Memory Management in Nix mark IV

Francisco J. Ballesteros

ABSTRACT

Memory management in Nix has been reworked recently to make the system more efficient. This paper reports the changes made to the system, the rationale behind the changes, and the resulting memory manager.

Introduction

Nix mark II was a reimplementation starting from a cleaner research kernel derived from Plan 9, named 9k, in which development we cooperated. Its memory management was a reimplementation and required more work for the reasons we enumerate. Most notably, the system had a few limitations that had to be addressed, some of them inherited from Plan 9:

- Not all the configured physical memory could be used. The physical memory allocator placed an arbitrary limit needed to let the system initialize correctly.
- The memory was split into kernel and user memory. The sizes were determined by constants, and not by actual usage of the system. The same happens to Plan 9.
- Several caches for files (the mount driver cache and the file image cache) were doing the same thing using different implementations. This also applies to Plan 9.
- The read-ahead mechanism could not exploit the cache because it could only keep a prefix for each cached file.
- All pages in the system were of the same size, and the MMU relied on *malloc* to allocate page table pages. This also applies in part to Plan 9.
- The implementation of *malloc* in the kernel relied on a fixed pool, so its size had to be static. This also applies in part to Plan 9.
- We had more page faults than needed. Despite caching binaries, there were page faults for text and stack which, in many case, could be avoided. The same happens to Plan 9.
- The page allocator was slow, because everyone was placing pressure on a single allocator for everything. The same happens to Plan 9.

Because of this, Nix mark IV includes the following changes, that we elaborate later.

- A new page allocator supporting multiple page sizes. The implementation does not rely on *malloc* and thus can be used to implement it.
- A modified implementation of *malloc* that can ask for large pages to add more core to the kernel heap.
- A modified MMU implementation for the K10 capable of using (some of) the new page sizes.

- A new cache for files and images, that relies on paged segments.
- A modified mount driver (with respect to the one described in [1]) relying on the new cache for concurrent reads and reading ahead.
- A modified implementation of segments, including adapted *rfork* and *exec* implementations, to reduce the number of page faults.

In what follows we describe the changes and the data structures used.

Physical and virtual memory

Early during initialization, the Address Space Map (ASM, inherited from 9k) determines the memory installed and creates the kernel space map to access physical memory and to support the kernel virtual space.

The resulting (initial kernel) address space is shown in the figure. The addresses shown are for the K10, other architectures (none at the moment) may change this, being it in the machine dependent layer. The figure shows also important addresses for user address spaces.

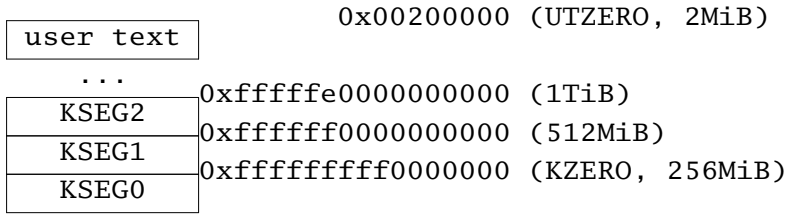

Figure 1 Important regions of the kernel address space.

The kernel uses high addresses, which have to be selected to permit the sign extension to build valid 64bit addresses (because the hardware does not really use 64 bits for addresses). The user text starts at *KTZERO* after enough space to leave an initial, perhaps 2M, page invalid.

KSEG0 maps the first few MiB of physical memory, usually 64MiB or less. This contains the kernel text and initial data. The rest of physical memory is mapped at KSEG2. The KSEG1 map is used to map the hardware page tables into themselves, so they can always be accessed without requiring extra kernel mappings (Such a nice implementation is Jim McKie's doing). The kernel mapping tries to use the largest pages supported by the HW when feasible, resorting to smaller pages for portions not aligned or not fully covering a single page. Usually, 1GiB, 2MiB and 4KiB page entries are used in the HW page tables implementing this.

All the memory, but for the first Mbyte (thanks to the HW design) and the kernel memory (up to the end of its BSS plus a preconfigured initial number of MiB) is given from the ASM to the page allocator as available for page allocation.

The physical memory pages are still called pages, and are represented by a *Page* structure. However, they are page frames. When allocated and given a virtual address, they become true (virtual memory) pages.

Instead of giving page frames a different structure, perhaps just an address, they are still handled with the conventional *Page* structure. The reason is that most of the book-keeping data needed for those allocating pages is usually the same, and thus it is easily kept in a single *Page* structure instead of requiring separate implementations to keep track of the pages at different places in the kernel.

In mark IV, pages can only exist within segments, and the only way to free a page is to free an entire segment. This is for most pages, but the kernel *malloc* and the MMU code both ask for individual pages; although they never release the pages they allocate, which keeps the initial sentence of this paragraph still true: for them the pages are page frames.

Assumptions and design guidelines

Nix does not have swapping (nor does it page out any page). Therefore, once a page is allocated, it remains in memory until set free.

There is another important assumption in nix: the system will not run out memory. That is, if it runs out of physical memory it is considered an error and the system panics. This, together with the removal of swapping, simplifies how memory is handled and can make the system faster. Those allocating memory must instead take care not to exhaust the system memory.

Counter measures like ''killbig'' are still in place but are likely to be replaced in the near future with something better.

An important design guideline is that caches tend to consume all unused memory. That is, free memory is to be used for the caches, until the system runs low on memory and reclaims cache space to release it to the free memory pool. For now, the implementation keeps hard-wired limits on space used at most for caching, but this restriction will be removed in favor of the design guideline.

In many cases, instead of allocating kernel data structures, using them, and then releasing their memory, the different subsystems keep little allocators of already existing but idle data structures. New structures needed are taken from them. This has two benefits: reduces the pressure on the kernel *malloc* implementation, and is potentially faster because there is no need to allocate and free them repeated times and also because part of their initialization can be performed once.

The idea is not new, and comes from the Slab allocator [2]. Unlike in the Slab, we simply keep the previously allocated structures cached in a free linked list. Thus, there is no extra complexity as it would be if we had introduced an Slab allocator.

Page allocation

Unlike in previous Nixes, page allocation is built directly on top of the machine dependent ASM. It does not rely on *malloc* or other artifacts and implements its own allocation for the structures used. We will discuss first how page allocation works and show the structures near the end of this section, once the discussion could make more clear how they work.

Initially, the memory banks noted by the ASM are split so that no bank contains memory in several NUMA proximity domains. That is, each bank is fully contained within a single domain. The kernel assigns each domain a color, and thus each bank has a single color (as each processor or *Mach* has; processes and segments are also assigned a preferred color).

The page allocator is actually a set of page allocators. Each one has a single page size and a single color. For each bank, one or more page allocators are built, to support allocation of page (frames) from it. Initially, each bank is given an allocator for the largest page size suitable for the bank. If the bank is not aligned to the largest pages contained in the bank, other allocators with smaller page sizes are created (for the unaligned heading, trailer, or for both when both exist).

The figure shows a set of initial, so called, top-level allocators. They are kept linked on global, per page size, lists. Thus, when a page of a given size is needed, a single list may be scanned to allocate it. In both the figure and the rest of the section we assume that the page sizes configured are 1GiB, 2MiB, 16KiB, and 4Kib, for clarity. This is just an example of one possible configuration.

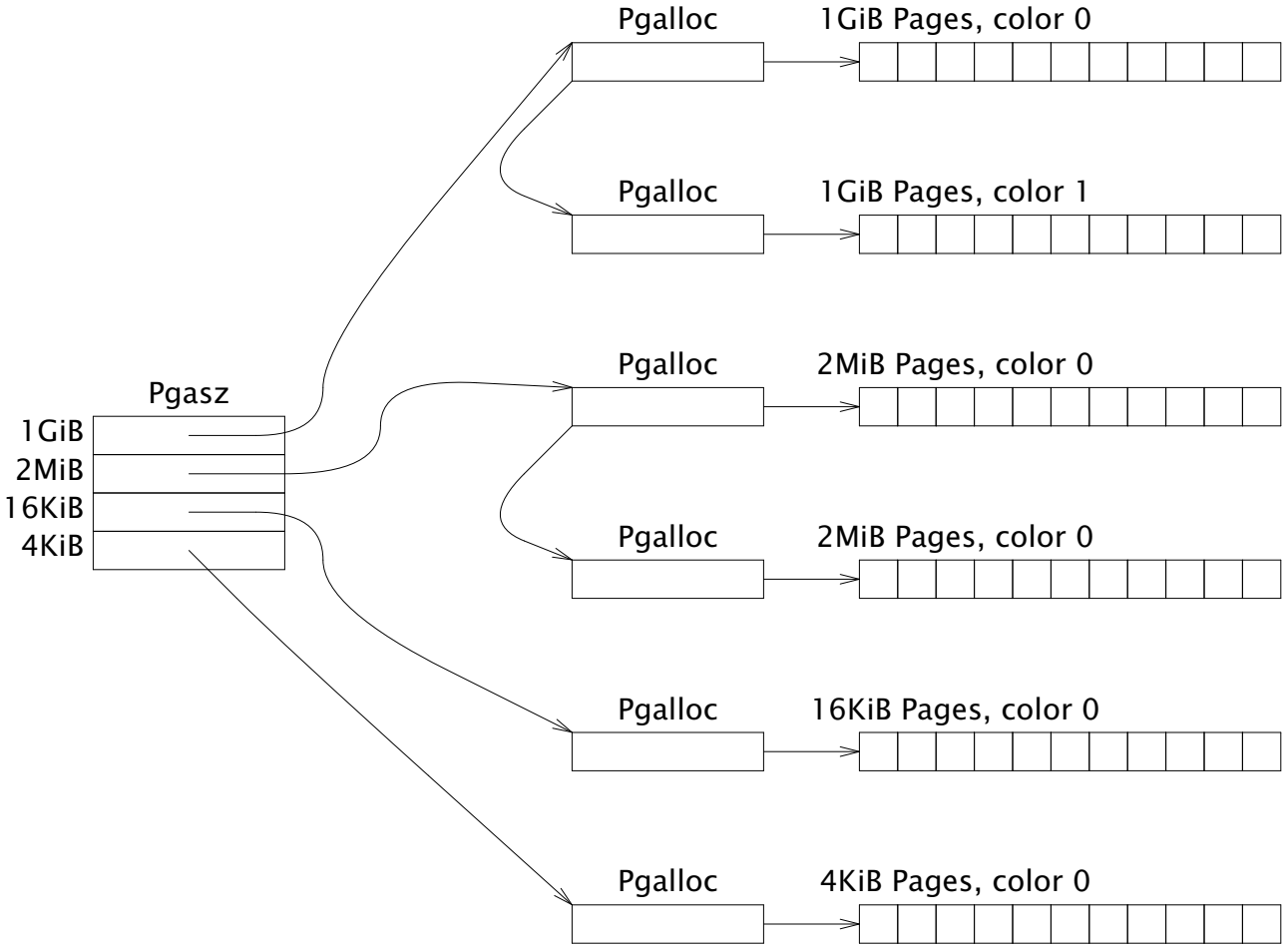

Figure 2 A set of top-level allocators is allocated to cover the memory reported by the ASM, using the largest page sizes for each region of memory depending on alignment and colors. Boxes represent structures, not memory. This example assumes that there are two banks with different colors and that the first bank was not aligned to 1GiB pages and required using also 2MiB, 16KiB, and 4KiB pages to cover its space. The 16KiB allocator is discussed later, ignore it by now.

Structures for top-level allocators are allocated very much like static data. The page allocator initialization takes physical memory to initialize the structures for all of them, including their arrays of *Page* structures. All these structures are kept on free lists so that we could allocate structures for the allocators and their pages without using *malloc* or static kernel data.

A page taken from any of the initial top-level allocators may be split at run time into smaller pages, by creating a new allocator for the smaller pages contained in the single larger page, and linking it in the list of allocators for the considered page size.

Later, when all the inner pages are released, they may be recombined back into the original larger page. This would remove the allocator for the smaller pages from the allocator list and release it.

Thus, the set of allocators is a hierarchy. The figure shows a set of initial top-level allocators with some other allocators resulting from page splits.

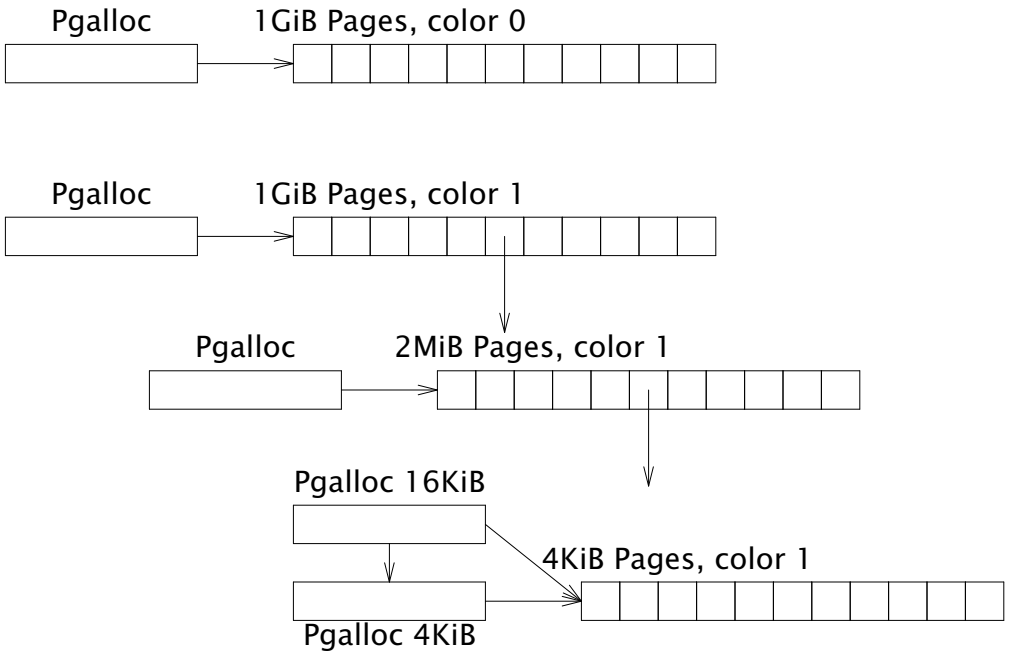

Figure 3 Pages from existing allocators are split into smaller pages handled by allocators built at run time. Here, one of the color-1 1GiB pages is split into 2MiB pages handled by a new allocator built at run time, and one of the 2MiB pages was split into an allocator for 16KiB pages, bundled from 4KiB pages.

Considering that 1GiB pages are too large and will be split almost for sure, all the structures for 2MiB page allocators resulting from splits are pre-allocated along with the top-level page allocator structures. They are then kept in the same free lists, and are taken from there when the splits actually happen. If a previously split 1GiB page is joined back when all the 2MiB pages within it are released, the structures are placed back on the free list to be reused later.

Structures for allocators resulting from the split of 2MiB pages are not pre-allocated. When a 2MiB page is split into 4KiB pages, the structure for the new 4KiB page allocator and its array of *Page* structures is allocated at the start of the memory in the 2MiB page being split. According to diagnostics of the current implementation, this implies a loss of 28 bytes per 2MiB page split due to internal fragmentation. This saves memory in structures for 2MiB pages that are used as such, without splitting into smaller pages. Thus, in the figure, the first tree allocators are pre-allocated. (the two top-level 1GiB page allocators, above, and the pre-allocated 2MiB page allocator, in the middle). The 4KiB page allocator shown at the bottom of the figure is allocated within the 2MiB page memory, leaving the rest of 4KiB pages free.

The page allocator (and the new MMU code) can handle page sizes not directly supported by the HW. This permits, for example, using 16KiB pages for user segments.

Allocation of sizes like 16KiB, close to the HW supported 4KiB, is problematic because of the implications for the smaller page size (4KiB in this case). The 16KiB allocator structures are embedded into the memory of the larger page (2MiB) being split, as explained before. However, the 4KiB page allocator for split 16KiB pages cannot be embedded into the 16KiB page (or we would waste 1/4 of the resulting pages for the structures), and it cannot be preallocated (or we would waste memory for larger pages not split).

To handle this case, a third type of allocator has been introduced. A bundle allocator allocates pages that are bundles of smaller pages. In this case, the pages belong to the allocator with the small size (4KiB) and may be bundled into larger pages (16KiB) by allocating several of them (4) with the right alignment constraints. Note that this is the opposite of splitting a page into smaller pages, although the net effect is that pages can still split and join.

A bundle (16KiB) allocator is usually embedded within a larger page (2MiB) being split and is allocated side by side with another embedded allocator for the smaller (4KiB) pages. There is a single *Page* array, owned by the smaller page allocator. However, at initialization time (when the 2MiB page is split), the bundle steals *Page* structures from the underlying (4KiB) page allocator to take as many bundles as possible, due to alignment constraints.

When stealing, only the first *Page* structure in each bundle is placed in the list of *Page* structures in the bundle allocator. Such page has the start address for the bundle and represents the larger page. Splitting a bundle is handled by placing such structure (and those for the other pages in the bundle, also stolen but not linked) back into the underlying allocator.

Joining is handled by detecting that a page free for the small size makes all the pages in a bundle available, and then stealing them and linking the first one as available in the bundle allocator. Like at initialization time.

The first structure is an array of *Pgasz* structures, each describing a set of allocator for a configured page size. We show now their definition and the configured initialization in our kernel.

```
/* Pgasz.type */
enum{PGprealloc, PGembed, PGbundle};

/*
 * One per configured page allocator page size.
 */
struct Pgasz
{
    usize    pgsz;        /* page size */
    uchar    pgszlg2;     /* log2 pgsz */
    uchar    atype;       /* allocation type */
    Pgalloc *pga;         /* allocator list */
    Pgalloc *last;        /* last in allocator list */
};
```

```
/*
 * Configured page sizes: 1G, 2M, 16k, 4k
 * 1G and 2M are preallocated for existing memory.
 * 16k are bundles of adjacent 4k pages embedded within 2M pages.
 *
 * - Keep sorted from larger to smaller sizes.
 * - If an entry is not PGprealloc, following ones can't be PGprealloc.
 * - The last entry can't be a bundle.
 * - The next to a bundle can't be a bundle.
 */
static Pgasz  pgasz[ ] = {
    { .pgszlg2 = 30, .atype = PGprealloc },
    { .pgszlg2 = 21, .atype = PGprealloc },
    { .pgszlg2 = 14, .atype = PGbundle },
    { .pgszlg2 = 12, .atype = PGembed },
};
```

The allocators linked on each *pgasz* allocate pages of that size, and are handled by a *Pgalloc* structure:

```
/*
 * One per page allocator.
 * There are one or more top-level allocators plus inner
 * allocators when pages are split into smaller pages.
 */
struct Pgalloc
{
    Page*    parent;       /* parent page or nil for top-level */
    Pgalloc*next;          /* in list of allocators for this pgasz */
    Pgalloc*prev;          /* in list of allocators for this pgasz */
    uintmem  start;        /* physical address in memory for page 0 */
    usize    npg;          /* number of pages */
    usize    nfree;        /* number of free pages */
    usize    nuser;        /* number of pages used by user segments */
    usize    nbundled;     /* number of pages stolen by bundles */
    usize    nsplit;       /* number of pages stolen by splits */
    uchar    szi;          /* index in pgasz array (page size) */
    uchar    color;        /* memory locality */
    Page *pg0;             /* first page in page array */
    Page *free;            /* unused pages */
    Pgalloc *bpga;         /* bundle allocator built upon us, if any */
};
```

Each page (frame or virtual) is represented by a *Page* structure:

```
struct Page
{
    Ref;
    QLock;
    uintptr  pa;           /* physical address in memory */
    uintptr  va;           /* virtual address for user pages */
    ushort   n;            /* paged in flag, mmu index */
    uchar    pgszlg2;      /* log2 pgasz[ .pga->szi].pgsz */
    uchar    bundlei;      /* page nb in page bundle [0..bundlesz-1] */
    Page *next;            /* used by mmu and pgalloc */
    Page *prev;            /* used by mmu and pgalloc */
    Pgalloc  *pga;         /* allocator this pg comes from */
};
```

The pages do not imply references on the segments. Instead, each reference from a

segment to a page adds one *ref* to that page. Pages kept in the free list have zero references counted. Processes waiting for page-ins in progress synchronize using the *QLock* on the page of interest. This permits locking a segment, looking at a page, and synchronizing on the page without holding the segment lock. Therefore, there is no need to release a segment lock and try again after the page-in has completed. Besides, one page is paged-in once, and not possibly multiple times as in the old code.

To aid in coalescing of free small pages (eg., 4KiB) into larger bundles (eg., 16KiB), *bundlei* indicates the position of the page into the bundle, counting from zero. Because the structures are allocated in an array, it is easy to locate peers in a bundle for coallescing. *Page* structures (and others) are double-linked to make it fast to release a given page, in part because of bundles.

This is the main loop within the routine allocating a page:

```
/* Find the smallest page already available */
for(i = npgasz-1; i >= 0; i--){
     if(pgasz[i].pgsz < sz)
          continue;
     ilock(&pgalk);
     for(pga = pgasz[i].pga; pga != nil; pga = pga->next){
          if(color >= 0 && pga->color != color)
               continue;
          pg = pganewpg(pga);
          if(pg != nil){
               /* Move pga to head if not yet there */
               if(pga != pgasz[i].pga){
                    unlinkpga(pga);
                    linkpga(pga, 1);
               }
               iunlock(&pgalk);
               goto found;
          }

     }
     iunlock(&pgalk);
}
DBG("newpg: no pages: sz %uld0, sz);
return nil;
```

The allocator is moved to the head of the list if it has free pages. Thus, most of the full allocators will not be visited while searching for pages. The rest of the routine takes care of splitting pages when the size found is not the one requested (to split the page as many times as we have to):

```
found:
/*
 * If the page is larger than we asked for, split it into
 * smaller pages until we get the desired size.
 */
DBG("newpg %#P pgsz %#ulx for %#ulx0, pg->pa, pgasz[i].pgsz, sz);
for(; pgasz[i].pgsz > sz && i < npgasz-1; i++){
     pgsz = pgasz[i].pgsz;
     if(pgasz[i].atype == PGbundle){
          pg = splitbundle(pg);
          continue;
     }
```

```
        DBG("newpg split %#p0, pg->pa);
        ppga = pga;
        switch(pgasz[i+1].atype){
        case PGprealloc:
             pga = newpga(pg->pa, pg->pa + pgsz, pga->color, i+1, pg);
             break;
        case PGbundle:
        case PGembed:
             pga = newepga(pg->pa, pg->pa + pgsz, pga->color, i+1, pg);
             break;
        default:
             pga = nil;
             panic("newpg: atype");
        }
        ilock(&pgalk);
        pg = pganewpg(pga);
        iunlock(&pgalk);
    }
    return pg;
```

Allocation is first attempted looking at the color asked, and then perhaps retried looking at any color. Thus, locality is honored but not enforced.

The kernel *malloc* asks for 2MiB pages when its initial pool of memory fills. It asks for one page at a time, when needing more core. Such page is never deallocated and its memory is used as an extra pool for the kernel heap. In a similar way, the MMU code asks for 4KiB pages used as Page Table pages. They are never released, but are kept cached when not needed, for later use.

The memory not given to the page allocator and occupied by the kernel is accounted as kernel memory. Pages for *malloc* and the MMU are also accounted as kernel memory. All other pages allocated are accounted as user memory (but for those used by the caches discussed later). Thus, the set of memory devoted for the kernel and the user is adjusted depending on actual system usage.

Virtual memory

Virtual memory handling is close to that of the previous implementations, but has important changes.

First, a process keeps an array of Segments, but there is no BSS segment. Instead, the data segment is sized to include the old BSS segment. The memory in the data segment that belongs to the BSS is cleared and not paged in from the executable file. This happen before for the last portion in the last page of the data segment (which holds BSS data). The code already had to check for the file length to decide which portion to read and which portion to clear.

The current implementation is exactly the same, only that because the data segment includes the BSS, there are pages in the data segment that are never read (not even in part) from the executable file. They are just cleared.

The structure for a *Segment* is similar to the standard one, but has been modified:

```
struct Segment
{
	Ref;
	QLock	lk;
	ushort	type;		/* segment type */
	uintptr	base;		/* virtual base */
	uintptr	top;		/* virtual top */
	usize	size;		/* size in pages */
	uchar	pgszlg2;	/* log2(size of pages in segment) */
	int	color;		/* memory locality */
	...
};
```

The segment now includes the size for the pages it uses, in the *pgszlg2* field. This size is supplied as a new parameter to *newseg*. It also includes the (preferred) color for the pages it uses.

Another important change is that *Pte* entries are configured to have the maximum segment size set to *SEGMAXSIZE*. Such value is reasonable for 32bit systems as well. The actual number of entries is derived from the maximum size, and not the other way around.

```
/*
 * virtual MMU, set to host up to SEGMAXSIZE bytes using
 * the smallest page size and PTEPERTAB entries per table.
 * The max segment size might be larger when using larger pages.
 * The max size is set fixed to be safe also on 32bit systems.
 */
enum
{
	PTEPERTAB = 256,
	SEGMAXSIZE = 0x7c000000u,
};
```

Segments, and their *Pte* entries are kept cached instead of free, for later use. Thus, two new fields in *Segment*, *first* and *last*, track the first and last entries used in the map. As a result, the map is always allocated and its changed only to grow the array when larger segments are allocated using recycled structures from smaller segments. Here *mapsize* is derived as indicated above.

```
struct Segment
{
	...
	uintptr	ptemapmem;	/* space mapped by one Pte in this segment */
	Pte	**map;
	Pte	**first;
	Pte	**last;
	int	mapsize;
	...
};
```

As segments are being used, new *Pte* entries are allocated and never deallocated. After some time of system usage, no more allocation for such structures is necessary.

Text and data segments are paged in from files (only text and initialized data). Once read, such data is kept cached. The old image cache has been reworked and is now a segment cache. These fields in the *Segment* structure are for implementing the

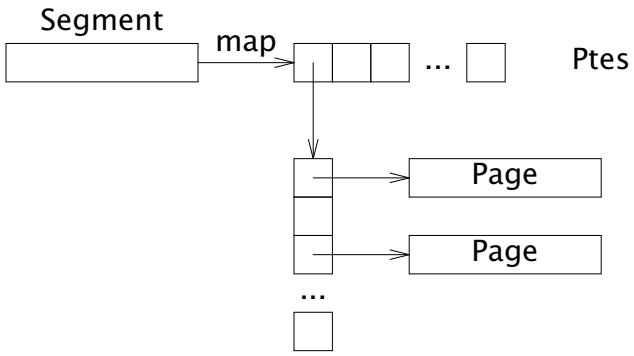

Figure 4 Segments keep a virtual MMU with pages, recording the range of entries used and never released, but recycled to allocate new segments.

segment cache, described later.

```
struct Segment
{
     ...
     ulong    fstart;          /* start address in file for demand load */
     ulong    flen;            /* length of segment in file */
     Chan *c;                  /* channel to text file */
     Segment  *src;            /* image the data comes from */
     Segment  *hash;           /* Qid hash chains */
     Segment  *lnext;          /* lru list */
     Segment  *lprev;          /* lru list */
     Segment  *anext;          /* alloc list */
     Segment  *fnext;          /* free list */
     int  used;                /* used since the last reclaim */
     ...
};
```

The mount driver cache is also using segments to cache file data. Its implementation relies on yet more fields in the *Segment* structure:

```
struct Segment
{
     ...
     Path *cpath;           /* debug only */
     Qid  cqid;
     Dev  *cdev;
     vlong    clength;
     vlong    nbytes;
     ...
};
```

Caches

The segment cache keeps a set of segments that include both text and data (each one). Each such segment represents the data in the file but already loaded into memory. These are called ‘‘source’’ segments and keep in the *c Chan* the reference to the file supplying the data. The cache is a hash table using *qids* and behaves very much like the old one.

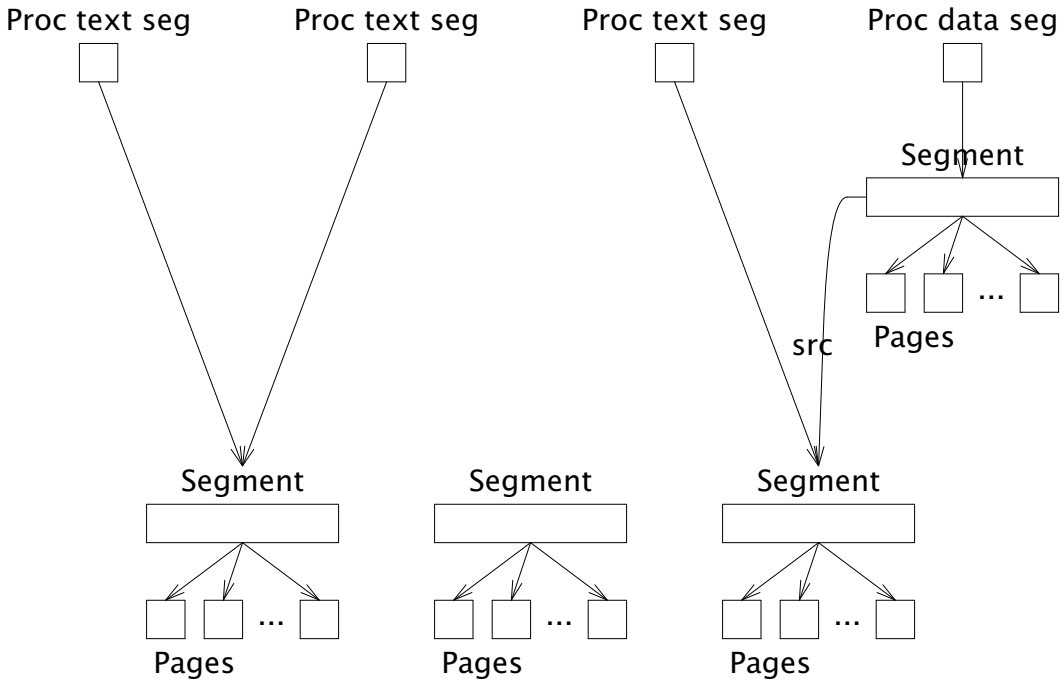

Figure 5 The segment cache keeps entire segments. User text segments are those in the cache. User data segments reference cached initialized data through their source field. User stack segments (not shown) hold anonymous memory never cached and are discussed later.

```
struct Segcache
{
     Lock;
     Segment  *hash[SHASHSIZE];
     Segment  *lru, *mru;       /* hd/tl of lru queue of cached segs */
     uint     nseg;             /* used or cached segs */

     QLock    reclaimlk;        /* one reclaimer at a time */
     int      nrcalls;          /* nb of seg reclaim calls */
     int      nreclaims;        /* nb of segs reclaimed */

};
```

However, it now keeps an LRU list for entire segments. Pages are no longer maintained in any LRU list. This is because Nix mark IV swaps out (actually, removes) entire segments and not single pages. Note that this never affect user data or stack segments because there is no swapping or paging support, by intention.

The user text segments are directly the source segments. These still include *Ptes* and *Pages* for paged-in data, but their *size*, *fstart*, and *flen* fields reflect the sizes for the text, so that data entries are never used or MMU installed for text segments. There is one reference to the segment from the cache hash table (counted in *ref*) plus one extra reference per process using it as its text.

User data segments are given their own *Segment* structures, which reference the cache sources using their *src* field. Their *size*, *fstart*, and *flen* (and their limits) reflect the data in the file. The data segment size includes the provision for the initial space of

the (old) BSS. We describe later how page faults are handled.

The figure shows the pointers from the per-process *Proc.seg* array entries into the segment structures. Of the three user text segments shown (the pointers near the top), two of them refer to the same cached segment (the bottom-left one), and the third refers to the bottom-right cached segment. The cached segment in the middle is not currently in use by any user segment, and would have a reference count of 1 (the left one would have 3, and the right one would have 3). The user data segment shown keeps its own *Segment* structure, referring to the cached segment (that holds initial paged-in data) through its source field. All pages shown in the figure (provided no more segments exist) would have a reference count of 1.

Stack segments (not shown in the figure) are anonymous memory and do not come from any cache. However, the kernel is recycles stack segments including not just their structures, but also their pages and memory. Most stacks are small and it does not pay to release their memory just to allocate it back again on the next *rfork*. Instead, the memory can be cleared (or not!, if you do not care much about security, although the top-most page should be cleared to provide reasonable defaults for the top-of-stack structure) and recycled.

When certain limits are reached, the segment cache scans the LRU list for segments not in use (with a reference count of just 1, for the hash), and release some of them. Here, only the memory is released. The structures used for the segments released are kept cached in the free list for later use.

Note that most pages in the segments have a reference count of 1, because text segments are directly shared and data segments copy their pages from the cache upon faults. However, using *rfork* to copy the data segment results in a copy of the segment that references the same pages. In such case, MMU entries are not (yet) installed for the pages, which now have a reference counter bigger than 1. Fault handling code pays attention to this to perform copy on reference for pages being shared.

The mount driver cache is similar in that it caches data using segments. It keeps a table of segments hashed using the file *qids* and is called by the mount driver code. The table maintains also an LRU of full segments, and removes upon request entire segments from the cache. Each segment is given its maximum possible size, placing a limit on the maximum file size that can be cached. This may change in the future.

This cache is further discussed later along with changes to the mount driver.

Page faults

In general, the segment faulting is locked and then the *Pte* and, perhaps, *Page* structure for the faulting address is located. Once located, the page is locked and the segment lock is released. Also, if the page is not there, a new page is allocated and placed into the segment structure, then it is locked and the segment lock is released.

A field in the page structure, *n*, indicates if the page has been paged in or is not yet initialized, thus processes synchronize on the page lock for paging in. Pages are larger than usual and they are read (using the segment cache) exploiting the new mount driver: a full series of read requests is issued for the entire page and then the replies are waited for.

When a data page is paged in, it is installed in both the source (cache) segment and in the faulting segment. Actually, one page is installed in the cache segment and then copied to another page installed in the user data segment. Thus, the translation for the user data is directly set as read-write, avoiding the need for further page faults and

favoring the use of memory in the right NUMA domain.

Further faults for pages already found in the cache simply copy the data from the page in the cache. For pages outside of the initialized data (i.e., old BSS pages), the page is not read, but is allocated and cleared as it could be expected.

After possibly paging in the page, if its reference count is greater than 1, then the page is shared and must be copied before installing the MMU entry for it.

That is all regarding page faulting. As it can be seen, it is quite simple. However, there are a few other changes intended to reduce the number of page faults suffered by the system.

One such change is that during *rfork* the MMU state is preserved. Instead of flushing all its state, only the entries for shared data segments are flushed. Text segments are read only and do not need to be flushed, and stack segments are copied, including their memory!, and are ready to be used. We copy the actual user stack (the memory) because stacks tend to be small and they are always referenced for writing in both the parent and the child process after a call to *rfork*.

Another aid is that when a segment cache image is attached as a text segment, all its existing entries are installed in the MMU for the process. Thus, there are no page faults for what we already have in memory. We do the same for the stack segment. Therefore, most of the page faults are now for the data segment.

The mount driver

The new mount driver exploits its new cache based on segments to become a simplified, but functionally equivalent, version of the Nix mark III mount driver. See [1] for a description of the software before the changes described here.

The main change is that the mount driver no longer includes specific code for reading ahead, and the old *Cio* structures used to perform asynchronous RPCs are gone. Instead, the cache is leveraged.

For non cacheable files, the mount driver issues one RPC at a time, as usual. That is so because most such files are devices.

For cacheable files, it depends on whether the server speaks 9Pix or 9P. In the former case, the *later* device issues batches of requests as explained in [1], in the latter case requests are issued one by one, although reading and writing can still proceed concurrently.

Cacheable files are always read and written by the mount driver cache, and not by the mount driver, unlike in previous versions. The cache keeps one segment per cached file, allocated to keep the maximum possible segment size starting at address zero. Data read from the file is kept in pages, although the actual file length as found during reads is kept to notice when to report an end of file to the user.

For reading, the mount driver cache scans all the pages involved in the addresses requested and copies the data available. If some data is found, the request returns before any contact with the server. If more data seems to be asked by the user, before returning, the read function starts a kernel process (usually awakes one already started) to read more data from the server into the cache. The next time the user asks for data, the data might have arrived and the user can save the sleep. This is in effect one implementation of read-ahead.

The length of the file noted in the segment structure is used to detect when we should report and end of file condition without asking the server for more data.

The processes reading data from the server operate by sending a concurrent series of read requests, and then waiting for their replies. Synchronization on the pages is similar to that of a user segment demand load, to that each one is read at most once.

Writing to the cache simply discards the cached data. However, the data is written concurrently by sending enough write requests to cover the data to be written and then waiting for the replies.

When a new file version is noticed on the server, or a *remove* request is issued for it, or the file is written, the cache discards its data. This is easier to do than being clever regarding what the resulting file data and file version would be, so it prevents races.

Selfish processes

There is an important optimization not described above, and not considered in the evaluation and the traces shown later. The idea is to let processes keep a few of the resources they release in case they are needed later.

In particular, we modified the process structure to keep up to 10 pages (of the size used for user segments). When a process releases a page and has less than 10 pages kept, it simply keeps the page without releasing it. Later, if a new page is needed it would first try to use one from the per-process pool. The pool is not released when a process dies. Instead, the pool is kept in the process structure and will be used again when a new process is allocated using it.

The trace output taken after applying this optimization shows that most of the pages are reused, and that for small cached programs about 1/3 of the allocations are satisfied with the per-process pool. Thus, the contention on the central page allocator is greatly reduced with this change.

Per process resource pool should be used with care. For example, our attempts to do the same with the kernel memory allocator indicated that it is not a good idea in this case. Memory allocations have very different sizes and some structures are very long lived while others are very short lived. Thus, what happen was that memory was wasted in per process pools and, at the same time, not many memory allocations could benefit from this technique.

In general, per-process allocation pools are a good idea when the structures are frequently used and have the same size. For example, this could be applied also to *Chan* and *Path* structures as used on Nix.

Early evaluation

To measure the impact of the different changes in the behavior of the system, we took the final system and run it with diagnostic output enabled for page allocation and page faults, and measured the different events of interest. Then we did the same disabling one or more of the improvements. The results are not fully precise because debugging output may miss some events sometimes. Further evaluation will use counters instead, and compare with a stock Plan 9 system.

We have to say that the impact of the changes is more dramatic than shown by the results, because early modifications for the mount driver and other parts of the system, on their own, do a good job reducing the impact of system load (e.g., by better caching).

The variations of the system executed are given these names in the results:

all

The standard Nix mark IV. All new features are in.

nopf

Prefaulting code for text and stack segments is disabled. Such code installs into the MMU entries for those pages already present in the segments (because of the cache or other optimizations). The alternative is installing entries on demand.

flush

Prefaulting code is disabled and the MMU is flushed on forks, as it is customary on Plan 9. The alternative to MMU flushes is flushing just the entries for the data segment (others do not have to be because of optimizations explained before).

nodeep

Deep stack forks are disabled. Deep stack forks imply copying the actual stack memory during forks. The alternative is the standard copy on reference to fork a stack segment.

nostk

Stack segments are not cached (their memory is not kept and recycled) and deep stack copies are not performed. The alternative is the standard construction by zero-fill on demand (due to page faults) and full deallocation and allocation of stacks when processes exit and are created.

none

There are no prefaulting code, the MMU is flushed on forks, deep stack copying is disabled, and stack memory is not cached. Quite similar to the state of the system before any modification was made, but for using 16KiB pages for user segments.

none4k

This is the old system. Like the previous variation, but using standard 4KiB pages for user segments.

The system booted normally from a remote file server to execute a shell instead of the full standard start script, and then we executed *pwd* twice. The first table reports the results for the second run of *pwd*, counting since we typed the command to the time when the shell printed its prompt after *pwd* completed. The first way to read the table is to compare any row with the first or the last one, to see the impact of a particular configuration.

The second table shows the same counters but for the entire execution of the system, from a hardware reset to the prompt after executing *pwd* twice.

Several things can be seen:

- going from the old to the new system means going from 68 down to 11 page faults, just for running *pwd* from the shell. For the entire boot process it means going from 843 down to 210.
- Using a more reasonable page size, without other optimizations, reduces a lot the number of page faults (as could be expected). We saw that almost all the 4K pages paged in are actually used for 16K pages, thus it pays to change the page size. Also, the new page size has a significant impact in the size of reads performed by the mount driver, because it enables concurrent reads for larger sizes.
- Deep stack copying reduces a little the page faults in the system, but it might not

bench	page allocs	page frees	mmu faults	page faults	page-ins
all	6	5	14	11	1
nopf	6	5	16	12	1
flush	6	5	22	18	1
nodeep	6	6	17	12	1
nostk	6	7	16	15	1
none	8	8	24	17	1
none4k	15	15	65	68	1

Table 1 Page allocations, page deallocations, page faults, MMU faults, and pages paged in for variations of the system on the second execution of a simple command.

bench	page allocs	page frees	mmu faults	page faults	page-ins
all	229	41	219	210	107
nopf	231	41	246	232	109
flush	224	38	311	283	109
nodeep	227	41	244	237	109
nostk	232	56	245	233	109
none	236	60	321	296	109
none4k	501	107	847	843	313

Table 2 Page allocations, page deallocations, page faults, MMU faults, and pages paged in for variations of the system for an entire boot and two executions of a simple command.

be worth if the time taken to zero out the stack pages kept in the cache is wasted when they are not used. In our system that does not seem to be case.

- More efforts should be taken to avoid flushing MMU state. As it could be expected, not flushing the MMU when it is not necessary reduces quite a bit the number of page faults.

As a reference, two appendixes list the trace output for the old and the new system for executing *pwd* the second time. It is illustrative to compare them.

Acknowledgements

We are grateful to Charles Forsyth and Jim McKie for their advice regarding this work, and to the new Plan 9 secret society.

Future work

The system described in this paper is operational, can be used to run commands and does not seem to have severe leaks, however, more testing is needed before others could use it.

Once the bugs behind the implementation have been dealt with, the new capabilities should be leveraged to better address locality in NUMA many-core systems. The present implementation is rudimentary is this respect, although it includes everything that is needed for better assignment policies.

More quantitative evaluation of this implementation must be conducted, measuring the impact of each feature added by selective configuration. Also, measures must use actual counters for events measured instead of debug output, because the output is to fully precise, otherwise a few events might be missed.

References

1. F. J. Ballesteros, A new mount table and protocol to make 9 and Nix faster, *GSyC-Tech. Rep.. http://lsub.org*, 2013.
2. J. Bonwick, The Slab Allocator: An Object-Caching Kernel Memory Allocator, *USENIX Technical Conference*, 1994.

A. Mark IV pwd trace output

```
% pwd
newpg 0x000000003fe44000 pgsz 0x4000 for 0x4000
fault pid 23 0x20f9e0 r
fault pid 21 0x400000 r
fixfault pid 23 s /bin/rc sref 9 Text 0x200000 addr 0x20c000 pg 0x1174000 r1 n1
fixfixfault faulted pid 23 s /bin/rc Text 0x200000 addr 0x20c000 pg 0x1174000 ref 1
fault pid 23 0x20a9d9 r
pid 21 s /bin/rc sref 1 Data 0x400000 addr 0x400000 pg 0x3fe6c000 r2 n1
newpg 0x000000003fe80000 pgsz 0x4000 for 0x4000
fixfaulted pid 21 s /bin/rc Data 0x400000 addr 0x400000 pg 0x3fe80000 ref 1
fault pid 21 0x404b30 w
fixfault pid 23 s /bin/rc sref 9 Text 0x200000 addr 0x208000 pfixfault pid 21 s /bin/rc sref 1 Data
0x400000 addr 0x404000 pg 0x3fe70000 r2 n1
newpg 0x000000003fe84000 pgsz 0x4000 for 0x4000
g 0x11a4000 r1 n1
fixfaulted pid 21 s /bin/rc Data 0x400000 addr 0x404000 pg 0x3fe84000 ref 1
fault pid 21 0x409afc r
fixfaulted pid 23 s /bin/rc Text 0x200000 addr 0x208000 pg 0x11a4000 ref 1
fault pid 23 0x400060 w
fixfault pid 21 s /bin/rc sref 1 Data 0x400000 addr 0x408000 pg 0x3fe78000 r2 n1
newpg 0x000000003fe88000 pgsz 0x4000 for 0x4000
fixfaulted pid 21 s /bin/rc Data 0x400000 addr 0x408000 pg 0x3fe88000 ref 1
fault pid 21 0x40c178 r
fixfault pid 23 s /bin/rc sref 1 Data 0x400000 addr 0x400000 pg 0x3fe6c000 r1 n1
fixfaulted pid 23 s /bin/rc Data 0x400000 addr 0x400000 pg 0x3fe6c000 ref 1
fault pid 23 0x202aec r
fixfault pid 23 s /bin/rc sref 9 Text 0x200000 fixfault pid 21 s /bin/rc sref 1 Data 0x400000 addr
0x40c000 pg 0x3fe74000 r2 n1
newpg 0x000000003feaddr 0x200000 pg 0x10e4000 r1 n1
fixfaulted pid 23 s /bin/rc Text 0x200000 addr 0x200000 pg 0x10e4000 ref 1
fault pid 23 0x40a698 r
8c000 pgsz 0x4000 for 0x4000
fixfaulted pid 21 s /bin/rc Data 0x400000 addr 0x40c000 pg 0x3fe8c000 ref 1
fixfault pid 23 s /bin/rc sref 1 Data 0x400000 addr 0x408000 pg 0x3fe78000 r1 n1
fixfaulted pid 23 s /bin/rc Data 0x400000 addr 0x408000 pg 0x3fe78000 ref 1
fault pid 23 0x212b47 r
fixfault pid 23 s /bin/rc sref 9 Text 0x200000 addr 0x210000 pg 0x11a8000 r1 n1
fixfaulted pid 23 s /bin/rc Text 0x200000 addr 0x210000 pg 0x11a8000 ref 1
fault pid 23 0x404b30 w
fixfault pid 23 s /bin/rc sref 1 Data 0x400000 addr 0x404000 pg 0x3fe70000 r1 n1
fixfaulted pid 23 s /bin/rc Data 0x400000 addr 0x404000 pg 0x3fe70000 ref 1
fault pid 23 0x40c17c r
fixfault pid 23 s /bin/rc sref 1 Data 0x400000 addr 0x40c000 pg 0x3fe74000 r1 n1
fixfaulted pid 23 s /bin/rc Data 0x400000 addr 0x40c000 pg 0x3fe74000 ref 1
fault pid 23 0x7ffffeffbfe0 w
fixfault pid 23 s '' sref 1 Stack 0x7ffffdffc000 addr 0x7ffffeff8000 pg 0x3fe60000 r1 n1
fixfaulted pid 23 s '' Stack 0x7ffffdffc000 addr 0x7ffffeff8000 pg 0x3fe60000 ref 1
pgfree pg 0x3fe6c000
pgfree pg 0x3fe70000
```

pgfree pg 0x3fe78000
pgfree pg 0x3fe74000
fault pid 23 0x400018 w
fixfault pid 23 s /bin/pwd sref 1 Data 0x400000 addr 0x400000 pg 0x0 r0 n-1
pagein pid 23 s 0x400000 addr 0x400000 soff 0x0
newpg 0x000000003fe74000 pgsz 0x4000 for 0x4000
fixfaulted pid 23 s /bin/pwd Data 0x400000 addr 0x400000 pg 0x3fe74000 ref 1
/usr/nemo
pgfree pg 0x3fe74000

B. Nix mark III pwd trace output

```
fault pid 21 0x2000c4 r
fixfault pid 21 s /bin/rc sref 7 Text 0x200000 addr 0x200000 pg 0x116c000 r1 n1
fixfaulted pid 21 s /bin/rc Text 0x200000 addr 0x200000 pg 0x116c000 ref 1
fault pid 21 0x20170a r
fixfault pid 21 s /bin/rc sref 7 Text 0x200000 addr 0x201000 pg 0x116f000 r1 n1
fixfaulted pid 21 s /bin/rc Text 0x200000 addr 0x201000 pg 0x116f000 ref 1
fault pid 21 0x205dfc r
fixfault pid 21 s /bin/rc sref 7 Text 0x200000 addr 0x205000 pg 0x1132000 r1 n1
fixfaulted pid 21 s /bin/rc Text 0x200000 addr 0x205000 pg 0x1132000 ref 1
fault pid 21 0x40b834 w
fixfault pid 21 s /bin/rc sref 1 Data 0x400000 addr 0x40b000 pg 0x11b3000 r1 n1
fixfaulted pid 21 s /bin/rc Data 0x400000 addr 0x40b000 pg 0x11b3000 ref 1
fault pid 21 0x407c78 r
fixfault pid 21 s /bin/rc sref 1 Data 0x400000 addr 0x407000 pg 0x11ac000 r1 n1
fixfaulted pid 21 s /bin/rc Data 0x400000 addr 0x407000 pg 0x11ac000 ref 1
fault pid 23 0x20f9e0 r
fixfault pid 23 s /bin/rc sref 9 Text 0x200000 addr 0x20f000 pg 0x1138000 r1 n1
fixfaulted pid 23 s /bin/rc Text 0x200000 addr 0x20f000 pg 0x1138000 ref 1
fault pid 23 0x7ffffffefe8 w
fixfault pid 23 s /bin/rc sref 1 Stack 0x7ffffefff000 addr 0x7fffffffe000 pg 0x11b0000 r2 n1
newpg 0x00000000011d2000 pgsz 0x1000 for 0x1000
fixfaulted pid 23 s '' Stack 0x7ffffefff000 addr 0x7fffffffe000 pg 0x11d2000 ref 1
fault pid 23 0x20a9d9 r
fixfault pid 23 s /bin/rc sref 9 Text 0x200000 addr 0x20a000 pg 0x1146000 r1 n1
fixfaulted pid 23 s /bin/rc Text 0x200000 addr 0x20a000 pg 0x1146000 ref 1
fault pid 23 0x20be91 r
fixfault pid 23 s /bin/rc sref 9 Text 0x200000 addr 0x20b000 pg 0x1135000 r1 n1
fixfaulted pid 23 s /bin/rc Text 0x200000 addr 0x20b000 pg 0x1135000 ref 1
fault pid 23 0x400060 w
fixfault pid 23 s /bin/rc sref 1 Data 0x400000 addr 0x400000 pg 0x11aa000 r2 n1
newpg 0x00000000011d0000 pgsz 0x1000 for 0x1000
fixfaulted pid 23 s /bin/rc Data 0x400000 addr 0x400000 pg 0x11d0000 ref 1
fault pid 23 0x202aec r
fixfault pid 23 s /bin/rc sref 9 Text 0x200000 addr 0x202000 pg 0x1130000 r1 n1
fixfaulted pid 23 s /bin/rc Text 0x200000 addr 0x202000 pg 0x1130000 ref 1
fault pid 23 0x40a698 r
fixfault pid 23 s /bin/rc sref 1 Data 0x400000 addr 0x40a000 pg 0x119e000 r2 n1
newpg 0x00000000011d6000 pgsz 0x1000 for 0x1000
fixfaulted pid 23 s /bin/rc Data 0x400000 addr 0x40a000 pg 0x11d6000 ref 1
fault pid 23 0x20977d r
fixfault pid 23 s /bin/rc sref 9 Text 0x200000 addr 0x209000 pg 0x1139000 r1 n1
fixfaulted pid 23 s /bin/rc Text 0x200000 addr 0x209000 pg 0x1139000 ref 1
fault pid 23 0x20dbe3 r
fixfault pid 23 s /bin/rc sref 9 Text 0x200000 addr 0x20d000 pg 0x113b000 r1 n1
fixfaulted pid 23 s /bin/rc Text 0x200000 addr 0x20d000 pg 0x113b000 ref 1
fault pid 23 0x212b47 r
fixfault pid 23 s /bin/rc sref 9 Text 0x200000 addr 0x212000 pg 0x113c000 r1 n1
fixfaulted pid 23 s /bin/rc Text 0x200000 addr 0x212000 pg 0x113c000 ref 1
fault pid 23 0x401338 r
```

```
fixfault pid 23 s /bin/rc sref 1 Data 0x400000 addr 0x401000 pg 0x11af000 r2 n1
newpg 0x00000000011ce000 pgsz 0x1000 for 0x1000
fixfaulted pid 23 s /bin/rc Data 0x400000 addr 0x401000 pg 0x11ce000 ref 1
fault pid 23 0x210670 r
fixfault pid 23 s /bin/rc sref 9 Text 0x200000 addr 0x210000 pg 0x113d000 r1 n1
fixfaulted pid 23 s /bin/rc Text 0x200000 addr 0x210000 pg 0x113d000 ref 1
fault pid 23 0x404b30 w
fixfault pid 23 s /bin/rc sref 1 Data 0x400000 addr 0x404000 pg 0x11ae000 r2 n1
newpg 0x00000000011d3000 pgsz 0x1000 for 0x1000
fixfaulted pid 23 s /bin/rc Data 0x400000 addr 0x404000 pg 0x11d3000 ref 1
fault pid 23 0x21107c r
fixfault pid 23 s /bin/rc sref 9 Text 0x200000 addr 0x211000 pg 0x1143000 r1 n1
fixfaulted pid 23 s /bin/rc Text 0x200000 addr 0x211000 pg 0x1143000 ref 1
fault pid 23 0x40c17c r
fixfault pid 23 s /bin/rc sref 1 Data 0x400000 addr 0x40c000 pg 0x11b2000 r2 n1
newpg 0x00000000011cf000 pgsz 0x1000 for 0x1000
fixfaulted pid 23 s /bin/rc Data 0x400000 addr 0x40c000 pg 0x11cf000 ref 1
fault pid 23 0x4097fc r
fixfault pid 23 s /bin/rc sref 1 Data 0x400000 addr 0x409000 pg 0x1196000 r2 n1
newpg 0x00000000011ca000 pgsz 0x1000 for 0x1000
fixfaulted pid 23 s /bin/rc Data 0x400000 addr 0x409000 pg 0x11ca000 ref 1
fault pid 23 0x20e02d r
fixfault pid 23 s /bin/rc sref 9 Text 0x200000 addr 0x20e000 pg 0x1147000 r1 n1
fixfaulted pid 23 s /bin/rc Text 0x200000 addr 0x20e000 pg 0x1147000 ref 1
fault pid 23 0x20cbb0 r
fixfault pid 23 s /bin/rc sref 9 Text 0x200000 addr 0x20c000 pg 0x1129000 r1 n1
fixfaulted pid 23 s /bin/rc Text 0x200000 addr 0x20c000 pg 0x1129000 ref 1
fault pid 23 0x40204a r
fixfault pid 23 s /bin/rc sref 1 Data 0x400000 addr 0x402000 pg 0x11a7000 r2 n1
newpg 0x00000000011c6000 pgsz 0x1000 for 0x1000
fixfaulted pid 23 s /bin/rc Data 0x400000 addr 0x402000 pg 0x11c6000 ref 1
fault pid 23 0x2086dc r
fixfault pid 23 s /bin/rc sref 9 Text 0x200000 addr 0x208000 pg 0x1165000 r1 n1
fixfaulted pid 23 s /bin/rc Text 0x200000 addr 0x208000 pg 0x1165000 ref 1
fault pid 23 0x213000 r
fixfault pid 23 s /bin/rc sref 9 Text 0x200000 addr 0x213000 pg 0x1142000 r1 n1
fixfaulted pid 23 s /bin/rc Text 0x200000 addr 0x213000 pg 0x1142000 ref 1
fault pid 23 0x4055b8 r
fixfault pid 23 s /bin/rc sref 1 Data 0x400000 addr 0x405000 pg 0x11a9000 r2 n1
newpg 0x00000000011d8000 pgsz 0x4000 for 0x1000
splitbundle pg 0x00000000011d8000
fixfaulted pid 23 s /bin/rc Data 0x400000 addr 0x405000 pg 0x11d8000 ref 1
fault pid 23 0x4081a8 r
fixfault pid 23 s /bin/rc sref 1 Data 0x400000 addr 0x408000 pg 0x11a8000 r2 n1
newpg 0x00000000011db000 pgsz 0x1000 for 0x1000
fixfaulted pid 23 s /bin/rc Data 0x400000 addr 0x408000 pg 0x11db000 ref 1
fault pid 23 0x407c78 r
fixfault pid 23 s /bin/rc sref 1 Data 0x400000 addr 0x407000 pg 0x11ac000 r2 n1
newpg 0x00000000011da000 pgsz 0x1000 for 0x1000
fixfaulted pid 23 s /bin/rc Data 0x400000 addr 0x407000 pg 0x11da000 ref 1
```

fault pid 23 0x406dd8 r
fixfault pid 23 s /bin/rc sref 1 Data 0x400000 addr 0x406000 pg 0x11ad000 r2 n1
newpg 0x00000000011d9000 pgsz 0x1000 for 0x1000
fixfaulted pid 23 s /bin/rc Data 0x400000 addr 0x406000 pg 0x11d9000 ref 1
fault pid 21 0x7fffffffefe8 w
fixfault pid 23 0x7ffffeffefe0 w
fixfault pid 23 s '' sref 1 Stack 0x7ffffdfff000 addr 0x7ffffeffe000 pg 0x0 r0 n-1
newpg 0x00000000011dc000 pgsz 0x4000 for 0x1000
splitbundle pg 0x00000000011dc000
fixfaulted pid 23 s '' Stack 0x7ffffdfff000 addr 0x7ffffeffe000 pg 0x11dc000 ref 1
pgfree pg 0x11d2000
pgfree pg 0x11d0000
pgfree pg 0x11ce000
pgfree pg 0x11c6000
pgfree pg 0x11d3000
pgfree pg 0x11d8000
pgfree pg 0x11d9000
pgfree pg 0x11da000
pgfree pg 0x11db000
pgfree pg 0x11ca000
pgfree pg 0x11d6000
pgfree pg 0x11cf000
faultfault pid 23 0x7fffffffef98 w
fixfault pid 23 s /bin/pwd sref 1 Stack 0x7ffffefff000 addr 0x7fffffffe000 pg 0x11dc000 r1 n1
fixfaulted pid 23 s '' Stack 0x7ffffefff000 addr 0x7fffffffe000 pg 0x11dc000 ref 1
fault pid 23 0x20008a r
fixfault pid 23 s /bin/pwd sref 3 Text 0x200000 addr 0x200000 pg 0x11cd000 r1 n1
fixfaulted pid 23 s /bin/pwd Text 0x200000 addr 0x200000 pg 0x11cd000 ref 1
fault pid 23 0x400018 w
fixfault pid 23 s /bin/pwd sref 1 Data 0x400000 addr 0x400000 pg 0x0 r0 n-1
pagein pid 23 s 0x400000 addr 0x400000 soff 0x0
newpg 0x00000000011cf000 pgsz 0x1000 for 0x1000
fixfaulted pid 23 s /bin/pwd Data 0x400000 addr 0x400000 pg 0x11cf000 ref 1
fault pid 23 0x201b6d r
fixfault pid 23 s /bin/pwd sref 3 Text 0x200000 addr 0x201000 pg 0x11d1000 r1 n1
fixfaulted pid 23 s /bin/pwd Text 0x200000 addr 0x201000 pg 0x11d1000 ref 1
 pid 21 s /bfault pid 23 0x2020ef r
fixfault pid 23 s /bin/pwd sref 3 Text 0x200000 addr 0x202000 pg 0x11d4000 r1 n1
fixfaulted pid 23 s /bin/pwd Text 0x200000 addr 0x202000 pg 0x11d4000 ref 1
fault pid 23 0x2040b2 r
fixfault pid 23 s /bin/pwd sref 3 Text 0x200000 addr 0x204000 pg 0x11d7000 r1 n1
fixfaulted pid 23 s /bin/pwd Text 0x200000 addr 0x204000 pg 0x11d7000 ref 1
/usr/nemo
fault pid 23 0x401098 r
fixfault pid 23 s /bin/pwd sref 1 Data 0x400000 addr 0x401000 pg 0x0 r0 n-1
pagein: zfod 0x401000
newpg 0x00000000011d6000 pgsz 0x1000 for 0x1000
fixfaulted pid 23 s /bin/pwd Data 0x400000 addr 0x401000 pg 0x11d6000 ref 1
pgfree pg 0x11dc000
pgfree pg 0x11cf000

```
pgfree pg 0x11d6000
in/rc sref 1 Stack 0x7ffffefff000 addr 0x7fffffffe000 pg 0x11b0000 r1 n1
fixfaulted pid 21 s '' Stack 0x7ffffefff000 addr 0x7fffffffe000 pg 0x11b0000 ref 1
fault pid 21 0x20f9e0 r
fixfault pid 21 s /bin/rc sref 7 Text 0x200000 addr 0x20f000 pg 0x1138000 r1 n1
fixfaulted pid 21 s /bin/rc Text 0x200000 addr 0x20f000 pg 0x1138000 ref 1
fault pid 21 0x20a9d9 r
fixfault pid 21 s /bin/rc sref 7 Text 0x200000 addr 0x20a000 pg 0x1146000 r1 n1
fixfaulted pid 21 s /bin/rc Text 0x200000 addr 0x20a000 pg 0x1146000 ref 1
fault pid 21 0x20bdca r
fixfault pid 21 s /bin/rc sref 7 Text 0x200000 addr 0x20b000 pg 0x1135000 r1 n1
fixfaulted pid 21 s /bin/rc Text 0x200000 addr 0x20b000 pg 0x1135000 ref 1
fault pid 21 0x400000 r
fixfault pid 21 s /bin/rc sref 1 Data 0x400000 addr 0x400000 pg 0x11aa000 r1 n1
fixfaulted pid 21 s /bin/rc Data 0x400000 addr 0x400000 pg 0x11aa000 ref 1
fault pid 21 0x20ddb3 r
fixfault pid 21 s /bin/rc sref 7 Text 0x200000 addr 0x20d000 pg 0x113b000 r1 n1
fixfaulted pid 21 s /bin/rc Text 0x200000 addr 0x20d000 pg 0x113b000 ref 1
fault pid 21 0x212ea2 r
fixfault pid 21 s /bin/rc sref 7 Text 0x200000 addr 0x212000 pg 0x113c000 r1 n1
fixfaulted pid 21 s /bin/rc Text 0x200000 addr 0x212000 pg 0x113c000 ref 1
fault pid 21 0x401338 r
fixfault pid 21 s /bin/rc sref 1 Data 0x400000 addr 0x401000 pg 0x11af000 r1 n1
fixfaulted pid 21 s /bin/rc Data 0x400000 addr 0x401000 pg 0x11af000 ref 1
fault pid 21 0x210670 r
fixfault pid 21 s /bin/rc sref 7 Text 0x200000 addr 0x210000 pg 0x113d000 r1 n1
fixfaulted pid 21 s /bin/rc Text 0x200000 addr 0x210000 pg 0x113d000 ref 1
fault pid 21 0x404b30 w
fixfault pid 21 s /bin/rc sref 1 Data 0x400000 addr 0x404000 pg 0x11ae000 r1 n1
fixfaulted pid 21 s /bin/rc Data 0x400000 addr 0x404000 pg 0x11ae000 ref 1
fault pid 21 0x409afc r
fixfault pid 21 s /bin/rc sref 1 Data 0x400000 addr 0x409000 pg 0x1196000 r1 n1
fixfaulted pid 21 s /bin/rc Data 0x400000 addr 0x409000 pg 0x1196000 ref 1
fault pid 21 0x211ba3 r
fixfault pid 21 s /bin/rc sref 7 Text 0x200000 addr 0x211000 pg 0x1143000 r1 n1
fixfaulted pid 21 s /bin/rc Text 0x200000 addr 0x211000 pg 0x1143000 ref 1
fault pid 21 0x20e02d r
fixfault pid 21 s /bin/rc sref 7 Text 0x200000 addr 0x20e000 pg 0x1147000 r1 n1
fixfaulted pid 21 s /bin/rc Text 0x200000 addr 0x20e000 pg 0x1147000 ref 1
fault pid 21 0x208574 r
fixfault pid 21 s /bin/rc sref 7 Text 0x200000 addr 0x208000 pg 0x1165000 r1 n1
fixfaulted pid 21 s /bin/rc Text 0x200000 addr 0x208000 pg 0x1165000 ref 1
fault pid 21 0x202c39 r
fixfault pid 21 s /bin/rc sref 7 Text 0x200000 addr 0x202000 pg 0x1130000 r1 n1
fixfaulted pid 21 s /bin/rc Text 0x200000 addr 0x202000 pg 0x1130000 ref 1
fault pid 21 0x40a698 r
fixfault pid 21 s /bin/rc sref 1 Data 0x400000 addr 0x40a000 pg 0x119e000 r1 n1
fixfaulted pid 21 s /bin/rc Data 0x400000 addr 0x40a000 pg 0x119e000 ref 1
fault pid 21 0x2097b7 r
fixfault pid 21 s /bin/rc sref 7 Text 0x200000 addr 0x209000 pg 0x1139000 r1 n1
```

```
fixfaulted pid 21 s /bin/rc Text 0x200000 addr 0x209000 pg 0x1139000 ref 1
fault pid 21 0x213000 r
fixfault pid 21 s /bin/rc sref 7 Text 0x200000 addr 0x213000 pg 0x1142000 r1 n1
fixfaulted pid 21 s /bin/rc Text 0x200000 addr 0x213000 pg 0x1142000 ref 1
fault pid 21 0x40c178 r
fixfault pid 21 s /bin/rc sref 1 Data 0x400000 addr 0x40c000 pg 0x11b2000 r1 n1
fixfaulted pid 21 s /bin/rc Data 0x400000 addr 0x40c000 pg 0x11b2000 ref 1
fault pid 21 0x20ce7e r
fixfault pid 21 s /bin/rc sref 7 Text 0x200000 addr 0x20c000 pg 0x1129000 r1 n1
fixfaulted pid 21 s /bin/rc Text 0x200000 addr 0x20c000 pg 0x1129000 ref 1
fault pid 21 0x204bba r
fixfault pid 21 s /bin/rc sref 7 Text 0x200000 addr 0x204000 pg 0x1133000 r1 n1
fixfaulted pid 21 s /bin/rc Text 0x200000 addr 0x204000 pg 0x1133000 ref 1
fault pid 21 0x402611 r
fixfault pid 21 s /bin/rc sref 1 Data 0x400000 addr 0x402000 pg 0x11a7000 r1 n1
fixfaulted pid 21 s /bin/rc Data 0x400000 addr 0x402000 pg 0x11a7000 ref 1
fault pid 21 0x405e00 r
fixfault pid 21 s /bin/rc sref 1 Data 0x400000 addr 0x405000 pg 0x11a9000 r1 n1
fixfaulted pid 21 s /bin/rc Data 0x400000 addr 0x405000 pg 0x11a9000 ref 1
fault pid 21 0x408d48 r
fixfault pid 21 s /bin/rc sref 1 Data 0x400000 addr 0x408000 pg 0x11a8000 r1 n1
fixfaulted pid 21 s /bin/rc Data 0x400000 addr 0x408000 pg 0x11a8000 ref 1
fault pid 21 0x20310e r
fixfault pid 21 s /bin/rc sref 7 Text 0x200000 addr 0x203000 pg 0x1136000 r1 n1
fixfaulted pid 21 s /bin/rc Text 0x200000 addr 0x203000 pg 0x1136000 ref 1
fault pid 21 0x40b834 w
fixfault pid 21 s /bin/rc sref 1 Data 0x400000 addr 0x40b000 pg 0x11b3000 r1 n1
fixfaulted pid 21 s /bin/rc Data 0x400000 addr 0x40b000 pg 0x11b3000 ref 1
fault pid 21 0x206ab9 r
fixfault pid 21 s /bin/rc sref 7 Text 0x200000 addr 0x206000 pg 0x113a000 r1 n1
fixfaulted pid 21 s /bin/rc Text 0x200000 addr 0x206000 pg 0x113a000 ref 1
fault pid 21 0x406820 r
fixfault pid 21 s /bin/rc sref 1 Data 0x400000 addr 0x406000 pg 0x11ad000 r1 n1
fixfaulted pid 21 s /bin/rc Data 0x400000 addr 0x406000 pg 0x11ad000 ref 1
fault pid 21 0x7fffffffce78 w
fixfault pid 21 s /bin/rc sref 1 Stack 0x7ffffefff000 addr 0x7fffffffc000 pg 0x11bd000 r1 n1
fixfaulted pid 21 s '' Stack 0x7ffffefff000 addr 0x7fffffffc000 pg 0x11bd000 ref 1
fault pid 21 0x2071d8 r
fixfault pid 21 s /bin/rc sref 7 Text 0x200000 addr 0x207000 pg 0x116b000 r1 n1
fixfaulted pid 21 s /bin/rc Text 0x200000 addr 0x207000 pg 0x116b000 ref 1
%
```