Locking#

The reader/writer lock on the address space protects:

- *struct mm_aspace*
- *struct mm_mmap*
- the particular *struct mm_object_ref* for that aspace
- possibly CPU specific stuff (e.g. the page tables)

The mutex on the memory object protects:

- *OBJECT*
- *struct mm_object_ref* list
- possibly CPU specific stuff (e.g. the portion of a page table mapped by an object)

When both have to be locked (usually), the address space must be locked before the object. This has implications when we want to do something to all the references of an object (e.g. *vmm_resize*). In those cases we'll have the object already locked, so we can't start locking the aspaces without causing deadlock scenerios. Instead the *memref_walk* function sets the struct mm_map *inuse* field as it traverses to each one. When a *struct mm_map* is freed, it gets put on a free list without disturbing any data except the *next* pointer. Allocations from the list check the *inuse* field and skip over any that have a non-zero value.

Page Fault Handling

In order to find the corresponding *struct mm_map* for a given virtual address, we need to have the mapping structures locked. Normally, this is done by locking the address space. However, when we're in *vmm_fault*, we are executing as part of the kernel and as such we can't make kernel calls. Instead the function calls *map_fault_lock* to perform the locking. This function is allowed to fail. If it does, we just send a pulse and let *fault_pulse* handle things at process time, where we can reliably gain exclusive access. This doesn't cost us any extra code, since *fault_pulse* has to do all the validation checking again anyway - the address space mappings might have changed in the interval between the fault and the start of the *fault_pulse* routine.

mlock/munlock and friends

There are three locking states for memory regions:

unlocked
: may be paged in/out

locked
: may not be paged in/out, may still fault on access/reference to maintain usage/modification stats

super-locked
: (happens when I/O privity is granted to a thread) no faulting allowed at all and covers the whole address space

For non-MAP_LAZY mappings, a locked region (via mlock or mlockall) is made readable (if PROT_READ) immediately, but may not be immediately writable (if PROT_WRITE) to track modifications. A super-locked address space is always fully mapped immediately.

For MAP_LAZY mappings, memory is not allocated/mapped until first reference for any of the above types. Once it's been referenced, it obeys the above rules - that means that it's a programmer error to touch a MAP_LAZY area in a critical region (interrupts disabled or an ISR) that hasn't already been referenced.

The other mmap flags don't interact with the locking state.

The default state of memory before a locking function is performed is currently indeterminate until we can do performance tuning. It might be that the default state will be different depending on what the underlying object is (e.g. shared memory object vs file).