Memory Hot(Un)Plug

This document describes generic Linux support for memory hot(un)plug with a focus on System RAM, including ZONE_MOVABLE support.

Introduction

Memory hot(un)plug allows for increasing and decreasing the size of physical memory available to a machine at runtime. In the simplest case, it consists of physically plugging or unplugging a DIMM at runtime, coordinated with the operating system.

Memory hot(un)plug is used for various purposes:

- The physical memory available to a machine can be adjusted at runtime, up- or downgrading the memory capacity. This dynamic memory resizing, sometimes referred to as "capacity on demand", is frequently used with virtual machines and logical partitions.
- Replacing hardware, such as DIMMs or whole NUMA nodes, without downtime. One example is replacing failing memory modules.
- Reducing energy consumption either by physically unplugging memory modules or by logically unplugging (parts of) memory modules from Linux.

Further, the basic memory hot(un)plug infrastructure in Linux is nowadays also used to expose persistent memory, other performance-differentiated memory and reserved memory regions as ordinary system RAM to Linux.

Linux only supports memory hot(un)plug on selected 64 bit architectures, such as x86_64, arm64, ppc64, s390x and ia64.

Memory Hot(Un)Plug Granularity

Memory hot(un)plug in Linux uses the SPARSEMEM memory model, which divides the physical memory address space into chunks of the same size: memory sections. The size of a memory section is architecture dependent. For example, x86_64 uses 128 MiB and ppc64 uses 16 MiB.

Memory sections are combined into chunks referred to as "memory blocks". The size of a memory block is architecture dependent and corresponds to the smallest granularity that can be hot(un)plugged. The default size of a memory block is the same as memory section size, unless an architecture specifies otherwise.

All memory blocks have the same size.

Phases of Memory Hotplug

Memory hotplug consists of two phases:

1. Adding the memory to Linux
2. Onlining memory blocks

In the first phase, metadata, such as the memory map ("memmap") and page tables for the direct mapping, is allocated and initialized, and memory blocks are created; the latter also creates sysfs files for managing newly created memory blocks.

In the second phase, added memory is exposed to the page allocator. After this phase, the memory is visible in memory statistics,

such as free and total memory, of the system.

Phases of Memory Hotunplug

Memory hotunplug consists of two phases:

1. Offlining memory blocks
2. Removing the memory from Linux

In the fist phase, memory is "hidden" from the page allocator again, for example, by migrating busy memory to other memory locations and removing all relevant free pages from the page allocator After this phase, the memory is no longer visible in memory statistics of the system.

In the second phase, the memory blocks are removed and metadata is freed.

Memory Hotplug Notifications

There are various ways how Linux is notified about memory hotplug events such that it can start adding hotplugged memory. This description is limited to systems that support ACPI; mechanisms specific to other firmware interfaces or virtual machines are not described.

ACPI Notifications

Platforms that support ACPI, such as x86_64, can support memory hotplug notifications via ACPI.

In general, a firmware supporting memory hotplug defines a memory class object HID "PNP0C80". When notified about hotplug of a new memory device, the ACPI driver will hotplug the memory to Linux.

If the firmware supports hotplug of NUMA nodes, it defines an object _HID "ACPI0004", "PNP0A05", or "PNP0A06". When notified about an hotplug event, all assigned memory devices are added to Linux by the ACPI driver.

Similarly, Linux can be notified about requests to hotunplug a memory device or a NUMA node via ACPI. The ACPI driver will try offlining all relevant memory blocks, and, if successful, hotunplug the memory from Linux.

Manual Probing

On some architectures, the firmware may not be able to notify the operating system about a memory hotplug event. Instead, the memory has to be manually probed from user space.

The probe interface is located at:

```
/sys/devices/system/memory/probe
```

Only complete memory blocks can be probed. Individual memory blocks are probed by providing the physical start address of the memory block:

```
% echo addr > /sys/devices/system/memory/probe
```

Which results in a memory block for the range [addr, addr + memory_block_size) being created.

> **Note**
>
> Using the probe interface is discouraged as it is easy to crash the kernel, because Linux cannot validate user input; this interface might be removed in the future.

Onlining and Offlining Memory Blocks

After a memory block has been created, Linux has to be instructed to actually make use of that memory: the memory block has to be "online".

Before a memory block can be removed, Linux has to stop using any memory part of the memory block: the memory block has to be "offlined".

The Linux kernel can be configured to automatically online added memory blocks and drivers automatically trigger offlining of memory blocks when trying hotunplug of memory. Memory blocks can only be removed once offlining succeeded and drivers may trigger offlining of memory blocks when attempting hotunplug of memory.

Onlining Memory Blocks Manually

If auto-onlining of memory blocks isn't enabled, user-space has to manually trigger onlining of memory blocks. Often, udev rules are used to automate this task in user space.

Onlining of a memory block can be triggered via:

```
% echo online > /sys/devices/system/memory/memoryXXX/state
```

Or alternatively:

```
% echo 1 > /sys/devices/system/memory/memoryXXX/online
```

The kernel will select the target zone automatically, depending on the configured `online_policy`.

One can explicitly request to associate an offline memory block with ZONE_MOVABLE by:

```
% echo online_movable > /sys/devices/system/memory/memoryXXX/state
```

Or one can explicitly request a kernel zone (usually ZONE_NORMAL) by:

```
% echo online_kernel > /sys/devices/system/memory/memoryXXX/state
```

In any case, if onlining succeeds, the state of the memory block is changed to be "online". If it fails, the state of the memory block will remain unchanged and the above commands will fail.

Onlining Memory Blocks Automatically

The kernel can be configured to try auto-onlining of newly added memory blocks. If this feature is disabled, the memory blocks will stay offline until explicitly onlined from user space.

The configured auto-online behavior can be observed via:

```
% cat /sys/devices/system/memory/auto_online_blocks
```

Auto-onlining can be enabled by writing `online`, `online_kernel` or `online_movable` to that file, like:

```
% echo online > /sys/devices/system/memory/auto_online_blocks
```

Similarly to manual onlining, with `online` the kernel will select the target zone automatically, depending on the configured `online_policy`.

Modifying the auto-online behavior will only affect all subsequently added memory blocks only.

> **Note**
>
> In corner cases, auto-onlining can fail. The kernel won't retry. Note that auto-onlining is not expected to fail in default configurations.

> **Note**
>
> DLPAR on ppc64 ignores the `offline` setting and will still online added memory blocks; if onlining fails, memory blocks are removed again.

Offlining Memory Blocks

In the current implementation, Linux's memory offlining will try migrating all movable pages off the affected memory block. As most kernel allocations, such as page tables, are unmovable, page migration can fail and, therefore, inhibit memory offlining from succeeding.

Having the memory provided by memory block managed by ZONE_MOVABLE significantly increases memory offlining reliability; still, memory offlining can fail in some corner cases.

Further, memory offlining might retry for a long time (or even forever), until aborted by the user.

Offlining of a memory block can be triggered via:

```
% echo offline > /sys/devices/system/memory/memoryXXX/state
```

Or alternatively:

```
% echo 0 > /sys/devices/system/memory/memoryXXX/online
```

If offlining succeeds, the state of the memory block is changed to be "offline". If it fails, the state of the memory block will remain unchanged and the above commands will fail, for example, via:

```
bash: echo: write error: Device or resource busy
```

or via:

```
bash: echo: write error: Invalid argument
```

Observing the State of Memory Blocks

The state (online/offline/going-offline) of a memory block can be observed either via:

```
% cat /sys/device/system/memory/memoryXXX/state
```

Or alternatively (1/0) via:

```
% cat /sys/device/system/memory/memoryXXX/online
```

For an online memory block, the managing zone can be observed via:

```
% cat /sys/device/system/memory/memoryXXX/valid_zones
```

Configuring Memory Hot(Un)Plug

There are various ways how system administrators can configure memory hot(un)plug and interact with memory blocks, especially, to online them.

Memory Hot(Un)Plug Configuration via Sysfs

Some memory hot(un)plug properties can be configured or inspected via sysfs in:

```
/sys/devices/system/memory/
```

The following files are currently defined:

`auto_online_blocks`	read-write: set or get the default state of new memory blocks; configure auto-onlining. The default value depends on the CONFIG_MEMORY_HOTPLUG_DEFAULT_ONLINE kernel configuration option. See the `state` property of memory blocks for details.
`block_size_bytes`	read-only: the size in bytes of a memory block.
`probe`	write-only: add (probe) selected memory blocks manually from user space by supplying the physical start address. Availability depends on the CONFIG_ARCH_MEMORY_PROBE kernel configuration option.
`uevent`	read-write: generic udev file for device subsystems.

> **Note**
>
> When the CONFIG_MEMORY_FAILURE kernel configuration option is enabled, two additional files `hard_offline_page` and `soft_offline_page` are available to trigger hwpoisoning of pages, for example, for testing purposes. Note that this functionality is not really related to memory hot(un)plug or actual offlining of memory blocks.

Memory Block Configuration via Sysfs

Each memory block is represented as a memory block device that can be onlined or offlined. All memory blocks have their device information located in sysfs. Each present memory block is listed under `/sys/devices/system/memory` as:

```
/sys/devices/system/memory/memoryXXX
```

where XXX is the memory block id; the number of digits is variable.

A present memory block indicates that some memory in the range is present; however, a memory block might span memory holes. A memory block spanning memory holes cannot be offlined.

For example, assume 1 GiB memory block size. A device for a memory starting at 0x100000000 is `/sys/device/system/memory/memory4`:

```
(0x100000000 / 1Gib = 4)
```

This device covers address range [0x100000000 ... 0x140000000)

The following files are currently defined:

`online`	read-write: simplified interface to trigger onlining / offlining and to observe the state of a memory block. When onlining, the zone is selected automatically.
`phys_device`	read-only: legacy interface only ever used on s390x to expose the covered storage increment.
`phys_index`	read-only: the memory block id (XXX).
`removable`	read-only: legacy interface that indicated whether a memory block was likely to be offlineable or not. Nowadays, the kernel return `1` if and only if it supports memory offlining.

state	read-write: advanced interface to trigger onlining / offlining and to observe the state of a memory block. When writing, online, offline, online_kernel and online_movable are supported. online_movable specifies onlining to ZONE_MOVABLE. online_kernel specifies onlining to the default kernel zone for the memory block, such as ZONE_NORMAL. online let's the kernel select the zone automatically. When reading, online, offline and going-offline may be returned.
uevent	read-write: generic uevent file for devices.
valid_zones	read-only: when a block is online, shows the zone it belongs to; when a block is offline, shows what zone will manage it when the block will be onlined. For online memory blocks, DMA, DMA32, Normal, Movable and none may be returned. none indicates that memory provided by a memory block is managed by multiple zones or spans multiple nodes; such memory blocks cannot be offlined. Movable indicates ZONE_MOVABLE. Other values indicate a kernel zone. For offline memory blocks, the first column shows the zone the kernel would select when onlining the memory block right now without further specifying a zone. Availability depends on the CONFIG_MEMORY_HOTREMOVE kernel configuration option.

Note

If the CONFIG_NUMA kernel configuration option is enabled, the memoryXXX/ directories can also be accessed via symbolic links located in the /sys/devices/system/node/node* directories.

For example:

```
/sys/devices/system/node/node0/memory9 -> ../../memory/memory9
```

A backlink will also be created:

```
/sys/devices/system/memory/memory9/node0 -> ../../node/node0
```

Command Line Parameters

Some command line parameters affect memory hot(un)plug handling. The following command line parameters are relevant:

memhp_default_state	configure auto-onlining by essentially setting /sys/devices/system/memory/auto_online_blocks.
movable_node	configure automatic zone selection in the kernel when using the contig-zones online policy. When set, the kernel will default to ZONE_MOVABLE when onlining a memory block, unless other zones can be kept contiguous.

See Documentation/admin-guide/kernel-parameters.txt for a more generic description of these command line parameters.

Module Parameters

Instead of additional command line parameters or sysfs files, the memory_hotplug subsystem now provides a dedicated namespace for module parameters. Module parameters can be set via the command line by predicating them with memory_hotplug. such as:

```
memory_hotplug.memmap_on_memory=1
```

and they can be observed (and some even modified at runtime) via:

```
/sys/module/memory_hotplug/parameters/
```

The following module parameters are currently defined:

memmap_on_memory	read-write: Allocate memory for the memmap from the added memory block itself. Even if enabled, actual support depends on various other system properties and should only be regarded as a hint whether the behavior would be desired. While allocating the memmap from the memory block itself makes memory hotplug less likely to fail and keeps the memmap on the same NUMA node in any case, it can fragment physical memory in a way that huge pages in bigger granularity cannot be formed on hotplugged memory.

online_policy	read-write: Set the basic policy used for automatic zone selection when onlining memory blocks without specifying a target zone. `contig-zones` has been the kernel default before this parameter was added. After an online policy was configured and memory was online, the policy should not be changed anymore. When set to `contig-zones`, the kernel will try keeping zones contiguous. If a memory block intersects multiple zones or no zone, the behavior depends on the `movable_node` kernel command line parameter: default to ZONE_MOVABLE if set, default to the applicable kernel zone (usually ZONE_NORMAL) if not set. When set to `auto-movable`, the kernel will try onlining memory blocks to ZONE_MOVABLE if possible according to the configuration and memory device details. With this policy, one can avoid zone imbalances when eventually hotplugging a lot of memory later and still wanting to be able to hotunplug as much as possible reliably, very desirable in virtualized environments. This policy ignores the `movable_node` kernel command line parameter and isn't really applicable in environments that require it (e.g., bare metal with hotunpluggable nodes) where hotplugged memory might be exposed via the firmware-provided memory map early during boot to the system instead of getting detected, added and onlined later during boot (such as done by virtio-mem or by some hypervisors implementing emulated DIMMs). As one example, a hotplugged DIMM will be onlined either completely to ZONE_MOVABLE or completely to ZONE_NORMAL, not a mixture. As another example, as many memory blocks belonging to a virtio-mem device will be onlined to ZONE_MOVABLE as possible, special-casing units of memory blocks that can only get hotunplugged together. *This policy does not protect from setups that are problematic with ZONE_MOVABLE and does not change the zone of memory blocks dynamically after they were onlined.*
auto_movable_ratio	read-write: Set the maximum MOVABLE:KERNEL memory ratio in % for the `auto-movable` online policy. Whether the ratio applies only for the system across all NUMA nodes or also per NUMA nodes depends on the `auto_movable_numa_aware` configuration. All accounting is based on present memory pages in the zones combined with accounting per memory device. Memory dedicated to the CMA allocator is accounted as MOVABLE, although residing on one of the kernel zones. The possible ratio depends on the actual workload. The kernel default is "301" %, for example, allowing for hotplugging 24 GiB to a 8 GiB VM and automatically onlining all hotplugged memory to ZONE_MOVABLE in many setups. The additional 1% deals with some pages being not present, for example, because of some firmware allocations. Note that ZONE_NORMAL memory provided by one memory device does not allow for more ZONE_MOVABLE memory for a different memory device. As one example, onlining memory of a hotplugged DIMM to ZONE_NORMAL will not allow for another hotplugged DIMM to get onlined to ZONE_MOVABLE automatically. In contrast, memory hotplugged by a virtio-mem device that got onlined to ZONE_NORMAL will allow for more ZONE_MOVABLE memory within *the same* virtio-mem device.
auto_movable_numa_aware	read-write: Configure whether the `auto_movable_ratio` in the `auto-movable` online policy also applies per NUMA node in addition to the whole system across all NUMA nodes. The kernel default is "Y". Disabling NUMA awareness can be helpful when dealing with NUMA nodes that should be completely hotunpluggable, onlining the memory completely to ZONE_MOVABLE automatically if possible. Parameter availability depends on CONFIG_NUMA.

ZONE_MOVABLE

ZONE_MOVABLE is an important mechanism for more reliable memory offlining. Further, having system RAM managed by ZONE_MOVABLE instead of one of the kernel zones can increase the number of possible transparent huge pages and dynamically allocated huge pages.

Most kernel allocations are unmovable. Important examples include the memory map (usually 1/64ths of memory), page tables, and kmalloc(). Such allocations can only be served from the kernel zones.

Most user space pages, such as anonymous memory, and page cache pages are movable. Such allocations can be served from ZONE_MOVABLE and the kernel zones.

Only movable allocations are served from ZONE_MOVABLE, resulting in unmovable allocations being limited to the kernel zones. Without ZONE_MOVABLE, there is absolutely no guarantee whether a memory block can be offlined successfully.

Zone Imbalances

Having too much system RAM managed by ZONE_MOVABLE is called a zone imbalance, which can harm the system or degrade performance. As one example, the kernel might crash because it runs out of free memory for unmovable allocations, although there is still plenty of free memory left in ZONE_MOVABLE.

Usually, MOVABLE:KERNEL ratios of up to 3:1 or even 4:1 are fine. Ratios of 63:1 are definitely impossible due to the overhead for the memory map.

Actual safe zone ratios depend on the workload. Extreme cases, like excessive long-term pinning of pages, might not be able to deal with ZONE_MOVABLE at all.

> **Note**
>
> CMA memory part of a kernel zone essentially behaves like memory in ZONE_MOVABLE and similar considerations apply, especially when combining CMA with ZONE_MOVABLE.

ZONE_MOVABLE Sizing Considerations

We usually expect that a large portion of available system RAM will actually be consumed by user space, either directly or indirectly via the page cache. In the normal case, ZONE_MOVABLE can be used when allocating such pages just fine.

With that in mind, it makes sense that we can have a big portion of system RAM managed by ZONE_MOVABLE. However, there are some things to consider when using ZONE_MOVABLE, especially when fine-tuning zone ratios:

- Having a lot of offline memory blocks. Even offline memory blocks consume memory for metadata and page tables in the direct map; having a lot of offline memory blocks is not a typical case, though.
- Memory ballooning without balloon compaction is incompatible with ZONE_MOVABLE. Only some implementations, such as virtio-balloon and pseries CMM, fully support balloon compaction.

  Further, the CONFIG_BALLOON_COMPACTION kernel configuration option might be disabled. In that case, balloon inflation will only perform unmovable allocations and silently create a zone imbalance, usually triggered by inflation requests from the hypervisor.
- Gigantic pages are unmovable, resulting in user space consuming a lot of unmovable memory.
- Huge pages are unmovable when an architectures does not support huge page migration, resulting in a similar issue as with gigantic pages.
- Page tables are unmovable. Excessive swapping, mapping extremely large files or ZONE_DEVICE memory can be problematic, although only really relevant in corner cases. When we manage a lot of user space memory that has been swapped out or is served from a file/persistent memory/... we still need a lot of page tables to manage that memory once user space accessed that memory.
- In certain DAX configurations the memory map for the device memory will be allocated from the kernel zones.
- KASAN can have a significant memory overhead, for example, consuming 1/8th of the total system memory size as (unmovable) tracking metadata.
- Long-term pinning of pages. Techniques that rely on long-term pinnings (especially, RDMA and vfio/mdev) are fundamentally problematic with ZONE_MOVABLE, and therefore, memory offlining. Pinned pages cannot reside on ZONE_MOVABLE as that would turn these pages unmovable. Therefore, they have to be migrated off that zone while pinning. Pinning a page can fail even if there is plenty of free memory in ZONE_MOVABLE.

  In addition, using ZONE_MOVABLE might make page pinning more expensive, because of the page migration overhead.

By default, all the memory configured at boot time is managed by the kernel zones and ZONE_MOVABLE is not used.

To enable ZONE_MOVABLE to include the memory present at boot and to control the ratio between movable and kernel zones there are two command line options: `kernelcore=` and `movablecore=`. See Documentation/admin-guide/kernel-parameters.rst for their description.

Memory Offlining and ZONE_MOVABLE

Even with ZONE_MOVABLE, there are some corner cases where offlining a memory block might fail:

- Memory blocks with memory holes; this applies to memory blocks present during boot and can apply to memory blocks

hotplugged via the XEN balloon and the Hyper-V balloon.

- Mixed NUMA nodes and mixed zones within a single memory block prevent memory offlining; this applies to memory blocks present during boot only.
- Special memory blocks prevented by the system from getting offlined. Examples include any memory available during boot on arm64 or memory blocks spanning the crashkernel area on s390x; this usually applies to memory blocks present during boot only.
- Memory blocks overlapping with CMA areas cannot be offlined, this applies to memory blocks present during boot only.
- Concurrent activity that operates on the same physical memory area, such as allocating gigantic pages, can result in temporary offlining failures.
- Out of memory when dissolving huge pages, especially when freeing unused vmemmap pages associated with each hugetlb page is enabled.

  Offlining code may be able to migrate huge page contents, but may not be able to dissolve the source huge page because it fails allocating (unmovable) pages for the vmemmap, because the system might not have free memory in the kernel zones left.

  Users that depend on memory offlining to succeed for movable zones should carefully consider whether the memory savings gained from this feature are worth the risk of possibly not being able to offline memory in certain situations.

Further, when running into out of memory situations while migrating pages, or when still encountering permanently unmovable pages within ZONE_MOVABLE (-> BUG), memory offlining will keep retrying until it eventually succeeds.

When offlining is triggered from user space, the offlining context can be terminated by sending a fatal signal. A timeout based offlining can easily be implemented via:

```
% timeout $TIMEOUT offline_block | failure_handling
```