QMS3204#

Proc and Kernel Free List Optimizations#

Introduction#

We have observed that fragmentation of the proc and kernel free lists can seriously degrade performance of any operation that requires kernel or proc resources. The purpose of this feature is to improve the performance of the system under such circumstances.

Memory management in the kernel and proc is done through functions in ker/nano_alloc.c. In here you will find implementations of _smalloc(), _sfree(), _srealloc() and many similar functions. All of these API functions are built on top of _sreallocfunc(), which maintains the free memory lists and allocates more memory as necessary. When the kernel or proc need to allocate memory, they invoke _smalloc() or one of the related functions, which in turn invokes _sreallocfunc(), which scans the free lists for a free block of memory that is large enough to satisfy the request. If no such free block exists, _sreallocfunc() allocates more memory from the system (using mmap()) to satisfy the request. When the kernel or proc are done with memory they free it using _sfree or one of the related functions, which in turn invokes _sreallocfunc(). When memory is freed, _sreallocfunc() places it on a free list -- memory is never returned to the system (never munmap()'d) once it has been allocated for the system.

The existing implementation uses three free lists, one for each of the kernel, proc and critical allocations (the critical free list is a last-gasp mechanism for allocations that must be succeed or the system will fail -- normally such things come from the kernel free list, but if the system is out of memory and the kernel free list is empty, a critical allocation might come from the critical free list which consists of 16K of memory preallocated at system initialization time).

Each free list is a singly-linked list of free memory blocks, sorted by address. Since this list is sorted by address, memory blocks can be coalesced when side-by-side blocks are freed. However, this sorted singly-linked list leads to performance issues when the list grows too long.

On allocation, we need to scan the list to find a block that is large enough to fulfill the allocation request. On memory deallocation, we need to scan the list to find the location where the new free block will be inserted. The cost of these scan operations increases linearly with the size of the free list.

Unfortunately, our system can lead to fragmentation, which in turn can lead to long free lists and thus poor performance. There are two approaches to improving performance: improve performance with long free lists, and modify system behaviour to prevent the long lists in the first place. Ideally both will be implemented (you might argue that reducing the length of the lists is adequate, but that presumes that you can guarantee short free lists in all systems. It is best to attempt to guarantee short free lists, but implement a more efficient algorithm to handle the situations where long free lists result regardless). This feature only addresses the first: it improves the performance of the system when a free list grows to be very long.

Data Structures#

The fundamental data structures in this design are the *free entry* and the *freelist*. A *free entry* is a piece of memory that may be used to satisfy a memory allocation. A *freelist* is a collection of free entries.

A *free entry* (or *free entry node*) is a data structure that describes a piece of free memory and which has necessary pointers for linkage in the freelist. A free entry occupies the memory that it represents. For example, if a free entry Q describes a piece of memory that starts at virtual address vaddr, then &Q==vaddr. In addition to the pointers for linkage into a freelist, a free entry structure has a size field. In terms of the C language and free entry is defined as a *struct*. However, the C language requires that all fields of the *struct* be defined, but

a free entry might not be big enough to include all linkage pointers. This works because the freelist linkage data structures require fewer pointers for smaller free entries. So, while the free entry has a concrete *struct* definition, the code which references it must be careful not to reference all fields of the *struct* if the free entry isn't large enough to contain them.

A *freelist* is a collection of free entries, structured to allow the collection to be searched and manipulated. A freelist is implemented as a set of skip-lists. A skip-list is an internal list with a characteristic size. Each skip-list list holds the elements of the freelist that are equal-to or larger than the characteristic size of the skip-list. Note we use 'size of the list' to refer to the skip-list's characteristic size, as opposed to 'length of the list' which refers to the number of elements on it.

Each free entry on the freelist can be on multiple internal lists, with larger free entries being on more lists. There will be a skip-list with a size of 0, so that all free entries will be on at least this skip-list. Other skip-lists have characteristic sizes chosen to optimize efficiency of searching the freelist. The original free list implementation was effectively a skip-list with a single size-0 skip-list.

Note that free entries on all skip-lists are sorted by the address of the free entries. We still want to coalesce side-by-side free blocks.

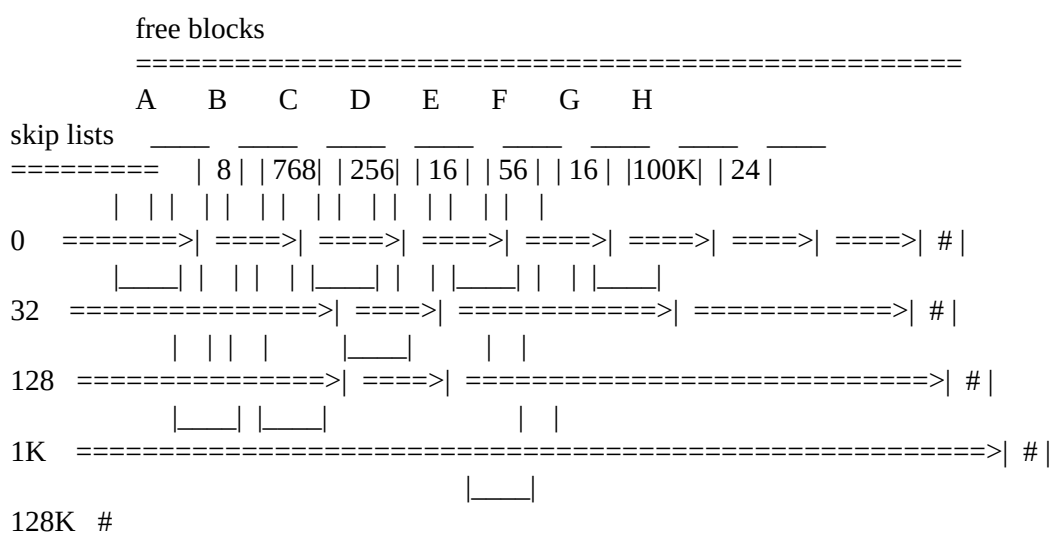
Each free entry contains a size field indicating the size of the free block of memory, along with one or more skip-list pointers (the number of skip-list pointers will be determined by the size of the free memory block). Since each free entry contains a 4-byte size and at least one 4-byte pointer, the minimum size of a free block is 8 bytes.

One question that must be addressed is why we have chosen to go with singly-linked lists for our skip-lists. We could get better performance using a doubly-linked list, or by using a different data structure entirely. However, any other data structure would require more than a single pointer for linkage (or we'd give up the ability to sort by address which means we can't coalesce adjacent free blocks) which in turn would mean the alignment and minimum block size would have to be bumped up to 16 bytes. This would, on average, mean every allocation would require 8 more bytes.

That might be a good trade-off (performance vs. memory) for some customers, but for the moment we've decided to go with a smaller memory footprint. Note that we can revisit this decision in the future. Perhaps a better solution would be to allow different data structures with different trade-offs to be implemented in different modules, and allow the customer to choose.

Example:#

In this example, the freelist has 5 skip-lists, with characteristic sizes of 0, 32, 128, 1K, and 128K. There are 8 free blocks at positions A through H (sorted by address order) with sizes of 8, 768, 256, 16, 56, 16, 100K and 24 bytes respectively. (note: '#' denotes a NULL pointer)



This data structure allows us to find a location in freelist much more quickly on average than we can a single singly-linked list, since we can skip over large portions of the freelist using the higher order skip-lists. While a single singly-linked list requires O(n) operations to find a location in a list length of N, a skip-list requires O(log(N)) operations.

Freelist Configuration#

A freelist configuration consists of the number and size of the skip-lists that implement the freelist. Different usage patterns will result in different ideal skip-list sizes. Through experimentation we have arrived at the following configurations for proc, kernel and critical freelists:

The proc freelist will have 7 skip-lists, of sizes 0, 16, 32, 64, 128, 256 and 1024 bytes. The proc heap is mostly used for pathmgr objects, which have sizes from around 30 to around 130 bytes, and the names of these objects (which are typically fairly short, say under 32 bytes). Some pathmgr objects have the names included in the object. The result is a fairly even distribution of object sizes from a few bytes (a name that's a few bytes long) to somewhat under 256 bytes (a large pathmgr object with a large embedded name). These sizes are chosen to distribute the object allocations across the different skip-lists. For some usage patterns, an additional skip-list with a size between 64 and 128 bytes might be useful.

The kernel freelist will have 6 skip-lists, of sizes 0, 16, 32, 64, 256 and 1024 bytes. The kernel heap is mostly used for objects on the soul lists. Once the souls on a soul list are exhausted, these objects are allocated individually. The kernel skip-list sizes are chosen to distribute the different object sizes onto different skip-lists. The 16-byte skip-list will get sync and pulse objects (16 and 24 bytes respectively). The 32-byte skip-list will get channel objects (32 bytes). The 64-bytes skip-list will get timer and connection objects (each 64 bytes long). The 256-byte skip-list will get process and thread objects, which are quite large.

The critical freelist will have a single zero-sized skip-list. The critical heap is never very large, so multiple skip-lists have limited return.

Code Structure#

The existing _sreallocfunc() routine is complex and convoluted, but once understood it can be seen to be a fair marvel of efficiency, not wasting any effort. Despite the complexity, the efficiency is important enough that the existing structure of _sreallocfunc() will be maintained. The manipulation of the freelist data structure will be extracted from the main-line _sreallocfunc() code into separate functions.

The _sreallocfunc() code can be paraphrased as:

```
_sreallocfunc(old_memory, old_size, new_size)

    new_memory = NULL

    if (old_memory)
        if (new_size) && (new_size < old_size)
            // we're deallocating the tail of an allocated piece of
            // memory.  Turn it into a normal dealloc...
            old_memory=old_memory+(old_size-new_size)
            old_size=old_size-new_size
            new_size=0
        find location of old_memory in freelist, and the next node
                    in the list beyond old_memory

    if (new_size)
        // i.e. we're allocating some memory

        if old_memory && next_node && old_memory+old_size==next_node
            // we're reallocating, and the memory immediately after
```

```
            // old_memory is free so we might be able to use it to
            // save reallocating the whole block.
            if (next_node.size == new_size-old_size)
               // it's exactly the right size
               remove next_node from freelist
               return old_memory
            if (next_node.size > new_size-old_size)
               remove new_size-old_size bytes from the beginning of
                  next_node and return the rest to the freelist
               return old_memory

         // we need to allocate new memory
         find a freelist entry of size >= new_size bytes

         if new_memory.size = new_size
            // block we found is exactly right.  Use it.
            remove new_memory from freelist

         else if new_memory.size > new_size
            // block we found is too big.  Pare off what we need
            remove new_size bytes from the beginning of new_memory
                        and return the rest to the freelist

         else
            // freelist doesn't have enough -- get more
            new_memory = allocate more memory from the system
            // This is how it works for proc freelist.  We'll ignore
            // differences with kernel & crit freelists for this discussion.
            add new_memory to the freelist
            go back and start over -- there will be enough  memory
                     on the freelist this time.

         if (old_memory)
            // it's a realloc
            copy old_size bytes from old_memory to new_memory

      if (old_memory)
         // we have some memory we don't need any more
         add old_size bytes at old_memory to the freelist

      return new_memory
```

This code structure will be maintained, but while the existing code implements the various list manipulation operations in-line, the new code will split them out into new functions. The new code will also be much better commented to explain this structure.

Freelist Operations#

There are four main operations that we need to perform on a freelist: *find*, *add*, *remove* and *splinter*.

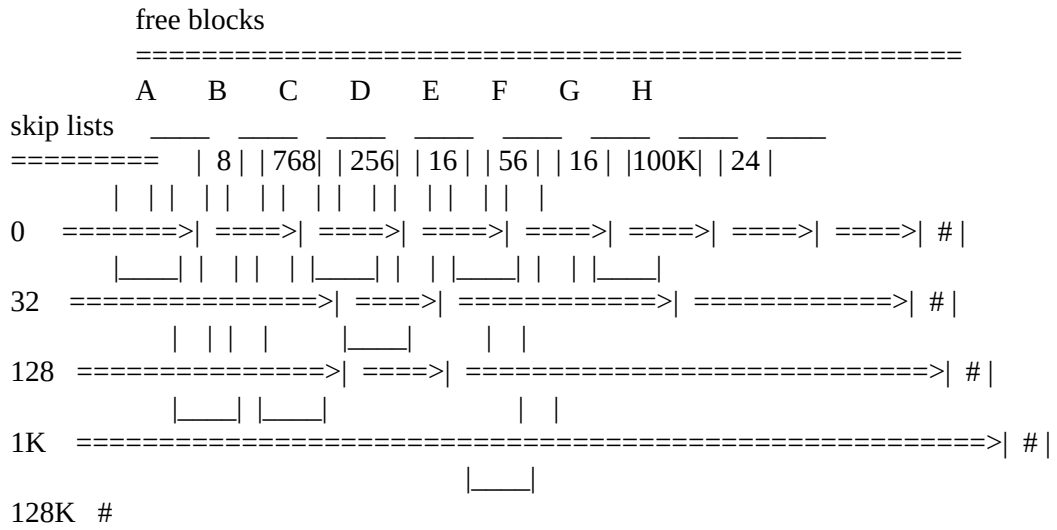
The *find* operation searches a freelist for a given address value, which might or might not actually be present in the freelist.

Searching the list is done progressively from lists with larger sizes to lists with smaller sizes. That is, we scan the largest skip-list until we've gone as far as we can, then move to the next smaller skip-list and continue the scan. This is repeated for all skip-lists, thus skipping over as many free entries (by using the larger skip-lists) as possible.

This algorithm works well (is efficient) as long as there aren't large runs of free blocks that all fit on the same skip-list -- in that event, performance degenerates to the same behaviour as a single skip-list.

The *find* operation returns a *location*. A *location* in a freelist is represented not as a pointer to any single free entry node, but rather to the set of free entry nodes before the address value in each skip list (recall that skip-lists are sorted by address value). This is important, because the singly-linked nature of the skip-lists prevents us from going backwards from a given node. If we need to manipulate a node, we usually need to find the previous nodes on the different skip-lists so that we can update their links. Thus after a *find* operation we are left with a list of free entry nodes, one per skip-list.

Referring to the previous example:



find(E) would return a *location* comprised of the list of nodes immediately before the address of node E on each skiplist. Note that node E itself is only on two skiplists -- this is irrelevant because *find* only considers the address of E, not its size. Since the freelist uses 5 skiplists, the result of *find*(E) will be a list of 5 nodes: { D, C, C, <head>, <head> }. On skip-list 0, D is the last node before the location of E. On skiplists 1 and 2, C is the last node before the location of E. On skiplists 3 and 4, there is no previous node so we use <head> to denote that the position of E is (or would be, if it were big enough) at the head of the skiplist. A *find* operation on any address value greater than D but less than E will return the same.

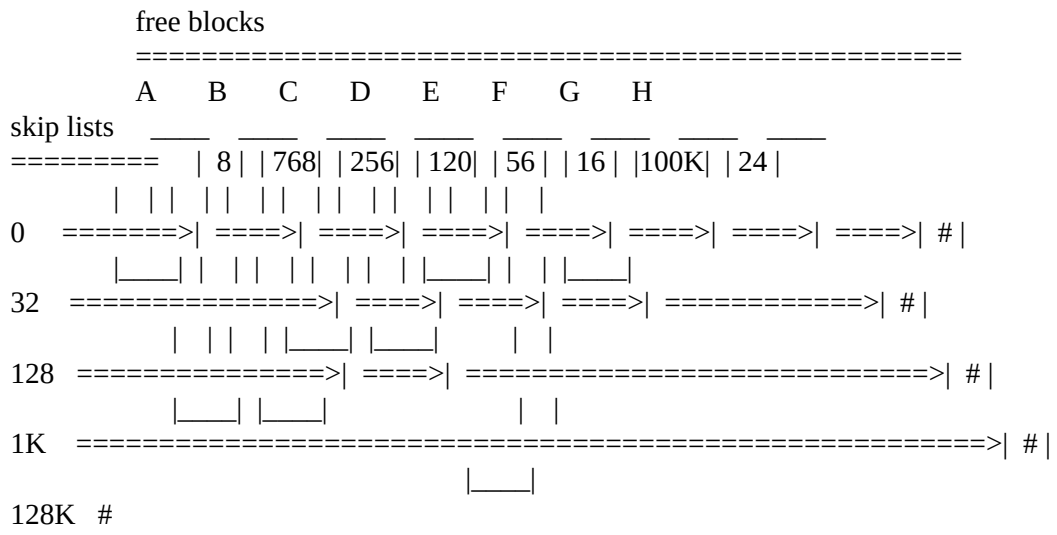
A *find* on an address value that points into a free entry node on the free list but not at the beginning of that node gives undefined results.

The *add* operation adds a new free entry node to a freelist.

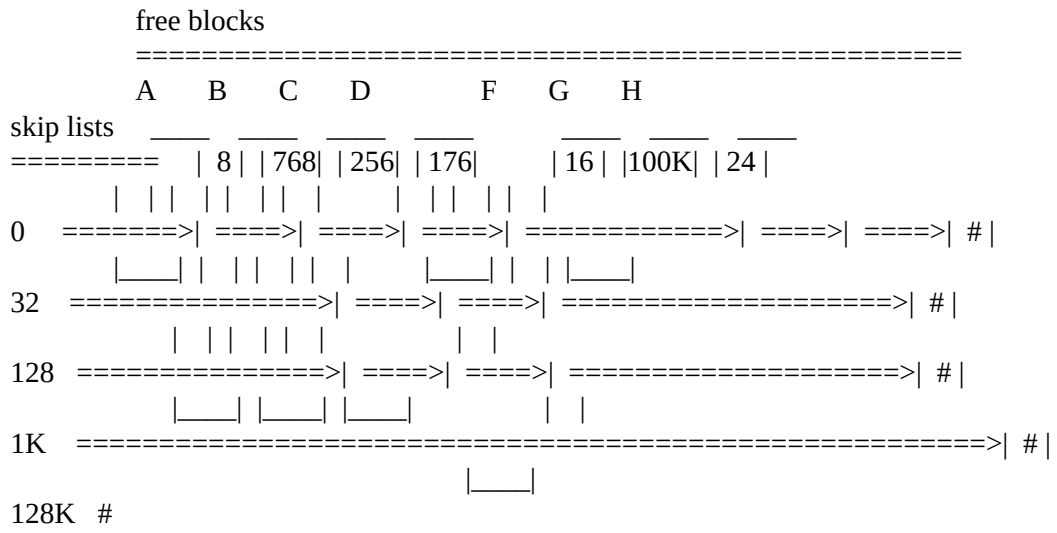
- The *find* operation is used to locate the place in the freelist where the new node will be placed.
- If the new node comes immediately after an existing node so that the two may be coalesced, then the existing node has its size field updated and the previous link pointers are adjusted if necessary (it may be necessary if the coalesce results in the previous node being placed on skip-lists that it didn't occupy before, in which case it must be added to them). Otherwise the new node is simply linked in.
- If the new node comes immediately before the next node in the freelist, then these two nodes may be merged: the new node's size is updated to include the next node, and the previous links are updated as necessary.

Referring to the example above, consider an *add*(Q), where Q is 104 byte free entry that fits between D and E and may be coalesced with both (thus Q=D+16 and E=Q+104 and the resulting coalesced block will be 16+104+56 bytes long). This *add* operation will go through the following steps:

- *find*(Q) returns a location set of { D, C, C, <head>, <head> }
- since Q==D+D.size, we need to coalesce D and Q. D.size = D.size + Q.size. D.size is now 120 so D needs to be added to skip-list 1. Thus D.next[1]=C.next[1] and C.next[1]=D:

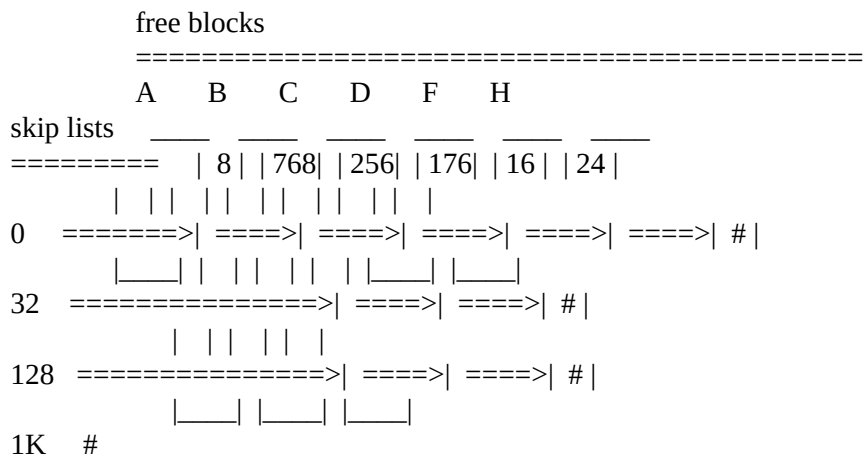


- since Q+Q.size==E, we need to coalesce Q and E (actually we need to coalesce D and E since we already coalesced D and Q). Thus D.size = D.size + E.size, D.next[0]=E.next[0], and D.next[1]=E.next[1]. D.size is now 176, so needs to be added to skip-list 2. Thus D.next[2] = C.next[2] and C.next[2] = D:



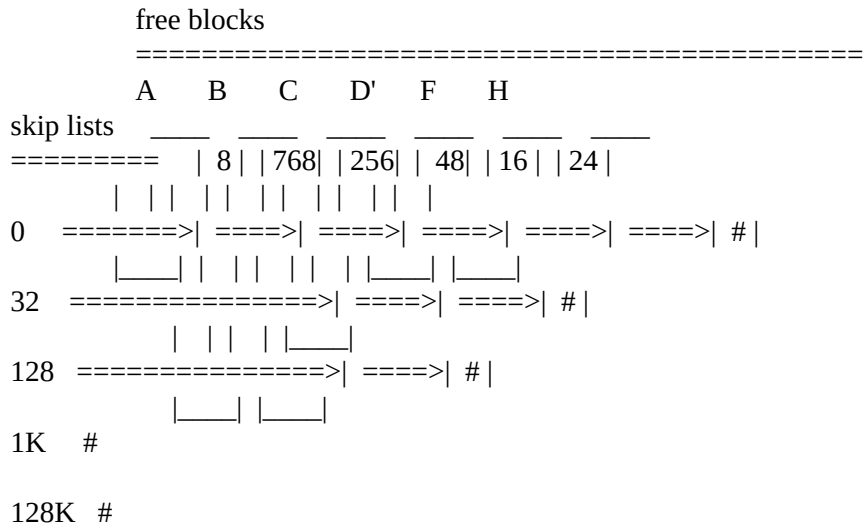
The *remove* operation removes a free entry node from a freelist. The *find* operation is used to locate the node in the freelist, and the next pointers of the previous nodes (located by *find*) are updated to point to the next node beyond the node being removed. We only need to update the skip-lists that the node being removed was on.

Using the previous examples, if we *remove*(G), we *find*(G) giving us { F, D, D, <head>, <head> }. To update the skip-lists, we assign F.next[0]=G.next[0], D.next[1]=G.next[1], D.next[2]=G.next[2], and <head>.next[3]=G.next[3]. Note that we don't update skip-list 4 links since G was not a member of skip-list 4.



The "splinter" operation removes a piece of a free entry node from a freelist. A free entry node is always splintered so that the start of the node is removed from the freelist and the tail of the node is left on the freelist. The *splinter* operation uses the *find* operation to locate the node in the freelist. The next pointers of the previous pointers (located by *find*) and the next pointers of the new splinter are updated to remove the old node from the skiplists and replace it with the new splinter. Care must be taken as the new splinter might be on fewer skiplists than the old node, so some of the larger skiplists might need to be adjusted to point around the new splinter.

Again following the previous example, if we splinter block D to remove 128 bytes, we're left with block D' of size 48. We perform a *find*(D) to give us { C, C, C, <head>, <head> }. We update C.next[0]=D', C.next[1]=D'. But D' is not on skip-list 2 where D was, so we need to update C.next[2]=D.next[2]:



Note that all operations are performed with a lock (the kernel is locked in the case of the kernel and critical freelists, and a mutex is held for the proc freelist) so we don't need to be concerned with mutual exclusion.

Simulations#

In order to ensure that the chosen skip list sizes were efficient, a simulation was run in the following manner:

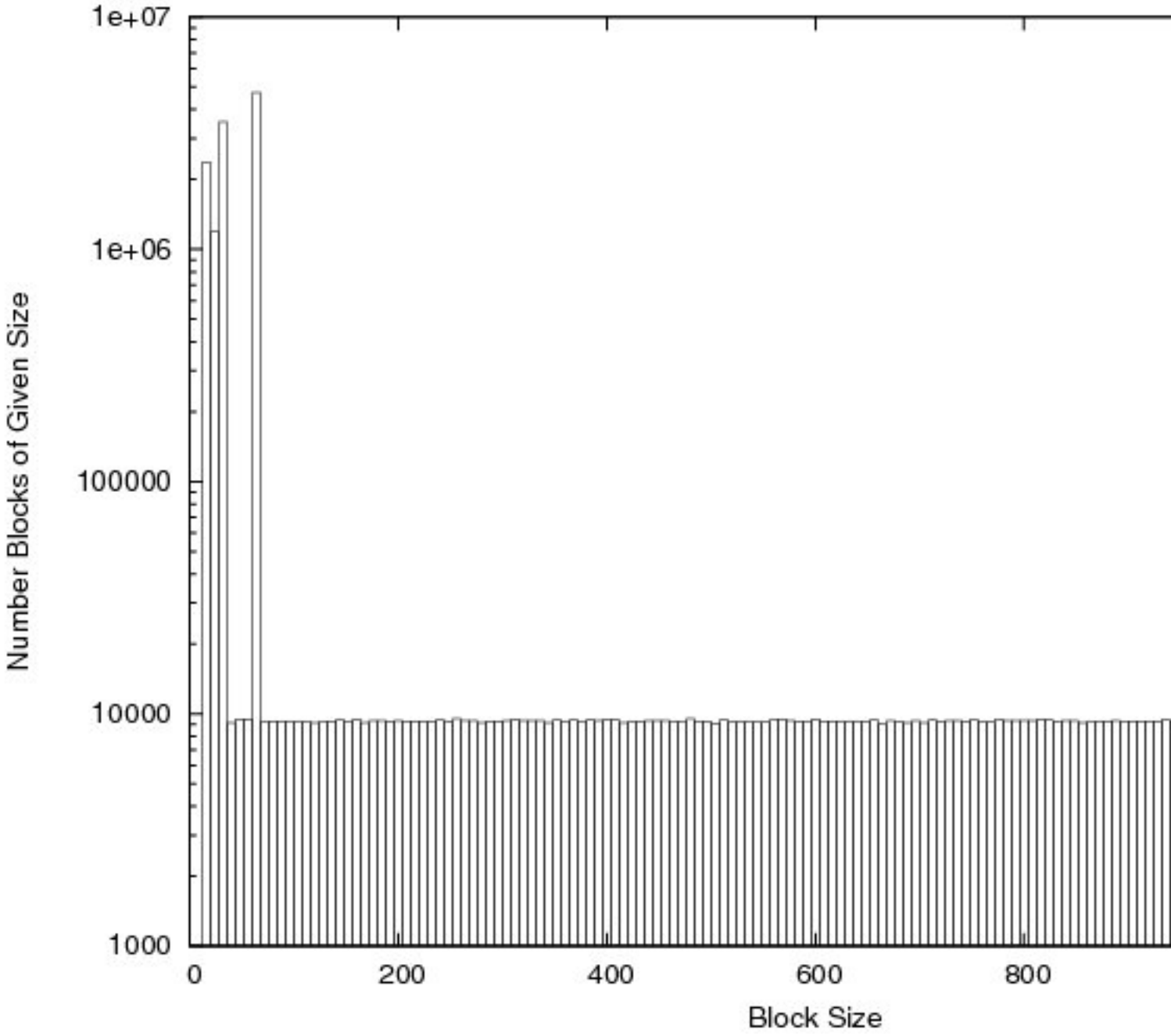
1. a fragmented free list was generated by "allocating" and "releasing" memory blocks a preset number of times. At each step the probability that the action was "allocate" was set to e^{\frac{-0.693147n}{N} } where N is the total number of actions simulated and n is the number of the action $(n \in {0, 1, 2, \dots, N-1})$. This meant that initially almost all actions were allocation (simulating the start-up of the system) and, by the end, allocations and releases were happening equally frequently. As memory was released, coalescence occurred when possible.
2. a set of `find()` targets was selected at random from the free areas. The number of such targets was the same as the number of free list fragments but the selection was made with replacement so some fragments were chosen several times and some not at all.
3. for each of the pre-initialised skip lists sizes based on the initial recommendations described above, for each of the `find()` targets, the mean number of skip list accesses required to reach the target was calculated.

The purpose of the simulations was to find a set of skip list sizes for the kernel and proc free lists that provided the least mean number of accesses.

Simulation Code#

The simulation was written in Python and made use of the simPy framework. The simulation program, analyseProcFreeList.py, and its supporting module statistics.py are held under source code control in the subversion repository.

Simulation Results#



The simulation was run 60 times for the kernel pattern of allocation described above. The figure above shews the sizes of the memory blocks allocated during the initialisation (note the logarithmic y axis). As can be seen, blocks of up to 1024 bytes were requested with blocks of 16 bytes (2,370,190 allocations), 24 bytes (1,190,908), 32 bytes (3,549,736) and 64 bytes (4,732,264) dominating. The number of fragments of the free list after initialisation varied across the simulations from 4181 to 4463.

Given the assumptions of the model, the simulations found that free list sizes of (0, 16, 32, 64, 128, 1024) were optimal in 18.3% of the simulations, (0, 16, 24, 32, 128, 1024) in 20\% of the simulations and (0, 24, 32, 64, 128, 1024) in the remaining 61.7%.

Design and Code Reviews#

The design of this feature was presented to the QNX OS team over the course of several meetings during which the various options were discussed and the design evolved. These meetings were held between July 21 2009 and July 30 2009. Participating in the design discussions and review were Brian Stecher, Colin Burgess, Attilla Danko, Steve Bergwerff, Neil Schellenberger, Peter Luscher, Scott Miller, Adrian Mardare and Alexander Koppel.

Final approval on the design was given by Brian Stecher and Colin Burgess with no action items.

Code review for this feature took place in the following discussion thread: http://community.qnx.com/sf/go/projects.core_os/discussion.osrev.topc8859.

This feature has no implications for the Safety Manual.