Memory Partitioning

(Design and Architecture Overview)

General Concepts

Partitioning is the means for dividing up resources for use amongst a collection of programs. A partition is the entity which represents some fraction of a resource and the rules on how that resource can be used.

Resources include basic objects like processor cycles and program store but also higher level objects like buffers, page tables, file descriptors, etc. There may also be different classes (or types) of a similar resource. Program store, or memory is the resource of concern in this document.

A memory partition is considered an abstract entity with a specific set of attributes. By associating processes with memory partitions, you are applying a set of resource constraints (partitioning rules) to the allocation of memory by associated processes for the purpose of object creation. This applies regardless of whether the allocations are the result of programmed memory allocations or memory allocations as the result of the flow of execution (ie. stack space, shared libraries, etc).

When a process creates objects, the class of memory that the object requires may be different than for other objects the process creates. The concept of memory classes is supported by the memory partitioning design such that different classes of memory (memory with different attributes) can be independently partitioned. If a particular process requires a particular class of memory, that process can be associated with a partition of that class, independent of the memory class requirements of any other process.

When partitioning memory, each memory class is partitioned independently. That is, a memory partition represents a single class of memory and therefore if a process creates objects that require different classes of memory, it must necessarily be associated with multiple partitions. Go back and read this again until you understand it!

A mechanism to “group” partitions is provided to allow a single, arbitrary name to represent a collection of *pseudo* partitions. A *pseudo* partition represents a *real* partition by a different name. This capability allows a process to associate with multiple partitions (like a scheduling partition and one or more memory partitions) using a single partition name (*see pseudo partitions below*).

Of importance to understand as well, is that although it is convenient to consider that it is processes that are associated with memory partitions, it is really the objects created by a process that are associated with a memory partition. This concept is what allows the existence of persistent objects (ie. objects like named shared memory) to exist beyond the lifetime of a process and still be governed by the constraints of the partition with which they were associated. Of course when no such persistent objects exist, the association of a process with a partition and the objects a process creates with a partition are analogous.

For simplicity, the use of process association is used throughout this document with the understanding that it is the objects created by associated processes that is implied.

Memory partitions of a given memory class may nest. Newly created memory partitions can subdivide a parent memory partition and form a hierarchy. This concept ensures that a process which creates a new partition, is constrained by the attributes of the partition that it is currently associated with. This “container” model provides security and is analogous to the operation of the adaptive partition scheduler.

Finally, as previously stated, partitions of different memory classes or types can be created and processes can be independently associated with one or more of these partitions as required. The POSIX typed memory interfaces can be used by processes to provide the basis for a simple class based memory allocation facility and the memory partitioning implementation will support the enforcement of partitioning rules based on the attributes specified on a per memory class basis, even if only one memory class, system memory, is utilized.

Memory Partition Attributes and Policies

As previously stated, a partition is an entity which represents the attributes and rules which govern the use of a resource. This section describes the partition attributes and rules that permit the effective use of memory partitioning by user programs.

All memory partitions have the following configurable attributes.

i. minimum size
   the minimum size attribute specifies the amount of a particular class of memory that has been reserved for exclusive use by objects which are associated with the partition.
ii. maximum size
   the maximum size attribute specifies the amount of a particular class of memory that can never be exceeded for use by objects which are associated with the partition.

Some concepts that emerge as a result of the aforementioned attributes are as follows

Reserved memory

- a non zero minimum value represents a reservation and not an allocation. That is, when a non zero minimum value is successfully configured for a partition, the memory is reserved for exclusive use by that partition in the allocator for that memory class. It is not preallocated
- allocations made against partitions with reservations will be accounted against unused reservations first and against available discretionary memory second

Discretionary memory

- The difference between the maximum and minimum values for a partition represents the amount of discretionary memory of that class available for the creation of objects associated with the partition. The value can be zero.
- Access to discretionary memory is dependent upon system partition configuration and is never guaranteed to be available

Memory Partition Types

Based upon the attributes and rules, memory partitions can be organized into four types or configurations.

1. Open
   minimum = 0, maximum = inf, discretionary only
   memory allocations made for objects associated with this partition type are subject to the availability of discretionary memory. There is no limit (beyond the limit of physical memory available to the allocator for that class) to the amount of memory that may be requested. Similarly, there is no guarantee that allocations made for objects associated with this partition will ever succeed.
   This type creates an environment similar to the case when there is no memory partitioning. If memory is available, the allocation request will succeed (notwithstanding factors unrelated to availability), otherwise it will fail

2. Guarantee
   minimum = N, maximum = inf, reserved + discretionary
   N bytes of the respective memory class is reserved and available for allocation to objects associated with the partition. There is no limit (beyond the limit of physical memory available to the allocator of that class) to the amount of memory that may be requested.
   This type can be used when it is known a priori that the objects associated with this partition will require a specific amount of memory but that any further memory requirements should be governed by the rules for discretionary memory allocations

3. Restricted
   minimum = 0, maximum = N, discretionary only
   memory allocations made for objects associated with this partition type are subject to the availability of discretionary memory and furthermore will not exceed the limit of N bytes.
   This type can be used when it is not known what the memory requirements are for the objects associated with this partition but that under no circumstances should the N bytes limit be exceeded

4. Sandbox
   minimum = N, maximum = N, reserved only
   N bytes, and only N bytes of the respective memory class is reserved and available for allocation to objects associated with the partition.
   This type can be used when it is precisely known what the memory requirements are for the objects associated with this partition or for situations where a course subdivision of the available memory class is desired. This can be used to hand out chunks of memory for further subdivision (using any of the defined partition types) by others

A fifth configuration which is not formally defined as a type combines the guarantee and restricted types. This configuration specifies a reserved amount but allows for a “not to exceed” buffer of discretionary allocations. This configuration can be used to assist in “tuning” a sandbox or for accommodating transient memory allocations in a guarantee partition.

<u>Memory Classes</u>

As previously stated, memory partitioning is performed on a per memory class basis. Memory classes are necessarily system specific however at least one memory class is always always present and automatically configured ... the sysram (system ram) class.

A memory class does not (necessarily) refer to a physical chunk of memory however it is most often the case that alternate classes of memory reside in separate physical chunks. Regardless, memory partitioning does not impose any particular attributes to memory classes so the suitability of a particular class of memory to a particular use is not the domain of memory partitioning. However, by appropriately configuring memory classes, the ability to partition, and hence manage them, is provided.

Memory classes are made available to the system via startup in the system page. They are made available for partitioning when they are introduced into the `/partition/mem` name space. With the exception of system ram, the POSIX typed memory interfaces are (currently) the means by which non sysram memory classes are made available for partitioning and subsequent allocation.

<u>The Name space</u>

Partitioning in general will take over the ***/partition/*** name space. Partitioning utilizes the POSIX name space to provide access to features of the partitioning modules. Memory and scheduling partitioning will take over the ***/partition/mem/*** and ***/partition/sched/*** name spaces respectively.

The reason for using the name space for access to the memory partitioning module is that it provides the following

- rudimentary access control through POSIX permissions
- the ability to view partition topology using well known and understood utilities (like ‘ls’)
- use of existing POSIX API’s (open(), close())

All configured memory classes will be made available in the name space as `/partition/mem/<memclass>`. The system ram memory class will always be present as `/partition/mem/sysram`. This memory class is used to account all non-specified memory allocations, like those from the `*alloc()` family of calls.

Before a memory class can be partitioned, the memory class is added to the system by creating a new entry in `/partition/mem/` using the memory classes name as registered in the system page. If a name is created under `./mem`, it is first checked to see if it is a defined memory class. If so, the class is added and may then be partitioned. If it does not, the name is treated as a *group* partition name (see ***Pseudo Partitions and Partition Groups***).

Since partitioning takes place on a per memory class basis, when a partition is created, it is created in the name space under the respective memory class. An example of this is the system partition which is created

by default as `/partition/mem/sysram/sys`. This partition is used to account all kernel memory allocations and all allocations by user processes which have not otherwise associated with a different partition. That is to say, that if additional partitions are not created, all allocations are accounted to the system partition. This is equivalent to the situation when memory partitioning is not used however the ability to obtain system ram and system partition metrics is provided.
If user processes will use their own partition(s) for sysram, then they should be created under the `./sysram` memory class.

A partition of a memory class (ie a *real* partition) will reside in the name space using the fully qualified path name ***/partition/mem/<memclass>/<partition>/*** and under the partition name will be visible the set of processes which are associated with that partition.
For example,

```
ls /partition/mem/sysram/sys
```

would produce a listing of all processes which are associated with the system partition.

```
1/        147468/   172047/   188435/   233493/   307223/   6/        90123/
12292/    159758/   188433/   188436/   237590/   4104/     7/        90125/
139273/   167952/   188434/   2/        3/        5/        73738/
```

and

```
ls /partition/mem/sysram/sys/1
```

would produce the following listing

```
partition/   as
```

All of the manipulations which could be performed on a process via the procfs path ***/proc/<pid>/*** can also be performed using the path ***/partition/mem/<memclass>/<partition>/<pid>/***.

When inspecting the name space ***/proc/<pid>/***, an additional entry will be present which exposes the partition name space for ***/proc/<pid>/***. This is the means by which the partitions which are associated with a specific process are exposed in the ***/proc/*** name space and as we will see later in the document, allows the modification of partition inheritance behaviours.

For example,

```
ls /proc/1
```

would produce the following listing (similar to above)

```
partition/   as
```

and

```
ls /proc/1/partition
```

would produce the following listing

```
mem/sysram/sys
```

However this view, unlike the ***/partition/mem/sysram/sys/*** view, does not expose the contents of ***./sys/*** as this would lead to name space recursion and confusion. Similarly, the ***/partition/mem/sysram/sys/<pid>/*** view does not expose the contents of ***./partition/*** for the same reasons.

In summary,
/proc/<pid>/partition/ exposes all of the partitions associated with ***/proc/<pid>***
/partition/mem/<memclass>/<partition>/ exposes all of the processes associated with ***./<partition>***

Memory Partition Configurations

Using the various partition types, memory partitions can be organized into various topologies or configurations that will allow them to be useful. By default the memory partitioning module will create a *system partition* of the *sysram* memory class. This partition is known as a root partition. The definition of a root partition is a partition which does not have a parent partition.

Because a certain amount of system ram is required during startup and for the kernel itself, the size of the system partition after startup determines what memory is available for partitioning. The system partition is created as an Open partition type by default but can be configured as a Guarantee partition with the procnto -R <size> option in the build file.

It is also possible to use the system partition as the root of a partition hierarchy although this is generally not advisable since the system partition can be impacted by the the configuration of the hierarchy.

Partition topologies can either be flat, in which all partitions are root partitions, or hierarchical, in which at least one root partition exists with 1 or more child partitions beneath it.

In a flat topology, the attributes specified for a partition are not based on the attributes of any other partition. Partitions of any type can be created so long as the rules for creation are satisfied. For example, you could not create 5 - 32 MB sandbox partitions of the same memory class with only 128 MB of physical memory of that class.

In a hierarchical topology, the attributes of a parent partition affect the attributes of the child. The following rules are used in a partition hierarchy

The rule of subdivision

- when a partition is created as the child of an existing partition, a non zero minimum configured in the child will be accounted as an allocation to the parent partition. This means that if the parent partition has any unallocated reserved memory, it will be used to satisfy some or all of the child reservation. This reservation is accounted up the entire partition hierarchy until is fully accounted for in either a parent partition or some combination of parent partitions and the allocator for the class.
- All allocations made against a partition which resides within a partition hierarchy, are also accounted within the entire hierarchy above the respective partition.

Partition rules govern whether the allocation of memory will be allowed to proceed, not whether the memory will be successfully allocated. There are many other reasons unrelated to availability why the allocator for a given class of memory may be unable to satisfy an allocation request.

The rules governing the use of the aforementioned attributes are as follows

- the maximum size attribute is always >= the minimum size attribute
- the minimum/maximum size attributes have a range from 0 -> infinity and are represented as 64 bit types
- minimum and maximum values can be modified subject to the aforementioned rules

Pseudo Partitions and Partition Groups

Up until now, the description of a partition has referred specifically to a *real* partition of a memory class. Pseudo partitions and partition group names provide a means of grouping *real* partitions for the purpose of user convenience.

A *real* partition has attributes and policies as described above and refers to partition of an actual resource. A *pseudo* partition is simply a name space reference to a *real* partition. A *partition group* is simply a name space reference to multiple *pseudo* partitions. Partition *group names* and *pseudo* partitions simplify process associations by allowing a process to be associated with a single *group name*. This eliminates the user having to associate a process with all of the partitions for each memory class it requires.

Partition groups and *pseudo* partitions is also the means by which partitions of different resources (like scheduling and one or more memory partitions) referenced using a common partition name.

An example configuration follows

real partitions ...
```
/partition/mem/sysram/sys
/partition/mem/sysram/p0
/partition/mem/”class 1”/p1
/partition/mem/”class 2”/p3
/partition/mem/”class 3”/p0
```

pseudo partitions ...

```
/partition/mem/my_pseudo_p0/p1 --> /proc/partition/mem/sysram/p0
/partition/mem/my_pseudo_p0/p2 --> /proc/partition/mem/"class 1"/p1
/partition/mem/my_pseudo_p0/p3 --> /proc/partition/mem/"class 2"/p3
/partition/mem/my_pseudo_p0/p4 --> /proc/partition/mem/"class 3"/p0
```

In this example, a partition group, `my_pseudo_p0` has been created to collectively refer to the partitions of 4 different memory classes.
Processes can simply associate with *pseudo* partition `/partition/mem/my_pseudo_p0` and they will have access to the 4 memory classes sysram, “class 1”, “class 2”, “class 3” and have their allocations governed by the attributes and policies of the respective *real* partitions p0, p1, p3, p0.

<u>**Process Association with Memory Partitions**</u>

When a process is created, it must be associated with at least one partition and that partition must be of the system memory class. This is required in order to satisfy allocations necessary to even create the process. Association with partitions of other memory classes is only required if the process creates objects which require those memory classes.
There will be 2 mechanisms to establish the partition associations of a child process.

1. Inheritance from the parent process
2. Explicitly specified using `posix_spawn()` and an appropriately initialized `posix_spawnattr_t` object

<u>Inheritance</u>

Partition inheritance is required in order to satisfy partition associations in the case where they can not be explicitly specified. This is the case for `fork()` and `spawn()`. The API’s simply provide no explicit control over partition associations. In order to resolve this, there are 2 behaviours, a default and an alternate. The default behaviour, as the name suggests, will take effect without doing anything extra. The alternative behaviour will be available through `posix_spawn()` (see discussion below).

Default Behaviour

By default, all of the memory partitions associated with the parent (creating) process which do not have their `mempart_flags_NO_INHERIT` flag set, will be inherited by the child (created) process.

Alternate Behaviour

By default, only the sysram memory class partition associated with the parent (creating) process will be inherited by the by the child (created) process.

The `mempart_flags_NO_INHERIT` flag can be set on a per partition basis for each process using the DCMD_ALL_SETFLAGS `devctl()`.

Explicitly Specified Associations - using posix_spawn()

The `posix_spawn()` call will behave similar to `fork()`/`spawn()` in terms of default behaviour however it provides an additional level of control via the `posix_spawn_file_actions_t` and `posix_spawnattr_t` objects. The combination of the new flag, `POSIX_SPAWN_SETMPART`, and the `posix_spawn_file_actions_t` and `posix_spawnattr_t` objects provides complete control over partition association as described in the following sections.

In order to utilize the `mempart_flags_NO_INHERIT` flag for the default behaviour, an additional file actions call, `posix_spawn_file_actions_devctl()`, will be defined which can be used to set or clear the `mempart_flags_NO_INHERIT` flag of a processes associated partitions. To explicitly control the inherited partition associations of the child process, the `posix_spawn_file_actions_t` object could be initialized as follows

```
posix_spawn_file_actions_addopen(.., fd, "/proc/<pid>/partition/mem/<memclass>/<partition>", ..);
posix_spawn_file_actions_adddevctl(.., fd, DCMD_ALL_GETFLAGS, &flags, ..);
flags &= ~mempart_flags_NO_INHERIT;          // to allow the partition to be inherited
    or
flags |= mempart_flags_NO_INHERIT;           // to prevent the partition from being inherited
posix_spawn_file_actions_adddevctl(.., fd, DCMD_ALL_SETFLAGS, &flags, ..);
posix_spawn_file_actions_addclose(.., fd);
```

This would be performed for each partition associated with the parent in order to modify the default inheritance behaviour.

The `mempart_flags_NO_INHERIT` flag is useful in controlling partition inheritance but it does not provide a means for associating a child process with partitions which the parent process is not associated with. In order to provide a means of associating a child process with a specific set of partitions, regardless of whether the parent process is associated with those partitions or not, the `POSIX_SPAWN_SETMPART` flag and `posix_spawnattr_t` attributes object will be utilized

The `posix_spawnattr_t` structure has been extended to accept a variable length array of partition name pointers. The names can be specified with a new `posix_spawnattr_set_mpart()` call. When calling `posix_spawn()` with a pointer *attrp* to the `posix_spawnattr_t` object the following partition inheritance behaviour will be obtained

- If the `POSIX_SPAWN_SETMPART` flag is **set** in the *spawn-flags* attribute of the object referenced by *attrp*, and the *spawn-mpart* attribute of the same object is **non-zero**, then the child's memory partition associations shall be as specified in the *spawn-mpart* attribute of the object referenced by *attrp*. There will be no implicit inheritance. Note that any `posix_spawn_file_actions_t` related to memory partitions will still be processed however because there is no inheritance, the results of those operations will be ignored

    This has the effect of allowing the calling process to explicitly specify which memory partitions the *child process should be associated with, including none. There will be no partition inheritance, although a partition which the parent is associated with can be specified for use by the child*

- If the `POSIX_SPAWN_SETMPART` flag is **set** in the *spawn-flags* attribute of the object referenced by *attrp*, and the *spawn-mpart* attribute of the same object is **zero**, then the alternative default behaviour for partition inheritance will be used (see above)

  This has the effect of allowing the caller to use the alternative partition inheritance behaviour. Because the POSIX_SPAWN_SETMPART flag must be explicitly specified, it will only take effect for applications which are "partitioning aware". The defined default behaviour will be used otherwise

- If the `POSIX_SPAWN_SETMPART` flag is **not set** in the *spawn-flags* attribute of the object referenced by *attrp*, then the default behaviour for partition inheritance will be used (see above)

  This has the effect of allowing posix_spawn() to provide default partition inheritance behaviour similar to fork() and spawn()

Shared Memory Allocations

Shared memory objects specifically are accounted to the partition in which they are created. This means that even if a process is associated with a different partition than the one which the shared memory object is associated with and that process `ftruncate()`'s the shared memory object to make it larger, the rules governing the allocation required to accomplish the `ftruncate()` are determined by the partition the shared memory object is associated with AND NOT the partition the `ftruncate()`'ing process is associated with. This requires that applications properly design their software so that shared memory objects are associated with the appropriate partition. It is a different problem, unrelated to memory partitioning, which establishes which processes may `ftruncate()` a shared memory object.

Memory Partition Metrics

Use of the path name space will provide the means to obtain memory partition metrics similar to other name space objects. This will allow the use of existing utilities as well as the creation of custom applications to obtain and display partition data.

The memory partition metrics will allow the user to suitably partition the available system memory and account for usage. Some internal data structures which represent an overhead to the operating system will be accounted for but will not otherwise be uniquely identifiable as being attributed to creation by a specific process.

Applications will be able to retrieve, as a minimum, the following information
- current configuration (attributes and policies)
- creation configuration (attributes and policies at the time the partition was created)
- current partition size

- highest partition size

Note that these metrics are only available from *real* partitions, not from *pseudo* (group name) partitions. This does not prevent an external application however from collecting data from all *real* partitions (utilizing a group partition name) and presenting the data in a suitable fashion.

Also note that similar information will be made available for each memory class as a whole.

Memory Partition Events

When an application attempts, either directly or indirectly, to allocate more memory than is permitted by the partition that the process is associated with, the behaviour will be identical to the situation in which no memory partitioning is being used and all of the available memory in a system has been allocated.

A suitably privileged process will be able to register for various types of partition events. These events may include

- size change events (threshold cross and delta)
- configuration change events
- process association and disassociation events
- child partition creation and destruction events

Also note that similar information will be made available for the memory class as a whole.

Issues of Security

Security, or “trusted” is a significant portion of a memory partitioning solution and in some instances may be the sole reason for its use. For the purpose of memory partitioning, the following context will be used

Configuration security

- the ability to prevent partition topology changes
- the ability to prevent illegal partition creation
- the ability to prevent partition destruction
- the ability the prevent partition configuration changes

Operational security

- the ability to ensure that guarantees are provided
- the ability to ensure that restrictions are enforced
- the ability to ensure that only authorized processes can be associated with partition

To these ends, the following mechanisms will be employed

POSIX file permissions

This mechanism will provide a rudimentary level of control based on the well understood user/group/world permissions.

- read permission allow metrics to be obtained as well as to register for events
- write permission allow configuration changes (including topology changes) to be made
- execute permissions allow association

Terminal Partition Policy

A terminal partition policy allows the partition to be configured such that no child partitions can be created. Once set TRUE, the policy can not be changed. This policy uniquely prevents the creation of child partitions although it does not prevent the changing of partition attributes. This policy can be used to prevent a hierarchical explosion of partitions while still allowing attribute modification (if appropriately privileged).

Configuration Lock Policy

The configuration lock policy allows all configuration attributes of a partition to be locked excluding the Terminal Partition policy. Once set TRUE, the policy can not be changed. This mechanism prevents any changes to be made to the partition including POSIX file permissions. It does however allow the creation of child partitions so that a locked down parent partition could be independently sub partitioned by an separate (appropriately privileged) organization.

Permanence Policy

The permanence policy prevents the destruction of a partition. Once set TRUE, the policy cannot be changed. This mechanism prevents partition removal independent of POSIX file permissions