the binding assignment of such an LWP is not changed.

MPC_SETLWPLDOM

This request synchronously assigns LWP (lightweight process) *lwpid* to locality domain *ldom*. The LWP may now run on any processor within the locality domain. The new locality domain assignment is returned. This option can be used to change the locality domain assignment of LWPs in any process.

The *lwpid* **MPC_SELF_LWPID** may be used to refer to the calling LWP.

The *ldom* **MPC_LDOMNOCHANGE** may be passed to read the current assignment. The *ldom* **MPC_LDOMFLOAT** may be used to break any specific-locality domain assignment. This allows the LWP to float to any locality domain.

When a processor is ready to choose another LWP to execute, and the highest priority **SCHED_FIFO** LWP is bound to a processor in a different locality domain, then that LWP will not be selected to execute on the selecting processor, but instead will wait for a processor on the locality domain to which it was bound. The selecting processor will then choose a lower priority LWP to execute on the processor.

NOTE: This option will not guarantee compliance with POSIX real-time scheduling algorithms.

Obtaining Processor and Locality Domain Binding Type

These options return the current binding type for the specified process or LWP.

```
int mpctl(mpc_request_t request, spu_t spu, pid_t pid);
int mpctl(mpc_request_t request, spu_t spu, lwpid_t lwpid);
```

The *request* argument determines the precise action to be taken by **mpctl()** and is one of the following:

MPC_GETPROCESS_BINDINGTYPE

Warning: This call is OBSOLETE and is only provided for backwards compatibility.

This request returns **MPC_ADVISORY** or **MPC_MANDATORY** to indicate the current binding type of the process specified by *pid*. The *spu* argument is ignored. If the target process has a binding type of something other than **MPC_MANDATORY** the value **MPC_ADVISORY** will be returned.

MPC_GETPROCESS_BINDVALUE

This request returns the current binding type of the process specified by *pid*. The *spu* argument is ignored.

Current valid return values are **MPC_NO_BINDING** (no binding), **MPC_SPU_BINDING** (advisory processor binding), **MPC_SPU_FORCED_BINDING** (processor binding), and **MPC_LDOM_BINDING** (locality domain binding). Other binding types may be added in future releases and returned via this option. Applications using this option should be written to handle other return values in order to continue working on future releases.

MPC_GETLWP_BINDINGTYPE

Warning: This call is OBSOLETE and is only provided for backwards compatibility.

This request returns **MPC_ADVISORY** or **MPC_MANDATORY** to indicate the current binding type of the LWP specified by *lwpid*. The *spu* argument is ignored. If the target LWP has a binding type of something other than **MPC_MANDATORY** the value **MPC_ADVISORY** will be returned.

MPC_GETLWP_BINDVALUE

This request returns the current binding type of the LWP specified by *lwpid*. The *spu* argument is ignored.

| m |

Current valid return values are **MPC_NO_BINDING** (no binding), **MPC_SPU_BINDING** (advisory processor binding), **MPC_SPU_FORCED_BINDING** (processor binding), and **MPC_LDOM_BINDING** (locality domain binding). Other binding types may be added in future releases and returned via this option. Applications using this option should be written to handle other return values in order to continue working on future releases.

Launch Policies

Each process shall have a launch policy. Each lightweight process shall have a launch policy. The launch policy for a lightweight process need not match the launch policy for the process. The launch policy determines the locality domain where the newly created process or LWP will be launched in a ccNUMA system. The locality domains covered by a process's or LWP's processor set are the available locality domains.

When a process creates another process (via **fork()** or **vfork()**), the child process will inherit the parent process's launch policy. The initial LWP in the child process will inherit the launch policy of the creating LWP (and not that of its process). Other LWPs in a multi-threaded process inherit their launch policy from the creating LWP.

For all launch policies, the target process or LWP is bound to the locality domain on which it was launched. The target is allowed to execute on any processor within that locality domain.

When setting a launch policy, if the target already has processor or locality domain binding, the existing binding will not be overwritten. Instead the locality domain in which the target is bound (whether locality domain binding or processor binding) will be used as the starting locality domain for implementing the launch policy.

When setting a process launch policy, the launch policy specified shall only be applied to the process. The launch policies of LWPs within the process shall not be affected.

The **mpctl()** interface currently supports the following launch policies:

MPC_LAUNCH_POLICY_RR
MPC_LAUNCH_POLICY_FILL
MPC_LAUNCH_POLICY_PACKED
MPC_LAUNCH_POLICY_LEASTLOAD
MPC_LAUNCH_POLICY_RR_TREE
MPC_LAUNCH_POLICY_FILL_TREE
MPC_LAUNCH_POLICY_NONE

| m |

When a launch policy is set for a process, it becomes the root of a new launch tree. The launch policy determines which processes become part of the launch tree. The new processes in the launch tree will be distributed among available locality domains based on the launch policy for that launch tree.

For **MPC_LAUNCH_POLICY_RR** and **MPC_LAUNCH_POLICY_FILL** launch policies, the root process and only its direct children form the launch tree. The new child process becomes the root of a new launch tree. Since the launch tree for these policies includes only the parent and its direct children, their distribution will be more deterministic.

For **MPC_LAUNCH_POLICY_RR_TREE** and **MPC_LAUNCH_POLICY_FILL_TREE** launch policies, any new process created by the root process or any of its descendants become part of the launch tree. When creating a new process with these policies, if the root of the launch tree has different launch policy than the creator of the new process, the new process becomes the root of a new launch tree. The locality domains selected for new processes in the tree are dependent on the order in which they are created. So, the process distribution for an application with several levels in the launch tree may vary across different runs.

When the launch policy for a process in a launch tree is changed, it becomes the root of a new launch tree. However, the distribution of existing processes in the old launch tree is not changed.

The LWP launch policy works the same as process launch policy except that LWP launch tree is contained within a process. When an LWP with a launch policy creates a new process, the initial LWP in the new process becomes the root of a new LWP launch tree.

The **MPC_LAUNCH_POLICY_NONE** indicates there is no explicit launch policy for the process or LWP. The operating system is free to select the optimal distribution of processes and LWPs. No explicit locality domain binding is applied to new processes and LWPs with **MPC_LAUNCH_POLICY_NONE** policy, unless it inherits the binding from the creator process or LWP.

If the processor set binding for a process or an LWP in a launch tree is changed to another processor set, that process or LWP becomes the root of a new launch tree. When creating a new process or an LWP, if the root of the launch tree is found to be in a different processor set, the new process or LWP is made the root of a new launch tree.

NOTE: locality domains are tightly tied to the physical components of the underlying system. As a result, the performance observed when using launch policies based on locality domains may vary from system to system. For example, a system which contains 4 locality domains, each containing 32 processors, may exhibit different performance behaviors from a system that contains 32 locality domains with 4 processors per domain. The launch policy that provides optimal performance on one system may not provide optimal performance on a different system for the same application.

For process launch policies use:

```
int mpctl(mpc_request_t request, ldom_t ldom, pid_t pid);
```

The *request* argument determines the precise action to be taken by `mpctl()` and is one of the following:

MPC_GETPROCESS_LAUNCH

This request currently returns **MPC_LAUNCH_POLICY_RR**, **MPC_LAUNCH_POLICY_FILL**, **MPC_LAUNCH_POLICY_PACKED**, **MPC_LAUNCH_POLICY_LEASTLOAD**, **MPC_LAUNCH_POLICY_RR_TREE**, **MPC_LAUNCH_POLICY_FILL_TREE**, or **MPC_LAUNCH_POLICY_NONE** to indicate the current launch policy of the process specified by *pid*. Other launch policies may be added in future releases and returned via this option. Applications using this option should be written to handle other return values in order to continue working on future releases. The *ldom* argument is ignored.

MPC_SETPROCESS_RR

This call establishes a round robin launch policy (**MPC_LAUNCH_POLICY_RR**) for the specified process. The successive child processes are launched on different locality domains in a round robin manner until all available locality domains have been used by processes in the launch tree. At that point, the selection of locality domains begins again from the original locality domain. The *ldom* argument is ignored.

MPC_SETPROCESS_FILL

This call establishes a fill first launch policy (**MPC_LAUNCH_POLICY_FILL**) for the specified process. The successive child processes are launched on the same locality domain as their parent process until one process has been created for each available processor in the domain. At that point, a new locality domain is selected and successive processes are launched there until there is one process per processor. All available locality domains will be used before the original domain is selected again. The *ldom* argument is ignored.

MPC_SETPROCESS_PACKED

This call establishes a packed launch policy (**MPC_LAUNCH_POLICY_PACKED**) for the specified process. The successive child processes are launched on the same locality domain as their parent process. The *ldom* argument is ignored.

MPC_SETPROCESS_LEASTLOAD

This call establishes a least loaded launch policy (**MPC_LAUNCH_POLICY_LEASTLOAD**) for the specified process. The successive child processes are launched on the least loaded locality domain in the processor set regardless of the location of their parent process. The *ldom* argument is ignored.

MPC_SETPROCESS_RR_TREE

This call establishes a tree based round robin launch policy (**MPC_LAUNCH_POLICY_RR_TREE**) for the specified process. This request differs from **MPC_SETPROCESS_RR** in which processes become part of the launch tree. This launch policy includes all descendents of the target process in the launch tree. The *ldom* argument is ignored.

MPC_SETPROCESS_FILL_TREE

This call establishes a tree based fill first launch policy (**MPC_LAUNCH_POLICY_FILL_TREE**) for the specified process. This request differs from **MPC_SETPROCESS_FILL** in which processes become part of the

launch tree. This launch policy includes all descendants of the target process in the launch tree. The *ldom* argument is ignored.

MPC_SETPROCESS_NONE

This call unsets any launch policy in the process. The system will employ a default, optimal policy in determining where the newly created process will be launched. The existing binding of the process is not changed. The *ldom* argument is ignored.

For LWP launch policies use:

```
int mpctl(mpc_request_t request, ldom_t ldom, lwpid_t lwpid);
```

The *request* argument determines the precise action to be taken by **mpctl()** and is one of the following:

MPC_GETLWP_LAUNCH

This *request* currently returns **MPC_LAUNCH_POLICY_RR**, **MPC_LAUNCH_POLICY_FILL**, **MPC_LAUNCH_POLICY_PACKED**, **MPC_LAUNCH_POLICY_LEASTLOAD**, **MPC_LAUNCH_POLICY_RR_TREE**, **MPC_LAUNCH_POLICY_FILL_TREE**, or **MPC_LAUNCH_POLICY_NONE** to indicate the current launch policy of the LWP specified by *lwpid*. Other launch policies may be added in future releases and returned via this option. Applications using this option should be written to handle other return values in order to continue working on future releases. The *ldom* argument is ignored.

MPC_SETLWP_RR This call establishes a round robin launch policy (**MPC_LAUNCH_POLICY_RR**) for the specified LWP. The successive child LWPs are launched on different locality domains in a round robin manner until all available locality domains have been used by LWPs in the launch tree. At that point, the selection of locality domains begins again from the original locality domain. The *ldom* argument is ignored.

MPC_SETLWP_FILL

This call establishes a fill first launch policy (**MPC_LAUNCH_POLICY_FILL**) for the specified LWP. The successive child LWPs are launched on the same locality domain as their parent LWP until one thread has been created for each available processor in the domain. At that point, a new locality domain is selected and successive LWPs are launched there until there is one LWP per processor. All available locality domains will be used before the original domain is selected again. The *ldom* argument is ignored.

MPC_SETLWP_PACKED

This call establishes a packed launch policy (**MPC_LAUNCH_POLICY_PACKED**) for the specified LWP. The successive child LWPs are launched on the same locality domain as their parent LWP. The *ldom* argument is ignored.

MPC_SETLWP_LEASTLOAD

This call establishes a least loaded launch policy (**MPC_LAUNCH_POLICY_LEASTLOAD**) for the specified LWP. The successive child LWPs are launched on the least loaded locality domain in the processor set regardless of the location of their parent LWP. The *ldom* argument is ignored.

MPC_SETLWP_RR_TREE

This call establishes a tree based round robin launch policy (**MPC_LAUNCH_POLICY_RR_TREE**) for the specified LWP. This request differs from **MPC_SETPROCESS_RR** in which LWPs become part of the launch tree. This launch policy includes all descendants of the target LWP in the launch tree. The *ldom* argument is ignored.

MPC_SETLWP_FILL_TREE

This call establishes a tree based fill first launch policy (**MPC_LAUNCH_POLICY_FILL_TREE**) for the specified LWP. This request differs from **MPC_SETPROCESS_FILL** in which LWPs become part of the launch tree. This launch policy includes all descendants of the target LWP in the launch tree. The *ldom* argument is ignored.

MPC_SETLWP_NONE

This call unsets any launch policy in the LWP. The system will employ a

default, optimal policy in determining where the newly created LWP will be launched. The existing binding of the LWP is not changed. The *ldom* argument is ignored.

To change the processor assignment, locality domain assignment, or launch policy of another process, the caller must either have the same effective user ID as the target process, or have the **PRIV_MPCTL** privilege.

Security Restrictions

Some or all of the actions associated with this system call require the **MPCTL** privilege. Processes owned by the superuser have this privilege. Processes owned by other users may have this privilege, depending on system configuration. See *privileges(5)* for more information about privileged access on systems that support fine-grained privileges.

RETURN VALUES

If **mpctl()** fails, **-1** is returned. If **mpctl** is successful, the value returned is as specified for that command/option.

NOTE: In some cases a negative number other than **-1** may be returned that indicates a successful return.

ERRORS

In general, **mpctl()** fails if one or more of the following is true:

- | | |
|----------|--|
| [EACCES] | <i>pid</i> or <i>lwpid</i> identifies a process or LWP that is not visible to the calling thread. |
| [EINVAL] | <i>request</i> is an illegal number. |
| [EINVAL] | <i>request</i> is MPC_GETNEXTSPU or MPC_GETNEXTSPU_SYS and <i>spu</i> identifies the last processor. Or <i>request</i> is MPC_GETNEXTLDM or MPC_GETNEXTLDM_SYS and <i>ldom</i> identifies the last locality domain. Or <i>request</i> is MPC_GETNEXTPROXIMATESPU or MPC_GETNEXTPROXIMATESPU_SYS and <i>spu</i> identifies the last proximate spu. |
| [EINVAL] | <i>request</i> is MPC_GETNUMPROXIMATESPUS or MPC_GETNUMPROXIMATESPUS_SYS or MPC_GETFIRSTPROXIMATESPU or MPC_GETFIRSTPROXIMATESPU_SYS and <i>spu</i> is not enabled. |
| [EINVAL] | <i>request</i> is to bind a process or an LWP to a processor or locality domain that is not in the processor set of the specified process or LWP. |
| [EPERM] | <i>request</i> is MPC_SETPROCESS , MPC_SETPROCESS_FORCE , or MPC_SETLDM , <i>spu</i> is not MPC_SPUNOCHANGE or MPC_LDMNOCHANGE , <i>pid</i> identifies another process, and the caller does not have the same effective user ID of the target process or does not have the PRIV_MPCTL privilege. |
| [EPERM] | <i>request</i> is MPC_SETPROCESS_RR , MPC_SETPROCESS_FILL , MPC_SETPROCESS_PACKED , MPC_SETPROCESS_LEASTLOAD , MPC_SETPROCESS_RR_TREE , MPC_SETPROCESS_FILL_TREE , or MPC_SETPROCESS_NONE , <i>pid</i> identifies another process, and the caller does not have the same effective user ID of the target process, or does not have the PRIV_MPCTL privilege. |
| [ESRCH] | <i>pid</i> or <i>lwpid</i> identifies a process or LWP that does not exist. |

SEE ALSO

getprivgrp(1), setprivgrp(1M), fork(2), getprivgrp(2), sysconf(2), pthread_processor_bind_np(3T), pthread_launch_policy_np(3T), privgrp(4), compartments(5), privileges(5).

| m |