1. OpenBSD Memory Allocator.

This documents malloc.c from CVS version 1.290: OpenBSD 7.4.

2. Also *dir_info*, *region_info*, *bigcache*, *smallcache*, *malloc_readonly*.

For stats: *malloc_leak*, *leaknode*.

The marked types should be replaced with their modern equivalents.

format *delete none*
format *u_char int*
format *u_int int*
format *u_int32_t int* /* */
format *u_short int*
format *uint32_t int*
format *uint64_t int*
format *uintptr_t int*
format *ushort int* /* */
format *chunk_head int*
format *chunk_info int*
format *leaktree int*

3. The memory allocator, hereafter called malloc like its namesake, is used in several contexts. When being built for the install media and anywhere else space is tight the symbol `MALLOC_SMALL` can be defined to disable gathering run-time statistics.

```
#ifndef MALLOC_SMALL
#define MALLOC_STATS
#endif
```

4. Malloc begins with a fairly standard set of headers, notably including `stdlib.h` which it provides part of the API for!

The unusual header in this list is `uvm/uvmexp.h` which exposes some parts of the OpenBSD kernel's virtual memory API.

If statistics are going to be gathered then additional headers are required including those for the `ktrace` API and `dlfcn.h` to locate information about functions in dynamically linked libraries.

Finally of course malloc must be aware of threads and requires some interfaces that are not part of the public API.

```
#include <sys/types.h>
#include <sys/queue.h>
#include <sys/mman.h>
#include <sys/sysctl.h>
#include <uvm/uvmexp.h>
#include <errno.h>
#include <stdarg.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#ifdef MALLOC_STATS
#include <sys/tree.h>
#include <sys/ktrace.h>
#include <dlfcn.h>
#endif
#include "thread_private.h"
#include <tib.h>
```

5. Malloc acquires the memory to divide up and organise for user-space in groups of one or more pages the length of `_MAX_PAGE_SHIFT`, defined in the system's architecture headers. For example on amd64 it's defined in `/usr/include/amd64/_types.h` to be 12, so a page is 2^{12} or 4096 bytes. Other platforms may differ with smaller and larger maximum page sizes. These macros ensure malloc's portability to all of these page sizes.

When a large allocation is requested a single contiguous range of pages is obtained from the kernel including an additional page at each side to guard against out-of-bounds reads or writes. Allocations smaller than half a page are served by dividing an operating system page into so-called "chunks" which are grouped together in "buckets" of other same-sized chunks.

```
#define MALLOC_PAGESHIFT _MAX_PAGE_SHIFT

#define MALLOC_MINSHIFT 4
#define MALLOC_MAXSHIFT (MALLOC_PAGESHIFT - 1)
#define MALLOC_PAGESIZE (1UL << MALLOC_PAGESHIFT)
#define MALLOC_MINSIZE (1UL << MALLOC_MINSHIFT)
#define MALLOC_PAGEMASK (MALLOC_PAGESIZE - 1)
#define MASK_POINTER(p) ((void *)(((uintptr_t)(p)) & ~MALLOC_PAGEMASK))

#define MALLOC_MAXCHUNK (1 << MALLOC_MAXSHIFT)
#define MALLOC_MAXCACHE 256
#define MALLOC_DELAYED_CHUNK_MASK 15
#ifdef MALLOC_STATS
#define MALLOC_INITIAL_REGIONS 512
#else
#define MALLOC_INITIAL_REGIONS (MALLOC_PAGESIZE/sizeof(struct region_info))
#endif
#define MALLOC_DEFAULT_CACHE 64
#define MALLOC_CHUNK_LISTS 4
#define CHUNK_CHECK_LENGTH 32

#define B2SIZE(b) ((b) * MALLOC_MINSIZE)
#define B2ALLOC(b) ((b) == 0 ? MALLOC_MINSIZE : (b) * MALLOC_MINSIZE)
#define BUCKETS (MALLOC_MAXCHUNK/MALLOC_MINSIZE)
```

6. We move allocations between half a page and a whole page towards the end, subject to alignment constraints. This is the extra headroom we allow. Set to zero to be the most strict.

```
#define MALLOC_LEEWAY 0
#define MALLOC_MOVE_COND(sz) ((sz) - mopts.malloc_guard < MALLOC_PAGESIZE - MALLOC_LEEWAY)
#define MALLOC_MOVE(p, sz) (((char *)(p)) +
    ((MALLOC_PAGESIZE - MALLOC_LEEWAY - ((sz) - mopts.malloc_guard)) & ~(MALLOC_MINSIZE - 1)))

#define PAGEROUND(x) (((x) + (MALLOC_PAGEMASK)) & ~MALLOC_PAGEMASK)
```

7. "Junking" is the process of filling an allocation with a pattern. The level of junking requested determines whether junking will be performed at all and when.

What to use for Junk. This is the byte value we use to fill with when the `J` option is enabled. Use `SOME_JUNK` right after alloc, and `SOME_FREEJUNK` right before free.

```
#define SOME_JUNK 0xdb      /* deadbeef */
#define SOME_FREEJUNK 0xdf      /* dead, free */
#define SOME_FREEJUNK_ULL 0xdfdfdfdfdfdfdfdfULL
```

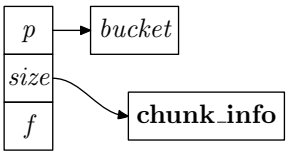
8. The mmap(2) system call requests pages from the operating system with one of these macros. `MMAPA` is identical to the usual `MMAP` except that requests a mapping at a specific virtual address. `MMAPNONE` requests a mapping with nether read nor write (nor exec) access which can be used as a guard page.

```
#define MMAP(sz, f)mmap (Λ, (sz), PROT_READ | PROT_WRITE, MAP_ANON | MAP_PRIVATE | (f), -1, 0)

#define MMAPNONE(sz, f)mmap (Λ, (sz), PROT_NONE, MAP_ANON | MAP_PRIVATE | (f), -1, 0)

#define MMAPA(a, sz, f)mmap
  ((a), (sz), PROT_READ | PROT_WRITE, MAP_ANON | MAP_PRIVATE | (f), -1, 0)
```

9. A "region" is any allocation of a page or more. Single-page allocations may be divided into chunks and a *region_info* object does double duty tracking both types, distinguished by p which when pointing to a large allocation will be page-aligned so the low `MALLOC_PAGESHIFT` bits will be zero and *size* is the length of the allocation.

If any of the low `MALLOC_PAGESHIFT` bits in p is set then the region is a page divided into chunks and *size* is really a pointer to a **chunk_info** object describing the chunks in the region. The low bits in p also describe the size of the chunks: if it's 1 then the chunks are zero bytes, otherwise the chunk size is 2^{bits-1}. The page the chunks are within is pointed to by the high bits of p as well as from the **chunk_info** object.

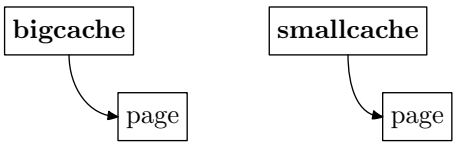
A **chunk_head** object is the head of a list of **chunk_info** object.



TODO: Is this still true now that chunks needn't be powers of two?
TODO: Double check that size really points to a **chunk_info** not a head.
TODO: Add large allocations to the diagram pointing left.
TODO: Find out what 'from' means for f (see also `STATS_SETF`?).

```
struct region_info {
    void *p;        /* page; low bits used to mark chunks */
    uintptr_t size;     /* size for pages, or chunk_info pointer */
#ifdef MALLOC_STATS
    void *f;        /* where allocated from */
#endif
};
LIST_HEAD(chunk_head, chunk_info);      /* Declare struct chunk_head. */
```

10. When an allocation is freed its mapping may be cached rather than being released immediately. There are two caches, for large allocations and small. Unlike the rest of the allocator the sizes refered to are in multiples of pages, not bytes so the largest cached small allocation will be `MAX_SMALLCACHEABLE_SIZE` $*$ `MALLOC_PAGESHIFT`, or $32 * 2^{12} = 128$KiB on common systems.



WIP

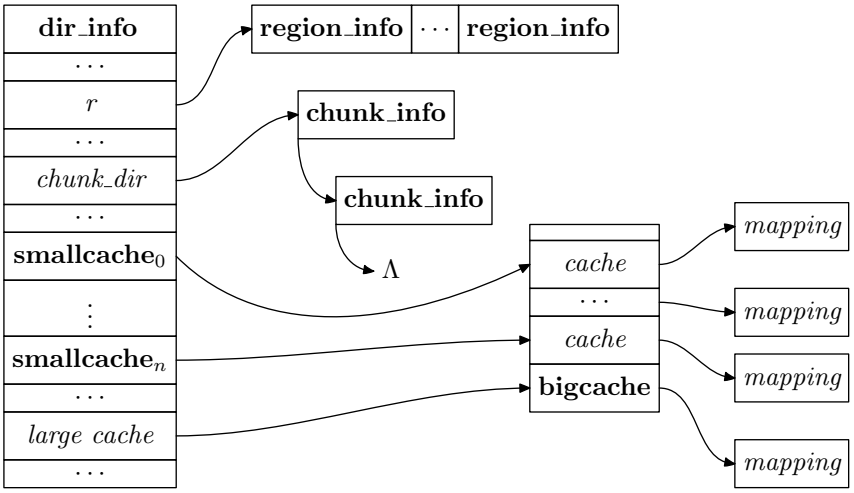
Two caches, one for "small" regions, one for "big". Small cache is an array per size, big cache is one array with different sized regions

If the total Nº of pages is larger than `BIGCACHE_FILL`, evict before inserting.

TODO: This diagram is awful.

```
#define MAX_SMALLCACHEABLE_SIZE 32
#define MAX_BIGCACHEABLE_SIZE 512
#define BIGCACHE_FILL(sz) (MAX_BIGCACHEABLE_SIZE * (sz)/4)
struct smallcache {
    void **pages;
    ushort length;
    ushort max;
};
struct bigcache {
    void *page;
    size_t psize;
};
```

11. Support is included for multiple threads using the allocator independently, including the ability to move allocations between threads, by maintaining several *dir_info* objects or "pools" and using locks to ensure a single thread accesses each one at a time.




```
struct dir_info {
    u_int32_t canary1;
    int active;      /* status of malloc */
    struct region_info *r;      /* region slots */
    size_t regions_total;      /* number of region slots */
    size_t regions_free;      /* number of free slots */
    size_t rbytesused;      /* random bytes used */
    char *func;      /* current function */
    int malloc_junk;      /* junk fill? */
    int mmap_flag;      /* extra flag for mmap */
    int mutex;
    int malloc_mt;      /* multi-threaded mode? */
    struct chunk_head chunk_info_list[BUCKETS + 1];      /* lists of free chunk info structs */
    struct chunk_head
        chunk_dir[BUCKETS + 1][MALLOC_CHUNK_LISTS];      /* lists of chunks with free slots */
    void *delayed_chunks[MALLOC_DELAYED_CHUNK_MASK + 1];      /* delayed free chunk slots */
    u_char rbytes[32];      /* random bytes */
    struct smallcache smallcache[MAX_SMALLCACHEABLE_SIZE];      /* free pages cache */
    size_t bigcache_used;
    size_t bigcache_size;
    struct bigcache *bigcache;
    void *chunk_pages;
    size_t chunk_pages_used;
#ifdef MALLOC_STATS
    size_t inserts;
    size_t insert_collisions;
    size_t finds;
    size_t find_collisions;
    size_t deletes;
    size_t delete_moves;
    size_t cheap_realloc_tries;
    size_t cheap_reallocs;
    size_t malloc_used;      /* bytes allocated */
    size_t malloc_guarded;      /* bytes used for guards */
    size_t pool_searches;      /* searches for pool */
    size_t other_pool;      /* searches in other pool */
#define STATS_ADD(x,y)  ((x) += (y))
```

```
#define STATS_SUB(x, y) ((x) -= (y))
#define STATS_INC(x) ((x)++)
#define STATS_ZERO(x) ((x) = 0)
#define STATS_SETF(x, y) ((x)->f = (y))
#else
#define STATS_ADD (x, y)      /* nothing */
#define STATS_SUB (x, y)      /* nothing */
#define STATS_INC (x)      /* nothing */
#define STATS_ZERO (x)      /* nothing */
#define STATS_SETF (x, y)      /* nothing */
#endif      /* MALLOC_STATS */
    u_int32_t canary2;
  };
```

12. Why is the *unmap* function is declared here?

```
static void unmap(struct dir_info *d, void *p, size_t sz, size_t clear);
```

13. A **chunk_info** object describes how a page has been divided into chunks. The space allocated for each object is larger than this structure suggests — the final element is treated as a "flexible array member" (**u_short** *bits*[];) supporting compilers that don't include those and is a bitmap describing which chunk are free. `MALLOC_BITS` is how many bits per **u_short** in the bitmap.

```
#define MALLOC_BITS (NBBY * sizeof(u_short))
  struct chunk_info {
    LIST_ENTRY (chunk_info) entries;
    void *page;      /* pointer to the page */
    u_short canary;
    u_short bucket;
    u_short free;      /* how many free chunks */
    u_short total;      /* how many chunks */
    u_short offset;      /* requested size table offset */
    u_short bits[1];      /* which chunks are free */
  };
```

14. There is a single instance of this *malloc_readonly* object that describes the truly global state, in particular an array of the active and inactive pools.

The "*malloc_readonly*" symbol refers to two things: the definition of this object and the variable that is the single instance of it. However the variable is rarely used directly as it's really a union with some padding to ensure it occupies a whole page, and the symbol *mopts* hides this detail.

As part of the initialisation the page this occupies is mapped `PROT_READ` to prevent tampering.

TODO: describe `.openbsd.mutable` & `_MALLOC_MUTEXES`.

```
struct malloc_readonly {
    struct dir_info *malloc_pool[_MALLOC_MUTEXES];      /* Main bookkeeping information */
    u_int malloc_mutexes;       /* how much in actual use? */
    int malloc_freecheck;       /* Extensive double free check */
    int malloc_freeunmap;       /* mprotect free pages PROT_NONE? */
    int def_malloc_junk;        /* junk fill? */
    int malloc_realloc;         /* always realloc? */
    int malloc_xmalloc;         /* xmalloc behaviour? */
    u_int chunk_canaries;       /* use canaries after chunks? */
    int internal_funcs;         /* use better recallocarray/freezero? */
    u_int def_maxcache;         /* free pages we cache */
    u_int junk_loc;         /* variation in location of junk */
    size_t malloc_guard;        /* use guard pages after allocations? */
#ifdef MALLOC_STATS
    int malloc_stats;       /* dump leak report at end */
    int malloc_verbose;         /* dump verbose statistics at end */
#define DO_STATS mopts.malloc_stats
#else
#define DO_STATS 0
#endif
    u_int32_t malloc_canary;        /* Matched against ones in pool */
};
static union {
    struct malloc_readonly mopts;
    u_char _pad[MALLOC_PAGESIZE];
} malloc_readonly
    __attribute__((aligned(MALLOC_PAGESIZE)))
    __attribute__((section(".openbsd.mutable")));
#define mopts malloc_readonly.mopts
```

15. Compile-time options which can be set by a program by declaring "**extern char** **malloc_options*" and setting the variable to a string constant that will be scanned by *omalloc_parsopt*.

```
char *malloc_options;
```

16. *wrterror* is used widely to display errors as they're encountered. *malloc_exit* is installed by `atexit(3)` to run *malloc_dump* when the process exits.

`PROTO_NORMAL` is part of OpenBSD's method to handle symbols shared between user-space, the kernel and the C compiler and is described in `/usr/src/lib/libc/include/README`.

```
static __dead void wrterror(struct dir_info *d, char *msg, ...)
    __attribute__((__format__(printf, 2, 3)));
#ifdef MALLOC_STATS
void malloc_dump(void);
PROTO_NORMAL(malloc_dump);
static void malloc_exit(void);
#endif
```

17. This function uses the GCC/Clang extension `__builtin_return_address(3)` to build a trace of stack frames of the callers to malloc.

```
#if defined (__aarch64__) ∨ defined (__amd64__) ∨ defined (__arm__)
  static inline void *caller(void)
  {
    void *p;
    switch (DO_STATS) {
    case 0: default: return Λ;
    case 1: p = __builtin_return_address(0);
      break;
    case 2: p = __builtin_return_address(1);
      break;
    case 3: p = __builtin_return_address(2);
      break;
    }
    return __builtin_extract_return_addr(p);
  }
#else
  static inline void *caller(void)
  {
    return DO_STATS ≡ 0 ? Λ : __builtin_extract_return_addr(__builtin_return_address(0));
  }
#endif
```

18. As per the description of regions, the low bits of $r\rightarrow p$ determine the size of a region: 0 means $r\rightarrow size$ holds the real size which will $\geq$ the page size, otherwise low bits is the bucket + 1.

```
#define REALSIZE(sz, r)(sz) = (uintptr_t)(r)→p & MALLOC_PAGEMASK,
      (sz) = ((sz) ≡ 0 ? (r)→size : B2SIZE((sz) − 1))
```

19. The lookup key is calculated from the address with this simple but fast algorithm.

```
  static inline size_t hash(void *p)
  {
    size_t sum;
    uintptr_t u;
    u = (uintptr_t) p ≫ MALLOC_PAGESHIFT;
    sum = u;
    sum = (sum ≪ 7) − sum + (u ≫ 16);
#ifdef __LP64__
    sum = (sum ≪ 7) − sum + (u ≫ 32);
    sum = (sum ≪ 7) − sum + (u ≫ 48);
#endif
    return sum;
  }
```

20. The first thing most malloc actions need to do with an unknown address is find the pool it came from, if there are multiple pools.

There will be multiple pools if one or more threads have been spawned otherwise there will actually be two but the first pool (zero) is always reserved for ... malloc internal? TODO.

```
  static inline struct dir_info *getpool(void)
  {
    if (mopts.malloc_pool[1] ≡ Λ ∨ ¬mopts.malloc_pool[1]→malloc_mt)
      return mopts.malloc_pool[1];
    else      /* first one reserved for special pool */
      return mopts.malloc_pool[1 + TIB_GET( )→tib_tid % (mopts.malloc_mutexes − 1)];
  }
```

21. Here is the error-reporting function *wrterror*. It obviously must not call anything which could recurse back into malloc.

```
static __dead void wrterror(struct dir_info *d, char *msg, ...)
{
  int saved_errno = errno;
  va_list ap;
  dprintf(STDERR_FILENO, "%s(%d) in %s(): ", __progname, getpid(),
      (d ≠ Λ ∧ d→func) ? d→func : "unknown");
  va_start(ap, msg);
  vdprintf(STDERR_FILENO, msg, ap);
  va_end(ap);
  dprintf(STDERR_FILENO, "\n");
#ifdef MALLOC_STATS
  if (DO_STATS ∧ mopts.malloc_verbose) malloc_dump();
#endif
  errno = saved_errno;
  abort();
}
```

22. Random bytes are used for various purposes within malloc. They are collected en masse in a buffer within each pool, doled out by *getrbyte* and replenished by *rbytes_init*.

```
static void rbytes_init(struct dir_info *d)
{
  arc4random_buf(d→rbytes, sizeof (d→rbytes));
  d→rbytesused = 1 + d→rbytes[0] % (sizeof (d→rbytes)/2);      /* +1 to account for this d→rbytes[0] */
}
static inline u_char getrbyte(struct dir_info *d)
{
  u_char x;
  if (d→rbytesused ≥ sizeof (d→rbytes)) rbytes_init(d);
  x = d→rbytes[d→rbytesused++];
  return x;
}
```

23. Application of the run-time options, called repeatedly by *omalloc_init* below.

```
static void omalloc_parseopt(char opt)
{
  switch (opt) {
  case '+': mopts.malloc_mutexes <<= 1;
    if (mopts.malloc_mutexes > _MALLOC_MUTEXES) mopts.malloc_mutexes = _MALLOC_MUTEXES;
    break;
  case '-': mopts.malloc_mutexes >>= 1;
    if (mopts.malloc_mutexes < 2) mopts.malloc_mutexes = 2;
    break;
  case '>': mopts.def_maxcache <<= 1;
    if (mopts.def_maxcache > MALLOC_MAXCACHE) mopts.def_maxcache = MALLOC_MAXCACHE;
    break;
  case '<': mopts.def_maxcache >>= 1; break;
  case 'c': mopts.chunk_canaries = 0; break;
  case 'C': mopts.chunk_canaries = 1; break;
#ifdef MALLOC_STATS
  case 'd': mopts.malloc_stats = 0; break;
  case 'D':
  case '1': mopts.malloc_stats = 1; break;
  case '2': mopts.malloc_stats = 2; break;
  case '3': mopts.malloc_stats = 3; break;
#endif /* MALLOC_STATS */
  case 'f': mopts.malloc_freecheck = 0; mopts.malloc_freeunmap = 0; break;
  case 'F': mopts.malloc_freecheck = 1; mopts.malloc_freeunmap = 1; break;
  case 'g': mopts.malloc_guard = 0; break;
  case 'G': mopts.malloc_guard = MALLOC_PAGESIZE; break;
  case 'j':
    if (mopts.def_malloc_junk > 0) mopts.def_malloc_junk--;
    break;
  case 'J':
    if (mopts.def_malloc_junk < 2) mopts.def_malloc_junk++;
    break;
  case 'r': mopts.malloc_realloc = 0; break;
  case 'R': mopts.malloc_realloc = 1; break;
  case 'u': mopts.malloc_freeunmap = 0; break;
  case 'U': mopts.malloc_freeunmap = 1; break;
#ifdef MALLOC_STATS
  case 'v': mopts.malloc_verbose = 0; break;
  case 'V': mopts.malloc_verbose = 1; break;
#endif /* MALLOC_STATS */
  case 'x': mopts.malloc_xmalloc = 0; break;
  case 'X': mopts.malloc_xmalloc = 1; break;
  default:
    dprintf(STDERR_FILENO, "malloc() warning: " "unknown char in MALLOC_OPTIONS\n");
    break;
  }
}
```

24. The first time any malloc function is called it initialises the allocater with this function. To find options it looks in turn at `sysctl(2)`, the environment and then the program.

TODO: Why is *junk_loc* allowed to be zero? *malloc_canary* is also used as a sentinel by *_malloc_init* to determine whether to initialise and *chunk_canaries* obviously needs a value to do its job.

```
static void omalloc_init(void)
{
  char *p, *q, b[16];
  int i, j;
  const int mib[2] = {CTL_VM, VM_MALLOC_CONF};
  size_t sb;      /* * Default options */
  mopts.malloc_mutexes = 8;
  mopts.def_malloc_junk = 1;
  mopts.def_maxcache = MALLOC_DEFAULT_CACHE;
  for (i = 0; i < 3; i++) {
    switch (i) {
    case 0: sb = sizeof (b);
      j = sysctl(mib, 2, b, &sb, Λ, 0);
      if (j ≠ 0) continue;
      p = b;
      break;
    case 1:
      if (issetugid( ) ≡ 0) p = getenv("MALLOC_OPTIONS");
      else continue;
      break;
    case 2: p = malloc_options;
      break;
    default: p = Λ;
    }
    for ( ; p ≠ Λ ∧ *p ≠ '\0'; p++) {
      switch (*p) {
      case 'S':
        for (q = "CFGJ"; *q ≠ '\0'; q++) omalloc_parseopt(*q);
        mopts.def_maxcache = 0;
        break;
      case 's':
        for (q = "cfgj"; *q ≠ '\0'; q++) omalloc_parseopt(*q);
        mopts.def_maxcache = MALLOC_DEFAULT_CACHE;
        break;
      default: omalloc_parseopt(*p);
        break;
      }
    }
  }
```

```
#ifdef MALLOC_STATS
    if (DO_STATS ∧ (atexit(malloc_exit) ≡ −1)) {
      dprintf(STDERR_FILENO, "malloc()␣warning:␣atexit(2)␣failed."
          "␣Will␣not␣be␣able␣to␣dump␣stats␣on␣exit\n");
    }
#endif
    while ((mopts.malloc_canary = arc4random( )) ≡ 0)
      ;
    mopts.junk_loc = arc4random( );
    if (mopts.chunk_canaries)
      do {
        mopts.chunk_canaries = arc4random( );
      } while ((u_char) mopts.chunk_canaries ≡ 0 ∨
            (u_char) mopts.chunk_canaries ≡ SOME_FREEJUNK);
  }
```

25. Initialising a new pool. Each pool's canaries differ by calculating them as the global canary `XOR` the value of the pool's address for *canary1* and its complement for *canary2*.

```
  static void omalloc_poolinit(struct dir_info *d, int mmap_flag)
  {
    int i, j;
    d→r = Λ;
    d→rbytesused = sizeof (d→rbytes);
    d→regions_free = d→regions_total = 0;
    for (i = 0; i ≤ BUCKETS; i++) {
      LIST_INIT(&d→chunk_info_list[i]);
      for (j = 0; j < MALLOC_CHUNK_LISTS; j++) LIST_INIT(&d→chunk_dir[i][j]);
    }
    d→mmap_flag = mmap_flag;
    d→malloc_junk = mopts.def_malloc_junk;
    d→canary1 = mopts.malloc_canary ⊕ (u_int32_t)(uintptr_t) d;
    d→canary2 = ∼d→canary1;
  }
```

26. Large regions are stored in a hash table. This function copies entries into a new larger table or creates the table for the first time. An allocation for the new table is obtained fresh from the kernel so that use-after-free bugs cannot affect its contents but the old table is put back into the cache of reusable regions like any other.

```
static int omalloc_grow(struct dir_info *d)
{
  size_t newtotal;
  size_t newsize;
  size_t mask;
  size_t i, oldpsz;
  struct region_info *p;
  if (d->regions_total > SIZE_MAX/sizeof(struct region_info)/2) return 1;
  newtotal = d->regions_total == 0 ? MALLOC_INITIAL_REGIONS : d->regions_total * 2;
  newsize = PAGEROUND(newtotal * sizeof(struct region_info));
  mask = newtotal - 1;
  p = MMAP(newsize, d->mmap_flag);      /* Don't use cache here, we don't want user uaf touch this */
  if (p == MAP_FAILED) return 1;
  STATS_ADD(d->malloc_used, newsize);
  STATS_ZERO(d->inserts);
  STATS_ZERO(d->insert_collisions);
  for (i = 0; i < d->regions_total; i++) {
    void *q = d->r[i].p;
    if (q != 0) {
      size_t index = hash(q) & mask;
      STATS_INC(d->inserts);
      while (p[index].p != 0) {
        index = (index - 1) & mask;
        STATS_INC(d->insert_collisions);
      }
      p[index] = d->r[i];
    }
  }
  if (d->regions_total > 0) {
    oldpsz = PAGEROUND(d->regions_total * sizeof(struct region_info));
    unmap(d, d->r, oldpsz, oldpsz);      /* clear to avoid meta info ending up in the cache */
  }
  d->regions_free += newtotal - d->regions_total;
  d->regions_total = newtotal;
  d->r = p;
  return 0;
}
```

27. Insert a new region entry. A slot is in use if the **region_info** object there has a value (p). This holds since non-`MAP_FIXED` mappings with hint 0 start at `BRKSIZ` — the address of the start of "the break" (`brk(2)`) as it was known of yore, now called the heap.

```
static int insert(struct dir_info *d, void *p, size_t sz, void *f)
{
  size_t index;
  size_t mask;
  void *q;
  if (d->regions_free * 4 < d->regions_total ∨ d->regions_total ≡ 0) {
    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 ≠ Λ) {
    index = (index − 1) & mask;
    q = d->r[index].p;
    STATS_INC(d->insert_collisions);
  }
  d->r[index].p = p;
  d->r[index].size = sz;
  STATS_SETF(&d->r[index], f);
  d->regions_free −−;
  return 0;
}
```

28. Locate an existing region entry in the table.

```
static struct region_info *find(struct dir_info *d, void *p)
{
  size_t index;
  size_t mask = d->regions_total − 1;
  void *q, *r;
  if (mopts.malloc_canary ≠ (d->canary1 ⊕ (u_int32_t)(uintptr_t) d) ∨ d->canary1 ≠ ∼d->canary2)
    wrterror(d, "internal struct corrupt");
  if (d->r ≡ Λ) return Λ;
  p = MASK_POINTER(p);
  index = hash(p) & mask;
  r = d->r[index].p;
  q = MASK_POINTER(r);
  STATS_INC(d->finds);
  while (q ≠ p ∧ r ≠ Λ) {
    index = (index − 1) & mask;
    r = d->r[index].p;
    q = MASK_POINTER(r);
    STATS_INC(d->find_collisions);
  }
  return (q ≡ p ∧ r ≠ Λ) ? &d->r[index] : Λ;
}
```

29. Delete a region entry from the table. The table is not reduced in size.

```
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;
  if (d->regions_total & (d->regions_total - 1)) wrterror(d, "regions_total not 2^x");
  d->regions_free++;
  STATS_INC(d->deletes);
  i = ri - d->r;
  for ( ; ; ) {
    d->r[i].p = Λ;
    d->r[i].size = 0;
    j = i;
    for ( ; ; ) {
      i = (i - 1) & mask;
      if (d->r[i].p ≡ Λ) 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];
      STATS_INC(d->delete_moves);
      break;
    }
  }
}
```

30. This function fills an area with junk for use-after-free and/or uninitialised-use detection.

```
static inline void junk_free(int junk, void *p, size_t sz)
{
  size_t i, step = 1;
  uint64_t *lp = p;
  if (junk ≡ 0 ∨ sz ≡ 0) return;
  sz /= sizeof(uint64_t);
  if (junk ≡ 1) {
    if (sz > MALLOC_PAGESIZE/sizeof(uint64_t)) sz = MALLOC_PAGESIZE/sizeof(uint64_t);
    step = sz/4;
    if (step ≡ 0) step = 1;
  }   /* Do not always put the free junk bytes in the same spot. There is modulo bias here, but
         we ignore that. */
  for (i = mopts.junk_loc % step; i < sz; i += step) lp[i] = SOME_FREEJUNK_ULL;
}
```

31. Check if some junk has been erroneously overwritten and abort with an error message if it has.

```
static inline void validate_junk(struct dir_info *pool, void *p, size_t sz)
{
  size_t i, step = 1;
  uint64_t *lp = p;
  if (pool->malloc_junk ≡ 0 ∨ sz ≡ 0) return;
  sz /= sizeof(uint64_t);
  if (pool->malloc_junk ≡ 1) {
    if (sz > MALLOC_PAGESIZE/sizeof(uint64_t)) sz = MALLOC_PAGESIZE/sizeof(uint64_t);
    step = sz/4;
    if (step ≡ 0) step = 1;
  }     /* see junk_free */
  for (i = mopts.junk_loc % step; i < sz; i += step) {
    if (lp[i] ≠ SOME_FREEJUNK_ULL) wrterror(pool, "write␣after␣free␣%p", p);
  }
}
```

32. When the program frees a large allocation — of a page or more — or when malloc releases an allocation it used internally the mapping originally acquired from the kernel may be cached rather then returned to the kernel straight away.

Two caches are maintained for allocations larger or smaller than `MAX_SMALLCACHEABLE_SIZE`. Allocations larger than `MAX_BIGCACHEABLE_SIZE` are always returned as soon as they're freed.

Opposed to the regular region data structure, the sizes in the cache are in `MALLOC_PAGESIZE` units.

```
static void unmap(struct dir_info *d, void *p, size_t sz, size_t clear)
{
  size_t psz = sz >> MALLOC_PAGESHIFT;
  void *r;
  u_short i;
  struct smallcache *cache;
  if (sz != PAGEROUND(sz) || psz == 0) wrterror(d, "munmap round");
  if (d->bigcache_size > 0 && psz > MAX_SMALLCACHEABLE_SIZE && psz <= MAX_BIGCACHEABLE_SIZE) {
    u_short base = getrbyte(d);       /* start in a random cache slot */
    u_short j;
    for (j = 0; j < d->bigcache_size/4; j++) {       /* don't look through all slots */
      i = (base + j) & (d->bigcache_size - 1);
      if (d->bigcache_used < BIGCACHE_FILL(d->bigcache_size)) {       /* if there's space ... */
        if (d->bigcache[i].psize == 0) break;       /* use an empty slot */
      }
      else {
        if (d->bigcache[i].psize != 0) break;       /* otherwise reclaim one */
      }
    }
    if (d->bigcache[i].psize != 0) {       /* if we didn't find a preferred slot, use random one */
      size_t tmp;
      r = d->bigcache[i].page;
      d->bigcache_used -= d->bigcache[i].psize;
      tmp = d->bigcache[i].psize << MALLOC_PAGESHIFT;
      if (!mopts.malloc_freeunmap) validate_junk(d, r, tmp);
      if (munmap(r, tmp))       /* evict the previously cached region */
        wrterror(d, "munmap %p", r);
      STATS_SUB(d->malloc_used, tmp);
    }
    if (clear > 0) explicit_bzero(p, clear);
    if (mopts.malloc_freeunmap) {
      if (mprotect(p, sz, PROT_NONE)) wrterror(d, "mprotect %p", r);
    }
    else junk_free(d->malloc_junk, p, sz);
    d->bigcache[i].page = p;
    d->bigcache[i].psize = psz;
    d->bigcache_used += psz;
    return;
  }
```

```
  if (psz > MAX_SMALLCACHEABLE_SIZE ∨ d→smallcache[psz − 1].max ≡ 0) {
    if (munmap(p, sz)) wrterror(d, "munmap␣%p", p);
    STATS_SUB(d→malloc_used, sz);
    return;
  }
  cache = &d→smallcache[psz − 1];
  if (cache→length ≡ cache→max) {      /* use a random slot */
    int fresh;
    i = getrbyte(d) & (cache→max − 1);
    r = cache→pages[i];
    fresh = (uintptr_t) r & 1;
    *(uintptr_t *) &r &= ∼1ULL;
    if (¬fresh ∧ ¬mopts.malloc_freeunmap) validate_junk(d, r, sz);
    if (munmap(r, sz)) wrterror(d, "munmap␣%p", r);
    STATS_SUB(d→malloc_used, sz);
    cache→length−−;
  }
  else i = cache→length;
  if (clear > 0) explicit_bzero(p, clear);
  if (mopts.malloc_freeunmap) mprotect(p, sz, PROT_NONE);
  else junk_free(d→malloc_junk, p, sz);
  cache→pages[i] = p;      /* fill slot */
  cache→length++;
}
```

33. Before requesting a new mapping from the kernel the cache is searched for a previously freed allocation that's big enough.

After acquiring a mapping from the kernel in *map* the new allocation is inserted into the pool's hashtable. After *unmap*, whether cached or not, the allocation is deleted. Thus an allocation is never (TODO: check) in a pool's hashtable and the cache simultaneously.

```
static void *map(struct dir_info *d, size_t sz, int zero_fill)
{
  size_t psz = sz >> MALLOC_PAGESHIFT;
  u_short i;
  void *p;
  struct smallcache *cache;
  if (mopts.malloc_canary ≠ (d→canary1 ⊕ (u_int32_t)(uintptr_t) d) ∨ d→canary1 ≠ ∼d→canary2)
    wrterror(d, "internal␣struct␣corrupt");
  if (sz ≠ PAGEROUND(sz) ∨ psz ≡ 0) wrterror(d, "map␣round");
  if (d→bigcache_size > 0 ∧ psz > MAX_SMALLCACHEABLE_SIZE ∧ psz ≤ MAX_BIGCACHEABLE_SIZE) {
    size_t base = getrbyte(d);      /* start in a random cache slot */
    size_t cached = d→bigcache_used;
    ushort j;
    for (j = 0; j < d→bigcache_size ∧ cached ≥ psz; j++) {      /* do search all available slots */
      i = (j + base) & (d→bigcache_size − 1);
      if (d→bigcache[i].psize ≡ psz) {
        p = d→bigcache[i].page;
        d→bigcache_used −= psz;
        d→bigcache[i].page = Λ;
        d→bigcache[i].psize = 0;
        if (¬mopts.malloc_freeunmap) validate_junk(d, p, sz);
        if (mopts.malloc_freeunmap) mprotect(p, sz, PROT_READ | PROT_WRITE);
        if (zero_fill) memset(p, 0, sz);
        else if (mopts.malloc_freeunmap) junk_free(d→malloc_junk, p, sz);
        return p;
      }
      cached −= d→bigcache[i].psize;
    }
  }
```

```
if (psz ≤ MAX_SMALLCACHEABLE_SIZE ∧ d→smallcache[psz − 1].max > 0) {
  cache = &d→smallcache[psz − 1];
  if (cache→length > 0) {
    int fresh;
    if (cache→length ≡ 1) p = cache→pages[−− cache→length];
    else {
      i = getrbyte(d) % cache→length;
      p = cache→pages[i];
      cache→pages[i] = cache→pages[−− cache→length];
    }
    fresh = (uintptr_t) p & 1UL;      /* check if page was not junked, i.e. "fresh" ... */
    *(uintptr_t *) &p &= ∼1UL;      /* ... we use the lsb of the pointer for that */
    if (¬fresh ∧ ¬mopts.malloc_freeunmap) validate_junk(d, p, sz);
    if (mopts.malloc_freeunmap) mprotect(p, sz, PROT_READ | PROT_WRITE);
    if (zero_fill) memset(p, 0, sz);
    else if (mopts.malloc_freeunmap) junk_free(d→malloc_junk, p, sz);
    return p;
  }
  if (psz ≤ 1) {
    p = MMAP(cache→max * sz, d→mmap_flag);
    if (p ≠ MAP_FAILED) {
      STATS_ADD(d→malloc_used, cache→max * sz);
      cache→length = cache→max − 1;
      for (i = 0; i < cache→max − 1; i++) {
        void *q = (char *) p + i * sz;
        cache→pages[i] = q;
        *(uintptr_t *) &cache→pages[i] |= 1UL;      /* mark pointer in slot as not junked */
      }
      if (mopts.malloc_freeunmap) mprotect(p, (cache→max − 1) * sz, PROT_NONE);
      p = (char *) p + (cache→max − 1) * sz;
      return p;      /* zero fill not needed, freshly mmapped */
    }
  }
}
p = MMAP(sz, d→mmap_flag);
if (p ≠ MAP_FAILED) STATS_ADD(d→malloc_used, sz);
return p;      /* zero fill not needed, freshly mmapped */
}
```

34. This function initialises a new **chunk_info** object. As described above each such object is not as big as **sizeof**(**struct chunk_info**) but is extended to include a bitmap of available chunks.

TODO: howmany(9) is not documented in OpenBSD.

```
static void init_chunk_info(struct dir_info *d, struct chunk_info *p, u_int bucket)
{
  u_int i;
  p→bucket = bucket;
  p→total = p→free = MALLOC_PAGESIZE/B2ALLOC(bucket);
  p→offset = bucket ≡ 0 ? 0xdead : howmany(p→total, MALLOC_BITS);
  p→canary = (u_short) d→canary1;
  i = p→total − 1;      /* set only the valid bits in the bitmap */
  memset(p→bits, 0xff, sizeof (p→bits[0]) * (i/MALLOC_BITS));
  p→bits[i/MALLOC_BITS] = (2U ≪ (i % MALLOC_BITS)) − 1;
}
```

35. A set of **chunk_info** objects is allocated when necessary to describe chunks of a certain size (*bucket*) by dividing up a new page and adding them all to a list of free **chunk_info** objects. As with the region hash table this page is allocated fresh from the kernel to avoid accidental use-after-free bugs.

TODO: Only does that if there are none available (perhaps explain buckets here).

```
static struct chunk_info *alloc_chunk_info(struct dir_info *d, u_int bucket)
{
  struct chunk_info *p;
  if (LIST_EMPTY(&d->chunk_info_list[bucket])) {
    const size_t chunk_pages = 64;
    size_t size, count, i;
    char *q;
    count = MALLOC_PAGESIZE/B2ALLOC(bucket);
    size = howmany(count, MALLOC_BITS);
    size = sizeof(struct chunk_info) + (size - 1) * sizeof(u_short);
    if (mopts.chunk_canaries) size += count * sizeof(u_short);
    size = _ALIGN(size);
    count = MALLOC_PAGESIZE/size;
      /* Don't use cache here, we don't want user uaf touch this */
    if (d->chunk_pages_used == chunk_pages || d->chunk_pages == 0) {
      q = MMAP(MALLOC_PAGESIZE * chunk_pages, d->mmap_flag);
      if (q == MAP_FAILED) return 0;
      d->chunk_pages = q;
      d->chunk_pages_used = 0;
      STATS_ADD(d->malloc_used, MALLOC_PAGESIZE * chunk_pages);
    }
    q = (char *)d->chunk_pages + d->chunk_pages_used * MALLOC_PAGESIZE;
    d->chunk_pages_used++;
    for (i = 0; i < count; i++, q += size) {
      p = (struct chunk_info *)q;
      LIST_INSERT_HEAD(&d->chunk_info_list[bucket], p, entries);
    }
  }
  p = LIST_FIRST(&d->chunk_info_list[bucket]);
  LIST_REMOVE(p, entries);
  if (p->total == 0) init_chunk_info(d, p, bucket);
  return p;
}
```

36. The notion of buckets has been glossed over until now.

Allocations of less than half a page are obtained by allocating a whole page and dividing it in to two or more chunks. Unallocated chunks in the remainder of the page are stored in a free list akin to the cache of large freed allocations. Such a free list is called a bucket.

This function requests a page from the cache and divides it into chunks in a bucket list before returning one.

```
static struct chunk_info *omalloc_make_chunks(struct dir_info *d, u_int bucket, u_int listnum)
{
  struct chunk_info *bp;
  void *pp;
  pp = map(d, MALLOC_PAGESIZE, 0);      /* Allocate a new bucket */
  if (pp ≡ MAP_FAILED) return Λ;
      /* memory protect the page allocated in the malloc(0) case */
  if (bucket ≡ 0 ∧ mprotect(pp, MALLOC_PAGESIZE, PROT_NONE) ≡ −1) goto err;
  bp = alloc_chunk_info(d, bucket);
  if (bp ≡ Λ) goto err;
  bp→page = pp;
  if (insert(d, (void *)((uintptr_t) pp | (bucket + 1)), (uintptr_t) bp, Λ)) goto err;
  LIST_INSERT_HEAD(&d→chunk_dir[bucket][listnum], bp, entries);
  if (bucket > 0 ∧ d→malloc_junk ≠ 0) memset(pp, SOME_FREEJUNK, MALLOC_PAGESIZE);
  return bp;
err: unmap(d, pp, MALLOC_PAGESIZE, 0);
  return Λ;
}
```

37. A function to work out how many bits a number needs to store it, an operation that in this the year of our Lord 2024 still requires all this noise.

TODO: This combines two preprocessor alternative function techniques when one will do (there is one earlier too).

```
#if defined (__GNUC__) ∧ __GNUC__ < 4
  static inline unsigned int lb(u_int x)
  {
#if defined (__m88k__)
    __asm__ __volatile__("ff1␣%0,␣%0": "=r"(x):"0"(x));
    return x;
#else     /* portable version */
    unsigned int count = 0;
    while ((x & (1U ≪ (sizeof(int) * CHAR_BIT − 1))) ≡ 0) {
      count++;
      x ≪= 1;
    }
    return (sizeof(int) * CHAR_BIT − 1) − count;
#endif
  }
#else     /* using built-in function version */
  static inline unsigned int lb(u_int x)
  {     /* I need an extension just for integer-length (: */
    return (sizeof(int) * CHAR_BIT − 1) − __builtin_clz(x);
  }
#endif
```

38. Storing a bucket list for each possible size between 0 and half a page would be hugely wasteful and slow so it makes a lot of sense to round up each size to a smaller set of predetermined sizes and group similarly-sized allocations within them. This naturally begs the question of what is the best size to make each successively-larger bucket.

OpenBSD determines the bucket sizes based on the "linear-log bucketing" algorithm described by Tony Finch at https://pvk.ca/Blog/2015/06/27/linear-log-bucketing-fast-versatile-simple/.

```
static inline unsigned int bin_of(unsigned int size)
{
  const unsigned int linear = 6;
  const unsigned int subbin = 2;
  unsigned int mask, range, rounded, sub_index, rounded_size;
  unsigned int n_bits, shift;

  n_bits = lb(size | (1U << linear));
  shift = n_bits - subbin;
  mask = (1ULL << shift) - 1;
  rounded = size + mask;      /* XXX: overflow. */
  sub_index = rounded >> shift;
  range = n_bits - linear;
  rounded_size = rounded & ~mask;
  return rounded_size;
}
```

39. Find the size of a bucket from the size of an allocation.

```
static inline u_short find_bucket(u_short size)
{
  if (size == 0) return 0;      /* malloc(0) is special */
  if (size < MALLOC_MINSIZE) size = MALLOC_MINSIZE;
  if (mopts.def_maxcache != 0) size = bin_of(size);
  return howmany(size, MALLOC_MINSIZE);
}
```

40. Allocations can be surrounded with "canary" bytes which are later checked and if they've been overwritten the process is aborted. This function fills in the bytes.

```
static void fill_canary(char *ptr, size_t sz, size_t allocated)
{
  size_t check_sz = allocated - sz;

  if (check_sz > CHUNK_CHECK_LENGTH) check_sz = CHUNK_CHECK_LENGTH;
  memset(ptr + sz, mopts.chunk_canaries, check_sz);
}
```

41. If an allocation request is small enough to be a chunk *malloc_bytes* finds one and returns it.
TODO: Are `ffs(3)` and *lb* related?

```
static void *malloc_bytes(struct dir_info *d, size_t size, void *f)
{
  u_int i, r, bucket, listnum;
  size_t k;
  u_short *lp;
  struct chunk_info *bp;
  void *p;
  if (mopts.malloc_canary ≠ (d→canary1 ⊕ (u_int32_t)(uintptr_t) d) ∨ d→canary1 ≠ ∼d→canary2)
    wrterror(d, "internal␣struct␣corrupt");
  bucket = find_bucket(size);
  r = ((u_int) getrbyte(d) ≪ 8) | getrbyte(d);
  listnum = r % MALLOC_CHUNK_LISTS;
  if ((bp = LIST_FIRST(&d→chunk_dir[bucket][listnum])) ≡ Λ) {      /* allocate a page of chunks */
    bp = omalloc_make_chunks(d, bucket, listnum);
    if (bp ≡ Λ) return Λ;
  }
  if (bp→canary ≠ (u_short) d→canary1) wrterror(d, "chunk␣info␣corrupted");
  i = (r/MALLOC_CHUNK_LISTS) % bp→total;      /* bias, as bp→total is not a power of 2 */
  lp = &bp→bits[i/MALLOC_BITS];      /* potentially start somewhere in a short */
  if (*lp) {
    int j = i % MALLOC_BITS;      /* j must be signed */
    k = ffs(*lp ≫ j);      /* TODO: doesn't use lb? */
    if (k ≠ 0) {
      k += j − 1;
      goto found;
    }
  }
  i /= MALLOC_BITS;      /* no bit halfway, go to next full short */
  for ( ; ; ) {
    if (++i ≥ howmany(bp→total, MALLOC_BITS)) i = 0;
    lp = &bp→bits[i];
    if (*lp) {
      k = ffs(*lp) − 1;
      break;
    }
  }
found:
  if (i ≡ 0 ∧ k ≡ 0 ∧ DO_STATS) {
    struct region_info *r = find(d, bp→page);
    STATS_SETF(r, f);
  }
  *lp ⊕= 1 ≪ k;      /* toggle the in-use bitmap */
  if (−−bp→free ≡ 0) LIST_REMOVE(bp, entries);      /* If no more are free, remove from free-list */
  k += (lp − bp→bits) * MALLOC_BITS;      /* Adjust to the real offset of that chunk */
  if (mopts.chunk_canaries ∧ size > 0) bp→bits[bp→offset + k] = size;
  k *= B2ALLOC(bp→bucket);
  p = (char *) bp→page + k;
  if (bp→bucket > 0) {
    validate_junk(d, p, B2SIZE(bp→bucket));
    if (mopts.chunk_canaries) fill_canary(p, size, B2SIZE(bp→bucket));
  }
  return p;
}
```

42. When an allocation is being freed its canaries are validated using this function.

```
static void validate_canary(struct dir_info *d, u_char *ptr, size_t sz, size_t allocated)
{
  size_t check_sz = allocated - sz;
  u_char *p, *q;
  if (check_sz > CHUNK_CHECK_LENGTH) check_sz = CHUNK_CHECK_LENGTH;
  p = ptr + sz;
  q = p + check_sz;
  while (p < q) {
    if (*p ≠ (u_char) mopts.chunk_canaries ∧ *p ≠ SOME_JUNK) {
      wrterror(d, "canary␣corrupted␣%p␣%#tx@%#zx%s", ptr, p - ptr, sz,
          *p ≡ SOME_FREEJUNK ? "␣(double␣free?)" : "");
    }
    p++;
  }
}
```

43. Find the chunk number on the page.

```
static uint32_t find_chunknum(struct dir_info *d, struct chunk_info *info, void *ptr,
    int check)
{
  uint32_t chunknum;
  if (info→canary ≠ (u_short) d→canary1) wrterror(d, "chunk␣info␣corrupted");
  chunknum = ((uintptr_t) ptr & MALLOC_PAGEMASK)/B2ALLOC(info→bucket);
  if ((uintptr_t) ptr & (MALLOC_MINSIZE - 1)) wrterror(d, "modified␣chunk-pointer␣%p", ptr);
  if (info→bits[chunknum/MALLOC_BITS] & (1U ≪ (chunknum % MALLOC_BITS)))
    wrterror(d, "double␣free␣%p", ptr);
  if (check ∧ info→bucket > 0) {
    validate_canary(d, ptr, info→bits[info→offset + chunknum], B2SIZE(info→bucket));
  }
  return chunknum;
}
```

44. Free a chunk, by marking its **chunk_info** object, and the page it's on if the page becomes empty.

```
static void free_bytes(struct dir_info *d, struct region_info *r, void *ptr)
{
  struct chunk_head *mp;
  struct chunk_info *info;
  uint32_t chunknum;
  uint32_t listnum;
  info = (struct chunk_info *) r->size;
  chunknum = find_chunknum(d, info, ptr, 0);
  if (chunknum ≡ 0) STATS_SETF(r, Λ);
  info->bits[chunknum/MALLOC_BITS] |= 1U ≪ (chunknum % MALLOC_BITS);
  info->free++;
  if (info->free ≡ 1) {      /* Page became non-full */
    listnum = getrbyte(d) % MALLOC_CHUNK_LISTS;
    mp = &d->chunk_dir[info->bucket][listnum];
    LIST_INSERT_HEAD(mp, info, entries);
    return;
  }
  if (info->free ≠ info->total) return;
  LIST_REMOVE(info, entries);
  if (info->bucket ≡ 0 ∧ ¬mopts.malloc_freeunmap)
    mprotect(info->page, MALLOC_PAGESIZE, PROT_READ | PROT_WRITE);
  unmap(d, info->page, MALLOC_PAGESIZE, 0);
  delete(d, r);
  mp = &d->chunk_info_list[info->bucket];
  LIST_INSERT_HEAD(mp, info, entries);
}
```

45. To find a chunked allocation *omalloc* defers to *malloc_bytes* above.

A larger allocation is searched for via the cache and returned to the pool's hash table. If the whole allocation including the optional guard page fits within a single page then it's shifted towards the end. This is one of many features introduced with the rewrite of malloc by Otto Moerbeek in CVS revision 1.92. It makes the base address of a large allocation (and thus by extension everything in it) slightly less predictable at very little cost.

```
static void *omalloc(struct dir_info *pool, size_t sz, int zero_fill, void *f)
{
  void *p;
  size_t psz;
  if (sz > MALLOC_MAXCHUNK) {
    if (sz ≥ SIZE_MAX − mopts.malloc_guard − MALLOC_PAGESIZE) {
      errno = ENOMEM;
      return Λ;
    }
    sz += mopts.malloc_guard;
    psz = PAGEROUND(sz);
    p = map(pool, psz, zero_fill);
    if (p ≡ MAP_FAILED) {
      errno = ENOMEM;
      return Λ;
    }
    if (insert(pool, p, sz, f)) {
      unmap(pool, p, psz, 0);
      errno = ENOMEM;
      return Λ;
    }
    if (mopts.malloc_guard) {
      if (mprotect((char *) p + psz − mopts.malloc_guard, mopts.malloc_guard, PROT_NONE))
        wrterror(pool, "mprotect");
      STATS_ADD(pool→malloc_guarded, mopts.malloc_guard);
    }
    if (MALLOC_MOVE_COND(sz)) {
      if (pool→malloc_junk ≡ 2)       /* fill whole allocation */
        memset(p, SOME_JUNK, psz − mopts.malloc_guard);
      p = MALLOC_MOVE(p, sz);       /* shift towards the end */
      if (zero_fill ∧ pool→malloc_junk ≡ 2)       /* fill zeros if needed and overwritten above */
        memset(p, 0, sz − mopts.malloc_guard);
    }
    else {
      if (pool→malloc_junk ≡ 2) {
        if (zero_fill) memset((char *) p + sz − mopts.malloc_guard, SOME_JUNK, psz − sz);
        else memset(p, SOME_JUNK, psz − mopts.malloc_guard);
      }
      else if (mopts.chunk_canaries)
        fill_canary(p, sz − mopts.malloc_guard, psz − mopts.malloc_guard);
    }
  }
  else {
    p = malloc_bytes(pool, sz, f);       /* takes care of SOME_JUNK */
    if (zero_fill ∧ p ≠ Λ ∧ sz > 0) memset(p, 0, sz);
  }
  return p;
}
```

46. Common function to detect recursion. Only print the error message once, to avoid making the problem potentially worse.

```
static void malloc_recurse(struct dir_info *d)
{
  static int noprint;
  if (noprint == 0) {
    noprint = 1;
    wrterror(d, "recursive call");
  }
  d->active--;
  _MALLOC_UNLOCK(d->mutex);
  errno = EDEADLK;
}
```

47. I don't like having to break the original function into pieces like this but *_malloc_init* is particularly large and divides neatly.

The malloc initialisation function is called in two different contexts: the first time malloc is used by the process calling the front-end API defined in this file and again when "`rthreads`", OpenBSD's threading implementation, is being initialised — see `/usr/src/lib/librthread/rthread.c`. This distinction is neatly represented here where this function is split into two pages. The first half is concerned with the first initialisation of malloc immediately after the process starts[1] and the whole function is protected by a global lock.

```
void _malloc_init(int from_rthreads)
{
  u_int i, j, nmutexes;
  struct dir_info *d;
  _MALLOC_LOCK(1);
  ⟨ Initialise global allocator settings 48 ⟩
  ⟨ Initialise threaded allocator settings 49 ⟩
  _MALLOC_UNLOCK(1);
}
DEF_STRONG(_malloc_init);
```

[1] This is not strictly true — see `/usr/src/libexec/ld.so/malloc.c`.

48. After options are parsed by *omalloc_init* and the canaries prepared the page that the global malloc settings object, **malloc_readonly** or *mopts*, is in is protected by:

* guard pages, making it impossible to access adjacent memory,
* an immutable mapping, so that the mode bits of those pages cannot change,
* every pool (**dir_info** object) is moved to a random offset within its available space, making it hard to guess its location,
* being made read-only, so it can't be overwritten,
* junking doesn't apply because this allocation will never be freed.

The number of pools initialised is dictated by *mopts.malloc_mutexes* which can be set during the process initialisation to pay the threading initialisation cost up-front. See *omalloc_parseopt*'s `+` and `-` flags.

```
⟨ Initialise global allocator settings 48 ⟩ ≡
  if (¬from_rthreads ∧ mopts.malloc_pool[1]) {
    _MALLOC_UNLOCK(1);
    return;
  }
  if (¬mopts.malloc_canary) {
    char *p;
    size_t sz, d_avail;
    omalloc_init( );      /* Allocate dir_infos with a guard page on either side. Also randomise offset
        inside the page at which the dir_infos lay (subject to alignment by 1 ≪ MALLOC_MINSHIFT) */
    sz = mopts.malloc_mutexes * sizeof (*d) + 2 * MALLOC_PAGESIZE;
    if ((p = MMAPNONE(sz, 0)) ≡ MAP_FAILED) wrterror(Λ, "malloc_init␣mmap1␣failed");
    if (mprotect(p + MALLOC_PAGESIZE, mopts.malloc_mutexes * sizeof (*d), PROT_READ | PROT_WRITE))
      wrterror(Λ, "malloc_init␣mprotect1␣failed");
    if (mimmutable(p, sz)) wrterror(Λ, "malloc_init␣mimmutable1␣failed");
    d_avail = (((mopts.malloc_mutexes * sizeof (*d) + MALLOC_PAGEMASK) & ∼MALLOC_PAGEMASK) −
        (mopts.malloc_mutexes * sizeof (*d))) ≫ MALLOC_MINSHIFT;
    d = (struct dir_info *)(p + MALLOC_PAGESIZE + (arc4random_uniform(d_avail) ≪
        MALLOC_MINSHIFT));
    STATS_ADD(d[1].malloc_used, sz);
    for (i = 0; i < mopts.malloc_mutexes; i++)      /* save each pool's random offset */
      mopts.malloc_pool[i] = &d[i];
    mopts.internal_funcs = 1;
    if (((uintptr_t) &malloc_readonly & MALLOC_PAGEMASK) ≡ 0) {
      if (mprotect(&malloc_readonly, sizeof (malloc_readonly), PROT_READ))
        wrterror(Λ, "malloc_init␣mprotect␣r/o␣failed");
      if (mimmutable(&malloc_readonly, sizeof (malloc_readonly)))
        wrterror(Λ, "malloc_init␣mimmutable␣r/o␣failed");
    }
  }
```

This code is used in section 47.

49. As hinted to at the end there when `rthreads` is initialising malloc the main purpose it to initialise more pools so that each can be locked by different threads independently. This half of the initialisation routine initialises any pools which were allocated above, particularly the default first two, that are (now) needed.

```
⟨ Initialise threaded allocator settings 49 ⟩ ≡
  nmutexes = from_rthreads ? mopts.malloc_mutexes : 2;
  for (i = 0; i < nmutexes; i++) {
    d = mopts.malloc_pool[i];
    d→malloc_mt = from_rthreads;
    if (d→canary1 ≡ ~d→canary2) continue;
    if (i ≡ 0) {
      omalloc_poolinit(d, MAP_CONCEAL);
      d→malloc_junk = 2;
      d→bigcache_size = 0;
      for (j = 0; j < MAX_SMALLCACHEABLE_SIZE; j++) d→smallcache[j].max = 0;
    }
    else {
      size_t sz = 0;
      omalloc_poolinit(d, 0);
      d→malloc_junk = mopts.def_malloc_junk;
      d→bigcache_size = mopts.def_maxcache;
      for (j = 0; j < MAX_SMALLCACHEABLE_SIZE; j++) {
        d→smallcache[j].max = mopts.def_maxcache ≫ (j/8);
        sz += d→smallcache[j].max * sizeof(void *);
      }
      sz += d→bigcache_size * sizeof(struct bigcache);
      if (sz > 0) {
        void *p = MMAP(sz, 0);
        if (p ≡ MAP_FAILED) wrterror(Λ, "malloc_init␣mmap2␣failed");
        if (mimmutable(p, sz)) wrterror(Λ, "malloc_init␣mimmutable2␣failed");
        for (j = 0; j < MAX_SMALLCACHEABLE_SIZE; j++) {
          d→smallcache[j].pages = p;
          p = (char *)p + d→smallcache[j].max * sizeof(void *);
        }
        d→bigcache = p;
      }
    }
    d→mutex = i;
  }
```

This code is used in section 47.

50. Simple allocation. Every function that can be called by a malloc user begins and ends with this prologue and epilogue (or some close variant of them).

It expects p/d to be declared as a pointer to a **dir_info** object, in many cases it's a call to *getpool* which will return Λ if there aren't any pools and malloc needs to be initialised.

```
#define PROLOGUE(p, fn)
  d = (p);
  if (d ≡ Λ) {
    _malloc_init(0);
    d = (p);
  }
  _MALLOC_LOCK(d→mutex);
  d→func = fn;
  if (d→active++) {
    malloc_recurse(d);
    return Λ;
  }
#define EPILOGUE( )
  d→active--;
  _MALLOC_UNLOCK(d→mutex);
  if (r ≡ Λ ∧ mopts.malloc_xmalloc) wrterror(d, "out␣of␣memory");
  if (r ≠ Λ) errno = saved_errno;
```

51. The simplest interface. Like most of these functions that use the backend described above, little extra description is necessary.

```
void *malloc(size_t size)
{
  void *r;
  struct dir_info *d;
  int saved_errno = errno;
  PROLOGUE(getpool( ), "malloc")
  r = omalloc(d, size, 0, caller( ));
  EPILOGUE( )
  return r;
}
DEF_STRONG(malloc);
```

52. A variant of *malloc* which keeps the allocated memory out of core dumps. See `MAP_CONCEAL` in mmap(2).

```
void *malloc_conceal(size_t size)
{
  void *r;
  struct dir_info *d;
  int saved_errno = errno;
  PROLOGUE(mopts.malloc_pool[0], "malloc_conceal")
  r = omalloc(d, size, 0, caller( ));
  EPILOGUE( )
  return r;
}
DEF_WEAK(malloc_conceal);
```

53. This function locates the pool an allocation came from by transferring the current thread's control of the pool lock already held to each pool in turn until the allocation is found.

```
static struct region_info *findpool(void *p, struct dir_info *argpool,
    struct dir_info **foundpool, char **saved_function)
{
  struct dir_info *pool = argpool;
  struct region_info *r = find(pool, p);
  STATS_INC(pool->pool_searches);
  if (r ≡ Λ) {
    u_int i, nmutexes;
    nmutexes = mopts.malloc_pool[1]->malloc_mt ? mopts.malloc_mutexes : 2;
    STATS_INC(pool->other_pool);
    for (i = 1; i < nmutexes; i++) {
      u_int j = (argpool->mutex + i) & (nmutexes - 1);
      pool->active--;
      _MALLOC_UNLOCK(pool->mutex);
      pool = mopts.malloc_pool[j];
      _MALLOC_LOCK(pool->mutex);
      pool->active++;
      r = find(pool, p);
      if (r ≠ Λ) {
        *saved_function = pool->func;
        pool->func = argpool->func;
        break;
      }
    }
    if (r ≡ Λ) wrterror(argpool, "bogus␣pointer␣(double␣free?)␣%p", p);
  }
  *foundpool = pool;
  return r;
}
```

54. *ofree* is the basis of *free* and the reallocating functions that may need a larger or smaller pool. It's another extremely large function which lends itself well to being broken into sequential pieces.

```
static void ofree(struct dir_info **argpool, void *p, int clear, int check, size_t argsz)
{
  struct region_info *r;
  struct dir_info *pool;
  char *saved_function;
  size_t sz;
  ⟨ Find the allocation's pool and check its metadata 55 ⟩
  if (sz > MALLOC_MAXCHUNK) {⟨ Free a large allocation 56 ⟩}
  else {⟨ Free a small allocation 57 ⟩}
  if (*argpool ≠ pool) {      /* Set the caller's in-use pool */
    pool->func = saved_function;
    *argpool = pool;
  }
}
```

55. Here *clear* is set if `MAP_CONCEAL` on the pool indicates the allocation's backing store must be cleared, and if *freezero* was called that its second (size) argument is correct.

```
⟨ Find the allocation's pool and check its metadata 55 ⟩ ≡
  r = findpool(p, *argpool, &pool, &saved_function);
  REALSIZE(sz, r);
  if (pool→mmap_flag) {      /* pool has MAP_CONCEAL */
    clear = 1;
    if (¬check) {
      argsz = sz;
      if (sz > MALLOC_MAXCHUNK) argsz -= mopts.malloc_guard;
    }
  }
  if (check) {
    if (sz ≤ MALLOC_MAXCHUNK) {
      if (mopts.chunk_canaries ∧ sz > 0) {
        struct chunk_info *info = (struct chunk_info *) r→size;
        uint32_t chunknum = find_chunknum(pool, info, p, 0);
        if (info→bits[info→offset + chunknum] < argsz)
          wrterror(pool, "recorded size %hu" " < %zu", info→bits[info→offset + chunknum], argsz);
      }
      else {
        if (sz < argsz) wrterror(pool, "chunk size %zu < %zu", sz, argsz);
      }
    }
    else if (sz − mopts.malloc_guard < argsz) {
      wrterror(pool, "recorded size %zu < %zu", sz − mopts.malloc_guard, argsz);
    }
  }
```

This code is used in section 54.

56. A large allocation's real base address is calculated if the allocation has been shifted, the guard pages' protection bits are relaxed then a large allocation is finally released to the cache and removed from the pool's region hash table.

```
⟨ Free a large allocation 56 ⟩ ≡
  if (¬MALLOC_MOVE_COND(sz)) {
    if (r→p ≠ p) wrterror(pool, "bogus pointer %p", p);
    if (mopts.chunk_canaries)
      validate_canary(pool, p, sz − mopts.malloc_guard, PAGEROUND(sz − mopts.malloc_guard));
  }
  else {      /* shifted towards the end */
    if (p ≠ MALLOC_MOVE(r→p, sz)) wrterror(pool, "bogus moved pointer %p", p);
    p = r→p;
  }
  if (mopts.malloc_guard) {
    if (sz < mopts.malloc_guard) wrterror(pool, "guard size");
    if (¬mopts.malloc_freeunmap) {
      if (mprotect((char *) p + PAGEROUND(sz) − mopts.malloc_guard, mopts.malloc_guard,
            PROT_READ | PROT_WRITE)) wrterror(pool, "mprotect");
    }
    STATS_SUB(pool→malloc_guarded, mopts.malloc_guard);
  }
  unmap(pool, p, PAGEROUND(sz), clear ? argsz : 0);
  delete(pool, r);
```

This code is used in section 54.

57. Validate and optionally canary check. A small allocation is similarly located within its bucket page but rather than being released immediately may be added to a set of chunks to check "later" for corruption — the "delayed chunks" — including double-free, out-of-bounds writes and other bugs.

A random spot is selected in the small buffer of delayed chunks and the newly freed chunk placed there after validating and finally releasing any delayed chunk that was already there.

```
⟨ Free a small allocation 57 ⟩ ≡
  void *tmp;
  u_int i;
  struct chunk_info *info = (struct chunk_info *) r->size;
  if (B2SIZE(info->bucket) ≠ sz) wrterror(pool, "internal␣struct␣corrupt");
  find_chunknum(pool, info, p, mopts.chunk_canaries);
  if (mopts.malloc_freecheck) {
    for (i = 0; i ≤ MALLOC_DELAYED_CHUNK_MASK; i++) {
      tmp = pool->delayed_chunks[i];
      if (tmp ≡ p) wrterror(pool, "double␣free␣%p", p);
      if (tmp ≠ Λ) {
        size_t tmpsz;
        r = find(pool, tmp);
        if (r ≡ Λ) wrterror(pool, "bogus␣pointer␣(""double␣free?)␣%p", tmp);
        REALSIZE(tmpsz, r);
        validate_junk(pool, tmp, tmpsz);
      }
    }
  }
  if (clear ∧ argsz > 0) explicit_bzero(p, argsz);
  junk_free(pool->malloc_junk, p, sz);
  i = getrbyte(pool) & MALLOC_DELAYED_CHUNK_MASK;
  tmp = p;
  p = pool->delayed_chunks[i];
  if (tmp ≡ p) wrterror(pool, "double␣free␣%p", p);
  pool->delayed_chunks[i] = tmp;
  if (p ≠ Λ) {
    r = find(pool, p);
    if (r ≡ Λ) wrterror(pool, "bogus␣pointer␣(double␣free?)␣%p", p);
    if (¬mopts.malloc_freecheck) {
      REALSIZE(sz, r);
      validate_junk(pool, p, sz);
    }
    free_bytes(pool, r, p);
  }
```

This code is used in section 54.

58. The common *free* function.

```
void free(void *ptr)
{
  struct dir_info *d;
  int saved_errno = errno;
  if (ptr ≡ Λ) return;      /* This is legal. */
  d = getpool( );
  if (d ≡ Λ) wrterror(d, "free() called before allocation");
  _MALLOC_LOCK(d→mutex);
  d→func = "free";
  if (d→active++) {
    malloc_recurse(d);
    return;
  }
  ofree(&d, ptr, 0, 0, 0);
  d→active--;
  _MALLOC_UNLOCK(d→mutex);
  errno = saved_errno;
}
DEF_STRONG(free);
```

59. The *freezero* is a variant of *free* that ensures the allocation's memory is explicitly discarded. It must be called with the original size of the allocation under the assumption that such important data will be tracked carefully enough. *freezero_p* is a simpler legacy implementation of *freezero* that's been deprecated.

```
static void freezero_p(void *ptr, size_t sz)
{
  explicit_bzero(ptr, sz);
  free(ptr);
}
void freezero(void *ptr, size_t sz)
{
  struct dir_info *d;
  int saved_errno = errno;
  if (ptr ≡ Λ) return;      /* This is legal. */
  if (¬mopts.internal_funcs) {
    freezero_p(ptr, sz);
    return;
  }
  d = getpool( );
  if (d ≡ Λ) wrterror(d, "freezero() called before allocation");
  _MALLOC_LOCK(d→mutex);
  d→func = "freezero";
  if (d→active++) {
    malloc_recurse(d);
    return;
  }
  ofree(&d, ptr, 1, 1, sz);
  d→active--;
  _MALLOC_UNLOCK(d→mutex);
  errno = saved_errno;
}
DEF_WEAK(freezero);
```

60. Reallocation. The function behind *realloc* et al. is large but less amenable to being broken up. By now though it should be easier to follow along.

```
static void *orealloc(struct dir_info **argpool, void *p, size_t newsz, void *f)
{
  struct region_info *r;
  struct dir_info *pool;
  char *saved_function;
  struct chunk_info *info;
  size_t oldsz, goldsz, gnewsz;
  void *q, *ret;
  uint32_t chunknum;
  int forced;
  if (p ≡ Λ) return omalloc(*argpool, newsz, 0, f);      /* realloc from Λ is equivalent to malloc */
  if (newsz ≥ SIZE_MAX − mopts.malloc_guard − MALLOC_PAGESIZE) {      /* bigger than possible */
    errno = ENOMEM;
    return Λ;
  }
  r = findpool(p, *argpool, &pool, &saved_function);      /* find the old allocation */
  REALSIZE(oldsz, r);
  if (oldsz ≤ MALLOC_MAXCHUNK) {      /* find a small allocation's chunk */
    if (DO_STATS ∨ mopts.chunk_canaries) {
      info = (struct chunk_info *) r→size;
      chunknum = find_chunknum(pool, info, p, 0);
    }
  }
  goldsz = oldsz;
  if (oldsz > MALLOC_MAXCHUNK) {      /* account for a large allocation's guard pages */
    if (oldsz < mopts.malloc_guard) wrterror(pool, "guard␣size");
    oldsz −= mopts.malloc_guard;
  }
  gnewsz = newsz;
  if (gnewsz > MALLOC_MAXCHUNK)      /* include guard pages if growing into a large allocation */
    gnewsz += mopts.malloc_guard;
  forced = mopts.malloc_realloc ∨ pool→mmap_flag;      /* force reallocation if MAP_CONCEAL set */
  if (newsz > MALLOC_MAXCHUNK ∧ oldsz > MALLOC_MAXCHUNK ∧ ¬forced) {
      /* First case: from n pages sized allocation to m pages sized allocation */
    size_t roldsz = PAGEROUND(goldsz);
    size_t rnewsz = PAGEROUND(gnewsz);
    if (rnewsz < roldsz ∧ rnewsz > roldsz/2 ∧
          roldsz − rnewsz < mopts.def_maxcache * MALLOC_PAGESIZE ∧
          ¬mopts.malloc_guard) {
      ret = p;
      goto done;
    }
    if (rnewsz > roldsz) {      /* try to extend existing region */
      if (¬mopts.malloc_guard) {
        void *hint = (char *) r→p + roldsz;
        size_t needed = rnewsz − roldsz;
        STATS_INC(pool→cheap_realloc_tries);
        q = MMAPA(hint, needed, MAP_FIXED | __MAP_NOREPLACE | pool→mmap_flag);
          /* __MAP_NOREPLACE is undocumented: fail if the space is unavailable */
```

```
    if (q ≡ hint) {      /* enlarged in-place */
      STATS_ADD(pool→malloc_used, needed);
      if (pool→malloc_junk ≡ 2) memset(q, SOME_JUNK, needed);
      r→size = gnewsz;
      if (r→p ≠ p) {      /* old pointer is moved */      /* TODO: but why if q ≡ hint? */
        memmove(r→p, p, oldsz);
        p = r→p;
      }
      if (mopts.chunk_canaries) fill_canary(p, newsz, PAGEROUND(newsz));
      STATS_SETF(r, f);
      STATS_INC(pool→cheap_reallocs);
      ret = p;
      goto done;
    }      /* else proceed as if (newsz ≠ oldsz) below */
  }
  else if (rnewsz < roldsz) {      /* shrink number of pages */
    if (mopts.malloc_guard) {      /* move guard page down */
      if (mprotect((char *) r→p + rnewsz − mopts.malloc_guard, mopts.malloc_guard, PROT_NONE))
        wrterror(pool, "mprotect");
    }
    if (munmap((char *) r→p + rnewsz, roldsz − rnewsz))      /* unmap (cache) excess space */
      wrterror(pool, "munmap␣%p", (char *) r→p + rnewsz);
    STATS_SUB(pool→malloc_used, roldsz − rnewsz);
    r→size = gnewsz;      /* update region's metadata */
    if (MALLOC_MOVE_COND(gnewsz)) {      /* shift the address if the change demands it */
      void *pp = MALLOC_MOVE(r→p, gnewsz);
      memmove(pp, p, newsz);
      p = pp;
    }
    else if (mopts.chunk_canaries) fill_canary(p, newsz, PAGEROUND(newsz));
    STATS_SETF(r, f);
    ret = p;
    goto done;
  }
  else {      /* number of pages remains the same */
    void *pp = r→p;
    r→size = gnewsz;
    if (MALLOC_MOVE_COND(gnewsz)) pp = MALLOC_MOVE(r→p, gnewsz);
    if (p ≠ pp) {      /* shift the base */
      memmove(pp, p, oldsz < newsz ? oldsz : newsz);
      p = pp;
    }
    if (p ≡ r→p) {
      if (newsz > oldsz ∧ pool→malloc_junk ≡ 2)
        memset((char *) p + newsz, SOME_JUNK, rnewsz − mopts.malloc_guard − newsz);
      if (mopts.chunk_canaries) fill_canary(p, newsz, PAGEROUND(newsz));
    }
    STATS_SETF(r, f);
    ret = p;
    goto done;
  }
}
```

```
  if (oldsz ≤ MALLOC_MAXCHUNK ∧ oldsz > 0 ∧
        newsz ≤ MALLOC_MAXCHUNK ∧ newsz > 0 ∧
        ¬forced ∧ find_bucket(newsz) ≡ find_bucket(oldsz)) {
      /* do not reallocate if new size fits good in existing chunk */
    if (pool→malloc_junk ≡ 2) memset((char *) p + newsz, SOME_JUNK, oldsz − newsz);
    if (mopts.chunk_canaries) {
      info→bits[info→offset + chunknum] = newsz;
      fill_canary(p, newsz, B2SIZE(info→bucket));
    }
    if (DO_STATS ∧ chunknum ≡ 0) STATS_SETF(r, f);
    ret = p;
  }
  else if (newsz ≠ oldsz ∨ forced) {      /* create new allocation */
    q = omalloc(pool, newsz, 0, f);
    if (q ≡ Λ) {
      ret = Λ;
      goto done;
    }
    if (newsz ≠ 0 ∧ oldsz ≠ 0) memcpy(q, p, oldsz < newsz ? oldsz : newsz);
    ofree(&pool, p, 0, 0, 0);
    ret = q;
  }
  else {      /* oldsz ≡ newsz */
    if (newsz ≠ 0) wrterror(pool, "realloc␣internal␣inconsistency");
    if (DO_STATS ∧ chunknum ≡ 0) STATS_SETF(r, f);
    ret = p;
  }
done:
  if (*argpool ≠ pool) {
    pool→func = saved_function;
    *argpool = pool;
  }
  return ret;
}
```

61. *realloc* is *orealloc*'s simplest interface.

```
void *realloc(void *ptr, size_t size)
{
  struct dir_info *d;
  void *r;
  int saved_errno = errno;
  PROLOGUE(getpool( ), "realloc")
  r = orealloc(&d, ptr, size, caller( ));
  EPILOGUE( )
  return r;
}
DEF_STRONG(realloc);
```

62. Cleared allocation. Before implementing other interfaces *calloc* is a variant of *malloc* that not only sets its contents to zero, but also safely multiples the size of an element with the number of them. This is *sqrt*(SIZE_MAX + 1), as *s1* ∗ *s2* ≤ SIZE_MAX if both *s1* < MUL_NO_OVERFLOW and *s2* < MUL_NO_OVERFLOW.

```
#define MUL_NO_OVERFLOW (1UL ≪ (sizeof(size_t) ∗ 4))
  void ∗calloc(size_t nmemb, size_t size)
  {
    struct dir_info ∗d;
    void ∗r;
    int saved_errno = errno;
    PROLOGUE(getpool( ), "calloc")
    if ((nmemb ≥ MUL_NO_OVERFLOW ∨ size ≥ MUL_NO_OVERFLOW) ∧
          nmemb > 0 ∧ SIZE_MAX/nmemb < size) {
      d→active −−;
      _MALLOC_UNLOCK(d→mutex);
      if (mopts.malloc_xmalloc) wrterror(d, "out␣of␣memory");
      errno = ENOMEM;
      return Λ;
    }
    size ∗= nmemb;
    r = omalloc(d, size, 1, caller( ));
    EPILOGUE( )
    return r;
  }
  DEF_STRONG(calloc);
```

63. Like *malloc_conceal*, *calloc_conceal* also ensures its contents are safe from core dumps (TODO: and swap?).

```
  void ∗calloc_conceal(size_t nmemb, size_t size)
  {
    struct dir_info ∗d;
    void ∗r;
    int saved_errno = errno;
    PROLOGUE(mopts.malloc_pool[0], "calloc_conceal")
    if ((nmemb ≥ MUL_NO_OVERFLOW ∨ size ≥ MUL_NO_OVERFLOW) ∧
          nmemb > 0 ∧ SIZE_MAX/nmemb < size) {
      d→active −−;
      _MALLOC_UNLOCK(d→mutex);
      if (mopts.malloc_xmalloc) wrterror(d, "out␣of␣memory");
      errno = ENOMEM;
      return Λ;
    }
    size ∗= nmemb;
    r = omalloc(d, size, 1, caller( ));
    EPILOGUE( )
    return r;
  }
  DEF_WEAK(calloc_conceal);
```

64. Array allocation. As its name suggests *orecallocarray* combines *calloc* and *realloc* to allocate and reallocate arrays of objects with safe multiplication and zeroing of new space. *orecollacarray* is the internal part of the implementation that expects the multiplication to have already been done.

It follows the familiar pattern: find the pool, check canaries etc., allocate new space, copy data, free old space.

```
static void *orecallocarray(struct dir_info **argpool, void *p, size_t oldsize, size_t newsize,
    void *f)
{
  struct region_info *r;
  struct dir_info *pool;
  char *saved_function;
  void *newptr;
  size_t sz;
  if (p ≡ Λ) return omalloc(*argpool, newsize, 1, f);
  if (oldsize ≡ newsize) return p;
  r = findpool(p, *argpool, &pool, &saved_function);
  REALSIZE(sz, r);
  if (sz ≤ MALLOC_MAXCHUNK) {
    if (mopts.chunk_canaries ∧ sz > 0) {
      struct chunk_info *info = (struct chunk_info *) r→size;
      uint32_t chunknum = find_chunknum(pool, info, p, 0);
      if (info→bits[info→offset + chunknum] ≠ oldsize)
        wrterror(pool, "recorded size %hu != %zu", info→bits[info→offset + chunknum], oldsize);
    }
    else {
      if (sz < oldsize) wrterror(pool, "chunk size %zu < %zu", sz, oldsize);
    }
  }
  else {
    if (sz − mopts.malloc_guard < oldsize)
      wrterror(pool, "recorded size %zu < %zu", sz − mopts.malloc_guard, oldsize);
    if (oldsize < (sz − mopts.malloc_guard)/2) wrterror(pool,
        "recorded size %zu inconsistent with %zu", sz − mopts.malloc_guard, oldsize);
  }
  newptr = omalloc(pool, newsize, 0, f);
  if (newptr ≡ Λ) goto done;
  if (newsize > oldsize) {
    memcpy(newptr, p, oldsize);
    memset((char *) newptr + oldsize, 0, newsize − oldsize);
  }
  else memcpy(newptr, p, newsize);
  ofree(&pool, p, 1, 0, oldsize);
done:
  if (*argpool ≠ pool) {
    pool→func = saved_function;
    *argpool = pool;
  }
  return newptr;
}
```

65. Like *freezero_p*, *recallocarray_p* is a deprecated implementation of *recallocarray*.

```
static void *recallocarray_p(void *ptr, size_t oldnmemb, size_t newnmemb, size_t size)
{
  size_t oldsize, newsize;
  void *newptr;
  if (ptr ≡ Λ) return calloc(newnmemb, size);
  if ((newnmemb ≥ MUL_NO_OVERFLOW ∨ size ≥ MUL_NO_OVERFLOW) ∧
      newnmemb > 0 ∧ SIZE_MAX/newnmemb < size) {
    errno = ENOMEM;
    return Λ;
  }
  newsize = newnmemb * size;
  if ((oldnmemb ≥ MUL_NO_OVERFLOW ∨ size ≥ MUL_NO_OVERFLOW) ∧
      oldnmemb > 0 ∧ SIZE_MAX/oldnmemb < size) {
    errno = EINVAL;
    return Λ;
  }
  oldsize = oldnmemb * size;
  if (newsize ≤ oldsize) {      /* Don't bother too much if we're shrinking just a bit, we do not
      shrink for series of small steps, oh well. */
    size_t d = oldsize − newsize;
    if (d < oldsize/2 ∧ d < MALLOC_PAGESIZE) {
      memset((char *) ptr + newsize, 0, d);
      return ptr;
    }
  }
  newptr = malloc(newsize);
  if (newptr ≡ Λ) return Λ;
  if (newsize > oldsize) {
    memcpy(newptr, ptr, oldsize);
    memset((char *) newptr + oldsize, 0, newsize − oldsize);
  }
  else memcpy(newptr, ptr, newsize);
  explicit_bzero(ptr, oldsize);
  free(ptr);
  return newptr;
}
```

66. The main interface to cleared array reallocation. Checks its arguments and calls *orecallocarray*.

```
void *recallocarray(void *ptr, size_t oldnmemb, size_t newnmemb, size_t size)
{
  struct dir_info *d;
  size_t oldsize = 0, newsize;
  void *r;
  int saved_errno = errno;
  if (¬mopts.internal_funcs) return recallocarray_p(ptr, oldnmemb, newnmemb, size);
  PROLOGUE(getpool(), "recallocarray")
  if ((newnmemb ≥ MUL_NO_OVERFLOW ∨ size ≥ MUL_NO_OVERFLOW) ∧
        newnmemb > 0 ∧ SIZE_MAX/newnmemb < size) {
    d→active−−;
    _MALLOC_UNLOCK(d→mutex);
    if (mopts.malloc_xmalloc) wrterror(d, "out␣of␣memory");
    errno = ENOMEM;
    return Λ;
  }
  newsize = newnmemb * size;
  if (ptr ≠ Λ) {
    if ((oldnmemb ≥ MUL_NO_OVERFLOW ∨ size ≥ MUL_NO_OVERFLOW) ∧
          oldnmemb > 0 ∧ SIZE_MAX/oldnmemb < size) {
      d→active−−;
      _MALLOC_UNLOCK(d→mutex);
      errno = EINVAL;
      return Λ;
    }
    oldsize = oldnmemb * size;
  }
  r = orecallocarray(&d, ptr, oldsize, newsize, caller());
  EPILOGUE()
  return r;
}
DEF_WEAK(recallocarray);
```

67. Aligned allocation. These algorithms return an address aligned to a given boundary. The allocations returned here can be used as normal including being resized but their special alignment is likely to be lost.

This is the workhorse of the simple algorithm: Allocate $sz + alignment$ bytes of memory, which must include a subrange of size bytes that is properly aligned. Unmap the other bytes, and then return that subrange.

```
static void *mapalign(struct dir_info *d, size_t alignment, size_t sz, int zero_fill)
{
  char *p, *q;
  if (alignment < MALLOC_PAGESIZE ∨ ((alignment − 1) & alignment) ≠ 0)
    wrterror(d, "mapalign␣bad␣alignment");
  if (sz ≠ PAGEROUND(sz)) wrterror(d, "mapalign␣round");
  if (alignment > SIZE_MAX − sz)      /* We need sz + alignment to fit into a size_t. */
    return MAP_FAILED;
  p = map(d, sz + alignment, zero_fill);
  if (p ≡ MAP_FAILED) return MAP_FAILED;
  q = (char *)(((uintptr_t) p + alignment − 1) & ∼(alignment − 1));
  if (q ≠ p) {
    if (munmap(p, q − p)) wrterror(d, "munmap␣%p", p);
  }
  if (munmap(q + sz, alignment − (q − p))) wrterror(d, "munmap␣%p", q + sz);
  STATS_SUB(d→malloc_used, alignment);
  return q;
}
```

68. This function follows the familiar pattern using *mapalign* in place of *map*.

```
static void *omemalign(struct dir_info *pool, size_t alignment, size_t sz, int zero_fill,
    void *f)
{
  size_t psz;
  void *p;
  if (sz > MALLOC_MAXCHUNK ∧ sz < MALLOC_PAGESIZE)      /* If between half a page and a page, */
    sz = MALLOC_PAGESIZE;                               /* ... avoid MALLOC_MOVE. */
  if (alignment ≤ MALLOC_PAGESIZE) {      /* max(size, alignment) rounded up to power of 2 is
        enough to assure the requested alignment. Large regions are always page aligned. */
    size_t pof2;
    if (sz < alignment) sz = alignment;
    if (sz < MALLOC_PAGESIZE) {
      pof2 = MALLOC_MINSIZE;
      while (pof2 < sz) pof2 ≪= 1;
    }
    else pof2 = sz;
    return omalloc(pool, pof2, zero_fill, f);
  }
  if (sz ≥ SIZE_MAX − mopts.malloc_guard − MALLOC_PAGESIZE) {
    errno = ENOMEM;
    return Λ;
  }
  if (sz < MALLOC_PAGESIZE) sz = MALLOC_PAGESIZE;
  sz += mopts.malloc_guard;
  psz = PAGEROUND(sz);
  p = mapalign(pool, alignment, psz, zero_fill);
  if (p ≡ MAP_FAILED) {
    errno = ENOMEM;
    return Λ;
  }
  if (insert(pool, p, sz, f)) {
    unmap(pool, p, psz, 0);
    errno = ENOMEM;
    return Λ;
  }
  if (mopts.malloc_guard) {
    if (mprotect((char *) p + psz − mopts.malloc_guard, mopts.malloc_guard, PROT_NONE))
      wrterror(pool, "mprotect");
    STATS_ADD(pool→malloc_guarded, mopts.malloc_guard);
  }
  if (pool→malloc_junk ≡ 2) {
    if (zero_fill) memset((char *) p + sz − mopts.malloc_guard, SOME_JUNK, psz − sz);
    else memset(p, SOME_JUNK, psz − mopts.malloc_guard);
  }
  else if (mopts.chunk_canaries) fill_canary(p, sz − mopts.malloc_guard, psz − mopts.malloc_guard);
  return p;
}
```

69. This is the POSIX interface to allocating aligned memory. The `posix_memalign(3)` manual says that memory returned from this function cannot be passed to *recallocarray* or *freezero*. TODO: Is this a left-over from before this implementation?

```
int posix_memalign(void **memptr, size_t alignment, size_t size)
{
  struct dir_info *d;
  int res, saved_errno = errno;
  void *r;
  if (((alignment - 1) & alignment) ≠ 0 ∨ alignment < sizeof(void *))
      /* Make sure that alignment is a large enough power of 2. */
    return EINVAL;
  d = getpool( );
  if (d ≡ Λ) {
    _malloc_init(0);
    d = getpool( );
  }
  _MALLOC_LOCK(d→mutex);
  d→func = "posix_memalign";
  if (d→active++) {
    malloc_recurse(d);
    goto err;
  }
  r = omemalign(d, alignment, size, 0, caller( ));
  d→active--;
  _MALLOC_UNLOCK(d→mutex);
  if (r ≡ Λ) {
    if (mopts.malloc_xmalloc) wrterror(d, "out␣of␣memory");
    goto err;
  }
  errno = saved_errno;
  *memptr = r;
  return 0;
err: res = errno;
  errno = saved_errno;
  return res;
}
DEF_STRONG(posix_memalign);
```

70. This is the BSD and C (as of `C18`) interface.

```
void *aligned_alloc(size_t alignment, size_t size)
{
  struct dir_info *d;
  int saved_errno = errno;
  void *r;
  if (((alignment - 1) & alignment) ≠ 0 ∨ alignment ≡ 0) {      /* alignment must be a power of 2 */
    errno = EINVAL;
    return Λ;
  }
  if ((size & (alignment - 1)) ≠ 0) {      /* per spec, size should be a multiple of alignment */
    errno = EINVAL;
    return Λ;
  }
  PROLOGUE(getpool( ), "aligned_alloc")
  r = omemalign(d, alignment, size, 0, caller( ));
  EPILOGUE( )
  return r;
}
DEF_STRONG(aligned_alloc);
```

71. Statistics. The rest of this file is only included if Malloc is being compiled with statistics gathering and reporting enabled.

```
#ifdef MALLOC_STATS
  ⟨ Memory allocator statistics (MALLOC_STATS) 72 ⟩
#endif
```

72. Micrologger. `KTR_USER_MAXLEN` is defined by `ktrace` as the "maximum length of passed data" and is (TODO: ?) used here to ensure print buffers don't overflow?

```
⟨ Memory allocator statistics (MALLOC_STATS) 72 ⟩ ≡
  static void ulog(const char *format, ...)
  {
    va_list ap;
    static char *buf;
    static size_t filled;
    int len;
    if (buf ≡ Λ) buf = MMAP(KTR_USER_MAXLEN, 0);
    if (buf ≡ MAP_FAILED) return;
    va_start(ap, format);
    len = vsnprintf(buf + filled, KTR_USER_MAXLEN − filled, format, ap);
    va_end(ap);
    if (len < 0) return;
    if (len > KTR_USER_MAXLEN − filled) len = KTR_USER_MAXLEN − filled;
    filled += len;
    if (filled > 0) {
      if (filled ≡ KTR_USER_MAXLEN ∨ buf[filled − 1] ≡ '\n') {
        utrace("malloc", buf, filled);
        filled = 0;
      }
    }
  }
```

See also sections 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, and 83.

This code is used in section 71.

73. Generate the structure and function definition code to implement **leaktree** as a red-black tree with *leaknode*s that contain a *malloc_leak* object. See `RBT_INIT(9)`.

```
⟨ Memory allocator statistics (MALLOC_STATS) 72 ⟩ +≡
  struct malloc_leak {
    void *f;
    size_t total_size;
    int count;
  };
  struct leaknode {
    RBT_ENTRY (leaknode) entry;
    struct malloc_leak d;
  };
  static inline int leakcmp(const struct leaknode *e1, const struct leaknode *e2)
  {
    return e1→d.f < e2→d.f ? −1 : e1→d.f > e2→d.f;
  }
  RBT_HEAD (leaktree, leaknode);
  RBT_PROTOTYPE (leaktree, leaknode, entry, leakcmp);
  RBT_GENERATE (leaktree, leaknode, entry, leakcmp);
```

74. ⟨ Memory allocator statistics (`MALLOC_STATS`) 72 ⟩ +≡

```
static void wrtwarning(const char *func, char *msg, ...)
{
  int saved_errno = errno;
  va_list ap;
  dprintf(STDERR_FILENO, "%s(%d) in %s(): ", __progname, getpid(), func ≠ Λ ? func : "unknown");
  va_start(ap, msg);
  vdprintf(STDERR_FILENO, msg, ap);
  va_end(ap);
  dprintf(STDERR_FILENO, "\n");
  errno = saved_errno;
}
```

75.

⟨ Memory allocator statistics (`MALLOC_STATS`) 72 ⟩ +≡

```
static void putleakinfo(struct leaktree *leaks, void *f, size_t sz, int cnt)
{
  struct leaknode key, *p;
  static struct leaknode *page;
  static unsigned int used;
  if (cnt ≡ 0 ∨ page ≡ MAP_FAILED) return;
  key.d.f = f;      /* TODO: the structure of leaknode is not documented */
  p = RBT_FIND(leaktree, leaks, &key);
  if (p ≡ Λ) {
    if (page ≡ Λ ∨ used ≥ MALLOC_PAGESIZE/sizeof(struct leaknode)) {
      page = MMAP(MALLOC_PAGESIZE, 0);
      if (page ≡ MAP_FAILED) {
        wrtwarning(__func__, strerror(errno));
        return;
      }
      used = 0;
    }
    p = &page[used++];
    p->d.f = f;
    p->d.total_size = sz * cnt;
    p->d.count = cnt;
    RBT_INSERT(leaktree, leaks, p);
  }
  else {
    p->d.total_size += sz * cnt;
    p->d.count += cnt;
  }
}
```

76. ⟨ Memory allocator statistics (MALLOC_STATS) 72 ⟩ +≡

```
static void dump_leaks(struct leaktree *leaks)
{
  struct leaknode *p;
  ulog("Leak report:\n");
  ulog("                  f      sum       #    avg\n");
  RBT_FOREACH(p, leaktree, leaks)
  {
    Dl_info info;
    const char *caller = p->d.f;
    const char *object = ".";
    if (caller ≠ Λ) {
      if (dladdr(p->d.f, &info) ≠ 0) {
        caller -= (uintptr_t) info.dli_fbase;
        object = info.dli_fname;
      }
    }
    ulog("%18p %7zu %6u %6zu addr2line -e %s %p\n", p->d.f, p->d.total_size, p->d.count,
        p->d.total_size/p->d.count, object, caller);
  }
}
```

77. ⟨ Memory allocator statistics (MALLOC_STATS) 72 ⟩ +≡

```
static void dump_chunk(struct leaktree *leaks, struct chunk_info *p, void *f, int fromfreelist)
{
  while (p ≠ Λ) {
    if (mopts.malloc_verbose) ulog("chunk %18p %18p %4zu %d/%d\n", p->page,
        ((p->bits[0] & 1) ? Λ : f), B2SIZE(p->bucket), p->free, p->total);
    if (¬fromfreelist) {
      size_t sz = B2SIZE(p->bucket);
      if (p->bits[0] & 1) putleakinfo(leaks, Λ, sz, p->total - p->free);
      else {
        putleakinfo(leaks, f, sz, 1);
        putleakinfo(leaks, Λ, sz, p->total - p->free - 1);
      }
      break;
    }
    p = LIST_NEXT(p, entries);
    if (mopts.malloc_verbose ∧ p ≠ Λ) ulog("       ->");
  }
}
```

78. ⟨ Memory allocator statistics (MALLOC_STATS) 72 ⟩ +≡

```
static void dump_free_chunk_info(struct dir_info *d, struct leaktree *leaks)
{
  int i, j, count;
  struct chunk_info *p;
  ulog("Free chunk structs:\n");
  ulog("Bkt) #CI                     page"      /* 1+21×␣ */
  "                  f size free/n\n");      /* 18+1+1×␣ */
  for (i = 0; i ≤ BUCKETS; i++) {
    count = 0;
    LIST_FOREACH(p, &d→chunk_info_list[i], entries)count++;
    for (j = 0; j < MALLOC_CHUNK_LISTS; j++) {
      p = LIST_FIRST(&d→chunk_dir[i][j]);
      if (p ≡ Λ ∧ count ≡ 0) continue;
      if (j ≡ 0) ulog("%3d) %3d ", i, count);
      else ulog("        ");      /* 9×␣ */
      if (p ≠ Λ) dump_chunk(leaks, p, Λ, 1);
      else ulog(".\n");
    }
  }
}
```

79. ⟨ Memory allocator statistics (MALLOC_STATS) 72 ⟩ +≡

```
static void dump_free_page_info(struct dir_info *d)
{
  struct smallcache *cache;
  size_t i, total = 0;
  ulog("Cached in small cache:\n");
  for (i = 0; i < MAX_SMALLCACHEABLE_SIZE; i++) {
    cache = &d→smallcache[i];
    if (cache→length ≠ 0)
      ulog("%zu(%u): %u = %zu\n", i + 1, cache→max, cache→length, cache→length * (i + 1));
    total += cache→length * (i + 1);
  }
  ulog("Cached in big cache: %zu/%zu\n", d→bigcache_used, d→bigcache_size);
  for (i = 0; i < d→bigcache_size; i++) {
    if (d→bigcache[i].psize ≠ 0) ulog("%zu: %zu\n", i, d→bigcache[i].psize);
    total += d→bigcache[i].psize;
  }
  ulog("Free pages cached: %zu\n", total);
}
```

80. ⟨ Memory allocator statistics (MALLOC_STATS) 72 ⟩ +≡

```
static void malloc_dump1(int poolno, struct dir_info *d, struct leaktree *leaks)
{
  size_t i, realsize;
  if (mopts.malloc_verbose) {
    ulog("Malloc dir of %s pool %d at %p\n", __progname, poolno, d);
    ulog("MT=%d J=%d Fl=%x\n", d->malloc_mt, d->malloc_junk, d->mmap_flag);
    ulog("Region slots free %zu/%zu\n", d->regions_free, d->regions_total);
    ulog("Finds %zu/%zu\n", d->finds, d->find_collisions);
    ulog("Inserts %zu/%zu\n", d->inserts, d->insert_collisions);
    ulog("Deletes %zu/%zu\n", d->deletes, d->delete_moves);
    ulog("Cheap reallocs %zu/%zu\n", d->cheap_reallocs, d->cheap_realloc_tries);
    ulog("Other pool searches %zu/%zu\n", d->other_pool, d->pool_searches);
    ulog("In use %zu\n", d->malloc_used);
    ulog("Guarded %zu\n", d->malloc_guarded);
    dump_free_chunk_info(d, leaks);
    dump_free_page_info(d);
    ulog("Hash table:\n");
    ulog("slot)  hash d  type                "      /* ...+15× */
    "page                    "      /* 18× */
    "f size [free/n]\n");
  }
  for (i = 0; i < d->regions_total; i++) {
    if (d->r[i].p != Λ) {
      size_t h = hash(d->r[i].p) & (d->regions_total - 1);
      if (mopts.malloc_verbose) ulog("%4zx) #%4zx %zd ", i, h, h - i);
      REALSIZE(realsize, &d->r[i]);
      if (realsize > MALLOC_MAXCHUNK) {
        putleakinfo(leaks, d->r[i].f, realsize, 1);
        if (mopts.malloc_verbose) ulog("pages %18p %18p %zu\n", d->r[i].p, d->r[i].f, realsize);
      }
      else dump_chunk(leaks, (struct chunk_info *) d->r[i].size, d->r[i].f, 0);
    }
  }
  if (mopts.malloc_verbose) ulog("\n");
}
```

81. ⟨ Memory allocator statistics (MALLOC_STATS) 72 ⟩ +≡

```
static void malloc_dump0(int poolno, struct dir_info *pool, struct leaktree *leaks)
{
  int i;
  void *p;
  struct region_info *r;
  if (pool ≡ Λ ∨ pool->r ≡ Λ) return;
  for (i = 0; i < MALLOC_DELAYED_CHUNK_MASK + 1; i++) {
    p = pool->delayed_chunks[i];
    if (p ≡ Λ) continue;
    r = find(pool, p);
    if (r ≡ Λ) wrterror(pool, "bogus pointer in malloc_dump %p", p);
    free_bytes(pool, r, p);
    pool->delayed_chunks[i] = Λ;
  }
  malloc_dump1(poolno, pool, leaks);
}
```

82. ⟨ Memory allocator statistics (`MALLOC_STATS`) 72 ⟩ +≡

```
void malloc_dump(void)
{
  int i;
  int saved_errno = errno;      /* XXX leak when run multiple times */
  struct leaktree leaks = RBT_INITIALIZER(&leaks);
  for (i = 0; i < mopts.malloc_mutexes; i++)  malloc_dump0(i, mopts.malloc_pool[i], &leaks);
  dump_leaks(&leaks);
  ulog("\n");
  errno = saved_errno;
}
DEF_WEAK(malloc_dump);
```

83. ⟨ Memory allocator statistics (`MALLOC_STATS`) 72 ⟩ +≡

```
static void malloc_exit(void)
{
  int save_errno = errno;
  ulog("******** Start dump %s *******\n", __progname);
  ulog("M=%u I=%d F=%d U=%d J=%d R=%d X=%d C=%d cache=%u ""G=%zu\n",
      mopts.malloc_mutexes, mopts.internal_funcs, mopts.malloc_freecheck,
      mopts.malloc_freeunmap, mopts.def_malloc_junk, mopts.malloc_realloc, mopts.malloc_xmalloc,
      mopts.chunk_canaries, mopts.def_maxcache, mopts.malloc_guard);
  malloc_dump( );
  ulog("******** End dump %s *******\n", __progname);
  errno = save_errno;
}
```

84. Index.

__aarch64__: 17.
__amd64__: 17.
__arm__: 17.
__asm__: 37.
__attribute__: 14, 16.
__builtin_clz: 37.
__builtin_extract_return_addr: 17.
__builtin_return_address: 17.
__dead: 16, 21.
__format__: 16.
__func__: 75.
__GNUC__: 37.
__LP64__: 19.
__MAP_NOREPLACE: 60.
__m88k__: 37.
__progname: 21, 74, 80, 83.
__volatile__: 37.
_ALIGN: 35.
_malloc_init: 24, 47, 50, 69.
_MALLOC_LOCK: 47, 50, 53, 58, 59, 69.
_MALLOC_MUTEXES: 14, 23.
_MALLOC_UNLOCK: 46, 47, 48, 50, 53, 58, 59, 62, 63, 66, 69.
_MAX_PAGE_SHIFT: 5.
_pad: 14.
abort: 21.
active: 11, 46, 50, 53, 58, 59, 62, 63, 66, 69.
aligned: 14.
aligned_alloc: 70.
alignment: 67, 68, 69, 70.
alloc_chunk_info: 35, 36.
allocated: 40, 42.
ap: 21, 72, 74.
arc4random: 24.
arc4random_buf: 22.
arc4random_uniform: 48.
argpool: 53, 54, 55, 60, 64.
argsz: 54, 55, 56, 57.
atexit: 24.
b: 24.
base: 32, 33.
bigcache: 2, 10, 11, 32, 33, 49, 79.
BIGCACHE_FILL: 10, 32.
bigcache_size: 11, 32, 33, 49, 79.
bigcache_used: 11, 32, 33, 79.
bin_of: 38, 39.
bits: 13, 34, 41, 43, 44, 55, 60, 64, 77.
bp: 36, 41.
BRKSIZ: 27.
bucket: 13, 34, 35, 36, 41, 43, 44, 57, 60, 77.
BUCKETS: 5, 11, 25, 78.
buf: 72.
B2ALLOC: 5, 34, 35, 41, 43.
B2SIZE: 5, 18, 41, 43, 57, 60, 77.
cache: 32, 33, 79.
cached: 33.
caller: 17, 51, 52, 61, 62, 63, 66, 69, 70, 76.
calloc: 62, 64, 65.
calloc_conceal: 63.
canary: 13, 34, 41, 43.
canary1: 11, 25, 28, 33, 34, 41, 43, 49.
canary2: 11, 25, 28, 33, 41, 49.
CHAR_BIT: 37.
cheap_realloc_tries: 11, 60, 80.
cheap_reallocs: 11, 60, 80.
check: 43, 54, 55.
check_sz: 40, 42.
chunk_canaries: 14, 23, 24, 35, 40, 41, 42, 45, 55, 56, 57, 60, 64, 68, 83.
CHUNK_CHECK_LENGTH: 5, 40, 42.
chunk_dir: 11, 25, 36, 41, 44, 78.
chunk_head: 9, 11, 44.
chunk_info: 9, 13, 34, 35, 36, 41, 43, 44, 55, 57, 60, 64, 77, 78, 80.
chunk_info_list: 11, 25, 35, 44, 78.
chunk_pages: 11, 35.
chunk_pages_used: 11, 35.
chunknum: 43, 44, 55, 60, 64.
clear: 12, 32, 54, 55, 56, 57.
cnt: 75.
count: 35, 37, 73, 75, 76, 78.
CTL_VM: 24.
d: 12, 16, 21, 22, 25, 26, 27, 28, 29, 32, 33, 34, 35, 36, 41, 42, 43, 44, 46, 47, 51, 52, 58, 59, 61, 62, 63, 65, 66, 67, 69, 70, 73, 78, 79, 80.
d_avail: 48.
def_malloc_junk: 14, 23, 24, 25, 49, 83.
def_maxcache: 14, 23, 24, 39, 49, 60, 83.
DEF_STRONG: 47, 51, 58, 61, 62, 69, 70.
DEF_WEAK: 52, 59, 63, 66, 82.
delayed_chunks: 11, 57, 81.
delete: 29, 44, 56.
delete_moves: 11, 29, 80.
deletes: 11, 29, 80.
dir_info: 2, 11, 12, 14, 16, 20, 21, 22, 25, 26, 27, 28, 29, 31, 32, 33, 34, 35, 36, 41, 42, 43, 44, 45, 46, 47, 48, 50, 51, 52, 53, 54, 58, 59, 60, 61, 62, 63, 64, 66, 67, 68, 69, 70, 78, 79, 80, 81.
Dl_info: 76.
dladdr: 76.
dli_fbase: 76.
dli_fname: 76.
DO_STATS: 14, 17, 21, 24, 41, 60.
done: 60, 64.
dprintf: 21, 23, 24, 74.
dump_chunk: 77, 78, 80.
dump_free_chunk_info: 78, 80.
dump_free_page_info: 79, 80.
dump_leaks: 76, 82.

EDEADLK: 46.
EINVAL: 65, 66, 69, 70.
ENOMEM: 45, 60, 62, 63, 65, 66, 68.
entries: 13, 35, 36, 41, 44, 77, 78.
entry: 73.
EPILOGUE: 50, 51, 52, 61, 62, 63, 66, 70.
err: 36, 69.
errno: 21, 45, 46, 50, 51, 52, 58, 59, 60, 61, 62, 63, 65, 66, 68, 69, 70, 74, 75, 82, 83.
explicit_bzero: 32, 57, 59, 65.
e1: 73.
e2: 73.
f: 9, 27, 41, 45, 60, 64, 68, 73, 75, 77.
ffs: 41.
fill_canary: 40, 41, 45, 60, 68.
filled: 72.
find: 28, 41, 53, 57, 81.
find_bucket: 39, 41, 60.
find_chunknum: 43, 44, 55, 57, 60, 64.
find_collisions: 11, 28, 80.
findpool: 53, 55, 60, 64.
finds: 11, 28, 80.
fn: 50.
forced: 60.
format: 72.
found: 41.
foundpool: 53.
free: 13, 34, 41, 44, 54, 58, 59, 65, 77.
free_bytes: 44, 57, 81.
freezero: 55, 59, 69.
freezero_p: 59, 65.
fresh: 32, 33.
from_rthreads: 47, 48, 49.
fromfreelist: 77.
func: 11, 21, 50, 53, 54, 58, 59, 60, 64, 69, 74.
getenv: 24.
getpid: 21, 74.
getpool: 20, 50, 51, 58, 59, 61, 62, 66, 69, 70.
getrbyte: 22, 32, 33, 41, 44, 57.
gnewsz: 60.
goldsz: 60.
h: 80.
hash: 19, 26, 27, 28, 29, 80.
hint: 60.
howmany: 34, 35, 39, 41.
i: 24, 25, 26, 29, 30, 31, 32, 33, 34, 35, 41, 47, 53, 57, 78, 79, 80, 81, 82.
index: 26, 27, 28.
info: 43, 44, 55, 57, 60, 64, 76.
init_chunk_info: 34, 35.
insert: 27, 36, 45, 68.
insert_collisions: 11, 26, 27, 80.
inserts: 11, 26, 27, 80.
internal_funcs: 14, 48, 59, 66, 83.
issetugid: 24.
j: 24, 25, 29, 32, 33, 41, 47, 53, 78.
junk: 30.
junk_free: 30, 31, 32, 33, 57.
junk_loc: 14, 24, 30, 31.
k: 41.
key: 75.
KTR_USER_MAXLEN: 72.
lb: 37, 38, 41.
leakcmp: 73.
leaknode: 2, 73, 75, 76.
leaks: 75, 76, 77, 78, 80, 81, 82.
leaktree: 73, 75, 76, 77, 78, 80, 81, 82.
len: 72.
length: 10, 32, 33, 79.
linear: 38.
LIST_EMPTY: 35.
LIST_ENTRY: 13.
LIST_FIRST: 35, 41, 78.
LIST_FOREACH: 78.
LIST_HEAD: 9.
LIST_INIT: 25.
LIST_INSERT_HEAD: 35, 36, 44.
LIST_NEXT: 77.
LIST_REMOVE: 35, 41, 44.
listnum: 36, 41, 44.
lp: 30, 31, 41.
malloc: 36, 51, 52, 60, 62, 65.
MALLOC_BITS: 13, 34, 35, 41, 43, 44.
malloc_bytes: 41, 45.
malloc_canary: 14, 24, 25, 28, 33, 41, 48.
MALLOC_CHUNK_LISTS: 5, 11, 25, 41, 44, 78.
malloc_conceal: 52, 63.
MALLOC_DEFAULT_CACHE: 5, 24.
MALLOC_DELAYED_CHUNK_MASK: 5, 11, 57, 81.
malloc_dump: 16, 21, 82, 83.
malloc_dump0: 81, 82.
malloc_dump1: 80, 81.
malloc_exit: 16, 24, 83.
malloc_freecheck: 14, 23, 57, 83.
malloc_freeunmap: 14, 23, 32, 33, 44, 56, 83.
malloc_guard: 6, 14, 23, 45, 55, 56, 60, 64, 68, 83.
malloc_guarded: 11, 45, 56, 68, 80.
MALLOC_INITIAL_REGIONS: 5, 26.
malloc_junk: 11, 25, 31, 32, 33, 36, 45, 49, 57, 60, 68, 80.
malloc_leak: 2, 73.
MALLOC_LEEWAY: 6.
MALLOC_MAXCACHE: 5, 23.
MALLOC_MAXCHUNK: 5, 45, 54, 55, 60, 64, 68, 80.
MALLOC_MAXSHIFT: 5.
MALLOC_MINSHIFT: 5, 48.
MALLOC_MINSIZE: 5, 6, 39, 43, 68.
MALLOC_MOVE: 6, 45, 56, 60, 68.
MALLOC_MOVE_COND: 6, 45, 56, 60.
malloc_mt: 11, 20, 49, 53, 80.
malloc_mutexes: 14, 20, 23, 24, 48, 49, 53, 82, 83.
malloc_options: 15, 24.

MALLOC_PAGEMASK: 5, 6, 18, 43, 48.
MALLOC_PAGESHIFT: 5, 9, 10, 19, 32, 33.
MALLOC_PAGESIZE: 5, 6, 14, 23, 30, 31, 32, 34, 35, 36, 44, 45, 48, 60, 65, 67, 68, 75.
malloc_pool: 14, 20, 48, 49, 52, 53, 63, 82.
malloc_readonly: 2, 14, 48.
malloc_realloc: 14, 23, 60, 83.
malloc_recurse: 46, 50, 58, 59, 69.
MALLOC_SMALL: 3.
malloc_stats: 14, 23.
MALLOC_STATS: 3, 4, 5, 9, 11, 14, 16, 21, 23, 24, 71.
malloc_used: 11, 26, 32, 33, 35, 48, 60, 67, 80.
malloc_verbose: 14, 21, 23, 77, 80.
malloc_xmalloc: 14, 23, 50, 62, 63, 66, 69, 83.
map: 33, 36, 45, 67, 68.
MAP_ANON: 8.
MAP_CONCEAL: 49, 52, 55, 60.
MAP_FAILED: 26, 33, 35, 36, 45, 48, 49, 67, 68, 72, 75.
MAP_FIXED: 27, 60.
MAP_PRIVATE: 8.
mapalign: 67, 68.
mask: 26, 27, 28, 29, 38.
MASK_POINTER: 5, 28.
max: 10, 32, 33, 49, 68, 79.
MAX_BIGCACHEABLE_SIZE: 10, 32, 33.
MAX_SMALLCACHEABLE_SIZE: 10, 11, 32, 33, 49, 79.
memcpy: 60, 64, 65.
memmove: 60.
memptr: 69.
memset: 33, 34, 36, 40, 45, 60, 64, 65, 68.
mib: 24.
mimmutable: 48, 49.
mmap: 8.
MMAP: 8, 26, 33, 35, 49, 72, 75.
mmap_flag: 11, 25, 26, 33, 35, 55, 60, 80.
MMAPA: 8, 60.
MMAPNONE: 8, 48.
mopts: 6, 14, 20, 21, 23, 24, 25, 28, 30, 31, 32, 33, 35, 39, 40, 41, 42, 44, 45, 48, 49, 50, 52, 53, 55, 56, 57, 59, 60, 62, 63, 64, 66, 68, 69, 77, 80, 82, 83.
mp: 44.
mprotect: 32, 33, 36, 44, 45, 48, 56, 60, 68.
msg: 16, 21, 74.
MUL_NO_OVERFLOW: 62, 63, 65, 66.
munmap: 32, 60, 67.
mutex: 11, 46, 49, 50, 53, 58, 59, 62, 63, 66, 69.
n_bits: 38.
NBBY: 13.
needed: 60.
newnmemb: 65, 66.
newptr: 64, 65.
newsize: 26, 64, 65, 66.
newsz: 60.
newtotal: 26.
nmemb: 62, 63.
nmutexes: 47, 49, 53.
noprint: 46.
object: 76.
offset: 13, 34, 41, 43, 55, 60, 64.
ofree: 54, 58, 59, 60, 64.
oldnmemb: 65, 66.
oldpsz: 26.
oldsize: 64, 65, 66.
oldsz: 60.
omalloc: 45, 51, 52, 60, 62, 63, 64, 68.
omalloc_grow: 26, 27.
omalloc_init: 23, 24, 48.
omalloc_make_chunks: 36, 41.
omalloc_parseopt: 23, 24, 48.
omalloc_parsopt: 15.
omalloc_poolinit: 25, 49.
omemalign: 68, 69, 70.
opt: 23.
orealloc: 60, 61.
orecallocarray: 64, 66.
orecollacarray: 64.
other_pool: 11, 53, 80.
p: 9, 12, 17, 19, 24, 26, 27, 28, 30, 31, 32, 33, 34, 35, 41, 42, 45, 48, 49, 53, 54, 60, 64, 67, 68, 75, 76, 77, 78, 81.
page: 10, 13, 32, 33, 36, 41, 44, 75, 77.
PAGEROUND: 6, 26, 32, 33, 45, 56, 60, 67, 68.
pages: 10, 32, 33, 49.
pof2: 68.
pool: 31, 45, 53, 54, 55, 56, 57, 60, 64, 68, 81.
pool_searches: 11, 53, 80.
poolno: 80, 81.
posix_memalign: 69.
pp: 36, 60.
printf: 16.
PROLOGUE: 50, 51, 52, 61, 62, 63, 66, 70.
PROT_NONE: 8, 14, 32, 33, 36, 45, 60, 68.
PROT_READ: 8, 14, 33, 44, 48, 56.
PROT_WRITE: 8, 33, 44, 48, 56.
PROTO_NORMAL: 16.
psize: 10, 32, 33, 79.
psz: 32, 33, 45, 68.
ptr: 40, 42, 43, 44, 58, 59, 61, 65, 66.
putleakinfo: 75, 77, 80.
q: 24, 26, 27, 28, 33, 35, 42, 60, 67.
r: 11, 28, 29, 32, 41, 44, 51, 52, 53, 54, 60, 61, 62, 63, 64, 66, 69, 70, 81.
range: 38.
RBT_ENTRY: 73.
RBT_FIND: 75.
RBT_FOREACH: 76.
RBT_GENERATE: 73.
RBT_HEAD: 73.

RBT_INITIALIZER: 82.
RBT_INSERT: 75.
RBT_PROTOTYPE: 73.
rbytes: 11, 22, 25.
rbytes_init: 22.
rbytesused: 11, 22, 25.
realloc: 60, 61, 64.
REALSIZE: 18, 55, 57, 60, 64, 80.
realsize: 80.
recallocarray: 65, 66, 69.
recallocarray_p: 65, 66.
region_info: 2, 5, 9, 11, 26, 27, 28, 29, 41, 44, 53, 54, 60, 64, 81.
regions_free: 11, 25, 26, 27, 29, 80.
regions_total: 11, 25, 26, 27, 28, 29, 80.
res: 69.
ret: 60.
ri: 29.
rnewsz: 60.
roldsz: 60.
rounded: 38.
rounded_size: 38.
save_errno: 83.
saved_errno: 21, 50, 51, 52, 58, 59, 61, 62, 63, 66, 69, 70, 74, 82.
saved_function: 53, 54, 55, 60, 64.
sb: 24.
section: 14.
shift: 38.
size: 9, 18, 27, 29, 35, 38, 39, 41, 44, 51, 52, 55, 57, 60, 61, 62, 63, 64, 65, 66, 68, 69, 70, 80.
SIZE_MAX: 26, 45, 60, 62, 63, 65, 66, 67, 68.
smallcache: 2, 10, 11, 32, 33, 49, 79.
SOME_FREEJUNK: 7, 24, 36, 42.
SOME_FREEJUNK_ULL: 7, 30, 31.
SOME_JUNK: 7, 42, 45, 60, 68.
sqrt: 62.
STATS_ADD: 11, 26, 33, 35, 45, 48, 60, 68.
STATS_INC: 11, 26, 27, 28, 29, 53, 60.
STATS_SETF: 9, 11, 27, 41, 44, 60.
STATS_SUB: 11, 32, 56, 60, 67.
STATS_ZERO: 11, 26.
STDERR_FILENO: 21, 23, 24, 74.
step: 30, 31.
strerror: 75.
sub_index: 38.
subbin: 38.
sum: 19.
sysctl: 24.
sz: 6, 8, 10, 12, 18, 27, 30, 31, 32, 33, 40, 42, 45, 48, 49, 54, 55, 56, 57, 59, 64, 67, 68, 75, 77.
s1: 62.
s2: 62.
TIB_GET: 20.
tib_tid: 20.
tmp: 32, 57.
tmpsz: 57.
TODO: 9, 10, 12, 14, 20, 24, 33, 34, 35, 37, 41, 60, 63, 69, 72, 75.
total: 13, 34, 35, 41, 44, 77, 79.
total_size: 73, 75, 76.
u: 19.
u_char: 11, 14, 22, 24, 42.
u_int: 14, 34, 35, 36, 37, 41, 47, 53, 57.
u_int32_t: 11, 14, 25, 28, 33, 41.
u_short: 13, 32, 33, 34, 35, 39, 41, 43.
uintptr_t: 5, 9, 18, 19, 25, 28, 32, 33, 36, 41, 43, 48, 67, 76.
uint32_t: 43, 44, 55, 60, 64.
uint64_t: 30, 31.
ulog: 72, 76, 77, 78, 79, 80, 82, 83.
unmap: 12, 26, 32, 33, 36, 44, 45, 56, 68.
used: 75.
ushort: 10, 33.
utrace: 72.
va_end: 21, 72, 74.
va_start: 21, 72, 74.
validate_canary: 42, 43, 56.
validate_junk: 31, 32, 33, 41, 57.
vdprintf: 21, 74.
VM_MALLOC_CONF: 24.
vsnprintf: 72.
wrterror: 16, 21, 28, 29, 31, 32, 33, 41, 42, 43, 45, 46, 48, 49, 50, 53, 55, 56, 57, 58, 59, 60, 62, 63, 64, 66, 67, 68, 69, 81.
wrtwarning: 74, 75.
x: 22, 37.
zero_fill: 33, 45, 67, 68.

⟨ Find the allocation's pool and check its metadata 55 ⟩ Used in section 54.
⟨ Free a large allocation 56 ⟩ Used in section 54.
⟨ Free a small allocation 57 ⟩ Used in section 54.
⟨ Initialise global allocator settings 48 ⟩ Used in section 47.
⟨ Initialise threaded allocator settings 49 ⟩ Used in section 47.
⟨ Memory allocator statistics (`MALLOC_STATS`) 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83 ⟩ Used in section 71.

OPENBSD MEMORY ALLOCATOR

	Section	Page
OpenBSD Memory Allocator	1	1
Simple allocation	50	31
Reallocation	60	36
Cleared allocation	62	39
Array allocation	64	40
Aligned allocation	67	43
Statistics	71	47
Index	84	53