A new malloc(3) for OpenBSD

Otto Moerbeek

otto@openbsd.org

Me?

- Developer since 2003
- Mainly userland work: patch, diff, dc, bc, privilege separated tcpdump, libc, ntpd, ...
- But also some kernel work: large partitions, ffs2 integration, time code

What’s malloc(3)?

- Kernel knows two ways of giving memory to an application: sbrk(2) and mmap(2)
- malloc takes memory from the kernel and manages it for the application
- so what’s managing?

Managing memory

- `void *malloc(size_t)`: get the application a region of memory
- `free(void *)`: release the memory
- `void *realloc(void *, size_t)`: resize, preserving contents as far as possible
- Details: alignment, 0 size, what happens to released memory?

Original *BSD malloc

- By Poul-Henning Kamp
- Get a contiguous region of memory from kernel using sbrk(2)
- Grow it when more is needed
- Manage pages by keeping an index, including free list. Index contains status per page.
- Manage chunk (sub page sized regions) by dividing a page and maintaining a bit map per chunk page

Original BSD malloc II

- Very predictable behavior
- Released memory can only be returned to the kernel in rare circumstances
- Meta data can leak to application, though more recent code in NetBSD uses mmap’ed memory for the page index.

Predictability is bad

- See for example the work of Ben Hawkes
- e.g. call free with a pointer, and that pointer will at some point be returned via malloc, even if the application is still using it!
- In combination with application bugs, this can be exploited

Next for OpenBSD

- A mmap(2) based malloc.
- mmap on OpenBSD returns range of pages at a random location
- Page dir was modified to allow for non-contiguous ranges of pages
- Linked list of page dir pages

Nice properties

- Addresses become more unpredictable
- Pages next to an allocated area are likely unmapped: free overrun protection
- Initially the page dir and free list were malloc’ed themselves, but that was changed later

Not so nice

- Due to caching of free pages, predictability comes back (at least partially)
- Free list maintenance might need memory to free memory
- For large address spaces, the page dir becomes very sparse

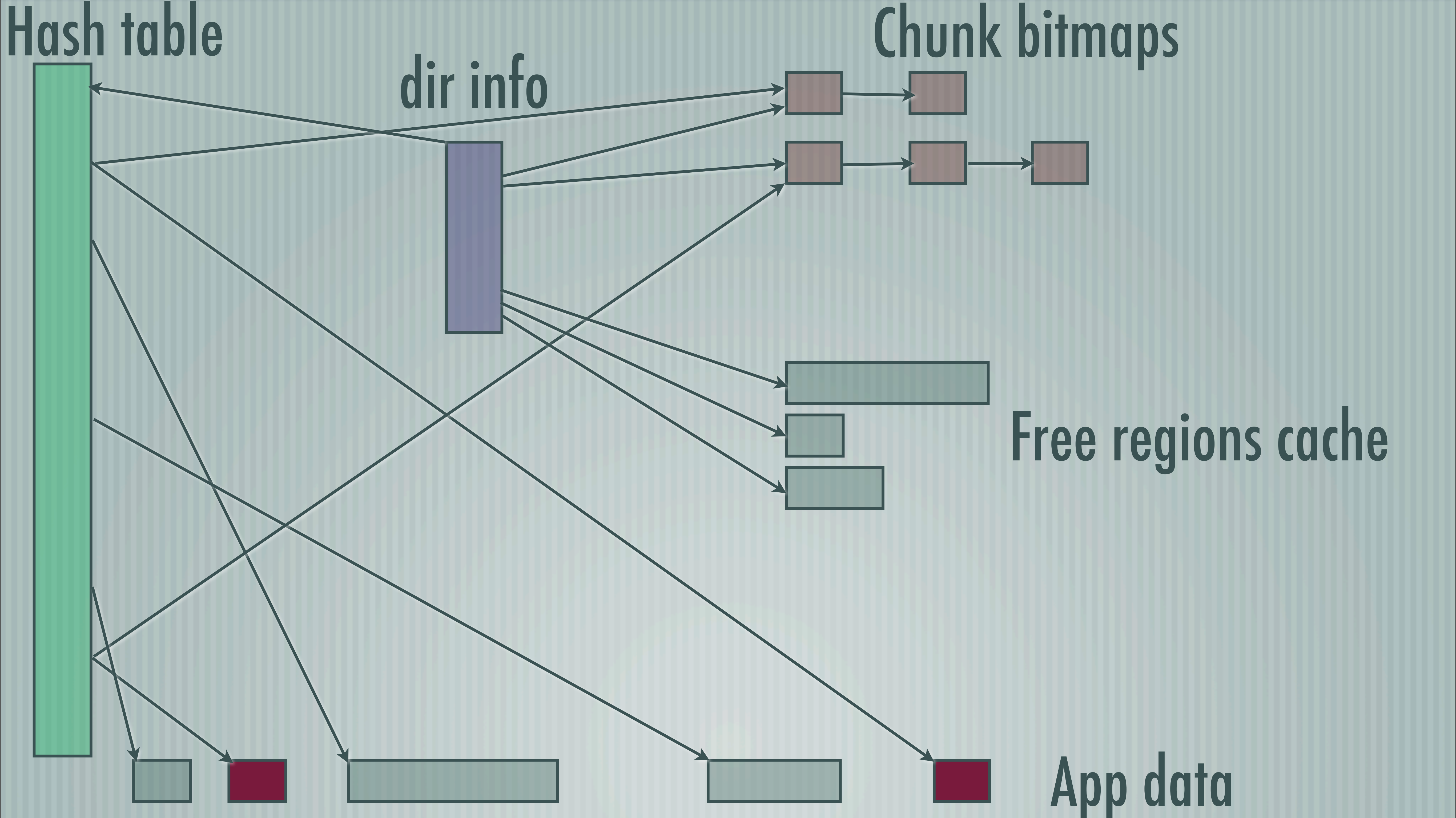
Design goals

- Simple
- Unpredictable
- Fast
- Less metadata space overhead
- Robust, e.g. freeing of a bogus pointer or a double free should be detected

Meta data

- Keep track of mmap’ed regions by storing their address and size into a hash table
- Keep existing data structure for chunk allocations
- A free region cache with a fixed number of slots

Overview of metadata



The hash table

```
struct region_info {
 void *p;        /* page; low bits used
                    to mark chunks */
 uintptr_t size; /* size for pages, or
                    chunk_info pointer */
};

static inline size_t hash(void *p)
{
 size_t sum;
 union {
  uintptr_t p;
  unsigned short a[sizeof(void *) /
    sizeof(short)];
 } u;
 u.p = (uintptr_t)p >> MALLOC_PAGESHIFT;
 sum = u.a[0];
 sum = (sum << 7) - sum + u.a[1];
#ifdef __LP64__
 sum = (sum << 7) - sum + u.a[2];
 sum = (sum << 7) - sum + u.a[3];
#endif
 return sum;
}
```

The pointers returned by mmap are already random

Simple hash function collapsing the bits

Insert

```
static int
insert(struct dir_info *d, void *p, size_t sz)
{
 size_t index, mask;
 void *q;

 if (d->regions_free * 4 < d->regions_total) {
  if (omalloc_grow(d))
   return 1;
 }
 mask = d->regions_total - 1;
 index = hash(p) & mask;
 q = d->r[index].p;
 STATS_INC(d->inserts);
 while (q != NULL) {
  index = (index - 1) & mask;
  q = d->r[index].p;
  STATS_INC(d->insert_collisions);
 }
 d->r[index].p = p;
 d->r[index].size = sz;
 d->regions_free--;
 return 0;
}
```

Hash table is grown if too full

Too full is >75% slots filled

Delete

```
static void
delete(struct dir_info *d, struct region_info *ri)
{
 /* algorithm R, Knuth Vol III section 6.4 */
 size_t mask = d->regions_total - 1;
 size_t i, j, r;

 d->regions_free++;
 i = ri - d->r;
 for (;;) {
  d->r[i].p = NULL;
  d->r[i].size = 0;
  j = i;
  for (;;) {
   i = (i - 1) & mask;
   if (d->r[i].p == NULL)
    return;
   r = hash(d->r[i].p) & mask;
   if ((i <= r && r < j) || (r < j && j < i) ||
       (j < i && i <= r))
    continue;
   d->r[j] = d->r[i];
   break;
  }
 }
}
```

- Straightforward from Knuth
- On delete, restore state as if the deleted item was never there
- Nice properties, as long as the hash function is good

Free regions cache

- Regions free'ed are kept for later reuse
- Large regions are unmapped directly
- If the number of pages cached gets too large, unmap some.
- Randomized search for fitting region, so region reuse is less predictable
- Optionally, pages in the cache are marked PROT_NONE

Some nice properties

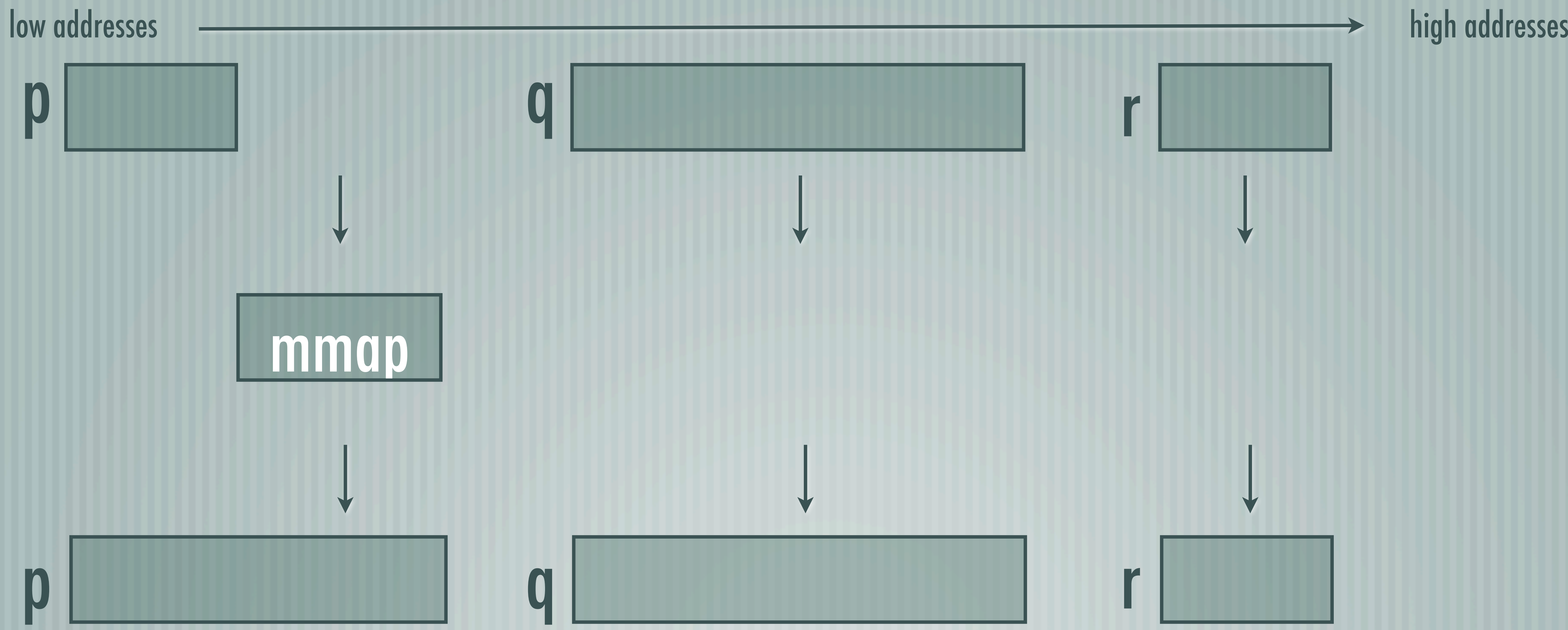
- Amortized cost of insert, find and delete are low
- Speed tests indicate a a 30% speedup compared to old malloc, though the gains are less in the final version due to more randomization in chunk allocation.
- `free(bogus)` is caught
- Memory is given back to kernel

Calloc

- `calloc(3)`: since pages returned by mmap are zero-filled, we do not need to zero them ourselves. We can use that for >= page sized allocations
- Nicely avoids page references until the pages are actually used

Realloc

Try to mmap next to the existing region if we are growing. There’s a high chance it’s available



If this doesn’t work out, fall back to new alloc and copy

PROT_READ is your friend

- Originally dir_info was in the bss section, having a predictable address
- Work by Damien Miller (djm@): protect dir_info and malloc options by mmap'ing the memory containing them and using PROT_READ to make it read only (optionally for page_dir)
- dir_info and chunk_info protected by canaries

Special

- Allocations between half a page and a page need a full page.
- Buffer overruns are more common than buffer underruns
- Taking into account alignment restrictions, we can shift the returned pointer so that the end of the region is near the end of the page
- An ancient bug in the code generated by yacc was discovered

Summary

- Faster
- More simple
- Never needs memory to free memory
- Robust
- Meta-data completely out-of-band
- Randomization in page, chunk, allocation and freeing.
- Since OpenBSD 4.4

Improvements?

- Lock contention for threaded apps
- Chunk randomization across multiple pages
- Improve kernel data structures to better handle fragmented vm
- sbrk(2) is still supported; this decreases available memory for apps, especially on 32-bit architectures

Thanks

- The OpenBSD community
- Especially Theo de Raadt en Damien Miller
- This audience