Physical Memory Defragmentation: Overview#

This feature comes from the problem represented by PR16405. As time passes, our physical memory starts to get fragmented. Eventually, even though there might be a significant amount of memory free in total, it is fragmented so that a request for a large piece of contiguous memory will fail.

Contiguous memory is often required for device drivers where the device uses DMA. The normal work-around is to ensure that all device drivers initialize early (before memory is fragmented) and hold onto their memory. This is a harsh restriction, particularly for embedded systems that might want to use different drivers depending on the actions of the user -- starting all possible device drivers simultaneously may not be feasible.

We have done work on the physical memory allocation algorithms in the 6.4.0 release which significantly reduces the amount of fragmentation that occurs. However, no matter how good your algorithms might be, certain usage patterns can lead to fragmentation.

The idea behind this feature is to defragment physical memory as necessary to support requests for contiguous memory.

It has been argued that the improvements in the 6.4.0 allocation algorithm are likely sufficient to satisfy immediate customer needs. However, no matter how smart our memory allocation algorithms, specific application behaviour can result in fragmented free memory. Consider a completely degenerate application that routinely allocates 8K of memory and then frees half of that. When such an application runs for long enough, it will reach a point where half of the system memory is free, but no free block is larger than 4K.

Thus, no matter how good our allocation routines are at avoiding fragmentation, in order to satisfy a request for contiguous memory it may be necessary to run some form of defragmentation algorithm.

Feature Description#

The term "fragmentation" can apply to both in-use memory and free memory. Memory that is in use by an application is considered fragmented if it is discontiguous (that is, a large allocation is satisfied with a number of smaller blocks of memory from different locations in the physical address map). Free memory is considered fragmented if it consists of small blocks separated by blocks of memory that are in use.

In disk-based file systems, fragmentation of in-use blocks is most important, as it impacts the read and write performance of the device. Fragmentation of free blocks is only important in that it leads to fragmentation of in-use blocks as new blocks are allocated. In general, users of disk-based systems don't care about allocating contiguous blocks, excepting as it impacts performance.

For the Neutrino memory system, both forms of fragmentation are important but for different reasons. If in-use memory is fragmented, it prevents the memory subsystem from using large page sizes to map the memory, which in turn leads to poorer performance than we might otherwise obtain. If free memory is fragmented, it prevents an application from allocating contiguous memory, which in turn might lead to complete failure of the application.

To satisfy the basic requirements of this feature, it is necessary to "defragment" free memory. That is, memory that is in use will be swapped for memory that is free in such a pattern that the free memory blocks coalesce into larger blocks that are sufficient to satisfy a request for contiguous memory.

However, we should not ignore the fragmentation of in-use memory so as not to prevent the use of large pages. In defragmenting free memory we must be careful to ensure that in-use memory fragmentation is not increased, and ideally our free memory defragmentation algorithm will decrease in-use memory fragmentation as well.

When an application allocates memory, it is provided by the operating system in quantums (a quantum is a 4K block of memory that exists on a 4K boundary). The operating system programs the MMU so that the application can reference the physical block of memory through a virtual address -- during operation the MMU translates a virtual address into a physical address. For example, a request for 16K of memory will be satisfied by four 4K quantums. The operating system will set aside the four physical blocks for the application and configure the MMU to ensure that the application can reference them through a 16K contiguous virtual address. However, these blocks may not be physically contiguous -- the operating system can arrange the MMU configuration (the virtual to physical mapping) so that non-contiguous physical addresses are accessed through contiguous virtual addresses.

The task of defragmentation consists of changing existing memory allocations and mappings to use different underlying physical pages. By swapping around the underlying physical quantums, we can consolidate the fragmented free blocks into contiguous runs. However, we must be careful to avoid moving certain types of memory where the virtual-to-physical mapping can't safely be changed. Memory allocated by the kernel and addressed through the one-to-one mapping area cannot be moved, as the one-to-one mapping area defines the mapping of virtual to physical addresses and we cannot change the physical address without also changing the virtual address. Memory that is locked by the application (see mlock) cannot be moved: by locking the memory, the application is indicating that moving the memory is not acceptable. An application that runs with io privity (see ThreadCtl _NTO_TCTL_IO flag) has all pages locked by default, as device drivers often require physical addresses. Finally, pages of memory that have mutex objects on them will not be moved in the initial version of this feature; while it would be possible to move these pages, mutex objects are registered with the kernel through their physical addresses, so moving a page with a mutex on it would require re-hashing the mutex object in the kernel -- this extra work will be skipped for now.

Defragmentation will be done, if necessary, when an application allocates a piece of contiguous memory. The application does this through the mmap() call, providing MAP_PHYS|MAP_ANON flags. Currently, if it is not possible to satisfy a MAP_PHYS allocation with contiguous memory we fail the mmap() call. Instead, with this feature, we will trigger a memory defragmentation algorithm that attempts to rearrange memory mappings across the system in order to allow the MAP_PHYS allocation to be satisfied.

During the memory defragmentation, the thread calling mmap() will be blocked. It is acceptable that the memory defragmentation take a significant amount of time but it must not cause significant delay in other system tasks or to significantly impact thread or interrupt latency. As other system tasks will be running simultaneously, the defragmentation algorithm must take into account that memory mappings can change while the algorithm is running.

Possible Future Development#

Optional Work#

As noted above, certain physical pages of memory cannot be moved as the virtual-to-physical address mapping must be locked. Also, in the initial implementation of this feature, pages that are not locked but that contain mutex objects will not be moved, as moving a page with a mutex object involves extra work in the kernel.

Once the initial feature implementation is complete, it would be relatively straight-forward to move pages that contain mutexes -- in addition to the rest of the work involved in moving the page, an extra call would need to be made into the kernel to remove the mutex objects on the page and add them back with the new physical addresses.

Time permitting, this will be done as part of this feature.

Further Possibilities#

We have discussed several alternatives and enhancements to the basic idea of defragmenting memory. In the end, we concluded that defragmenting memory on the mmap call is always necessary, and that while additional features might improve the system, they are not necessary.

Ideas discussed included modifying our memory allocation algorithm so that by preference we use memory fragments to support a memory allocation request. This would prevent memory becoming fragmented as quickly as it does. We would need to model the algorithm to get a better idea of how it might behave. However, it seems clear that this sort of an algorithm change would be favourable for certain systems (where defragment-on-mmap happens frequently) at the expense of making some applications run more slowly since they would not get as much benefit from large page sizes.

Another idea would be to defragment memory in the background, using the idle task. When the system is idle, we could run an algorithm that attempts to consolidate free memory into contiguous runs. We have talked about using the idle task for a number of different tasks in the past. In most cases its simply a lack of resources that has prevented us from implementing the ideas. The notion of defragmenting memory while at idle is likely to follow the same path -- the importance and benefit of idle-time defragmenting is arguable, and we have many higher priorities.

Feature Design#

It is not always safe to change memory mappings. It can be safe because of the nature of virtual memory systems -- the application usually does not know or care what physical memory blocks back the virtual memory addresses it is given when it allocates memory. However, for some applications, the physical address of its underlying memory *is* important -- case in point are the very DMA-based device drivers that this feature is intended to assist: their DMA frames cannot be moved in the physical address space without breaking the application. Further, memory allocated for the kernel (and referenced through the platform's 1-to-1 window) cannot be moved as the memory's virtual address is tightly tied (through the 1-to-1 mapping) to the memory's physical address.

Further, it is not always desirable to change memory mappings. Neutrino provides "large page" support on most hardware platforms. If an application has a large contiguous memory allocation the MMU can be programmed to use a large page representation. This is more efficient than multiple small pages. So, even if an application doesn't require physically contiguous memory, having physically contiguous memory can improve efficiency. If our memory defragmentation algorithm breaks up a large page, the application performance will suffer.

Thus our defragmentation algorithm must take into account that not all memory mappings can be changed, and that even where its possible to change a mapping, it may not be desirable.

In the initial release of this feature, we will only implement the ability to compact free memory. Further, we will do the minimum amount of work necessary to satisfy the current allocation request. In the future we intend to implement extensions that will allow a more thorough defragmentation to run in the background (or on user request).

In order to satisfy a memory allocation, we may need a block of contiguous memory. If no such block exists, we need to create one. Our algorithm consists of the following steps:

1. examine memory and characterize its usage
2. identify a candidate run of quantums that we can use to satisfy the allocation request
3. flush all in-use blocks out of the candidate range

We must be sensitive to the fact that memory usage might change as we're running, so when we get to flushing out our candidate area we might discover that we can no longer move a quantum as required to use this range. In that case it is necessary to go back and try again.

This algorithm will be called the "minimal_compacting" algorithm.

Characterize Memory Usage#

In order to characterize memory usage, we will take a two-pronged approach.

1. examine the physical allocator quantum data structures to determine what free quantums are available, and what quantums are used by the system
2. examine all memory objects in the system to determine how non-system memory is used.

To this end we will introduce new "walk" functions for walking physical memory block heads and memory objects.

The minimal_compacting algorithm will walk physical memory quantums looking at each quantum's flags to determine which quantums are free and which are used by the system. It will then walk all memory objects to determine how they make use of physical memory. This information will be used to build up a list of memory ranges with their associated usages.

Memory List Data Structure#

To track memory usage, we will introduce a new data structure, named a 'memlist'. A 'memlist' will consist of a linked list of 'memblock's. Each 'memblock' will represent a range of physical memory addresses that share common usage.

A memblock's usage indicates whether a block is:

- free
- inuse by the system
- inuse by a memory object, with optional additional characteristics:
  - locked
  - has a sync object on it

Operations on a memlist are:

- init: initializing a memlist to an empty state
- add_block: define the usage of a range of memory - this might involve:
  - adding a new memblock to the memlist in the appropriate location
  - modifying an existing memblock to extend its range
  - splitting a memblock into two or three memblocks
- find_run_set: finding a run of memblocks in a memlist that meet certain criteria
- destroy: destroying a memlist and freeing all associated memory

Identifying a Candidate Range#

In order to identify a candidate range to which we can compact free memory blocks, we will implement a memlist find_run_set function that allows us to find a run of memblocks that best meet a certain criteria. This function will

1. accept 'qualifier' and 'comparator' functions as parameters,

2. scan the memlist looking for runs of memblocks that satisfy the given qualifier function
3. sort the resulting acceptable runs using the given comparator function

For our purposes, the qualifier function is simple: if the block is free or can be moved (that is, isn't owned by the system, isn't locked, and doesn't have a sync object on it), it qualifies for a candidate run.

The comparator function is more difficult. The find_run_set function with the comparator will present us with two possible candidate runs (which might be too small, just big enough, or far bigger than we need) and we need to compare the two and judge which is better.

For our purposes, we have a minimum requirement on the size of the run. If one run doesn't meet this requirement while the other does, the larger run is superior. If both runs are of sufficient size, selecting one or the other as superior is much more difficult. We need to consider how much of each run is within the kernel's 1-to-1 area (since we aren't moving system objects, we don't want to use up the 1-to-1 area if we can avoid it), how much of each run is free (and thus will require fewer quantum moves to compact), whether clearing the run would break up objects that can currently use large page sizes, etc.

Thus the comparator is something that can be endlessly tuned. For our first pass, we will use the following algorithm:

1. are the runs both big enough? If not, which ever is bigger is superior.
2. if both runs are big enough, which has the less space inside the 1-to-1 area? The run with the least overlap with the 1-to-1 area is superior.
3. if both runs are big enough and both share the same overlap with the 1-to-1 area, then the smaller run is superior (we don't want to expend a larger run than we need to).

Once we have run find_run_set, we have found a range of memblocks that satisfy our requirements. This range may be much larger than we require, however. We trim it to the minimum required size to define our candidate range.

Compacting Free Memory#

Once we have chosen a candidate range, we need to compact free memory to that range. This stage consists of examining all memory objects in the system to determine if they make any use of the range. If they do, those quantums that fall within the range must be swapped for free quantums outside the range.

While we are running the compacting step, we want to ensure that memory allocations performed by other system tasks do not interfere. We want to prevent, if possible, the pa_alloc() routine from allocating memory in our candidate range. To this end, we will modify the kerext_pa_alloc() routine so that it preferentially avoids the candidate range if such exists.

We will introduce functions to the physical allocator subsystem that allows the defragmentation code to register or clear candidate ranges. The kerext_pa_alloc() routine will be modified so that where it currently loops over a list of 'restrictions', it now loops over restrictions modified by the candidate range first and then loops over the restrictions again (this time unmodified) if necessary.

With the behaviour of pa_alloc() modified in this fashion, compacting free memory consists of:

- walk the set of memory objects. For each object:
    - walk the object's memory. For each memory run:
        - if the run overlaps the candidate region
            - pa_alloc() a new set of quantums
            - swap the old quantums for the new
            - free the old quantums

Of these steps, only swapping the old quantums for the new requires new functionality.

Swapping Pages#

int defrag_swap(OBJECT *obp, struct pa_quantum* curquant, struct pa_quantum* newquant);

We assume that the input to the algorithm is an in-use memory quantum and the object that uses the memory, and a free memory quantum. The goal is to swap the inuse quantum for the free quantum and update the object so that it refers to the new. Note that we are also assuming that the given inuse quantum is safe to be swapped.

The basic algorithm for swapping an in-use page for a free page is straight forward, excepting for the issue of mutual exclusion with other system tasks that are executing in parallel, possibly on a separate CPU.

We must ensure that our design is compatible with the following considerations:

- other threads or processes reading or writing memory while we are swapping pages
- an ISR reading or writing memory while we are swapping pages
- a process unmapping memory while we are swapping pages
- a page of memory is referenced by multiple address spaces

Fortunately ISRs are not an issue, because all processes with ISRs are super-locked (this happens when they are granted IO-privity -- kerop_thread_ctrl() invokes memmgr.mlock() which locks all current mappings and sets aspace flag MM_ASFLAG_LOCKALL, which will cause subsequent mappings to be locked as they are established). Thus a page that is used by a process that has an ISR will never be a candidate for swapping.

The object lock is key. It is used to prevent other threads from changing the object or their relationship with it during the swap process. However, once the object lock is obtained we cannot then lock an address space. If we need both an address space and an object locked, we must lock the address space first (this is the order used elsewhere in the memory manager code, and reversing the order here could result in a deadlock). It is strongly undesirable to lock address spaces for the page swap algorithm, as we would need to lock them all before we locked the object, and this could be quite intrusive to system operation. (The alternative, of unlocking the object between address space manipulations is fraught with race conditions and other difficulties as unlocking the object allows other processes to modify it while we're in the middle of our algorithm). Thus we require an algorithm that does not require locking address spaces. Fortunately the object lock is sufficient for our purpose here.

The algorithm for the swap is as follows:

1. lock the object
2. run through all aspaces that reference the object and remove any write permissions from their page tables (but not from their mm_maps) for the quantum
3. copy the old page to the new page
4. change the object to reference the new page
5. for each process that references the object, change the page tables and flush the TLBs (if the object has been mapped)
6. unlock the object

If any processes attempt to write to the old page after we've removed their write permissions, they will get a page fault and the fault handler will attempt to lock the object. Since we have the object locked, they will be blocked. Thus we are guaranteed that nobody will be able to modify the page once we start copying it. However, while we are copying the page, any process that gets a page fault associated with the object will block (the fault handler will attempt to lock the object as part of updating the page table permissions). Reading from the page will be unaffected if the page was already mapped in (the read might access either the old or the new page depending on whether it happens before or after we update the process's page tables).

Thus a process that attempts to write to the page as we're swapping it out, or that attempts any access to an unmapped area of the object) will be blocked, but processes can read from the old page unhindered if the page is already mapped.

Similarly, any attempt to unmap the memory referenced by the object will cause vmm_munmap() (through ms_unmap()) to lock the object, so the munmap can't proceed until the page swap is complete.

Removing Write Permissions#

When we initially set up a process's memory mapping to a memory object, we often initially set the page table mapping to be read only, even if the process's access to the object is read/write. In this manner, when the process attempts to write to the memory object, it causes a page fault. During the processing of the page fault to enable write access in the process's page tables we can track the fact that the memory object has been changed.

The purpose of removing the write permissions that a process might have in the page tables that refer to a page we're about to swap out is to ensure that we get a page fault if the process attempts to write to the page. Since the fault handler has to lock the object, we force the write attempt to be blocked while we are copying the underlying memory page.

Note that we are only changing the process's page tables -- we are not changing its mm_maps. Thus when we are done the process will automatically get its page table write permissions back the next time it attempts to write to the page.

To update the page tables we will walk all object references (using the memref_walk() function) and for each reference to the object that actually references the quantum and is mapped in through the page tables, we will invoke pte_map() to remove write permissions.

We can test to see if a quantum is used in an object reference using a new function map_quantum_to_vaddr(), defined below. We can test to see if a page is already mapped in a process's page tables and if it already has write permissions using the cpu_vmm_vaddrinfo() routine.

In the subsequent step of copying the quantum we need to know if the page has been referenced, and the virtual address and mm_map structure used in the reference. If we find a reference to the page during the removal of write permissions we can store this information so that the copy operation doesn't need to search it out again.

```
//
// remove_write_permissions pseudo-code
//
// This is a worker function for memref_walk invoked on an object.  For each reference, remove
// any associated write permissions to the given quantum.  If the quantum is referenced, set
// aside some information about if for later use.
//

static int
remove_write_permissions(struct mm_object_ref *or, struct mm_map *mm, void *d)
{
  use map_quantum_to_vaddr (below) to determine if the quantum is used in
                        this object reference/mm_map.

  if the quantum isn't referenced in this mm_map
    return EOK so the walk continues

  if we haven't found a reference to the object before this
    note the vaddr, mm, and object offset of the quantum for this reference

  use cpu_vmm_vaddrinfo to determine the protections associated with
                      the vaddr for this reference
```

```
  if there were write permissions
    use pte_map to remove the write permissions

  return EOK so the walk continues
}


//
// map_quantum_to_vaddr() pseudo-code
//
// Determine if quantum <pquant> is referenced in <mm> and, if so, determine <pquant>'s
// offset in the object and return <pquant>'s virtual address.
//

static uintptr_t
map_quantum_to_vaddr( struct mm_map* mm, struct pa_quantum* pquant, off64_t* poff)
{

  run memobj_pmem_walk_mm using mm and qtovfunc (below) to find out if the quantum
                    is referenced in the given mm

  if a reference wasn't found
    return 0 so the pmem_walk continues

  if poff was given
    set *poff = offset of the mm_map in the object(mm->offset) +
                offset of the quantum in the mm_map (found by qtovfunc)

  return vaddr of pquant = start vaddr of mm (mm->start) +
                offset of the quantum in the mm_map (found by qtovfunc)
}


static int
qtovfunc(OBJECT * obp, off64_t off, struct pa_quantum *pq, unsigned num, void *data)
{
  if we find data->pquant in pq[0] through pq[num-1]
    calculate offset of data->pquant in mm_map (= off + offset of data->pquant in pq[])
    indicate that a reference was found
    return -1 to terminate pmem_walk
  else
    return EOK to allow pmem_walk to continue
}
```

Note that invoking pte_map() will cause the flow of execution to enter the kernel. Thus we are entering and exiting the kernel once for each reference of each process to the given quantum.

Copying the Page#

Copying the page would be straight forward, except that it is necessary to set up virtual address mappings to the pages before the copy can take place.

Ideally we could use pte_temp_map() to set up virtual addresses for both pages, however pte_temp_map() can only set up one mapping at a time -- because it doesn't reserve the assigned address space range when it sets up a mapping, the attempt to set up a second mapping can result in reusing the same virtual address.

The alternative is to use memmgr.mmap() to set up the virtual address mapping. We have the object and the offset of the page within it, so we could set up a shared mapping to the object at the given offset. However,

testing this shows than when we remove the mapping (using memmgr.munmap), the munmap code is not sensitive to the fact that an anonymous object might have multiple mappings to it, and frees up the quantum when the reference in unmapped.

We also have the physical address of the page, so we can set up a MAP_PHYS mapping to the page. This solution works, but we need to be careful of the colour of the virtual address on systems that use cache colours. For this we take advantage of the fact that if we specify a suggested virtual address to memmgr.mmap but don't specify the MAP_FIXED flag, memmgr.mmap may or may not use our suggested virtual address but it will preserve the address's colour. This is not a documented behaviour, however, so we must be careful that this behaviour does not change in the future. Comments will be added to vmm_mmap.c to try to ensure that this functionality remains.

While we could use memmgr.mmap(MAP_PHYS) for both pages, pte_temp_map() is more efficient since it doesn't require reserving the mapping in the address space. Thus we will use pte_temp_map() for one mapping, and memmgr.mmap(MAP_PHYS) for the other.

```
static int
pa_copy_quantum(PROCESS* prp, defrag_data_t* data)
{
  determine a value to use as a vaddr hint to memmgr.mmap:
    if we found a reference to the old quantum while removing write permissions
      vaddr_hint = the vaddr of the reference we found
    else
      vaddr_hint = memobj_colour(object, offset of quantum in object)

  determine correct protection bits to use for mmap call:
    prot = PROT_READ
    if we found a reference to the old quantum while removing write permissions
    and that reference specified PROT_NOCACHE
      prot |= PROT_NOCACHE

  src_vaddr = memmgr.mmap(), using vaddr_hint, prot, MAP_PHYS, paddr of old quantum


  use pte_temp_map(vaddr_hint, user mm_map found while removing write permissions) to set up a
  mapping to the new quantum and invoke newpage_mapped() (below)

  memmgr.munmap(src_vaddr)
}

int
newpage_mapped(void* vaddr, size_t len, void* data)
{
  CRASHCHECK(len != PAGESIZE);
  memcpy(vaddr, data.src, PAGESIZE);
  if user mapping of PROT_NOCACHE exists, flush data cache
  if user mapping of PROT_EXEC exists, invalidate instruction cache
}
```

Note that pte_temp_map() accepts as a parameter a pointer to an mm_map structure that shows how the quantum was used. It uses this parameter to determine the type of the object and the flags used in mapping (in particular, whether or not PROT_NOCACHE) was specified. We will modify pte_temp_map so that it can accept a null value for this parameter and use reasonable defaults (which include not setting PROT_NOCACHE). In our newpage_mapped function, we will take into account how the page was used and flush the caches if necessary.

Alternatives that could be considered include:

- modifying pte_temp_map so that it can map multiple pages -- there doesn't seem to be any benefit to this relative to using memmgr.mmap()
- modifying vmm_munmap() to support multiple mappings to a virtual object so we can do a MAP_SHARED map to the object at the appropriate offset - this might be necessary in the future in any case (to support proc/<pid>/as and MM_ANMEM_MULTI_REFS flag) but for now it's simpler to use MAP_PHYS.

Changing the Object Reference#

The quantums associated with an object are maintained in a singly linked list from the object's 'pmem' field. To change the object from using the new quantum instead of the old, we need to change the list linkage. Since the object is locked, this is safe.

We can effect the reference change using the existing functions memobj_pmem_del() and memobj_pmem_add_pq().

Note, however, that memobj_pmem_del frees the deleted quantums. We need to hold on to the old quantums until we have updated any page tables that referenced the old quantums. We will make a minor modification to memobj_pmem_del so that it takes an additional parameter that indicates whether or not it should free the quantums when it has deleted them from the object.

Update Page Tables#

Updating a process's page tables is done in a manner similar to that used for removing the write permissions from an object's page references.

We will use memref_walk() to walk all references to the object, map_quantum_to_vaddr() to verify that a given reference uses the quantum, cpu_vmm_vaddrinfo() to verify that the quantum is mapped, and pte_map() to change the mapping. In this case we will pass in the protection flags unchanged, but specify the paddr of the new quantum.

Note that invoking pte_map() will cause the flow of execution to enter the kernel. Thus we are entering and exiting the kernel once for each reference of each process to the given quantum.

Performance Analysis#

To perform the page swap, this algorithm enters the kernel up to twice for each reference to the object that owns the inuse quantum (once for each pte_map call in removing write permissions and updating the page tables). Further, walking memory objects requires entering the kernel to ensure mutual exclusion with the kernel modifying those memory objects. Entering the kernel as many times as this is a rather high cost, but is unavoidable without moving the whole operation into the kernel.

Design and Code Reviews#

The design was presented to the entire kernel team on Thursday, January 29th.

An initial code review was held on Wednesday, February 11th. In attendance: Brian Stecher, Colin Burgess, Mike Kisel, Doug Bailey.

actions from code review:

- make sure changes to common.mk files are checked in under appropriate PRs (i.e. they are not part of this feature)

- change to shutdown_nto.c was deferred (not part of this feature)
- change vmm_configure.c so that defragmentation is disabled by default at this point
- we will reverse the logic of the PAA_FLAG_NO_DEFRAG flag. That is, we will replace the flag with PAA_FLAG_ALLOC_DEFRAG and modify all locations where it is used to reverse their logic. The reasoning behind this is to ensure that we only allow defragmentation where it is explicitly enabled.
    - Note that even single page allocations can benefit from the defragmentation algorithm, since if they fail because they can't meet their restrict list, defragmentation can free up a page within the restrict list by pushing it out. This can be important for kernel memory allocations.
- in memmgr_map.c, leave the original structure of the for(;;) loop (instead of the do/while I replaced it with)
- in memmgr_map.c, remove the "#ifndef NDEBUG kprintf" or replace it with a kerext_slogf
- in memmgr_map.c, modify the loop so that it will iterate a maximum of 8 times (8 is an arbitrary choice and should be commented as such)
- in memmgr_map.c, add a comment indicating that if the allocation fails with ENOMEM because it failed to get necessary kernel resources, but there is otherwise sufficient free memory for the allocation, we will keep looping on invoking the compaction routine and eventually get kicked out by our maximum number of loops. This is expected (but highly unlikely) behaviour.
- in pa.c, I don't need the spinlock on the avoidance area -- careful ordering, atomic_order and a write barrier are sufficient. Add comment explaining why this is sufficient.
- in pa_blkhead_walk, change FIXME to a reference to the existing PR that already describes this problem in other places
- move pa_process_is_paddr64_safe to another location (maybe mm_anmem.c?)
- in memobj_pmem_del(), move pmem_stats_hook and MEMCLASS_PID_FREE_INKER calls so they are only done when free_it is set. Make sure there is no memory accounting done in pa_free or pa_alloc.
- for anmem_memobj_walk use QueryObject/proc_lock_pid/clone object/proc_unlock_pid to ensure mutex safety
- in memmgr_defrag.c, use mmap instead of _scalloc to allocate data structure memory. This will allow the memory to be returned to the pa system instead of to the kernel heap.
- in memmgr_defrag.c, check return codes from memlist_add_block and if we run out of system memory, look into trying to proceed with the defrag operation anyways.
- in fdmem_memobj_walk(), move mutex lock inside loop so that it isn't held for whole walk, use object clone to prevent destruction of object while we don't hold the fdmem mutex.

further discussion:

- in order to fail a memoffset call if the memory isn't locked, within the memmgr handler, once I have the paddr (however it is obtained) I can look at the quantums directly to see if they are locked.
- when contiguous memory is allocated, we need to make sure it is locked so that it can't be moved as part of defragmentation. We can't rely on the process to lock it after allocation, as it could end up being moved before they can lock it. We can mlock_all before the allocation, but then any other threads that happen to be allocating memory will also end up with locked memory. We could pass a flag down to the memory allocator, but that will use up a flag that we're already short on. This item needs more thought.

followup:

- code updated as required and checked in: http://community.qnx.com/integration/viewcvs/viewcvs.cgi?root=coreos_pub&rev=213323&system=exsy1001&view=rev
- further code changes and inspection: http://community.qnx.com/sf/discussion/do/listPosts/projects.core_os/discussion.osrev.topc6239
- final check-in: http://community.qnx.com/integration/viewcvs/viewcvs.cgi?root=coreos_pub&rev=213706&system=exsy1001&view=rev
- reviewed full list of actions and confirmed that all items have been addressed in the final code with the exception of deferred items (identified as such in the action list)