# Various Requirements

This document collects my otherwise stray and disorganized thoughts on the requirements of various features and subsystems of the Precursor microkernel.

(Editorial note from 2021: Apologies that this is a Word doc; Markdown wasn’t a thing yet when I first wrote this.)

## Panic Requirements

There are times when we will need to shut the kernel down NOW. Conflicting with this is the need to let the user know somehow, which means we can only shut down the kernel at NOW + n, where n is the time it takes to let the user know, and also the trouble we may cause along the way.

Here are the requirements for panic:

* Must be callable from everywhere in the kernel, if possible.
* Must print out a friendly message incorporating some kind of debugging information.
* To the highest degree possible, whatever information is displayed on the screen during a panic must be useful for the kernel developer in terms of being able to locate and fix the bug that caused the panic.
* Must optionally print out the contents of the current trap frame. Optional because the caller (e.g. – Executive’s exception handlers) will decide whether such information is appropriate.
* In the future, may print out a partial dump of the current trap frame’s kernel stack (in the case of a kernel re-entry).
* Must reset the kernel display before outputting anything.
* MUST NOT CAUSE PANICS (either directly, via badly designed circular control flow, or indirectly, via exceptions that call panic). In other words, panic must not call panic, and panic must NEVER cause exceptions. This suggests some global state to indicate whether the system is panicking.
* On an SMP system, as soon as one processor invokes panic, the system is in panic mode. All other processors must be halted immediately.
* Must either halt the system or reboot it after some configured delay.
* Reboot and halt functionality must be callable outside the context of a panic (i.e. – normal shutdown situation). In this case, different “non-panicky” messages will be output.

Here are the facilities required by panic in order to implement the above requirements:

* Formatted output (KOut, TextWriter, & DisplayTextStream from KRunTime; KernelDisplay from the HAL).
* Output of TrapFrame contents (must go in architecture-specific part of KRunTime or Executive; will use KOut in KRunTime for formatted output). Same goes for kernel stack dump.
* KernelDisplay for resetting the display.
* Some facility is required to “remember” that the system is panicking in case panic causes an exception and re-enters the kernel, or in the unlikely case on an SMP system that one of the other processors calls panic before being notified that it must halt.
* Some facilities are needed in the HAL to halt and reboot the machine.
* A facility is needed for accurate timing based on busy-waiting (for reboot delays). This should probably be in the HAL.

Here are some constraints imposed by the current kernel design:

* Panic is not callable from the HAL, since it depends on KRunTime functionality.

Panic may not be entirely callable from KRunTime, since parts of it may be implemented in higher layers (i.e. – Executive). Perhaps panic can be split so that a subset of its functionality can be called from KRunTime.

* Panic cannot be called if the system is already panicking. This condition must be enforced globally across all processors in an SMP system. If a panic detects that the system is already panicking, it must immediately hard-reset the system.

### Whether KRunTime Needs Panic (Or, Where Should Panic Go?)

There is an important tradeoff to consider when deciding where Panic should live (Executive or KRunTime). First, the hard requirements must be dealt with. Parts of panic that *must* be in Executive include:

* Deciding whether to print TrapFrame & kernel stack (the actual mechanism *might* be ok to put in KRunTime – the only objection is on the basis of consistency, since KRunTime isn’t supposed to deal in machine-specific stuff).
* Deciding whether to panic or do a normal shutdown (KRunTime doesn’t have sufficient information to know when to do what).

There isn’t anything else that Executive absolutely *must* handle (that I can think of). Whether to put the bulk of functionality in Executive or KRunTime comes down to a judgement call about the responsibilities and scope of these two subsystems.

On one hand, it seems likely that putting most of the functionality in KRunTime is more beneficial since it will allow KRunTime itself to use the majority of the functionality (see first requirement above – panic must be callable from as much of the kernel as possible). On the other hand, if KRunTime calls panic directly, then the machine state dumping (if required) must be in KRunTime as well, which is somewhat of a breach of the design.

Whether machine state dumping is required in KRunTime or not depends on how much error handling in KRunTime can be done with assertions (which include file & line info). The problem of where to dump machine state cannot be mitigated by finding a creative way to do it in the HAL without having the HAL call back to KRunTime, because this is impractical (the HAL needs itoa()-like functionality at the very least, which would be silly to duplicate in both KRunTime and the HAL).

## Re-entrancy Protection

Don’t get too paranoid about unexpected kernel re-entry. It can only happen in the following circumstances:

* Synchronous – there’s something wrong with the kernel code anyway. Find it and fix it.
* Asynchronous – NMI. Chances are it won’t continue to interrupt itself indefinitely.

The goal is to prevent the low-level interrupt/exception dispatching code in Processor to infinitely reenter itself. This is not a problem for NMIs, as mentioned above. In general it won’t be a big problem with exceptions either, since they should be rare. However, there is an obvious weak point in the form of KDebug\_assert(). Again, failures should be rare, but it feels better to know that failed assertions will still fail gracefully, even if they’re triggered within the Processor class itself. To that end, it makes sense to re-direct Processor\_triggerDebugTrap() during critical times.

After examining the Processor dispatch code carefully, it has been deemed simple enough to be verifiably exception-free. It isn’t worth the performance cost to fiddle with triggerDebugTrap() out of simple paranoia. A maintenance note has been put in the code warning not to use assertions on that part of the code path.

## Error Handling Requirements

<FIXME>

## Memory Management (Take 1)

The current idea for Precursor’s memory manager is to implement L4’s hierarchical address spaces. This requires the following functionality in the kernel:

* A way at boot-time to divide physical memory so that a certain “tithe” goes to the kernel, and the rest goes to our version of “sigma 0” (must find a better name for this…). I’m thinking 10% of physical RAM minus the fixed-cost kernel code, data + bss at this point, but it’s mostly a shot in the dark. The only way to do better is to try, measure, and optimize. Any system to dynamically reclaim memory for the kernel requires either exokernel-style visible revocation, or other really complex research-y things I’m not prepared to do in this, my very first OS.
* A physical memory manager that can dole out frames to different parts of the kernel dynamically. The parts I currently foresee are the heap, thread stacks, user-mode read-only information pages, and perhaps lookaside-lists. This should do proper page-colouring if possible (as an interesting exercise if nothing else). At the very least its interface should be designed with page-colouring in mind.
* An initial physical memory manager that provides memory to the creation of “sigma 0”. Since the memory used to create the initial address space must all be physically contiguous, a bitmap makes the most sense here. Having a distinct manager here makes sense in the context of portability, since it separates the architecture-specific memory-map interpretation from the architecture-independent creation of the initial address space. Also, there is no overhead once “sigma 0” starts running, since the entire bitmap can be thrown away. This is a tradeoff however, because it would probably be simpler to create “sigma 0” directly from the architecture-specific memory map.
* A virtual memory manager that manages page directories, page tables, and the mapping database. Page directories will be created per-address-space. Page tables will be allocated on demand rather than pre-allocated. I have not thought about how to reclaim unused page tables, or whether this is even worth doing. I have also not thought yet about what the best way is to map these structures into kernel space to operate on them. I have also not studied the L4 mapping database in detail. For now I can imagine designing a similar one in an ad-hoc manner and creating it with memory from the kernel heap.
* Something to manage memory-mapped devices. Not sure how to detect them, or whether this is even possible. The GRUB memory map does not include them in a form that is consumable according to the Multiboot spec (some of the reserved regions look suspiciously like VGA and PCI areas, but any “type” besides “available RAM” is undefined by the Multiboot spec). Perhaps I will need to consult Ralph Brown’s Interrupt List, or (ick) call the BIOS somehow. This kinda screams for a custom bootloader.

First a note about limitations. I don’t think there is any way for an L4-style scheme to handle PAE, or in general, any case where the physical address space is larger than the virtual one. There are two reasons for this. The first is that sigma0 is supposed to have an identity-mapped address space, which makes it impossible to get at the extra 60 GB in the PAE case. The second is that even if you dispensed with identity-mapping, you’d still need a scheme that simply doesn’t fit the recursive address space model (or at best, you’d need to partition the system into 16 completely separate memory regions with 16 different pagers). Oh well. 64-bit will reverse this situation, so I don’t think it’s worth worrying about. (Actually, 64-bit may not reverse the situation – since fewer than 64 of the address bits are actually used for the virtual address space, it is still possible for the physical address space to be larger. However, it will probably be a while before this becomes a problem).

In L4, sigma0 is supposed to have the physical address space completely identity-mapped. It then uses a special protocol to give the remaining memory to the kernel. This is a little strange IMO, so I will go for the opposite approach – the kernel will grant everything but the “tithe” to sigma0, so sigma0 will be identity-mapped with a few holes. An interesting question is where should these holes be? The kernel itself really needs to live above 1MB so that its load address is predictable. If we continue to use the 4MB kernel page, this will mean a 4MB hole at the beginning of sigma0, which seems like a bit much (and precludes running on systems with 4MB RAM or less). It seems advantageous to put the rest of the kernel’s private memory at the end of the available physical RAM.

There is actually another bigger problem with holes in sigma0 – what about the kernel’s virtual space? How much does it need? This must be determined at design time if the kernel is to go in the higher-half, but that would preclude the possibility of a dynamic “tithe” calculated at boot-time. Maybe L4 is in the lower-half, thus avoiding the problem (except that it creates a problem for apps, which wouldn’t know what the lowest possible base address could be). Maybe the L4 kernel is relocatable. There is evidence to suggest that L4 itself is just a module loaded by GRUB, and sigma0 itself is actually “booted”.

The way to fix the virtual hole problem is as follows:

1. Make the higher half as high as possible (3.5 GB in a 32-bit address space for example).
2. At boot time, calculate the “tithe” (physical memory minus I/O regions minus fixed kernel code, data + bss, minus user-usable regions). If the amount of physical memory minus the tithe (but including I/O holes, the kernel, etc.) goes past the higher half, then scale it back to just below the higher half (i.e. – increase the tithe). If the tithe would have been 10% anyway, then 0.5 GB is 12.5%, which is not too bad. 64-bit should become more popular soon anyway, and this problem will go away for a while.
3. Profit!

The above solution unfortunately does not address the PCI configuration space, which, if identity-mapped, overlaps with the higher half. I suspect that L4 handles this by being in the lower half. It’s beginning to seem silly to treat physical memory as the same resource as virtual memory (Singularity treats them differently).

A philosophical note to guide these thoughts – without paging, there can be no invisible revocation. Without an exokernel architecture, there can be no visible revocation. If you have neither (like Singularity), then I don’t see how you can prevent apps from DOSing the kernel’s memory manager, short of terminating aberrant processes (which is a bit heavy-handed IMO). I think Singularity gets away with it by providing stock GCs that each app uses. Since the GCs are trusted code, the kernel trusts them to call “free” whenever they can.

I am not sure if a 4MB page for the kernel has any negative consequences for memory-mapped I/O to the first meg. There is a section in the Intel manual about memory aliasing that seems vague and scary (see the related thread on Mega-Tokyo). If I can verify that this evilness doesn’t affect me, then that’s one less reason to abandon the 4MB page (it still leaves the reason given above about having a hole in sigma0 though).

### Physical Memory Allocator Algorithms

Here are some design concerns for physical memory allocators:

1. Does it support multiple page sizes?
2. Does it support tracking non-RAM address regions (e.g. – memory mapped devices)?
3. How fast can it allocate an arbitrary frame?
4. How fast can it allocate a specific frame?
5. How fast can it allocate a frame of a specific colour?
6. How fast can it free a frame?
7. How fast can it tell you whether a particular frame is free or not?
8. Is it practical to protect itself against a frame being allocated twice?
9. Is it practical to protect itself against a frame being freed twice?
10. How much space overhead does it introduce?
11. Does it require interaction with the virtual memory manager?
12. Is it practical to protect itself against a “bad” frame being allocated or freed?
13. Is it practical to have a lock-free implementation?

For Precursor’s kernel PMM, concerns 1, 2, and 4 do not matter. The usage characteristics will be as follows:

* The kernel PMM will be used to manage the “tithe” only. This means that for a 32-bit physical address space, it will be managing at most 512 MB. This limits the amount of space overhead that we need to worry about for the PMM.
* The kernel PMM will allocate/free a few frames at a time when thread stacks are created/destroyed.
* The kernel PMM will allocate a few frames as needed when the kernel heap is expanded. If any freeing happens for the kernel heap, it will probably be in bulk and fairly infrequent (GC-like behaviour).
* The kernel PMM will allocate a handful of frames at a time when page tables are created. This will happen on demand as address spaces are created and expanded. They will probably be freed in bulk and relatively infrequently (GC-like behaviour) when address spaces are destroyed. This applies both to user address spaces and to the kernel’s own page tables.

Some algorithms:

#### Bitmap

|  |  |
| --- | --- |
| Concern | Fulfillment |
| Does it support multiple page sizes? | Yes |
| Does it support tracking non-RAM address regions (e.g. – memory mapped devices)? | Yes, if you use more than one bit per frame. |
| How fast can it allocate an arbitrary frame? | O(n), or O(1) in the best case |
| How fast can it allocate a specific frame? | O(1) |
| How fast can it allocate a frame of a specific colour? | O(n) |
| How fast can it free a frame? | O(1) |
| How fast can it tell you whether a particular frame is free or not? | O(1) |
| Is it practical to protect itself against a frame being allocated twice? | Yes |
| Is it practical to protect itself against a frame being freed twice? | Yes |
| How much space overhead does it introduce? | O(n): PagesOfRam/8\*BitsPerFrame (e.g. – for 128 KPages and 2 bits per frame, the bitmap would be 32KB) |
| Does it require interaction with the virtual memory manager? | Only upon initialization to map in the bitmap. |
| Is it practical to protect itself against a “bad” frame being allocated or freed? | Yes, if you use more than one bit per frame. |
| Is it practical to have a lock-free implementation? | Yes. |

#### Free Frame Stack Stored in Free Frames

|  |  |
| --- | --- |
| Concern | Fulfillment |
| Does it support multiple page sizes? | No |
| Does it support tracking non-RAM address regions (e.g. – memory mapped devices)? | No |
| How fast can it allocate an arbitrary frame? | O(1) |
| How fast can it allocate a specific frame? | O(n) |
| How fast can it allocate a frame of a specific colour? | O(1) |
| How fast can it free a frame? | O(1) |
| How fast can it tell you whether a particular frame is free or not? | O(n) |
| Is it practical to protect itself against a frame being allocated twice? | Yes, assuming the invariant that a free frame exists only once in the free frame stack. |
| Is it practical to protect itself against a frame being freed twice? | Yes, although it’s not perfect. You’d have to mark free pages with a special signature and test for it on free. The signature could be faked, but it makes accidental misuse basically impossible. |
| How much space overhead does it introduce? | O(1): PointerBytes\*NumOfLists (e.g. – for 32-bit pointers and 16 page colour lists, the overhead would be 64 bytes.) |
| Does it require interaction with the virtual memory manager? | Yes |
| Is it practical to protect itself against a “bad” frame being allocated or freed? | No. You’d have to search a “bad” list on every free, which would make free O(n). |
| Is it practical to have a lock-free implementation? | For freeing frames, yes. For allocating, it doesn’t seem practical since you’d need DCAS, and even if you had it, contention could get bad. |

#### Free Frame Stack Stored Separately

|  |  |
| --- | --- |
| Concern | Fulfillment |
| Does it support multiple page sizes? | No |
| Does it support tracking non-RAM address regions (e.g. – memory mapped devices)? | No |
| How fast can it allocate an arbitrary frame? | O(1) |
| How fast can it allocate a specific frame? | O(n) |
| How fast can it allocate a frame of a specific colour? | O(1) |
| How fast can it free a frame? | O(1) |
| How fast can it tell you whether a particular frame is free or not? | O(n) |
| Is it practical to protect itself against a frame being allocated twice? | Yes, assuming the invariant that a free frame exists only once in the free frame stack. |
| Is it practical to protect itself against a frame being freed twice? | Not unless you put a signature in the frame and check it, but that negates much of the benefit of storing the stack separately. |
| How much space overhead does it introduce? | O(n): PointerBytes\*PagesOfRam + PointerBytes (e.g. – for 128 KPages and 32-bit pointers, the overhead would be 512KB + 4 bytes.) Note that this is the worst-case. The more that’s allocated, the less stack space is needed in principle. |
| Does it require interaction with the virtual memory manager? | Yes, if you grow the stack dynamically. Otherwise, if you pre-allocate, only upon initialization to map in the stack(s). |
| Is it practical to protect itself against a “bad” frame being allocated or freed? | No. You’d have to search a “bad” list on every free, which would make free O(n). |
| Is it practical to have a lock-free implementation? | If you don’t pre-allocate, then almost certainly no. Even if you do pre-allocate, you’d need DCAS, which doesn’t exist on x86. |

#### Doubly-Linked Array List

|  |  |
| --- | --- |
| Concern | Fulfillment |
| Does it support multiple page sizes? | No |
| Does it support tracking non-RAM address regions (e.g. – memory mapped devices)? | No |
| How fast can it allocate an arbitrary frame? | O(1) |
| How fast can it allocate a specific frame? | O(1) |
| How fast can it allocate a frame of a specific colour? | O(1) |
| How fast can it free a frame? | O(1) |
| How fast can it tell you whether a particular frame is free or not? | O(1) |
| Is it practical to protect itself against a frame being allocated twice? | Yes |
| Is it practical to protect itself against a frame being freed twice? | Yes |
| How much space overhead does it introduce? | O(n): 2\*PointerBytes\*PagesOfRam + PointerBytes (e.g. – for 128 KPages and 32-bit pointers, the overhead would be 1MB + 4 bytes. |
| Does it require interaction with the virtual memory manager? | Only upon initialization to map in the array. |
| Is it practical to protect itself against a “bad” frame being allocated or freed? | Yes |
| Is it practical to have a lock-free implementation? | No |

#### Hybrid Bitmap and Separate Free Frame Stack

|  |  |
| --- | --- |
| Concern | Fulfillment |
| Does it support multiple page sizes? | No |
| Does it support tracking non-RAM address regions (e.g. – memory mapped devices)? | No |
| How fast can it allocate an arbitrary frame? | O(1) in the best case, O(n) in the worst case |
| How fast can it allocate a specific frame? | O(k), where k is the size of the stack |
| How fast can it allocate a frame of a specific colour? | O(1) |
| How fast can it free a frame? | O(1) |
| How fast can it tell you whether a particular frame is free or not? | O(1) |
| Is it practical to protect itself against a frame being allocated twice? | Yes |
| Is it practical to protect itself against a frame being freed twice? | Yes |
| How much space overhead does it introduce? | O(n): PagesOfRam/8\*BitsPerFrame + PointerBytes\*StackSize (e.g. – for 128 KPages, 2 bits per frame, 32-bit pointers, and 1024 stack entries, the total storage required would be 36KB) |
| Does it require interaction with the virtual memory manager? | Only upon initialization to map in the stack(s) and bitmap. |
| Is it practical to protect itself against a “bad” frame being allocated or freed? | Yes, if you use two bits per frame |
| Is it practical to have a lock-free implementation? | Only for the bitmap portion. |

#### Hybrid Scheme Design Decision Tree

Basic structure: There is a bitmap covering all managed physical memory (1 = free, 0 = allocated), plus a fixed-size stack of free frame numbers.

Question 1: What is the authoritative source on whether a particular frame is allocated or free?

Answer 1.1: Just the bitmap. In this case, if a frame is popped off the stack, its corresponding bit in the bitmap must be cleared. Likewise, if a frame is pushed on the stack, its corresponding bit in the bitmap must be set.

Answer 1.2: Just the stack. This is impossible, since the stack cannot track all of physical memory.

Answer 1.3: The stack for frames on the stack, and the bitmap for everything else. In this case, if a frame is popped off the stack, its corresponding bit in the bitmap must be cleared since it is no longer on the stack. When a frame is pushed on the stack, its bit need not be set, since the fact that it is free is already denoted by its presence on the stack.

Question 2: If a frame is freed and the stack is full, what happens?

Answer 2.1: The frame’s bit in the bitmap is set and the stack is unchanged. This fits with both answers 1.1 and 1.3 above.

Answer 2.2: The frame replaces a frame already on the stack. There is actually no point in doing this, since there is no need to track the most-recently freed frames. All we need is a cache of frames to avoid scanning the bitmap.

It seems that it would be easiest to create a lock-free implementation of scheme 1.3 above. In this scheme, the bitmap and the stack can be treated as completely independent PMMs. The dynamics work like this:

* The initial state is all bits 1 and an empty stack.
* On allocate(), the stack is checked first. If it is empty, the bitmap is scanned and the bit for the chosen frame is cleared. Otherwise, a frame is popped off the stack.
* On free(), the stack is checked first. If it is full, the frame’s bit in the bitmap is set. Otherwise, the frame is pushed on the stack.

With a “last allocated” pointer, a bitmap can be made to work almost as fast as the hybrid approach, so the hybrid is probably not worth it.

### Multiboot Info Translator Design Decision Tree

Problem: GRUB provides a data structure needed by the kernel. Its base address is known, but the extents of its sub-structures are not, making it non-trivial to map it into the initial virtual address space.

End Goal: Have the Multiboot info, or some required sub-set of it, mapped into the virtual address space before calling main().

1. Q: Will the translator be an intermediate boot loader between GRUB and the kernel?
   1. A: Yes. This translator will be linked to physical 1MB and will run with paging disabled. It will gather any information or do any translation required of the Multiboot info, then enable paging before calling the kernel. It will ensure that the Multiboot info or its extracted by-products will be accessible in virtual memory when the kernel starts.
   2. A: No. The kernel will do the translation. Q: Will we just panic if the Multiboot info goes beyond the first 4MB of the physical address space?
      1. A: Yes. This is very easy to implement, either before or after paging is enabled. However, it is quite limited.
      2. A: No. Let’s try to actually translate something. Q: Will the kernel translate anything before enabling paging?
         1. A: Yes. The code to implement the translation must be position-independent, which probably means it must be hand-written in assembler. The purpose of this code is to make as much information as possible available in the virtual address space when paging is enabled. Q: Will info be extracted from the Multiboot structure or otherwise copied around in memory?
            1. A: Yes. Q: Will the extracted info go into a pre-allocated area of the kernel’s data section?

A: Yes. Q: Will we just try to copy all the Multiboot structures as-is?

A: Yes. There will be a hard limit on the size of the Multiboot structure, but the algorithm to copy the info will be quite simple. It is difficult to predict what the limit should be, making this approach problematic.

A: No. We can just copy the main structure, and record the highest end-address of all the modules, the kernel image itself, and the Multiboot structures themselves. This, along with the mem\_upper field of the Multiboot structure, gives us a certain amount of physical memory to play with for the rest of the translation process. In order for this approach to make sense, the rest of the translation process must happen after paging is enabled. Otherwise, it is pointless to have copied the Multiboot structure in the first place.

A: No. We will try to use areas of memory that we discover to be free to hold the extracted info. Memory is known to be free if:

* It is on the Multiboot memmap as a free RAM area or is in lower or upper memory, and
* It does not overlap the Multiboot structures themselves, and
* It does not overlap the kernel image itself, and
* It does not overlap any of the other modules loaded by GRUB as indicated by the Multiboot module list.

In order to use memory in this way fairly easily, we must know ahead of time how much we will need. It obviously must be virtually contiguous when it is mapped later, but... Q: Will we require the memory to be physically contiguous?

A: Yes. This approach introduces a tighter limit, but will be easier to implement.

A: No. The algorithm will have to take pagination into account and must map accordingly. In position-independent assembler code, this will be very difficult.

* + - * 1. A: No. In this case, we intend to try and map the Multiboot structure before paging is enabled. This is equivalent to doing so after paging is enabled, except less maintainable since the code must be position-independent. Therefore, this solution doesn’t make much sense.
      1. A: No. The Multiboot info must be made virtually accessible somehow. In the following cases, parts of the Multiboot info structure may reside at physical addresses corresponding to the kernel’s virtual addresses (or the virtual addresses of kernel modules loaded by GRUB). This area of the physical address space is called “the forbidden zone” for the sake of discussion. Note that the “forbidden zone” cannot be statically known, since it depends on the location of modules loaded by GRUB. Therefore, this branch of possible solutions requires some smarts before paging is enabled to detect the full extent of the forbidden zone. Q: Will the Multiboot info be faulted in with special page fault handling?
         1. A: Yes. A special page fault handler will be used that identity-maps the Multiboot info structure on demand using 4MB pages (so that no page tables need to be allocated). We must explicitly test parts of the structure to see if they overlap the forbidden zone, even though we are otherwise using a PF handler to map on-demand. We just panic if part of the Multiboot info overlaps the forbidden zone, since otherwise we’d have to move parts of it around, making identity-mapping impossible. This is an unlikely case, but nevertheless it represents a limitation of this approach.
         2. A: No. In this case, the first part of the Multiboot info will be mapped “by hand”. Subsequent parts of the structure will be mapped using 4MB pages and pointers to them translated “by hand”. Q: Do we just panic if part of the Multiboot info overlaps the forbidden zone?

A: Yes. This is an unlikely case, but nevertheless represents a limitation of this approach.

A: No. We are free to move parts of the structure around in memory, since we can adjust the pointers to them. However, we need a place to which to move them. This implies that some lower and upper-memory detection must have taken place before paging was enabled. Q: Do we rely solely on the lower and upper memory to hold parts of the structure that we copy?

A: Yes. This is yet another limitation that otherwise simplifies the algorithm.

A: No. We can copy parts of the memmap into this free area and use it to discover more free areas. If the free area is not big enough to hold the entire memmap, we can examine one region at a time and absorb the free memory that way. It’s complex, but completely free of artificial limitations.

#### Solutions Identified

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Solution | Description | Complexity | C | Limitations | L | Score |
| 1.1 | Extra bootloader stage. | Affects build environment and requires designing a protocol between the new loader and the kernel. Actual implementation is easy. | 9 | None. | 0 | 9 |
| 1.2.1 | Panic if Multiboot outside 4MB. | Very little. | 1 | If GRUB decides to put the Multiboot structure outside the first 4MB, the system won’t boot. | 8 | 9 |
| 1.2.2.1.1.1.1 | Copy entire Multiboot structure into pre-allocated kernel area before paging. | Not bad – some position-independent assembler, but the algorithm is very simple. | 2 | Whatever guess we make on the max size of the Multiboot structure is the limit. | 10 | 12 |
| 1.2.2.1.1.1.2 | Copy the main structure into a pre-allocated area along with enough info to use upper memory, and then do the rest after paging is enabled (**partial solution**). | Moderate – some position-independent assembler, but the algorithm is relatively simple. Don’t forget to add the complexity of the post-paging part of the solution. | 3 | None, but this is a partial solution only. | 0 | 3 |
| 1.2.2.1.1.2.1 | Use upper-memory info to copy a summary of the Multiboot structure into physically contiguous RAM before paging is enabled. | High – lots of PIC assembler doing relatively complex things. | 7 | There is an unlikely chance that the summarized info won’t fit within a single 4MB frame, or that there aren’t enough contiguous 4MB frames to hold it. | 3 | 10 |
| 1.2.2.1.1.2.2 | Use upper-memory info to copy a summary of the Multiboot structure into not-necessarily physically contiguous RAM before paging is enabled. | Extreme – like the previous solution, only does a lot of what the VMM would be doing. | 10 | None. | 0 | 10 |
| 1.2.2.1.2 | Just like 1.2.2.2.2, only more complex. Therefore, pointless. | Extreme | 10 | None. | 0 | 10 |
| 1.2.2.2.1 | On-demand mapping | High – must calculate “forbidden zone” before enabling paging, and must co-ordinate with PF handler. | 7 | If any of the Multiboot structures overlap the “forbidden zone”, the system won’t boot. | 3 | 10 |
| 1.2.2.2.2.1 | Like the previous, but without the PF handler. | Moderate – like the previous, but without the PF handler. | 6 | Same as previous. | 3 | 9 |
| 1.2.2.2.2.2.1 | “Hand-map” the structure, moving parts out of “the forbidden zone” as necessary into upper memory. | High – like the previous, but with extra pre-paging logic to detect free memory and post-paging logic to use it. | 8 | It is possible that there isn’t enough free upper memory to hold the parts of the structure that must be moved. | 2 | 10 |
| 1.2.2.2.2.2.2 | Same as previous, but uses all available RAM. | Very High – like the previous, but incrementally adds more free RAM as the mmap is processed. | 9 | None. | 0 | 9 |

#### BootLoaderInfoTranslator Algorithm

This is the algorithm for producing a BootLoaderInfo structure from the MultibootInfo structure. It assumes that if the algorithm runs with paging enabled, that the MultibootInfo structure is present in its entirety in virtual memory and its pointers point to the correct virtual addresses.

The BootLoaderInfo structure is held in memory discovered to be free during the translation process. Therefore, the first step in this process must be to determine how much free memory is needed. This is a function of how many modules there are, how many memory regions are free (not including those known to be used), and how much string data needs to be copied over.

The next step is to find enough free memory to hold the BootLoaderInfo. This requires traversing the Multiboot info as if it contained a proper free list that takes into account the kernel image, modules, and Multiboot structure itself. This can be done in stages:

1. First, sort the Multiboot mmap and module list. Bubble sort would suffice here, for the sake of simplicity.
2. Next, test the mmap and module lists to make sure none of the entries overlap.
3. Next, wrap the “available RAM” entries of the mmap (or mem\_lower and mem\_upper) in a simple forward iterator.
4. Next, wrap the “other” entries of mmap, the Multiboot structure extents, kernel image extent, and module list in a complex forward iterator (i.e. – something that pretends that all of these disparate pieces of info are really a single sorted sequence of used regions).
5. Finally, wrap the two forward iterators in another complex iterator that is able to superimpose the used regions on top of the free regions. It presents a single, correct, sorted sequence of free regions.

Step 4 can used the technique of step 5 to break down the “used” list into multiple “virtual sequences”. This makes the overall structure more complex while making each iterator simpler.

Once an authoritative “free list” can be read from the Multiboot structures, enough free memory must be found to hold the BootLoaderInfo. In theory, this memory need not be physically contiguous, but if it’s not, it must be possible to make it virtually contiguous. This makes the algorithm very tricky:

1. Search the list for a region of physical memory large enough to hold BootLoaderInfo. If one is found, we’re done – make sure it’s all virtually mapped.
2. If a large enough region is not found, find the first region ending on a 4MB boundary and make sure it is virtually mapped. Store its address as the beginning of the “used list” for the BootLoaderInfo.
3. At this point, we must incorporate the temporary “used list” into our “virtual iterator” of physical memory regions.
4. Using the new list, find the first region beginning on a 4MB boundary that’s large enough to hold the rest of the BootLoaderInfo. If one is found, make sure it is mapped virtually after the previously allocated page (this may require re-mapping all previously allocated pages if this region is in the kernel’s 4MB page) and write its physical address to the beginning of the previously allocated region.
5. No such region is found. If the remaining portion of the BootLoaderInfo is 4MB or less, then panic – there isn’t enough contiguous memory to do the job.
6. Otherwise, try to find a 4MB-aligned 4MB free region. If one is found, virtually map it after the previously allocated page and write its physical address to the beginning of the previously allocated region. Go to step 4.
7. Otherwise, panic – there isn’t enough memory to do the job.

Everything after step 1 is extremely complex and probably not worth the effort.

Another way to avoid the need for contiguous memory to store BootLoaderInfo is to change the way BootLoaderInfo records physical memory. Instead of having an array of free regions, it can instead contain a bitmap of free frames. The advantage is that it is possible to put an upper bound on the amount of memory required for the memory portion of BootLoaderInfo. The disadvantages are:

* We’re prematurely breaking memory up into frames, which is really the job of whoever initializes the PMM.
* It is still fairly complex to incrementally create the bitmap while allocating and mapping memory for it.
* The BootLoaderInfo itself is still variable-size, since the module list is of unknown length.

Ultimately, it is still best to stick with the algorithm above, skipping steps 2 through 7 and relying on there being enough free contiguous physical memory to hold the BootLoaderInfo.

See “pmm approaches.txt” for the complete algorithm.

### BootLoaderInfo Requirements

In the x86 version of Precursor, a multiboot-compliant bootloader such as GRUB creates a MultibootInfo structure and leaves it at some location in physical memory. There are other non-Multiboot bootloaders for x86, and none of the bootloaders for other architectures are Multiboot-compliant (which makes sense, since Multiboot is x86-specific). Some of the Multiboot structure is architecture-specific (such as BIOS drives and VESA video modes) and some is architecture-neutral (memory map, kernel command line, module list).

Things we might want in priority order:

1. To make the info accessible to the kernel in virtual memory without any hassles (i.e. – it would be nice if it were already mapped when the kernel is first booted or shortly thereafter).
2. For the parts of the info that cannot be reclaimed, they should cause as little fragmentation as possible. They should not interfere with the normal operations of the physical or virtual memory managers.
3. To have the parts of the kernel that consume the architecture-neutral parts of the Multiboot info be completely portable to other architectures (and by extension, non-Multiboot bootloaders).
4. Wherever possible we must avoid arbitrary limitations on memory use during booting that would cause booting to fail on memory-constrained machines (or machines where, through some twisted flaw, the Multiboot info fragments the available free memory too much to be usable – although this should be nearly impossible).
5. To make it easy for the kernel to reclaim memory used by transient parts of the Multiboot info (i.e. – parts that are not needed after kernel initialization, such as the memory map). Don’t bother.
6. To support non-Multiboot bootloaders on x86. Why bother?

Things which just aren’t possible:

* Having a completely architecture-neutral protocol between the bootloader and the kernel. Some of that info is architecture-specific, and there is nothing that can be done about it.

Requirements #1 and #2 together suggest that either a second-stage bootloader or the kernel initialization routine should relocate and pack the Multiboot structures into page-aligned memory somewhere and then map them into the kernel’s address space (i.e. – in the higher half). This may conflict with item #4.

Requirement #2 alone would suggest that as an alternative, the kernel can relocate and re-map the Multiboot info in a later stage of initialization. However, this conflicts with item #1, and is therefore not a recommended solution.

Requirement #3 can be solved in one of two ways – either by translating the architecture-neutral parts of the Multiboot info into an internal kernel structure, or by wrapping it with an architecture-neutral API. The API solution works best in the absence of requirement #5 (which is fine; see next item). The translation solution works with the first proposed solution for requirements #1 and #2, although it makes them more complex. On the balance of things, this makes the API solution more attractive. Note that either solution must ensure that the memory used by the non-transient portions of the Multiboot info structure (if any) must be subtracted from the free RAM list. Otherwise, the supposedly architecture-neutral consumer of the free RAM list must know about the Multiboot structure, which is a contradiction. In the translation solution, there is no problem because the translated structure itself can be walked by the consuming code since the translated structure is itself architecture-neutral. In the API solution, the Multiboot info extents must somehow be subtracted from the free list dynamically as it is requested by the caller.

Requirement #4 is a lot easier to handle with an intermediate bootloader. Without one, there is little point in bending over backwards to satisfy this requirement.

Requirement #5 makes requirements #1 and #2 more difficult to fulfill, therefore it will be dropped. However, in dropping it, it makes requirement #2 arguably even more important (i.e. – if you can never reclaim any of the memory, it had better not be using precious resources like 4MB pages, even if it consumes very little physical memory).

Requirement #6 will be dropped out of hand. If necessary, a custom chain-loaded Multiboot-compliant bootloader can be built.

## Memory Management (Take 2 – December 30, 2005)

There were many problems with the L4-style scheme that necessitated a re-design:

* (Minor) Memory must be divided at boot-time between the kernel and applications.
* (Medium) L4 can’t handle PAE, and will have trouble if physical addresses are ever larger than virtual ones on 64-bit architectures (it is possible).
* (Severe) The L4 scheme requires (IMO) a lower-half kernel. This is because the need to identity-map sigma0 would put a higher-half kernel in conflict with the PCI configuration space (or something like that… I still need to learn more about PCI). This is a pretty big problem.
* (Minor) L4’s scheme can lead to conceptually complex systems with multiple pagers. This level of flexibility is overkill for Precursor.

### Alternatives to L4’s Hierarchical Address Spaces

One of the big problems with L4’s scheme is that it melds together the management of the virtual and physical address spaces. It is also somewhat complex and difficult to understand. Precursor does not require the level of flexibility that the L4 scheme provides. This section explores an alternative motivated by a few goals:

* Precursor should make it obvious how a driver is to allocate specific physical address ranges.
* Precursor should support high-performance message-passing based on page table manipulation (see the section on IPC below). Not sure about the TLB shootdown overhead of this though…
* Precursor must still be able to support a variety of swap mechanisms (e.g. – disk, network, or none at all), if not a variety of paging policies.

#### Use Cases for a VMM

This section describes a few typical “use cases” for a virtual memory manager. These will help in analyzing the various design options available.

##### Use Case #1: Process Start

1. A process asks the OS to create a new process. The OS is directed to load the executable code for the new process from a particular image on disk.
2. The OS creates a new address space.
3. The OS maps the executable image file into the part of the address space specified by the file’s header.
4. The OS creates a new thread and sets it to begin executing instructions in the executable code.
5. The new thread starts executing and immediately triggers a page fault and is blocked.
6. The OS handles the page fault by bringing in the first page of executable code from the image file on disk.
7. The OS unblocks the thread, which re-executes the faulting instruction and continues happily along.

##### Use Case #2: Working Set Trimming

1. Periodically, the OS attempts to “steal” pages from a running process by marking them as invalid and keeping them on a standby list.
2. Shortly after stealing some pages, the process references one of them and causes a page fault.
3. The OS handles the “soft” page fault by taking the page off the standby list and marking it valid again. The OS resumes the faulting thread.
4. Some time later, the OS notices that none of the other stolen pages have been referenced. It frees the clean pages so that they can be zeroed and re-used later.
5. The OS writes the dirty pages to disk and frees each one after it is written.

#### Proposal #1: In-kernel VMM Mechanisms with Policy in External Process Manager

In this scheme, the kernel is responsible for all VMM mechanisms, but it must rely on a trusted user-space server (called the Process Manager) to handle all policy, as well as the swapping mechanism (to disk, in this case).

##### Use Case #1 Analysis

* (Step #1): The calling process sends a message to the Process Manager.
* (Step #2): The Process Manager makes a system call to the kernel, which creates a new address space.
* (Step #3): The Process Manager opens the executable file (by making IPC calls to the file system) and reads its header. It sets up data structures that create the mapping between the executable image and the new address space.
* (Step #4): The Process Manager makes a system call for the kernel to create a new thread.
* (Step #5): The Process Manager replies to the original caller, giving the kernel a chance to schedule the newly-created thread for execution. When the new thread executes, it triggers a page fault and blocks.
* (Step #6): As read, with “the OS” being the Process Manager.
* (Step #7): Same as step 6. The Process Manager unblocks the thread by replying to it with a new page.

##### Use Case #2 Analysis

* (Step #1): A thread in the Process Manager occasionally runs, making kernel system calls to scan working sets for non-accessed pages to steal. When it finds one, it makes another system call to transfer the page from the target address space to its own.
* (Step #2): A thread in the victim process references one of the stolen pages and causes a page fault. The kernel sends a PF message to the Process Manager.
* (Step #3): The Process Manager notes that the faulting address points to a page on its standby list (it would have to track this state somewhere for fast lookup, perhaps in a tree of hash tables). It makes a system call to transfer the page back to the target process and moves it from the standby list to the working set list.
* (Step #4): Eventually, the Process Manager’s background thread runs again. It makes some system calls to release the clean pages back to the kernel for zeroing.
* (Step #5): The Process Manager uses the file system (via IPC) to write the dirty pages back to disk.

#### Proposal #2: In-kernel VMM with External Pager

In this scheme, the kernel is responsible for all VMM policy, and all mechanisms except for persisting pages. The kernel must rely on a trusted user-space server (called the Pager) to handle the transfer of pages to and from an external medium (disk in this case).

##### Use Case #1 Analysis

* (Steps #1 through #5): Same as for proposal #1.
* (Step #6): The kernel handles the page fault and notices that the page is externally mapped. It sends a message to the Process Manager, which uses the file system to begin reading in chunks of the file from disk and give them to the kernel.
* (Step #7): The kernel dismisses the page fault and resumes the thread.

##### Use Case #2 Analysis

* (Step #1): A kernel background thread performs working-set trimming as per NT, by scanning and manipulating the page tables directly.
* (Step #2): A thread references a page on the standby list, causing a page fault that is handled by the kernel.
* (Step #3): The kernel handles the “soft” PF as per NT.
* (Steps #4 and #5): The kernel’s working set trimmer frees some standby pages and sends messages to the Process Manager/swapper instructing it to write the dirty pages out to disk.

### Proposal #1 Details

I have chosen to implement proposal #1 because it allows me to implement a minimal kernel sooner and worry about the more complex aspects of memory management later. It also provides a measure of safety, as the Process Manager will be in its own address space. It seems less performant than proposal #2, but my goal is to get a microkernel working as quickly as possible, not to design the fastest hobbyist microkernel ever.

With that said, here are the details for proposal #1. The kernel will implement all physical memory management, as well as minimal non-RAM physical address space management. The kernel will also implement the virtual memory management mechanisms, but not the policies. The Process Manager will invoke these mechanisms via a set of system calls that only it can invoke. These are detailed below.

#### RegisterPageFaultHandler( chid, affinitymask )

This registers the given channel for page fault handling. When a page fault occurs, the kernel will send the PF message to one of the registered channels. It will choose the channel based on processor affinity, or else it will just choose the next one with a blocked receiver if none have an affinity.

#### RegisterCriticalThread()

This tells the kernel that the calling thread will be handling page faults incurred by the Process Manager itself. That means that if the critical thread itself causes a page fault, the kernel will terminate it. This will probably bring down the entire system.

#### WaitForPageFault( chid, &asid, &vaddr, &flags )

This blocks the calling thread until the kernel dispatches a page fault message to it. Note that once it is initialized, the Process Manager handles its own page faults. This is how it allocates memory to itself.

#### CreateAddressSpace( &asid )

This creates a new empty address space. The kernel allocates a new page directory and associates it with a new address-space identifier (asid).

#### DeleteAddressSpace( asid )

This deletes an address space, freeing all physical pages that it maps.

#### PageControl( asid, vaddr, npages, &flags )

This can be used to query or modify the protection of the given virtual address region. This can only be used to query user pages, not kernel pages, so the only protection information is readable/writable/executable. Accessed/dirty status information can also be obtained when querying.

#### TrimKernelWorkingSet()

The Process Manager calls this when Alloc fails due to memory exhaustion. The kernel indicates in the return value whether it was able to free up some of its reserved memory or not (i.e. – memory that is on the kernel’s own free lists but is otherwise not being used). If it was, the Process Manager should try its allocation again. If it fails the second time, then the overall allocation operation should be considered a failure.

#### Alloc( chid, paddr, npages, flags, isContiguous )

The Process Manager calls this to respond to a page fault message. It maps one or more physical pages into the faulting thread’s address space (possibly its own). How the pages are obtained depends on the paddr parameter:

* If paddr is NULL, the kernel allocates physical pages (using colouring for performance reasons, if the kernel supports it). The call fails if not enough pages are available.
* If paddr is not NULL…
  + …and corresponds to a region with at least one allocated physical page, the call fails.
  + …and corresponds to a large enough range of free physical pages, the kernel allocates those pages.
  + …and does not correspond to physical memory…
    - …and corresponds to a kernel-reserved region of the physical address space, the call fails.
    - …and corresponds to a free region of the physical address space, the kernel maps the region. It does not “allocate” anything since it does not track ownership of such regions.

The flags parameter controls the protection of the mapped pages. Note that when the kernel allocates physical RAM, it sets a reference count for those pages to 1. This is used by the Share() and Free() calls described later.

#### Share( chid, srcasid, srcvaddr, npages, flags )

The Process Manager calls this to respond to a page fault message. It maps one or more pages from an address space into the faulting thread’s address space. The given virtual address region must be mapped and outside the kernel’s virtual address space. The flags parameter controls the protection on the target pages. If the any of the pages being shared are physical RAM, the kernel tracks the sharing by increasing the appropriate reference count(s).

#### Free( asid, vaddr, npages )

This unmaps the given virtual address region from the target address space. The call fails if the region is not already mapped. If any pages in the region correspond to physical RAM, the kernel frees those pages only if the appropriate reference counts decrement to zero.

#### Transfer( chid, srcasid, srcvaddr, npages, flags )

The Process Manager calls this to respond to a page fault message. It maps one or more pages from the source address space into the faulting thread’s address space and subsequently unmaps them from the source address space. The given virtual address region must be mapped and outside the kernel’s virtual address space. The flags parameter controls the protection on the target pages. There is no need for the kernel to manage reference counts, since the number of references does not change in this case.

### Proposal #1 Physical Memory Allocator Analysis

Of the design concerns for physical memory allocators, the concerns 1 and 2 do not matter for this memory management scheme. Contrast this to the L4 scheme, where concern 4 also does not matter. The usage characteristics also differ:

* The PMM will manage all physical memory – both that owned by the kernel and that owned by user processes. It will likely do this with an array of tracking structures, the size of which will be determined by the highest physical address of usable RAM in the system. For the 32-bit non-PAE version of Precursor, this can mean more than 3GB worth of physical address space to track.
* The kernel’s usage characteristics will mirror those given for the L4 scheme.
* Since this scheme manages all physical memory, the usage characteristics of user processes must be taken into account. Since these are unpredictable, this requires us to use a PMM scheme that is performant in the general case.

With this scheme, the best PMM structure is an array containing several doubly-linked lists of frames (the lists must be doubly-linked in order to support removal from the middle of a list). Each list will represent a different state:

* Bad list: frames that have generated errors, frames that correspond to reserved regions of the physical address space (e.g. – motherboard resources), or frames that correspond to unused (from the kernel’s perspective) regions of the physical address space.
* Free lists: one per page colour.
* Zero lists: one per page colour.
* Kernel free lists: one per page colour.
* Below 16MB free list: for special DMA memory.
* Unlinked frames, which are allocated. These entries will contain a reference count for sharing purposes, and will be linked to other allocated frames for temporary book-keeping purposes.

### Proposal #1 Pseudo-code

The following is pseudo-code for the four main memory-management system calls.

#### Alloc( chid, paddr, npages, flags, contigFlags )

If page tables for destination asid is not already mapped in Hyperspace on the current

processor

Map it.

End If

If paddr is NULL or paddr is in PFDB

// Cases:

// paddr contig below16 result

// NULL 0 0 alloc anywhere, prefer >16MB

// NULL x 1 alloc contiguous <16MB

// NULL 1 0 alloc contiguous anywhere, prefer >16MB

// paddr x x alloc contiguous at paddr

If paddr is not NULL

allocateContiguousAt( paddr, npages )

Else If contigFlags are all clear

allocate( npages, faulting vaddr )

Else

If below16 is clear

allocateContiguous( npages, false, nox64k )

End If

If allocation failed or not done yet

allocateContiguous( npages, true, nox64k )

End If

End If

If the allocation failed

Return an error.

End If

Zero any of the allocated frames that are not already zeroed.

Lock the target region starting at the faulting vaddr.

Map the allocated frames to the target region using the given flags.

Do a TLB shootdown for all newly-mapped vaddrs on all processors that have the

faulting asid currently active.

Unlock the target region.

Else If paddr is in a reserved area

Return an error.

Else

Lock the target region starting at the faulting vaddr.

Map npages frames @paddr to the target region using the given flags.

Do a TLB shootdown for all newly-mapped vaddrs on all processors that have the

faulting asid currently active.

Unlock the target region.

End if

#### +PageFrameDB\_allocate( nframes, colourHint )

// Don't need to check the first allocation – the rest of the logic works out ok for the

// boundary condition.

firstFrameNumber = allocateAny( colourHint )

nframes--

currentFrameNumber = firstFrameNumber

While currentFrameNumber != 0 and nframes != 0

colourHint += PAGE\_SIZE

nextFrameNumber = allocateAny( colourHint )

// No need to lock freshly-allocated PFDB entries.

pfdb[currentFrameNumber].next = nextFrameNumber

currentFrameNumber = nextFrameNumber

If currentFrameNumber != 0

nframes--

If nframes == 0

pfdb[currentFrameNumber].next = 0

End If

End If

End While

// Must rollback if we couldn't allocate enough.

If nframes != 0

freeChain( firstFrameNumber )

Return 0.

Else

Return currentFrameNumber.

End If

#### -PageFrameDB\_allocateAny( colourHint )

// NOTE: This implementation ignores colourHint.

zero\_list.lock()

frame = zero\_list.allocate( pfdb, ALLOCATED\_ZERO )

zero\_list.unlock()

If (frame != 0)

Return frame.

End If

free\_list.lock()

frame = free\_list.allocate( pfdb, ALLOCATED )

free\_list.unlock()

If (frame != 0)

Return frame.

End If

below\_16\_free\_list.lock()

frame = below\_16\_free\_list.allocate( pfdb, ALLOCATED )

below\_16\_free\_list.unlock()

Return frame.

#### -PageFrameDB\_kernelAllocate( colourHint )

// NOTE: This implementation ignores colourHint.

kernel\_list.lock()

frame = kernel\_list.allocate( pfdb, ALLOCATED\_FOR\_KERNEL\_USE )

kernel\_list.unlock()

If (frame != 0)

Return frame.

End If

// The kernel doesn't need zero pages as much as user processes do.

free\_list.lock()

frame = free\_list.allocate( pfdb, ALLOCATED\_FOR\_KERNEL\_USE )

free\_list.unlock()

If (frame != 0)

Return frame.

End If

zero\_list.lock()

frame = zero\_list.allocate( pfdb, ALLOCATED\_FOR\_KERNEL\_USE )

zero\_list.unlock()

If (frame != 0)

Return frame.

End If

below\_16\_free\_list.lock()

frame = below\_16\_free\_list.allocate( pfdb, ALLOCATED\_FOR\_KERNEL\_USE )

below\_16\_free\_list.unlock()

Return frame.

#### -PageFrameDB\_kernelFree( frameNumber )

assert( frameNumber in PFDB and type is ALLOCATED\_FOR\_KERNEL\_USE and refcount == 1 )

pfdb[frameNumber].refcount--

If frameNumber is above 16MB

list = kernel\_list

Else

list = below\_16\_free\_list

End If

list.lock()

list.free( pfdb, FREE\_FOR\_KERNEL\_USE, frameNumber )

list.unlock()

#### +PageFrameList\_allocate( pfdb, allocationType )

assert( allocationType is ALLOCATED, ALLOCATED\_ZERO, or ALLOCATED\_FOR\_KERNEL\_USE )

If list is empty

Return 0.

End If

firstFrame = list\_head

list\_head = pfdb[firstFrame].next

pfdb[list\_head].prev = 0

pfdb[firstFrame].type = allocationType

pfdb[firstFrame].refcount = 1

pfdb[firstFrame].next = 0

Return firstFrame

#### -PageFrameDB\_freeChain( frameNumber )

assert( frameNumber in PFDB )

currentFrameNumber = frameNumber

While currentFrameNumber != 0

// No locks needed when touching exclusively-owned allocated pages.

assert( pfdb[currentFrameNumber].refcount == 1 )

pfdb[currentFrameNumber].refcount--

nextFrameNumber = pfdb[currentFrameNumber].next

free( currentFrameNumber )

currentFrameNumber = nextFrameNumber

End While

#### -PageFrameDB\_free( frameNumber )

assert(

(frameNumber is in PFDB) and

(pfdb[frameNumber].type is ALLOCATED or ALLOCATED\_ZERO)

)

// This implementation ignores the colour of frameNumber.

If frameNumber is above 16MB

list = free\_list

Else

list = below\_16\_free\_list

End If

list.lock()

list.free( pfdb, FREE, frameNumber )

list.unlock()

#### +PageFrameList\_free( pfdb, freeType, frameNumber )

assert( freeType is FREE, FREE\_ZERO, or FREE\_FOR\_KERNEL\_USE )

assert( frameNumber is in PFDB )

assert( pfdb[frameNumber].refcount == 0 )

assert(

pfdb[frameNumber].type is ALLOCATED or ALLOCATED\_ZERO or ALLOCATED\_FOR\_KERNEL\_USE

)

pfdb[frameNumber].type = freeType

pfdb[frameNumber].prev = 0

pfdb[frameNumber].next = list\_head

pfdb[list\_head].prev = frameNumber

list\_head = frameNumber

#### +PageFrameDB\_allocateContiguousAt( startFrameNumber, nframes ) (option #1 \*\*\*FIXME: resurrect option #2)

assert( startFrameNumber in PFDB and is not 0 )

If (startFrameNumber + nframes – 1) would overflow or is not in PFDB

Return 0.

End If

// Don't need to check the first allocation – the rest of the logic works out ok for the

// boundary condition.

lockAllLists()

firstFrameNumber = allocateSpecific( startFrameNumber )

unlockAllLists()

nframes--

currentFrameNumber = firstFrameNumber

While currentFrameNumber != 0 and nframes != 0

lockAllLists()

nextFrameNumber = allocateSpecific( currentFrameNumber + 1 )

unlockAllLists()

// No need to lock freshly-allocated PFDB entries.

pfdb[currentFrameNumber].next = nextFrameNumber

currentFrameNumber = nextFrameNumber

If currentFrameNumber != 0

nframes--

If nframes == 0

pfdb[currentFrameNumber].next = 0

End If

End If

End While

// Must rollback if we couldn't allocate enough.

If nframes != 0

freeChain( firstFrameNumber )

Return 0.

Else

Return currentFrameNumber

End If

#### -PageFrameDB\_allocateSpecific( frameNumber )

assert( frameNumber in PFDB )

If pfdb[frameNumber].type not FREE or FREE\_ZERO

Return 0.

End If

If frameNumber < 16MB

list = below\_16\_list

If pfdb[frameNumber].type == FREE

allocationType = ALLOCATED

Else

allocationType = ALLOCATED\_ZERO

End If

Else If pfdb[frameNumber].type == FREE

list = free\_list

allocationType = ALLOCATED

Else

list = zero\_list

allocationType = ALLOCATED\_ZERO

End If

Return list.allocateSpecific( pfdb, allocationType, frameNumber )

#### +PageFrameList\_allocateSpecific( pfdb, allocationType, frameNumber )

assert( allocationType is ALLOCATED, ALLOCATED\_ZERO, or ALLOCATED\_FOR\_KERNEL\_USE )

assert( frameNumber in PFDB )

assert( pfdb[frameNumber].type is not FREE or FREE\_ZERO )

prevFrame = pfdb[frameNumber].prev

nextFrame = pfdb[frameNumber].next

If prevFrame != 0

pfdb[prevFrame].next = nextFrame

End If

If nextFrame != 0

pfdb[nextFrame].prev = prevFrame

End If

pfdb[frameNumber].type = allocationType

pfdb[frameNumber].refcount = 1

pfdb[frameNumber].next = 0

Return frameNumber

#### +PageFrameDB\_allocateContiguous( nframes, below16, nox64k )

If nframes is outside PFDB

Return 0.

End If

If below16

If nox64k and (64k < (1 + nframes))

startFrameNumber = 64k

Else

startFrameNumber = 1

End If

Else

startFrameNumber = 16MB // Already on 64KB boundary.

End If

If (startFrameNumber + nframes – 1) would overflow

Return 0.

End If

lastFrameNumber = startFrameNumber + nframes – 1

lockAllLists()

While (below16 and (lastFrameNumber < 16MB)) or

(!below16 and (lastFrameNumber is in PFDB))

For frameNumber = startFrameNumber; frameNumber <= lastFrameNumber; frameNumber++

If pfdb[frameNumber].type is not FREE or FREE\_ZERO

If (frameNumber + 1) would overflow

Break

End If

startFrameNumber = frameNumber + 1

If (startFrameNumber + nframes – 1) would overflow

Break

End If

lastFrameNumber = startFrameNumber + nframes – 1

If nox64k

If ((nframes < 64k) and

(lastFrameNumber is above next 64k boundary

after startFrameNumber)) or

((64k <= nframes) and

(startFrameNumber not on 64k boundary)

startFrameNumber = next highest 64k boundary

If (startFrameNumber + nframes – 1) would overflow

Break

End If

lastFrameNumber = startFrameNumber + nframes – 1

End If

End If

Continue

End If

End For

allocateSpecific( startFrameNumber )

For frameNumber = startFrameNumber + 1; frameNumber <= lastFrameNumber;

frameNumber++

allocateSpecific( frameNumber )

pfdb[frameNumber – 1].next = frameNumber

End For

pfdb[lastFrameNumber].next = 0

unlockAllLists()

Return startFrameNumber

End While

unlockAllLists()

Return 0.

#### Share( chid, srcasid, srcvaddr, npages, flags )

If page tables for destination and source asids are not already mapped in Hyperspace on

the current processor

Map them.

End If

If srcasid is invalid or srcvaddr is in kernel-space

Return error.

End If

// Whenever regions are locked or unlocked, do them in whatever order guarantees no

// deadlock (e.g. – lock lower-address regions first and unlock them last).

Find the npages PTEs starting at srcvaddr in srcasid.

Lock this source region.

If any of the PTEs are invalid

Unlock the source region.

Return error.

End If

Find the npages PTEs starting at the faulting vaddr.

Lock this target region.

If any of the PTEs are valid

Unlock the target region.

Unlock the source region.

Return error.

End If

For each PTE in source region

Get PFN from PTE.

If PFN is in PFDB

If !PFDB\_addRef( PFN in PTE )

Unlock the target region.

Unlock the source region.

Return error.

End If

End If

assert( PFN is not in a reserved area )

End for

Copy npages PTEs from source page tables to destination page tables.

Apply protection flags to destination PTEs.

Do a TLB shootdown for all destination PTEs on all processors that have the faulting asid

currently active.

Unlock the target region.

Unlock the source region.

#### +PageFrameDB\_addRef( frameNumber )

assert( frameNumber is in PFDB )

If frameNumber is non-RAM

Return true. // Trivial success.

Else

assert( frameNumber is allocated or allocated zero )

Do

oldRefCount = atomic\_read( &pfdb[frameNumber].refcount )

If oldRefCount is at maximum

Return false.

Else

// This tests for a conflict between addRef() and release(), which

// must be prevented by proper locking of virtual address regions.

assert( oldRefCount != 0 )

newRefCount = oldRefCount + 1

End If

While !CAS( &pfdb[frameNumber].refcount, oldRefCount, newRefCount )

Return true.

End If

#### Free( asid, vaddr, npages )

If asid is invalid or vaddr is in kernel-space

Return error.

End If

If page tables for asid is not already mapped in Hyperspace on the current processor

Map it.

End If

Find the npages PTEs starting at vaddr in asid.

Lock this target region.

For each PTE in the target region

If PTE is invalid

Unlock the target region.

Return error.

End If

Get PFN from PTE.

If PFN is in PFDB

PFDB\_release( PFN in PTE )

End If

assert( PFN is not in a reserved area )

Make PTE invalid

End For

Do a TLB shootdown for the target vaddrs on all processors that have the asid currently

active.

Unlock the target region.

#### +PageFrameDB\_release( frameNumber )

assert( frameNumber is in PFDB )

If frameNumber is non-RAM

Return. // Trivial success.

Else

assert( frameNumber is allocated or allocated zero )

Do

oldRefCount = atomic\_read( &pfdb[frameNumber].refcount )

// Make sure some other release() didn't get there first. If so, we have a

// concurrent double-deletion bug. This must be prevented by properly

// locking the virtual address regions.

assert( oldRefCount != 0 )

newRefCount = oldRefCount – 1

While !CAS( &pfdb[frameNumber].refcount, oldRefCount, newRefCount )

If newRefCount == 0

free( frameNumber )

End If

End If

#### Transfer( chid, srcasid, srcvaddr, npages, flags )

If page tables for destination and source asids are not already mapped in Hyperspace on

the current processor

Map them.

End If

If srcasid is invalid or srcvaddr is in kernel-space

Return error.

End If

// Whenever regions are locked or unlocked, do them in whatever order guarantees no

// deadlock (e.g. – lock lower-address regions first and unlock them last).

Find the npages PTEs starting at srcvaddr in srcasid.

Lock this source region.

If any of the PTEs are invalid

Unlock the source region.

Return error.

End If

Find the npages PTEs starting at the faulting vaddr.

Lock this target region.

If any of the PTEs are valid

Unlock the target region.

Unlock the source region.

Return error.

End If

Copy npages PTEs from source page tables to destination page tables.

Apply protection flags to destination PTEs.

Make all source PTEs invalid.

Do a TLB shootdown on all processors that have either srcasid or the faulting asid

currently active.

// For the next two statements, unlock the regions in whatever order they were locked

// (e.g. – unlock higher-address regions first).

Unlock the target region.

Unlock the source region.

## Use Cases for IPC

IPC in Precursor will be based on message-passing. My initial ideas on this are focused on small message transfers via copying. Here are the things I plan to support:

* Small synchronous message passing (ala QNX send-receive-reply).
* Very small asynchronous message passing (blocking receive, non-blocking send, ala QNX events).

This leaves open the question of very large messages. Asynchronous message passing is out of the question for large messages if any copying is involved, as this will require large buffers in the kernel. This leaves a few possibilities:

* Synchronous copy-based messaging. This could include scatter-gather and streaming options as QNX does. This is pretty solid stuff, although somewhat inefficient due to the copying. In some use cases, this doesn’t matter because at least one copy would be required anyway and scatter-gather can solve a lot of problems here (especially for file system caches that may not be virtually contiguous). (Not sure if this is really true. It seems that it pays to avoid gathering until DMA’ing to the output device in case the device supports gather DMA, and the inverse for scattering – don’t scatter until copying into the application’s buffers.)
* Page-based messaging. The sender could unmap a set of pages from its own address space and have the OS map them into the receiver’s address space (this is more robust than sharing the memory). This could be synchronous or asynchronous, since the amount of data transferred is really small. This would allow for zero-copy transfers, but certain use cases might make it impractical due to the requirement for transferring page-aligned buffers.

To sort out which of these schemes Precursor should use, I will go through a few use cases/thought experiments that look at possible network stack and file system implementations on top of the Precursor microkernel.

Here is my current stack of questions:

1. How important is zero-copy? What should Precursor’s IPC look like in order to allow for low latency, high throughput, and security on I/O intensive tasks? The answers will come from these use-cases.
2. Given the requirements for IPC, what sort of memory management facilities should the entire Precursor OS offer to make that happen?
3. Given the general requirements for the entire OS’ memory management scheme, what parts of it should best be implemented in the microkernel (from the perspectives of performance and security)? Paging to disk is good use case to look at here.
4. How does the chosen microkernel VMM implementation deal with allocation of physical address space to device drivers?
   1. Side question – how do we detect those memory-mapped I/O regions without using the BIOS?
5. How will the microkernel VMM be implemented?
6. How will it be initialized?
7. What does it need from the current PMM design?
8. From the BootLoaderInfo?
9. What should I do next in the implementation of BootLoaderInfo and PMM initialization?

### File System Use Cases

Some fun facts about hard disks that will help inform these use cases:

* The typical size of a sector of a hard disk is 512 bytes.
* Today’s IDE drives found in most PCs have between 2 and 8 MB of onboard cache.

<FIXME>

### Network Protocol Stack Use Cases

Some fun facts about TCP/IP networking that will help inform these use cases:

* Precursor will probably only support Ethernet, which has a maximum frame size of 1500 bytes.
* The default MTU of IP is 576 bytes. If an IP datagram larger than the MTU must be transmitted, it must be fragmented (the IP headers support this) and re-assembled at the final destination. If fragments don’t arrive in time, the IP datagram is retransmitted in its entirety.
* The maximum size of an IP datagram is 64KB.
* UDP is really close to IP, and will just be treated as a simple extension to IP.
* TCP segments can be of arbitrary size, but it makes the most sense to fit a single segment into a single IP datagram. It seems quite pointless to do otherwise.

Tanenbaum’s top 7 list of system design rules for network performance is summarized as follows (items listed in order of importance):

1. CPU speed is more important than network speed.
2. Reduce packet count to reduce software overhead.
3. Minimize context switches.
4. Minimize copying.
5. You can buy more bandwidth but not lower delay.
6. Avoiding congestion is better than recovering from it.
7. Avoid timeouts.

The items we care about are 1, 3, and 4, and to a lesser extent, 2.

First, some ground rules for these use cases:

* Precursor is based on a microkernel architecture. This means that the protocol stack and NIC driver cannot be in kernel-space. It also means that the NIC driver in particular must be in a separate process from applications, since it must have enough privilege to talk to the NIC hardware directly.
* One thing Xok has taught me is that the minimum requirements for inter-application security are to ensure that each app has access only to packets that belong to it, and to prevent forgery of outgoing headers. This requires another process to de-multiplex incoming packets, and to verify outgoing headers, respectively.
* It is possible to have the protocol stack and NIC driver in separate processes. The advantage of this is better isolation of these two components from each other (i.e. – a bug in one cannot crash the other). The disadvantage is that it would probably be a major drag on performance since there would now be two inter-process boundaries to cross instead of just one (meaning more context switches, and possibly more copies). Having the NIC driver and protocol stack in the same process gives roughly inverted tradeoffs (plus it requires shared libraries to avoid an exploding number of possible statically-linked driver/protocol combinations).
  + The cost of a failure in the NIC and/or protocol stack is worth examining.
    - If the NIC driver fails but the protocol stack does not, the driver can simply be restarted and apps need never know something was wrong. This can only fail if the binary for the NIC driver is on a network file system (not advisable).
    - If the protocol stack fails, it’s irrelevant whether the driver failed or not, since all connection state is lost at that point. The protocol stack can be re-started, but apps will need to re-establish their connections.
* It is possible to have the network stack implemented as a library running in the application (this is roughly the Xok model). In this model, the NIC driver must have a little bit of protocol knowledge to de-multiplex incoming packets (in Xok, this is done with DPFs – downloadable packet filters). It is unclear whether this offers any performance advantages over putting the protocol stack in the same process with the NIC driver. It offers a slight reliability advantage in that if the protocol stack fails, only the application itself is affected (i.e. – other apps are not affected). It may have some disadvantages in terms of system fairness, but more research is needed in this area.
* The actors in the following use cases will be the Application, the TCP Stack, and the NIC Driver.
* The use cases will not discuss how the Application, TCP/IP Stack, or NIC Driver are implemented. It will also not discuss how IPC is implemented. It will discuss ownership of memory, however, since this is relevant to security.
* The OS cannot protect against a poorly-designed TCP/IP implementation. Therefore the following use cases will assume a well-designed TCP/IP implementation (e.g. – one that uses Nagle’s algorithm to avoid silly window syndrome, and buffers where appropriate to improve performance).

#### Use Case #1: Sending a Small Amount of Data over a TCP/IP Network

1. The Application has some data that it wishes to send over a socket. This data is very small, so it resides in a single buffer.
2. The Application calls the appropriate socket function to inform the TCP stack that it wishes to send some data.
3. The TCP stack is invoked. It notes that the amount of data to be sent is small enough to fit in an optimally-sized TCP segment (i.e. – a segment that is small enough to fit in a datagram less than the MTU), but it is so small that Nagle’s algorithm kicks in.
4. The TCP stack adjusts its cached header block for the current connection with the appropriate sequence number, etc.
5. The TCP stack invokes the IP stack, pointing it at the header and data block to be encapsulated (this probably does not require a copy since TCP and IP should be running in the same process).
6. The IP stack adjusts its own prototype header block as necessary and invokes the NIC driver, pointing it at the header and data blocks.
7. The NIC driver uses DMA to copy the headers and data into the onboard RAM for transmission.
8. The network card transmits the frame.
9. At some point, the Application decides to send a bit more data. It calls the appropriate socket function, pointing it at its buffer.
10. The TCP stack notes that the amount of data to send is still very small, and would not be enough to break it out of Nagle’s algorithm. Therefore, it needs to buffer the data until it receives an acknowledgement from the receiver for the first segment sent.
11. The TCP stack buffers a snapshot of the Application’s data (i.e. – data that the Application itself can no longer modify).
12. The Application slowly fills up the TCP stack’s buffer (with let’s say 10 send requests) until finally an acknowledgement arrives for the first segment.
13. An interrupt occurs, invoking the kernel. The kernel routes the interrupt to the NIC driver. The NIC driver obtains the packet from the network card and dismisses the interrupt. It then delivers the packet to the IP stack, which strips off the IP header and delivers the remainder to the TCP stack (routed to the Application’s original connection, of course).
14. The TCP stack sees that it has just received an acknowledgement for the first segment sent. Therefore, it puts together a TCP header and passes that, along with all the buffered data to the IP stack.
15. The IP stack puts together an IP header and passes that, along with the TCP header and data to the NIC driver.
16. The NIC driver DMAs the headers and data to the network card for transmission.

#### Use Case #2: Sending a Large Amount of Data over a TCP/IP Network

1. The Application has some data that it wishes to send over a socket. This data is very large and is split into many buffers in the Application’s memory space.
2. The Application calls the appropriate socket function to inform the TCP stack that it wishes to send some data that must be gathered from various buffers.
3. The TCP stack is invoked. It notes that the amount of data to be sent is too large to fit in an optimally-sized TCP segment (i.e. – a segment that is small enough to fit in a datagram less than the MTU). The data is also larger than the receiver’s last advertised window size.
4. Either the Application is blocked or a snapshot of its data is taken at this point.
5. The TCP stack adjusts its cached header block for the current connection with the appropriate sequence number, etc.
6. The TCP stack invokes the IP stack, pointing it at the header and portion of the data to be encapsulated (this probably does not require a copy since TCP and IP should be running in the same process).
7. The IP stack adjusts its own prototype header block as necessary and invokes the NIC driver, pointing it at the header and data blocks.
8. The NIC driver uses DMA to copy the headers and data into the onboard RAM for transmission.
9. The network card transmits the frame.
10. The TCP stack repeats the above steps (let’s say 10 times) until all of the Application’s data has been transmitted.

#### Use Case #3: Receiving Non-Fragmented Data over a TCP/IP Network

1. The NIC generates an interrupt, invoking the kernel.
2. The kernel routes the interrupt to the NIC driver.
3. The NIC driver DMAs the received frame from the NIC and dismisses the interrupt.
4. The NIC driver hands the new frame up to the protocol stack.
5. The IP stack is invoked. It inspects the header and routes the packet to the appropriate TCP connection.
6. The TCP stack is invoked, receiving the packet stripped of its IP headers.
7. If the Application is ready to receive data right now (blocking case), the TCP stack populates its target buffer with the received data (minus the TCP headers).
8. If the Application is not ready to receive data right now (non-blocking case), the TCP stack buffers the data. Later on, the Application invokes the TCP/IP stack to retrieve the buffered data.

#### Use Case #4: Receiving Fragmented Data over a TCP/IP Network

1. The NIC generates an interrupt, invoking the kernel.
2. The kernel routes the interrupt to the NIC driver.
3. The NIC driver DMAs the received frame from the NIC and dismisses the interrupt.
4. The NIC driver hands the new frame up to the protocol stack.
5. The IP stack is invoked. It inspects the header and sees that the frame is a fragment of a packet. It buffers the fragment and waits for the next one (space for the buffer is no problem, since an IP packet can be at most 64KB in length).
6. Steps 1 through 5 repeat until all the fragments are accounted for (let’s say there are 10 fragments that arrive after the first one). At this point, the IP stack passes the complete packet up to the TCP stack.
7. The TCP stack is invoked, receiving the packet stripped of its IP headers.
8. If the Application is ready to receive data right now (blocking case), the TCP stack populates its target buffer with the received data (minus the TCP headers).
9. If the Application is not ready to receive data right now (non-blocking case), the TCP stack buffers the data. Later on, the Application invokes the TCP/IP stack to retrieve the buffered data.

#### Observations for the Network Use Cases

|  |  |  |  |
| --- | --- | --- | --- |
| Protocol stack is application library | | Protocol stack is driver library | |
| **App-Stack interface** | **Stack-driver interface** | **App-stack interface** | **Stack-driver interface** |
| Can be blocking or non-blocking, but a non-blocking interface requires copying of data to prevent corruption. | Some kind of downloadable packet filter and header verifier must be supported by the network driver. | Can be blocking or non-blocking. A non-blocking interface does not require copying if page transfers are used. Page transfers would require a non-socket-compatible API or at best an optimization of sockets for properly-aligned buffers. | Calls can be blocking or non-blocking. On send(), copying is only required if the NIC does not support gather DMA. On recv(), no copying is required as buffers can simply be passed between the two layers. |
| Reduces the number of context switches in cases where the sender’s data is buffered due to Nagle’s algorithm. | There will be a context switch per packet sent. I don’t think there is enough flexibility in the TCP/IP standard to allow for buffering here (i.e. – when we say “send a packet”, we really mean it). | The number of context switches on a recv() will be reduced by Clark’s solution regarding silly window syndrome. | Context switches are required for almost every received frame, since it is likely that the NIC driver process will not be running at the time the NIC generates an interrupt. |
| Does not reduce context switches on packet reception, since the client app must reconstruct fragmented packets, for example. | Copies can be avoided via page transfers. Setting up address space for these transfers would be relatively easy, since the data sizes would be small. Page alignment could be part of the driver-stack interface. | There will generally be a context switch for every call to send() unless some “uninformed” buffering is done in the Application. |  |
| A lot of distributed control information must be made available to all apps using the protocol library. | Scatter/gather could be implemented either for copying or page transfers. |  |  |

#### Performance Analysis for the Network Use Cases

##### Protocol stack is application library

Use case #1:

* (Step #6) Either a gather copy or gather page transfer is required to transfer the header and data blocks to the NIC driver when the first segment is transmitted.
* (Step #6) There is a context switch to (and from, if blocking) the NIC driver when transferring the first segment.
* (Steps #9 through #12) For each of the Application’s 10 subsequent send requests, the data being sent is copied into the TCP stack’s buffers.
* (Step #13) A context switch to the NIC driver occurs when the NIC generates an interrupt signaling the arrival of the acknowledgement.
* (Step #13) A context switch to the Application (specifically the IP stack) occurs when the NIC transfers the acknowledgement to the IP stack.
* (Step #13) A single-block copy or page transfer is required in order to transfer the acknowledgement to the IP stack.
* (Step #15) Another gather copy or page transfer is required to transfer the remaining data to the NIC driver.
* (Step #15) Another context switch to the NIC driver is required to transfer the remaining data. A smart protocol between the TCP/IP stack and the NIC driver here will prevent the need for a context switch back to the stack and then back to the NIC driver again (i.e. – use a message-passing model rather than a call-return model).
* Grand totals:
  + 2 gather copies or page transfers (1 per packet sent)
  + 1 block copy or page transfer
  + 10 buffer copies (1 per send() call)
  + 5 context switches (Stack -> NIC driver -> Stack, Other App -> NIC driver -> Stack -> NIC driver), 4 if the protocol between the Stack and the NIC driver is non-blocking (i.e. – the first transition is Stack -> NIC driver, but not back again). (1 per packet sent, 2 if blocking, plus 2 per ack received.)

Use case #2:

* (Step #4): In the non-blocking case, a buffer copy is required in order to take a snapshot of the Application’s data prior to transmission.
* (Step #7): A context switch to the NIC driver is required. Another one back is required in the blocking case.
* (Step #7): A gather copy or page transfer is required to send the header and data blocks to the NIC driver.
* (Step #10): Repeat the above two steps for each of the 10 remaining packets to transmit.
* Grand totals:
  + 11 gather copies or page transfers (1 per packet sent)
  + 1 buffer copy in the non-blocking case
  + 22 context switches (Stack -> NIC driver -> Stack), 11 if the protocol between the Stack and the NIC driver is non-blocking (i.e. – the first transition is Stack -> NIC driver, but not back again). (1 per packet sent, 2 if blocking.)

Use case #3:

* (Step #2): A context switch to the NIC driver occurs when the packet arrives.
* (Step #4): A context switch to the Application is needed to transfer the received packet to the protocol stack.
* (Step #4): A block copy or page transfer is required to transfer the received packet to the protocol stack.
* (Step #7 or #8): A buffer copy is required to deliver the data to the application.
* Grand totals:
  + 1 block copy or page transfer (1 per packet received)
  + 1 buffer copy (1 per recv() call)
  + 2 context switches (Other App -> NIC Driver -> Stack)

Use case #4:

* (Step #2): A context switch to the NIC driver occurs when the packet arrives.
* (Step #4): A context switch to the Application is needed to transfer the received fragment to the protocol stack.
* (Step #4): A block copy or page transfer is required to transfer the received fragment to the protocol stack.
* (Step #6): Repeat all of the above steps 10 more times, once for each subsequent fragment.
* (Step #8 or #9): A buffer copy is required to deliver the data to the application.
* Grand totals:
  + 11 block copies or page transfers (1 per fragment received)
  + 1 buffer copy (1 per recv() call)
  + 22 context switches (Other App -> NIC Driver -> Stack)

##### Protocol stack is driver library

Use case #1:

* (Step #3) A block copy or page transfer is required to transfer the data from the Application to the TCP/IP Stack.
* (Step #3) A context switch is required to transfer data from the Application to the TCP/IP Stack.
* (Step #7) A buffer copy is required if the NIC hardware does not support gather DMA.
* (Step #9) If blocking, a context switch back to the Application is required after the first send() call.
* (Steps #9 through #12) For each of the Application’s 10 subsequent send requests, a context-switch to (and from, if blocking) the TCP/IP stack is required.
* (Steps #9 through #12) For each of the Application’s 10 subsequent send requests, a block copy or page transfer is required to send the data to the TCP/IP stack.
* (Step #13) A context switch to the NIC driver occurs when the NIC generates an interrupt signaling the arrival of the acknowledgement.
* Grand totals:
  + 11 block copies or page transfers (1 per send() call)
  + 1 buffer copy, but only if the NIC does not support gather DMA
  + 23 context switches (App -> Stack -> App x 11, Other App -> NIC driver), 12 if the protocol between the Stack and the NIC driver is non-blocking (i.e. – the first transition is App -> Stack, but not back again). (1 per send() call, 2 if blocking, plus 1 per ack received.)

Use case #2:

* (Step #4): A gather copy or page transfer is required to send a snapshot of the Application’s data to the TCP/IP stack.
* (Step #4): A context switch to the TCP/IP stack is required. Another one back is required in the blocking case.
* Grand totals:
  + 1 gather copy or page transfer (1 per send() call)
  + 2 context switches (App -> Stack -> App), 1 if the protocol between the Stack and the NIC driver is non-blocking (i.e. – the first transition is Stack -> NIC driver, but not back again). (1 per send() call, 2 if blocking.)

Use case #3:

* (Step #2): A context switch to the NIC driver occurs when the packet arrives.
* (Step #7 or #8): A block copy or page transfer is required to transfer the received packet to the application.
* (Step #7 or #8): A context switch to the application is required to deliver the data (and one from the application in the non-blocking case).
* Grand totals:
  + 1 block copy or page transfer (1 per recv() call)
  + 3 context switches (Other App -> NIC Driver, App -> NIC Driver -> App), 2 in the blocking case (Other App -> NIC Driver -> App)

Use case #4:

* (Step #2): A context switch to the NIC driver occurs when the packet arrives.
* (Step #6): Repeat the above step 10 more times, once for each subsequent fragment.
* (Step #8 or #9): A block copy or page transfer is required to deliver the data to the application.
* (Step #8 or #9): A context switch is required to deliver the data to the application (and another one in the non-blocking case).
* Grand totals:
  + 1 block copy or page transfer (1 per recv() call)
  + 13 context switches (Other App -> NIC Driver x 10, App -> Stack -> App), 12 in the blocking case (Other App -> NIC Driver x 10, Stack -> App)

### Conclusions

The current answers to the current stack of questions are as follows:

1. Zero-copy is pretty important. Not necessarily because of large data transfers, but because data transfers add up. Also, zero-copy plus asynchronous messages seems like an elegant way to keep message queuing simple while avoiding context switches. Note that zero-copy synchronous rendezvous can be implemented as an optimization of the copying case if the message buffers are page-aligned.
2. Precursor should allow for the aforementioned page-transfer facility, even if it doesn’t implement it. A gather version would also be helpful. Control over the locking of these pages and proper handling of concurrent access is vital to the success of this scheme.
3. As much of the messaging system as possible should be in the microkernel in order to avoid excessive context switching <FIXME> – Not sure how paging to disk would work just yet…).
4. Allocation of physical address space to device drivers will be handled directly via system calls, just like in Singularity.
   1. <FIXME> Side question – how do we detect those memory-mapped I/O regions without using the BIOS?
5. <FIXME> How will the microkernel VMM be implemented?
6. <FIXME> How will it be initialized?
7. <FIXME> What does it need from the current PMM design?
8. <FIXME> From the BootLoaderInfo?
9. <FIXME> What should I do next in the implementation of BootLoaderInfo and PMM initialization?

<FIXME>

## Virtual Memory Manager Requirements

The in-kernel virtual memory manager must support the page mapping operations described above for user-level address spaces. It must also manage the kernel’s address space – both per-process and global portions of it.

As with all parts of the Precursor microkernel, one of the most important goals of the VMM is to be portable to other hardware architectures. MMUs on various CPU architectures work quite differently, but can be broadly classified into two categories:

* Hardware-managed TLBs (x86, x64, Alpha, PowerPC)
* Software-managed TLBs (MIPS, Sparc)

The VMM will call into a portable interface (probably just function prototypes linked differently for different platforms) that will encapsulate this major difference between MMUs.

Here are some typical tasks that an OS must perform when dealing with a hardware-managed TLB:

* Allocating page tables and adding new page directory entries for them.
* Adding new valid page table entries.
* Handling in-kernel page faults for lazily-initialized shared kernel page tables. For page directory faults, the handler copies in global page directory entries for the new page tables. There shouldn’t be page table faults, since all kernel page tables for shared regions are shared.
* Marking page table entries as not-present.
* Reclaiming page tables by marking page directory entries as not-present, sometimes multiple copies across processors (e.g. – for shared kernel page tables pointed to by distinct page directories).
* Invalidating TLB entries whenever page table entries are modified (sometimes across processors, e.g. – when a page or page table is marked as not-present).

Here are some typical tasks that an OS must perform when dealing with a software-managed TLB:

* Allocating space for whatever mapping structures are required (software TLBs, inverted page tables, etc.).
* Adding new valid page table entries to the tables.
* Handling in-kernel page faults for lazily-initialized shared kernel pages. The handler will consult whatever structures are necessary to get the TLB entry for the new shared kernel page.
* Marking pages as not-present in the mapping structures.
* Reclaiming mapping structures? Not sure how this would work.
* Invalidating TLB entries whenever mapping structures are modified (sometimes across processors, e.g. – when a page is marked not-present).
* Handling TLB miss faults.

## IPC, Take Two

The user-space VMM scheme described above has proven to be impractical, in particular due to concurrency issues around the “dirty” bit in each PTE. I have decided instead to go with an NT-style in-kernel VMM. However, I still want paging I/O to be handled outside the microkernel, so the next step is to design a kernel-pager interface. This requires taking a closer look at IPC.

The networking use cases above seem to suggest that asynchronous is better than synchronous. However, there are too many practical problems with asynchronous that Singularity solves better than I ever could. For example:

* Using Brendan’s asynchronous page-transfer IPC would minimize the size of channel queues, but it incurs the cost of a TLB shoot down on each message pass – bad!
* There is no elegant way to prevent one thread from spamming another thread’s endpoint.

For these reasons, we will go with synchronous copying, with page copy-on-write as an optimization. The NX bit will be used on CPUs where it is available to avoid buffer overrun attacks (the kernel will never copy more than it is told the receiver can hold, but the receiver could have a bug and overstate the size of its buffer). There will also be tiny asynchronous “events” which can be sent through the same channels as regular messages.

One thing that I do want to preserve is the “endpoint-oriented” nature of Singularity IPC. This is the basis for a capability-based security model, and solves all sorts of “spamming” problems. To summarize:

* Two threads (same or different processes) communicate through a channel, which is composed of two endpoints.
* Each thread owns one or both endpoints of a given channel, although if it owns both it cannot use it.
* The primitives for message-passing on an endpoint are send, receive, and reply, just like in QNX.
* The primitives for event-passing on an endpoint are sendevent and receive (the same receive as for messages).
* There is a special primitive that can “listen” on several endpoints at once, notifying the caller when at least one of them has a message or event pending.
* send() can only send a message on the channel if the other end is not send-blocked or reply-blocked (to avoid deadlock). It is impossible to send() a message if the other end is replying, since that would imply that the would-be sender is currently reply-blocked (a contradiction). The sender is blocked until the other thread calls receive() (in the case of multi-receive, the receiver must also choose this channel). Note that send() is still allowed to proceed even if there are events pending, as long as all the other preconditions hold.
* receive() will fail if the other end is already receive-blocked or reply-blocked (to avoid deadlock). It will return immediately with the message if the other end is already send-blocked. It will return immediately with the highest-priority event if one is pending. If a message and one or more events are pending, whichever has the highest priority will be received. If the thread sending a message has the same priority as the highest-priority pending event, the message will be received first. If either a message or event is received, the receiving thread will inherit the priority of the sending thread. It will block if the other end is not using the channel currently and there are no events pending.
* reply() will fail if the other end is not reply-blocked. It is non-blocking and unblocks the other end with an ack message. Note that reply() is still allowed to proceed even if there are events pending, as long as all the other preconditions hold. The priority of the replying thread reverts back to normal.
* sendevent() will fail if there isn’t enough room on the target endpoint’s queue, which should be a rare occurrence, especially if the queues can grow dynamically. Note that events with the same priority and payload value can be compressed together in the queue using a counter. The caller explicitly specifies the priority of the event, in contrast to send(), which uses the priority of the sending thread.
* listen() will fail under the same conditions as receive() if any of the channels are not in the correct state. It blocks if no messages or events are pending. It will return an indication of which channel has the highest-priority pending message or event as soon as any message or event is available. If there is a tie between an event and a message within an endpoint or between endpoints, the same rules apply as for receive(). If there is a message-message or event-event tie between endpoints, the chosen endpoint is non-deterministic from the caller’s point of view.
* All calls will fail if they are applied to an endpoint that is not owned by the calling thread, or if the calling thread owns both endpoints.
* There is a special system call to grant ownership of an endpoint from the calling thread to another thread in the same process.

Messages that can be sent over a channel via send() or reply() (that require different system calls):

* Untyped contiguous buffers (optimization for page-aligned transfers).
* Endpoints themselves (this is the only way to transfer ownership of an endpoint to a thread in another process).
* Scatter-gather messages that are combinations of the above types.

## VMM, Take Two

Here is the second attempt at designing Precursor’s VMM.

### Why No Memory-Mapped Files?

I want to aim for a sealed-process architecture, which means in part that two processes should only be able to communicate with each other through channels. If two processes communicate through file I/O, they’re still using channels. However, if memory-mapped files are allowed, one process might see changes that another process is making to the file in a somewhat non-deterministic way. This could be avoided if the file system ensures that files opened for memory-mapped I/O are locked. However, this means that every file system will have to be trusted. I have decided not to go this route as it is too risky to have such a large trusted base.

### Why Paging At All?

There is an argument to be made that if one of the goals is to reduce the trusted base as much as possible, why bother implementing a trusted paging server? 64-bit machines are going to be the norm soon. Why not just insist that app developers stop treating memory as an infinite resource and tell admins to buy more RAM when needed? I actually think there is some merit to this argument, but in the end it doesn’t matter, because I want to learn how to implement paging just to satisfy my own curiosity. Perhaps my next OS will omit paging altogether, as Singularity does.

### Pager Protocol

This is the protocol between the microkernel and an external pager.

#### Goals

* “Do no harm” – a hard page fault caused by one thread should not adversely affect other threads that are either running or will be running soon after the fault is triggered.
* Prioritized paging – the kernel should inform the pager of the relative priority of in-page requests. Out-page requests don’t really have relative priorities. They are all either important (in low-memory situations) or not (if there is other disk I/O traffic). This is no longer true when there are in-page collisions with out-page I/O, however. In that case the out-page operation inherits the priority of the faulting thread.
* Scalability – the protocol should allow scaling to a large number of CPUs.
* Latency – actually not that important, since I/O time dominates page fault handling.

Of the above goals, scalability is the sketchiest. It is perhaps most closely related to “do no harm” in the sense that page fault processing should be processor-local whenever possible and access to shared spinlocks or data structures should be avoided.

#### Design

There are two main circumstances in which this protocol will be used:

1. For in-page I/O to handle a hard page fault.
2. For out-page I/O when the kernel wishes to evict some dirty pages from memory.

Case 1 requests will be generated by the kernel. When a page fault occurs, the kernel will check state bits in the PTE. If the page is committed and not in transition (i.e. – on the standby or modified lists), it is a hard fault (case 1). The kernel will send the remaining PTE bits to the external pager. It will receive the page back in the reply, filled with the appropriate code or data, along with the PTE bits again for confirmation. There are a few reasons why the pager should allocate the memory instead of the kernel:

* The pager can pre-allocate buffers for this purpose, resulting in fewer allocation operations.
* It saves an expensive TLB shootdown that would otherwise be required when unmapping the allocated page from the kernel’s address space to send it to the pager to be filled.

Of course, there is at least one TLB shootdown to remove the page from the pager’s address space, but this is avoidable if there is one single-threaded pager per CPU (FIXME – investigate this option further). The pager itself might incur extra shootdowns in sending the page to and from a disk or file system driver, but this is not really the kernel’s problem (again, Singularity solves this problem far better than I ever could).

Case 1 raises some questions:

1. How does the protocol handle collided page faults?
2. In what context does the kernel block waiting for the in-page I/O?
3. How does the kernel get the right ID to put in the PTE in the first place?
4. How does the pager deal with incoming requests from any number of kernel threads?
5. How does the kernel thread get an endpoint from the pager in the first place?
6. How does the kernel avoid infinite loops or deadlock if the pager itself causes a page fault?
7. How does the protocol handle page fault collisions with out-page I/O?
8. How does the kernel avoid infinite loops or deadlock if the pager itself causes out-page I/O?
9. How does the kernel inform the pager that it must increase the priority of an in-progress request due to collision?

Answer 1: The kernel changes the state of the PTE to “in-paging” and points it at a block of control information (a queue of threads waiting for the in-page I/O, among other things). If another faulting thread sees the “in-paging” state, it will queue itself waiting for that page, change its state to blocked, and call the scheduler. The thread that receives the reply from the pager will finish the page fault processing. The threads queued on the control information will simply ensure that the operation was successful and then either resume the user-mode thread or signal an in-page I/O error.

Answer 2: The kernel sends its in-page request to the pager as an “event”, which is a very small (probably 2 machine words) asynchronous message. Since it is a non-blocking operation, but the faulting thread has nothing else useful to do, it will block itself on the PF control block (i.e. – it will add itself to a wait queue and invoke the scheduler). When the pager wants to complete an in-page request, it makes a special “free”-like system call to the kernel, passing it the filled-in page and the disk address. The kernel uses the disk address in an associative lookup to find the PF control block. In other words, the start of an in-page request occurs in the kernel context of the faulting thread, and the completion of an in-page request occurs in the kernel context of a pager thread.

Answer 3: See case 2 (out-page requests).

Answer 4: There is no need for the pager to respond immediately to any kernel in-page request, since the affected CPU would have already been scheduled away to some other thread (or the pager, if it is available). Also, I/O is a greater source of latency than the inability of the pager to respond immediately. A good policy would be for the pager to have a listener thread per CPU, and some number of worker threads greater than the number of CPUs. Each listener thread would be responsible for waiting for PF events for its CPU and dispatching requests to the worker threads. For the sake of scalability, the worker threads should probably be divided into groups based on CPU affinity (i.e. – a single group of worker threads assigned to a CPU should handle all paging requests issues by threads on that same CPU). This is just a suggestion and cannot be enforced by the kernel.

Answer 5:

1. The first process the kernel creates is the Process Manager. It alone has the ability to create other processes, including “critical” processes (i.e. – ones that cannot use pageable memory), non-critical processes (i.e. – normal processes that can use pageable memory), and the pager process itself.
2. The pager process alone has the ability to make certain system calls, including a call to register a “pager service” endpoint with each CPU.
3. The kernel puts each received “pager service” endpoint in CPU-local storage. No non-critical processes can be created until such an endpoint is registered for each CPU. Keeping the endpoints CPU-local prevents unnecessary thrashing of other CPUs and caches.
4. When a thread causes a hard page fault, it first sets up the PTE and control block in an atomic manner. It initializes a count in the control block to indicate that a thread has already faulted, so that subsequent collisions are detected and queued immediately. It then transfers the ownership of the pager service endpoint to itself (threads running in kernel space can do this) and sends a request event to the pager with the disk address from the PTE.
5. Once the kernel thread has sent the event, it blocks itself on the wait handle in the control block.
6. Eventually, a worker thread in the pager invokes the aforementioned special “free-like” system call, fulfilling the original page fault request.

Answer 6: The Process Manager is responsible for creating the pager, as well as any auxiliary processes that it might use (e.g. – disk driver, file system driver, etc.). The Process Manager uses a special flag in the “create process” system call that indicates that each new process is to be a “critical process” – that is, they cannot cause hard page faults and they cannot allocate virtual memory lazily or without requesting that it be locked in physical memory. The former case causes system failure; the latter case causes an error to be returned from the system call. This implies that loading the executable code for these critical processes must be done all at once rather than using demand-paging.

Answer 7: See case 2 (out-page requests).

Answer 8: See case 2 (out-page requests).

Answer 9: Each colliding thread first checks the control block against its priority. If its thread priority is higher, it sets the control block’s priority to match and sends an event to the pager service endpoint, in much the same manner as a faulting thread would. The event is set to the thread’s priority and includes a code indicating that this is a “priority boost” request, and it also includes the disk address for the block being read or written. It is up to the pager to find the in-progress request for that disk address and boost its priority accordingly. This can be done by having an associative lookup from disk address to worker thread. Once the pager service thread finds the appropriate worker thread, it can boost its priority to match its own priority (since it would have inherited the priority of the request event). This same mechanism can be used to set the priority of the worker thread in the first place.

Case 2 requests are generated in order to reduce the size of the modified list. These requests are periodically issued by kernel, which uses “events” (small, asynchronous messages normally used to signal an interrupt) to notify the pager that its modified page list is getting too big (the kernel can use the same pager service endpoints discussed for case 1). The reason to use asynchronous events is to prevent a critical system thread (the kernel’s hypothetical “modified page writer” thread, which would call the pager) from getting blocked indefinitely. It is quite simple for the pager, upon receiving an out-page event, to send a batch request for out-pages to the kernel, and to send such written-out pages back – it simply uses system calls. The in-kernel processing is done in the context of the pager thread that makes the system call. The system calls are somewhat specialized and look a lot like VM alloc/free operations, but they can only be called by the pager. The first two calls are highly specialized and are used to inform the kernel of new disk blocks that have been allocated to pages that have never been written out before (see Answer 3). The third call is an “alloc-like” operation that receives pages and a list of disk addresses from the kernel. The kernel will have constructed a PF control block for each disk address that it will use to track the pending operations. Each PTE pointing to a page about to be removed from the modified list will be updated to point at its corresponding control block. The fourth call is a “free-like” operation that sends the exact same information back to the kernel. Upon receiving the pages and corresponding disk addresses, the kernel uses the same associative lookup that it uses for in-page responses to find the corresponding control block for each page. From here, it can finish processing, including resolving any collisions that may have occurred. Note that the pager technically does not need to return the same physical pages back to the kernel that it originally received, as long as they have the same contents. However, it would be foolish to do so since it implies an unnecessary copy somewhere.

Answer 3: When a process first allocates a page, the kernel increases the commit charge by a certain amount, but does not ask the pager to allocate swap space for the new pages immediately. (Note that the pager must keep the kernel informed about the maximum allowable commit charge so that the kernel knows when to fail memory allocation requests.) The VADDRs for the new pages are initialized as usual, and in the typical case, no physical memory is allocated until the first page fault occurs. One way or another, the kernel ends up allocating a physical page and creating a PTE to map that page. The kernel sets up the PFDB entry for the allocated page with a blank “original PTE contents” (aside from protection bits). The first time the kernel decides to evict the page when it’s dirty, it changes the “PFN of PTE” field (if there is one) to point to the pager’s page tables, transfers the page to the pager’s working set, and waits for it to be returned eventually. Before this waiting period, it sends an event to the pager service endpoint, informing it that it needs new disk blocks. In response, the pager makes a special system call to query for a given number of such requests. The kernel returns a list of address space IDs and virtual addresses (for optimization purposes). The pager makes a second system call, returning a list of the same size containing the ASIDs and vaddrs, along with their allocated disk addresses. The kernel then puts each disk address in the “original PTE contents” field for the next time the page is evicted altogether.

Answer 7: Note that there is a difference between a “soft-fault” on a page on the kernel’s modified list, and a “hard-fault” colliding with a page that belongs to the pager and currently has an out-paging I/O in progress. The former case is dealt with by the kernel in the obvious way, while the latter case must be dealt with differently. It could be handled by either the kernel or the pager, but it is almost certainly much simpler for the kernel to handle it. When the kernel sends a page to the pager to be written out to disk, it removes it from the modified list and puts it in the pager’s working set, then changes the state of the PTE to “out-paging”, and points it at a block of control information. This control information includes a queue of threads waiting for the out-page I/O, among other things. If another faulting thread sees the “out-paging” state, it will queue itself waiting for that page, change its state to blocked, and call the scheduler. When the kernel receives the page back from the pager via the special “free”-like system call, it will finish the page fault processing. The threads queued on the control information will simply ensure that the operation was successful and then either resume the user-mode thread or signal an out-page I/O error. The page itself will be put back in the original process’ working set in this case. It would normally go back on the standby list.

Answer 8: The same mechanism that protects the pager against recursive page faults also works for out-page I/O. If every allocation request from the pager must return physical memory immediately or fail, then it cannot block waiting for pages to be evicted from memory.

#### Note to Self

Allocation in general should fail if there are no free, zero, or standby pages. User processes have the option of waiting for the modified page writer to do its thing and trying again later. The kernel has no choice – it must fail (and probably hard) if this ever happens.

#### Protocol Summary

**Pager Service Events:**

* Event( Priority, PageFault, BlockAddress )
* Event( Priority, PriorityBoost, BlockAddress )
* Event( Priority, AllocBlocks, 0 ) <FIXME – How does the kernel know not to keep spamming the pager with these?>
* Event( Priority, WriteBlocks, 0 ) <FIXME – ibid>

**Pager System Calls:**

* RegisterPagerServiceEndpoint( Endpoint )
* CompletePageFault( BlockAddress, vaddrToFree )
* GetBlockAllocRequests( maxNumRequests, BlockAllocRequest[] outBuffer )
  + BlockAllocRequest = {ASID, vaddr}
* TryToAllocBlocks( numResponses, BlockAllocResponse[] inOutBuffer )
  + BlockAllocResponse = {BlockAllocRequest, BlockAddress, bool succeeded}
* TakeOutPageIORequests( maxNumRequests, startVaddrToAlloc, BlockAddress[] outBuffer )
* CompleteOutPageIO( numRequests, startVaddrToFree, BlockAddress[] inBuffer )

**Event( Priority, PageFault, BlockAddress )**

**Sent by the kernel to the pager service channel when a page fault occurs.**

* **Priority – the priority of the thread that faulted**
* **PageFault – indicator that this is a page fault event**
* **BlockAddress – address of the starting block to read; also acts as an identifier for the in-page request for the kernel.**

**Event( Priority, PriorityBoost, BlockAddress )**

**Sent by the kernel to the pager service channel when there is a collided page fault on either an in-page or out-page operation, and the faulting thread has a higher priority than the request’s current priority.**

* **Priority – the priority of the thread that faulted**
* **PriorityBoost – indicator that this is a priority boost event**
* **BlockAddress – address of the starting block currently being read or written; also acts as an identifier for the in-page or out-page operation for the kernel.**

**Event( Priority, AllocBlocks, 0 )**

**Sent by the kernel to the pager service channel when the “modified new” list gets too large. This is how the kernel informs the pager that it should allocate some disk blocks in bulk.**

* **Priority – the priority of the request to allocate blocks**
* **AllocBlocks – indicator that this is an alloc blocks event**

**Event( Priority, WriteBlocks, 0 )**

**Sent by the kernel to the pager service channel when the “modified” list gets too large. This is how the kernel informs the pager that it should perform some out-page I/O.**

* **Priority – the priority of the request to write out dirty pages**
* **WriteBlocks – indicator that this is a write blocks event**

**RegisterPagerServiceEndpoint( Endpoint )**

**Registers the given endpoint as the pager-service endpoint for the calling CPU. Can only be called by the pager.**

**CompletePageFault( BlockAddress, vaddrToFree )**

**Informs the kernel that a page fault request has been completed. Can only be called by the pager.**

* **BlockAddress – the starting block that was read. This acts as an identifier for the original page fault request sent out by the kernel.**
* **vaddrToFree – virtual address in the calling process of the page read from disk. It will be unmapped from the pager’s address space and mapped into the faulting address space.**

**GetBlockAllocRequests( maxNumRequests, BlockAllocRequest[] outBuffer )**

**BlockAllocRequest = {ASID, vaddr}**

**Fetches requests from the kernel to allocate blocks for dirty pages that have never been written out before. Can only be called by the pager.**

* **maxNumRequests – the number of BlockAllocRequests that can fit in the target buffer.**
* **outBuffer – virtual address of buffer in pager’s address space that will receive up to maxNumRequests BlockAllocRequests structures.**

**Each BlockAllocRequest contains an address space ID and virtual address of the page for which to allocate a block. In addition to identifying the allocation request (needed for TryToAllocBlocks later), this virtual address location information acts as locality hints to the pager to help it allocate disk blocks close together for pages that are close together in the same virtual address space. Although it is possible for the pager to use this information to implement memory-mapped files, this is not recommended since it violates the sealed process architecture.**

**TryToAllocBlocks( numResponses, BlockAllocResponse[] inOutBuffer )**

**BlockAllocResponse = {BlockAllocRequest, BlockAddress, bool succeeded}**

**Informs the kernel of blocks that have been allocated for dirty pages that have never been written out before. Can only be called by the pager.**

* **numResponses – the number of BlockAllocResponse structures contained in inOutBuffer.**
* **inOutBuffer – virtual address of buffer in pager’s address space that holds BlockAllocResponse structures to send to and be updated by the kernel.**

**Each BlockAllocResponse contains a BlockAllocRequest indicating which page the new block address is allocated to, the BlockAddress itself, and a flag to be set by the kernel indicating whether or not it accepts the allocation. If, between the time the pager obtains a BlockAllocRequest and the time it sends back a BlockAllocResponse, the page moved off of the modified new list for some reason, then the kernel rejects the allocation and the pager should free the allocated block address. Pages might move off the modified new list if another pager thread allocates blocks to them first, or if they are soft-faulted back into their working sets. This makes the protocol more complicated, but keeps processing stateless from the kernel’s point of view.**

**TakeOutPageIORequests( maxNumRequests, startVaddrToAlloc, BlockAddress[] outBuffer )**

**Fetches requests from the kernel to write out dirty pages. Can only be called by the pager.**

* **maxNumRequests – the number of BlockAddresses that can fit in outBuffer and the maximum number of pages that can be mapped into the caller’s address space contiguously starting at startVaddrToAlloc.**
* **startVaddrToAlloc – page-aligned virtual address in the caller’s address space to which dirty pages will be mapped.**
* **outBuffer – buffer into which the kernel copies up to maxNumRequests block addresses. Each page mapped to the pager corresponds to the block address at the same relative offset in outBuffer. For example, the page at startVaddrToAlloc + PAGE\_SIZE \* 5 corresponds to the block address in outBuffer[5]. Each BlockAddress indicates to where the dirty page should be written, and also acts as an identifier for the out-page request for the kernel.**

**The kernel forms out-page requests by removing entries from the modified list. That is why the name of this function begins with “take”.**

**CompleteOutPageIO( numRequests, startVaddrToFree, BlockAddress[] inBuffer )**

**Indicates to the kernel that a batch of out-page requests have been completed. Can only be called by the pager.**

* **numRequests – the number of BlockAddresses in inBuffer and the number of pages mapped at startVaddrToFree that are no longer dirty and should be returned to their rightful owners.**
* **startVaddrToFree – page-aligned virtual address in the caller’s address space at which newly cleaned pages are mapped.**
* **inBuffer – contains BlockAddresses corresponding to the pages mapped at startVaddrToFree. These indicate to the kernel which pages go with which out-page requests.**

The kernel will unmap the pages from startVaddrToFree and either put them on the standby list, or map them into working sets if collided page faults occurred during out-page I/O.

#### General VMM System Calls

<FIXME: Create lifecycle state machines for the following.>

Note to self: Create Process flow

* Parent sends request message and endpoint to Process Manager
* Process Manager creates a new process, which includes its own new address space and initial thread. The thread start-up code is determined by the kernel. The system call includes a parameter allowing the Process Manager to send an endpoint to the new process.
* The new process sends a message back to the Process Manager requesting its executable code.
* The Process Manager uses the appropriate file system to read in the executable code (or grabs its cached copy) and sends it to the new process (COW optimization will help a lot here) <FIXME: Is there any way to support demand paging of executable code?>, along with the endpoint sent by the parent process in the original create process request. The child process sends a wakeup message to its parent to notify it that it has started.
* The Process Manager replies to the parent process, which then receives on its endpoint. It continues execution once it receives the wakeup message from its child.

**CreateProcess( endpoint, isCritical, isDriver )**

Can be called only by the Process Manager.

* Endpoint through which executable code will be sent
* Flag indicating whether the new process is a critical process (i.e. – cannot cause paging activity).
* Flag indicating whether the new process is a driver process and should be allowed to hook interrupts and allocate specific regions of physical memory.

**PageControl( vaddr, npages, &protFlags, &isLocked )**

This can be used to query or modify the protection of the given virtual address region. This can only be used to query user pages, not kernel pages, so the only protection information is

readable/writable/executable. Can also query/modify the locked state of a page.

**Alloc( vaddr, size, allocFlag, protFlags, isLocked )**

This is the standard allocation system call, where the kernel chooses the physical pages (using colouring, if supported). The physical allocation doesn’t happen right away by default.

* vaddr – target virtual address; must be page-aligned. Can be null, in which case the kernel decides where to map the allocated region.
* size – size of region to allocate; rounded up to the page granularity.
* allocFlag – mutually exclusive options – reserve, commit, nolazy (i.e. – commit and allocate physical storage immediately).
* protFlags – page protection (read/write/execute)
* isLocked – if true, allocFlag must be “nolazy”. Required to be true when a critical process makes this system call.

**DeviceAlloc( vaddr, paddr, size, allocFlags, protFlags )**

Special allocation call intended for use by device drivers (cannot be called by regular processes). Physical pages are allocated immediately (equivalent to Alloc’s “nolazy” option) and locked in memory. How the pages are obtained depends on the paddr parameter:

* If paddr is NULL, the kernel allocates physical pages (using colouring for performance reasons, if the kernel supports it and if contiguity is not requested). The call fails if not enough pages are available. Note that if contiguity is not requested, then allocFlags are ignored and this call is equivalent to Alloc( vaddr, size, nolazy, protFlags, true ).
* If paddr is not NULL…
  + …and corresponds to a region with at least one allocated physical page, the call fails.
  + …and corresponds to a large enough range of free physical pages, the kernel allocates those pages (contiguity is assumed).
  + …and does not correspond to physical memory…
    - …and corresponds to a kernel-reserved region of the physical address space, the call fails.
    - …and corresponds to a free region of the physical address space, the kernel maps the region. It does not “allocate” anything since it does not track ownership of such regions.

vaddr can be NULL, in which case the kernel decides where to map the allocated region.

The size parameter is rounded up to the page granularity.

The allocFlags parameter has several options that can be combined (note that all are ignored unless paddr is NULL):

* Contiguous – physical pages allocated must be contiguous. Ignored and assumed to be 1 if paddr is non-NULL.
* Below16MB – physical pages allocated must be below 16MB. “Contiguous” is implied by this option. Mutually exclusive with Below4GB. If this option and Below4GB are both zero, pages above 16MB are preferred, and on 64-bit systems, pages above 4GB are preferred.
* Below4GB – physical pages allocated must be below 4GB. “Contiguous” is implied by this option. Mutually exclusive with Below16MB. If this option and Below16MB are both zero, pages above 16MB are preferred, and on 64-bit systems, pages above 4GB are preferred.
* NoCross64KB – physical region allocated must not cross a 64KB boundary. “size” parameter must be less than 64KB or the call fails.

The protFlags parameter controls the protection of the mapped pages.

**Free( vaddr, size, freeFlag )**

Frees the given region to some degree.

* vaddr – base virtual address of the target region. If freeFlag is “Release”, must be the same address returned by Alloc() or DeviceAlloc().
* size – size of the region to free in pages, or zero to free the entire region. Must be zero if “Release” is specified for freeFlag.
* freeFlag – mutually exclusive options:
  + Decommit – frees the physical page if mapped (unlocking it if necessary), decreases the system commit charge, but leaves the region of virtual address space reserved.
  + Release – Implies Decommit, and furthermore frees the region of virtual address space.