Complete source code of `dlmalloc`

Written by Doug Lea
Typeset by John Wickerson

Contents

Chapter 1

Introduction

```
/*
  This is a version (aka dlmalloc) of malloc/free/realloc written by
  Doug Lea and released to the public domain, as explained at
  http://creativecommons.org/licenses/publicdomain.  Send questions,
  comments, complaints, performance data, etc to dl@cs.oswego.edu

* Version 2.8.4 Wed May 27 09:56:23 2009  Doug Lea  (dl at gee)

   Note: There may be an updated version of this malloc obtainable at
           ftp://gee.cs.oswego.edu/pub/misc/malloc.c
         Check before installing!
```

1.1 Quickstart

```
  This library is all in one file to simplify the most common usage:
  ftp it, compile it (-O3), and link it into another program. All of
  the compile-time options default to reasonable values for use on
  most platforms.  You might later want to step through various
  compile-time and dynamic tuning options.

  For convenience, an include file for code using this malloc is at:
     ftp://gee.cs.oswego.edu/pub/misc/malloc-2.8.4.h
  You don't really need this .h file unless you call functions not
  defined in your system include files.  The .h file contains only the
  excerpts from this file needed for using this malloc on ANSI C/C++
  systems, so long as you haven't changed compile-time options about
  naming and tuning parameters.  If you do, then you can create your
  own malloc.h that does include all settings by cutting at the point
  indicated below. Note that you may already by default be using a C
  library containing a malloc that is based on some version of this
  malloc (for example in linux). You might still want to use the one
  in this file to customize settings or to avoid overheads associated
  with library versions.
```

1.2 Vital statistics

```
Supported pointer/size_t representation:       4 or 8 bytes
     size_t MUST be an unsigned type of the same width as
     pointers. (If you are using an ancient system that declares
     size_t as a signed type, or need it to be a different width
     than pointers, you can use a previous release of this malloc
     (e.g. 2.7.2) supporting these.)

Alignment:                                     8 bytes (default)
     This suffices for nearly all current machines and C compilers.
     However, you can define MALLOC_ALIGNMENT to be wider than this
     if necessary (up to 128bytes), at the expense of using more space.

Minimum overhead per allocated chunk:   4 or  8 bytes (if 4byte sizes)
                                        8 or 16 bytes (if 8byte sizes)
     Each malloced chunk has a hidden word of overhead holding size
     and status information, and additional cross-check word
     if FOOTERS is defined.

Minimum allocated size: 4-byte ptrs:  16 bytes    (including overhead)
                        8-byte ptrs:  32 bytes    (including overhead)

     Even a request for zero bytes (i.e., malloc(0)) returns a
     pointer to something of the minimum allocatable size.
     The maximum overhead wastage (i.e., number of extra bytes
     allocated than were requested in malloc) is less than or equal
     to the minimum size, except for requests >= mmap_threshold that
     are serviced via mmap(), where the worst case wastage is about
     32 bytes plus the remainder from a system page (the minimal
     mmap unit); typically 4096 or 8192 bytes.

Security: static-safe; optionally more or less
     The "security" of malloc refers to the ability of malicious
     code to accentuate the effects of errors (for example, freeing
     space that is not currently malloc'ed or overwriting past the
     ends of chunks) in code that calls malloc.  This malloc
     guarantees not to modify any memory locations below the base of
     heap, i.e., static variables, even in the presence of usage
     errors.  The routines additionally detect most improper frees
     and reallocs.  All this holds as long as the static bookkeeping
     for malloc itself is not corrupted by some other means.  This
     is only one aspect of security -- these checks do not, and
     cannot, detect all possible programming errors.

     If FOOTERS is defined nonzero, then each allocated chunk
     carries an additional check word to verify that it was malloced
     from its space.  These check words are the same within each
     execution of a program using malloc, but differ across
     executions, so externally crafted fake chunks cannot be
```



```
     freed. This improves security by rejecting frees/reallocs that
     could corrupt heap memory, in addition to the checks preventing
     writes to statics that are always on.  This may further improve
     security at the expense of time and space overhead.  (Note that
     FOOTERS may also be worth using with MSPACES.)

     By default detected errors cause the program to abort (calling
     "abort()"). You can override this to instead proceed past
     errors by defining PROCEED_ON_ERROR.  In this case, a bad free
     has no effect, and a malloc that encounters a bad address
     caused by user overwrites will ignore the bad address by
     dropping pointers and indices to all known memory. This may
     be appropriate for programs that should continue if at all
     possible in the face of programming errors, although they may
     run out of memory because dropped memory is never reclaimed.

     If you don't like either of these options, you can define
     CORRUPTION_ERROR_ACTION and USAGE_ERROR_ACTION to do anything
     else. And if if you are sure that your program using malloc has
     no errors or vulnerabilities, you can define INSECURE to 1,
     which might (or might not) provide a small performance improvement.

Thread-safety: NOT thread-safe unless USE_LOCKS defined
     When USE_LOCKS is defined, each public call to malloc, free,
     etc is surrounded with either a pthread mutex or a win32
     spinlock (depending on WIN32). This is not especially fast, and
     can be a major bottleneck.  It is designed only to provide
     minimal protection in concurrent environments, and to provide a
     basis for extensions.  If you are using malloc in a concurrent
     program, consider instead using nedmalloc
     (http://www.nedprod.com/programs/portable/nedmalloc/) or
     ptmalloc (See http://www.malloc.de), which are derived
     from versions of this malloc.

System requirements: Any combination of MORECORE and/or MMAP/MUNMAP
     This malloc can use unix sbrk or any emulation (invoked using
     the CALL_MORECORE macro) and/or mmap/munmap or any emulation
     (invoked using CALL_MMAP/CALL_MUNMAP) to get and release system
     memory.  On most unix systems, it tends to work best if both
     MORECORE and MMAP are enabled.  On Win32, it uses emulations
     based on VirtualAlloc. It also uses common C library functions
     like memset.

Compliance: I believe it is compliant with the Single Unix Specification
     (See http://www.unix.org). Also SVID/XPG, ANSI C, and probably
     others as well.
```

1.3 Overview of algorithms

```
This is not the fastest, most space-conserving, most portable, or
```

```
  most tunable malloc ever written. However it is among the fastest
  while also being among the most space-conserving, portable and
  tunable.  Consistent balance across these factors results in a good
  general-purpose allocator for malloc-intensive programs.

  In most ways, this malloc is a best-fit allocator. Generally, it
  chooses the best-fitting existing chunk for a request, with ties
  broken in approximately least-recently-used order. (This strategy
  normally maintains low fragmentation.) However, for requests less
  than 256bytes, it deviates from best-fit when there is not an
  exactly fitting available chunk by preferring to use space adjacent
  to that used for the previous small request, as well as by breaking
  ties in approximately most-recently-used order. (These enhance
  locality of series of small allocations.)  And for very large requests
  (>= 256Kb by default), it relies on system memory mapping
  facilities, if supported.  (This helps avoid carrying around and
  possibly fragmenting memory used only for large chunks.)

  All operations (except malloc_stats and mallinfo) have execution
  times that are bounded by a constant factor of the number of bits in
  a size_t, not counting any clearing in calloc or copying in realloc,
  or actions surrounding MORECORE and MMAP that have times
  proportional to the number of non-contiguous regions returned by
  system allocation routines, which is often just 1. In real-time
  applications, you can optionally suppress segment traversals using
  NO_SEGMENT_TRAVERSAL, which assures bounded execution even when
  system allocators return non-contiguous spaces, at the typical
  expense of carrying around more memory and increased fragmentation.

  The implementation is not very modular and seriously overuses
  macros. Perhaps someday all C compilers will do as good a job
  inlining modular code as can now be done by brute-force expansion,
  but now, enough of them seem not to.

  Some compilers issue a lot of warnings about code that is
  dead/unreachable only on some platforms, and also about intentional
  uses of negation on unsigned types. All known cases of each can be
  ignored.

  For a longer but out of date high-level description, see
     http://gee.cs.oswego.edu/dl/html/malloc.html
```

1.4 MSPACES

```
  If MSPACES is defined, then in addition to malloc, free, etc.,
  this file also defines mspace_malloc, mspace_free, etc. These
  are versions of malloc routines that take an "mspace" argument
  obtained using create_mspace, to control all internal bookkeeping.
  If ONLY_MSPACES is defined, only these versions are compiled.
  So if you would like to use this allocator for only some allocations,
  and your system malloc for others, you can compile with
```



```
  ONLY_MSPACES and then do something like...
    static mspace mymspace = create_mspace(0,0); // for example
    #define mymalloc(bytes)  mspace_malloc(mymspace, bytes)

  (Note: If you only need one instance of an mspace, you can instead
  use "USE_DL_PREFIX" to relabel the global malloc.)

  You can similarly create thread-local allocators by storing
  mspaces as thread-locals. For example:
    static __thread mspace tlms = 0;
    void*  tlmalloc(size_t bytes) {
      if (tlms == 0) tlms = create_mspace(0, 0);
      return mspace_malloc(tlms, bytes);
    }
    void  tlfree(void* mem) { mspace_free(tlms, mem); }

  Unless FOOTERS is defined, each mspace is completely independent.
  You cannot allocate from one and free to another (although
  conformance is only weakly checked, so usage errors are not always
  caught). If FOOTERS is defined, then each chunk carries around a tag
  indicating its originating mspace, and frees are directed to their
  originating spaces.
*/
```

Chapter 2

Compile-time options

```
/*
Be careful in setting #define values for numerical constants of type
size_t. On some systems, literal values are not automatically extended
to size_t precision unless they are explicitly casted. You can also
use the symbolic values MAX_SIZE_T, SIZE_T_ONE, etc below.

WIN32                    default: defined if _WIN32 defined
  Defining WIN32 sets up defaults for MS environment and compilers.
  Otherwise defaults are for unix. Beware that there seem to be some
  cases where this malloc might not be a pure drop-in replacement for
  Win32 malloc: Random-looking failures from Win32 GDI API's (eg;
  SetDIBits()) may be due to bugs in some video driver implementations
  when pixel buffers are malloc()ed, and the region spans more than
  one VirtualAlloc()ed region. Because dlmalloc uses a small (64Kb)
  default granularity, pixel buffers may straddle virtual allocation
  regions more often than when using the Microsoft allocator.  You can
  avoid this by using VirtualAlloc() and VirtualFree() for all pixel
  buffers rather than using malloc().  If this is not possible,
  recompile this malloc with a larger DEFAULT_GRANULARITY.

MALLOC_ALIGNMENT         default: (size_t)8
  Controls the minimum alignment for malloc'ed chunks.  It must be a
  power of two and at least 8, even on machines for which smaller
  alignments would suffice. It may be defined as larger than this
  though. Note however that code and data structures are optimized for
  the case of 8-byte alignment.

MSPACES                  default: 0 (false)
  If true, compile in support for independent allocation spaces.
  This is only supported if HAVE_MMAP is true.

ONLY_MSPACES             default: 0 (false)
  If true, only compile in mspace versions, not regular versions.

USE_LOCKS                default: 0 (false)
  Causes each call to each public routine to be surrounded with
  pthread or WIN32 mutex lock/unlock. (If set true, this can be
  overridden on a per-mspace basis for mspace versions.) If set to a
  non-zero value other than 1, locks are used, but their
```



```
  implementation is left out, so lock functions must be supplied manually,
  as described below.

USE_SPIN_LOCKS           default: 1 iff USE_LOCKS and on x86 using gcc or MSC
  If true, uses custom spin locks for locking. This is currently
  supported only for x86 platforms using gcc or recent MS compilers.
  Otherwise, posix locks or win32 critical sections are used.

FOOTERS                  default: 0
  If true, provide extra checking and dispatching by placing
  information in the footers of allocated chunks. This adds
  space and time overhead.

INSECURE                 default: 0
  If true, omit checks for usage errors and heap space overwrites.

USE_DL_PREFIX            default: NOT defined
  Causes compiler to prefix all public routines with the string 'dl'.
  This can be useful when you only want to use this malloc in one part
  of a program, using your regular system malloc elsewhere.

ABORT                    default: defined as abort()
  Defines how to abort on failed checks.  On most systems, a failed
  check cannot die with an "assert" or even print an informative
  message, because the underlying print routines in turn call malloc,
  which will fail again.  Generally, the best policy is to simply call
  abort(). It's not very useful to do more than this because many
  errors due to overwriting will show up as address faults (null, odd
  addresses etc) rather than malloc-triggered checks, so will also
  abort.  Also, most compilers know that abort() does not return, so
  can better optimize code conditionally calling it.

PROCEED_ON_ERROR           default: defined as 0 (false)
  Controls whether detected bad addresses cause them to bypassed
  rather than aborting. If set, detected bad arguments to free and
  realloc are ignored. And all bookkeeping information is zeroed out
  upon a detected overwrite of freed heap space, thus losing the
  ability to ever return it from malloc again, but enabling the
  application to proceed. If PROCEED_ON_ERROR is defined, the
  static variable malloc_corruption_error_count is compiled in
  and can be examined to see if errors have occurred. This option
  generates slower code than the default abort policy.

DEBUG                    default: NOT defined
  The DEBUG setting is mainly intended for people trying to modify
  this code or diagnose problems when porting to new platforms.
  However, it may also be able to better isolate user errors than just
  using runtime checks.  The assertions in the check routines spell
  out in more detail the assumptions and invariants underlying the
  algorithms.  The checking is fairly extensive, and will slow down
  execution noticeably. Calling malloc_stats or mallinfo with DEBUG
  set will attempt to check every non-mmapped allocated and free chunk
```

```
  in the course of computing the summaries.

ABORT_ON_ASSERT_FAILURE   default: defined as 1 (true)
  Debugging assertion failures can be nearly impossible if your
  version of the assert macro causes malloc to be called, which will
  lead to a cascade of further failures, blowing the runtime stack.
  ABORT_ON_ASSERT_FAILURE cause assertions failures to call abort(),
  which will usually make debugging easier.

MALLOC_FAILURE_ACTION     default: sets errno to ENOMEM, or no-op on win32
  The action to take before "return 0" when malloc fails to be able to
  return memory because there is none available.

HAVE_MORECORE             default: 1 (true) unless win32 or ONLY_MSPACES
  True if this system supports sbrk or an emulation of it.

MORECORE                  default: sbrk
  The name of the sbrk-style system routine to call to obtain more
  memory.  See below for guidance on writing custom MORECORE
  functions. The type of the argument to sbrk/MORECORE varies across
  systems.  It cannot be size_t, because it supports negative
  arguments, so it is normally the signed type of the same width as
  size_t (sometimes declared as "intptr_t").  It doesn't much matter
  though. Internally, we only call it with arguments less than half
  the max value of a size_t, which should work across all reasonable
  possibilities, although sometimes generating compiler warnings.

MORECORE_CONTIGUOUS       default: 1 (true) if HAVE_MORECORE
  If true, take advantage of fact that consecutive calls to MORECORE
  with positive arguments always return contiguous increasing
  addresses.  This is true of unix sbrk. It does not hurt too much to
  set it true anyway, since malloc copes with non-contiguities.
  Setting it false when definitely non-contiguous saves time
  and possibly wasted space it would take to discover this though.

MORECORE_CANNOT_TRIM      default: NOT defined
  True if MORECORE cannot release space back to the system when given
  negative arguments. This is generally necessary only if you are
  using a hand-crafted MORECORE function that cannot handle negative
  arguments.

NO_SEGMENT_TRAVERSAL       default: 0
  If non-zero, suppresses traversals of memory segments
  returned by either MORECORE or CALL_MMAP. This disables
  merging of segments that are contiguous, and selectively
  releasing them to the OS if unused, but bounds execution times.

HAVE_MMAP                 default: 1 (true)
  True if this system supports mmap or an emulation of it.  If so, and
  HAVE_MORECORE is not true, MMAP is used for all system
  allocation. If set and HAVE_MORECORE is true as well, MMAP is
  primarily used to directly allocate very large blocks. It is also
```



```
  used as a backup strategy in cases where MORECORE fails to provide
  space from system. Note: A single call to MUNMAP is assumed to be
  able to unmap memory that may have be allocated using multiple calls
  to MMAP, so long as they are adjacent.

HAVE_MREMAP               default: 1 on linux, else 0
  If true realloc() uses mremap() to re-allocate large blocks and
  extend or shrink allocation spaces.

MMAP_CLEARS               default: 1 except on WINCE.
  True if mmap clears memory so calloc doesn't need to. This is true
  for standard unix mmap using /dev/zero and on WIN32 except for WINCE.

USE_BUILTIN_FFS            default: 0 (i.e., not used)
  Causes malloc to use the builtin ffs() function to compute indices.
  Some compilers may recognize and intrinsify ffs to be faster than the
  supplied C version. Also, the case of x86 using gcc is special-cased
  to an asm instruction, so is already as fast as it can be, and so
  this setting has no effect. Similarly for Win32 under recent MS compilers.
  (On most x86s, the asm version is only slightly faster than the C version.)

malloc_getpagesize         default: derive from system includes, or 4096.
  The system page size. To the extent possible, this malloc manages
  memory from the system in page-size units.  This may be (and
  usually is) a function rather than a constant. This is ignored
  if WIN32, where page size is determined using getSystemInfo during
  initialization.

USE_DEV_RANDOM             default: 0 (i.e., not used)
  Causes malloc to use /dev/random to initialize secure magic seed for
  stamping footers. Otherwise, the current time is used.

NO_MALLINFO                default: 0
  If defined, don't compile "mallinfo". This can be a simple way
  of dealing with mismatches between system declarations and
  those in this file.

MALLINFO_FIELD_TYPE        default: size_t
  The type of the fields in the mallinfo struct. This was originally
  defined as "int" in SVID etc, but is more usefully defined as
  size_t. The value is used only if  HAVE_USR_INCLUDE_MALLOC_H is not set

REALLOC_ZERO_BYTES_FREES    default: not defined
  This should be set if a call to realloc with zero bytes should
  be the same as a call to free. Some people think it should. Otherwise,
  since this malloc returns a unique pointer for malloc(0), so does
  realloc(p, 0).

LACKS_UNISTD_H, LACKS_FCNTL_H, LACKS_SYS_PARAM_H, LACKS_SYS_MMAN_H
LACKS_STRINGS_H, LACKS_STRING_H, LACKS_SYS_TYPES_H,  LACKS_ERRNO_H
LACKS_STDLIB_H                default: NOT defined unless on WIN32
  Define these if your system does not have these header files.
```

```
  You might need to manually insert some of the declarations they provide.

DEFAULT_GRANULARITY        default: page size if MORECORE_CONTIGUOUS,
                                system_info.dwAllocationGranularity in WIN32,
                                otherwise 64K.
      Also settable using mallopt(M_GRANULARITY, x)
  The unit for allocating and deallocating memory from the system.  On
  most systems with contiguous MORECORE, there is no reason to
  make this more than a page. However, systems with MMAP tend to
  either require or encourage larger granularities.  You can increase
  this value to prevent system allocation functions to be called so
  often, especially if they are slow.  The value must be at least one
  page and must be a power of two.  Setting to 0 causes initialization
  to either page size or win32 region size.  (Note: In previous
  versions of malloc, the equivalent of this option was called
  "TOP_PAD")

DEFAULT_TRIM_THRESHOLD    default: 2MB
      Also settable using mallopt(M_TRIM_THRESHOLD, x)
  The maximum amount of unused top-most memory to keep before
  releasing via malloc_trim in free().  Automatic trimming is mainly
  useful in long-lived programs using contiguous MORECORE.  Because
  trimming via sbrk can be slow on some systems, and can sometimes be
  wasteful (in cases where programs immediately afterward allocate
  more large chunks) the value should be high enough so that your
  overall system performance would improve by releasing this much
  memory.  As a rough guide, you might set to a value close to the
  average size of a process (program) running on your system.
  Releasing this much memory would allow such a process to run in
  memory.  Generally, it is worth tuning trim thresholds when a
  program undergoes phases where several large chunks are allocated
  and released in ways that can reuse each other's storage, perhaps
  mixed with phases where there are no such chunks at all. The trim
  value must be greater than page size to have any useful effect.  To
  disable trimming completely, you can set to MAX_SIZE_T. Note that the trick
  some people use of mallocing a huge space and then freeing it at
  program startup, in an attempt to reserve system memory, doesn't
  have the intended effect under automatic trimming, since that memory
  will immediately be returned to the system.

DEFAULT_MMAP_THRESHOLD       default: 256K
      Also settable using mallopt(M_MMAP_THRESHOLD, x)
  The request size threshold for using MMAP to directly service a
  request. Requests of at least this size that cannot be allocated
  using already-existing space will be serviced via mmap.  (If enough
  normal freed space already exists it is used instead.)  Using mmap
  segregates relatively large chunks of memory so that they can be
  individually obtained and released from the host system. A request
  serviced through mmap is never reused by any other request (at least
  not directly; the system may just so happen to remap successive
  requests to the same locations).  Segregating space in this way has
  the benefits that: Mmapped space can always be individually released
```



```
  back to the system, which helps keep the system level memory demands
  of a long-lived program low.  Also, mapped memory doesn't become
  'locked' between other chunks, as can happen with normally allocated
  chunks, which means that even trimming via malloc_trim would not
  release them.  However, it has the disadvantage that the space
  cannot be reclaimed, consolidated, and then used to service later
  requests, as happens with normal chunks.  The advantages of mmap
  nearly always outweigh disadvantages for "large" chunks, but the
  value of "large" may vary across systems.  The default is an
  empirically derived value that works well in most systems. You can
  disable mmap by setting to MAX_SIZE_T.

MAX_RELEASE_CHECK_RATE   default: 4095 unless not HAVE_MMAP
  The number of consolidated frees between checks to release
  unused segments when freeing. When using non-contiguous segments,
  especially with multiple mspaces, checking only for topmost space
  doesn't always suffice to trigger trimming. To compensate for this,
  free() will, with a period of MAX_RELEASE_CHECK_RATE (or the
  current number of segments, if greater) try to release unused
  segments to the OS when freeing chunks that result in
  consolidation. The best value for this parameter is a compromise
  between slowing down frees with relatively costly checks that
  rarely trigger versus holding on to unused memory. To effectively
  disable, set to MAX_SIZE_T. This may lead to a very slight speed
  improvement at the expense of carrying around more memory.
*/

/* Version identifier to allow people to support multiple versions */
#ifndef DLMALLOC_VERSION
#define DLMALLOC_VERSION 20804
#endif /* DLMALLOC_VERSION */

#ifndef WIN32
#ifdef _WIN32
#define WIN32 1
#endif  /* _WIN32 */
#ifdef _WIN32_WCE
#define LACKS_FCNTL_H
#define WIN32 1
#endif /* _WIN32_WCE */
#endif  /* WIN32 */
#ifdef WIN32
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#define HAVE_MMAP 1
#define HAVE_MORECORE 0
#define LACKS_UNISTD_H
#define LACKS_SYS_PARAM_H
#define LACKS_SYS_MMAN_H
#define LACKS_STRING_H
#define LACKS_STRINGS_H
#define LACKS_SYS_TYPES_H
```

```
#define LACKS_ERRNO_H
#ifndef MALLOC_FAILURE_ACTION
#define MALLOC_FAILURE_ACTION
#endif /* MALLOC_FAILURE_ACTION */
#ifdef _WIN32_WCE /* WINCE reportedly does not clear */
#define MMAP_CLEARS 0
#else
#define MMAP_CLEARS 1
#endif /* _WIN32_WCE */
#endif  /* WIN32 */

#if defined(DARWIN) || defined(_DARWIN)
/* Mac OSX docs advise not to use sbrk; it seems better to use mmap */
#ifndef HAVE_MORECORE
#define HAVE_MORECORE 0
#define HAVE_MMAP 1
/* OSX allocators provide 16 byte alignment */
#ifndef MALLOC_ALIGNMENT
#define MALLOC_ALIGNMENT ((size_t)16U)
#endif
#endif  /* HAVE_MORECORE */
#endif  /* DARWIN */

#ifndef LACKS_SYS_TYPES_H
#include <sys/types.h>  /* For size_t */
#endif  /* LACKS_SYS_TYPES_H */

#if (defined(__GNUC__) && ((defined(__i386__) ||  defined(__x86_64__)))) \
         || (defined(_MSC_VER) && _MSC_VER>=1310)
#define SPIN_LOCKS_AVAILABLE 1
#else
#define SPIN_LOCKS_AVAILABLE 0
#endif

/* The maximum possible size_t value has all bits set */
#define MAX_SIZE_T           (~(size_t)0)

#ifndef ONLY_MSPACES
#define ONLY_MSPACES 0     /* define to a value */
#else
#define ONLY_MSPACES 1
#endif  /* ONLY_MSPACES */
#ifndef MSPACES
#if ONLY_MSPACES
#define MSPACES 1
#else   /* ONLY_MSPACES */
#define MSPACES 0
#endif  /* ONLY_MSPACES */
#endif  /* MSPACES */
#ifndef MALLOC_ALIGNMENT
#define MALLOC_ALIGNMENT ((size_t)8U)
#endif  /* MALLOC_ALIGNMENT */
```



```
#ifndef FOOTERS
#define FOOTERS 0
#endif  /* FOOTERS */
#ifndef ABORT
#define ABORT  abort()
#endif  /* ABORT */
#ifndef ABORT_ON_ASSERT_FAILURE
#define ABORT_ON_ASSERT_FAILURE 1
#endif  /* ABORT_ON_ASSERT_FAILURE */
#ifndef PROCEED_ON_ERROR
#define PROCEED_ON_ERROR 0
#endif  /* PROCEED_ON_ERROR */
#ifndef USE_LOCKS
#define USE_LOCKS 0
#endif  /* USE_LOCKS */
#ifndef USE_SPIN_LOCKS
#if USE_LOCKS && SPIN_LOCKS_AVAILABLE
#define USE_SPIN_LOCKS 1
#else
#define USE_SPIN_LOCKS 0
#endif /* USE_LOCKS && SPIN_LOCKS_AVAILABLE. */
#endif /* USE_SPIN_LOCKS */
#ifndef INSECURE
#define INSECURE 0
#endif  /* INSECURE */
#ifndef HAVE_MMAP
#define HAVE_MMAP 1
#endif  /* HAVE_MMAP */
#ifndef MMAP_CLEARS
#define MMAP_CLEARS 1
#endif  /* MMAP_CLEARS */
#ifndef HAVE_MREMAP
#ifdef linux
#define HAVE_MREMAP 1
#else   /* linux */
#define HAVE_MREMAP 0
#endif  /* linux */
#endif  /* HAVE_MREMAP */
#ifndef MALLOC_FAILURE_ACTION
#define MALLOC_FAILURE_ACTION  errno = ENOMEM;
#endif  /* MALLOC_FAILURE_ACTION */
#ifndef HAVE_MORECORE
#if ONLY_MSPACES
#define HAVE_MORECORE 0
#else   /* ONLY_MSPACES */
#define HAVE_MORECORE 1
#endif  /* ONLY_MSPACES */
#endif  /* HAVE_MORECORE */
#if !HAVE_MORECORE
#define MORECORE_CONTIGUOUS 0
#else   /* !HAVE_MORECORE */
#define MORECORE_DEFAULT sbrk
```

```
#ifndef MORECORE_CONTIGUOUS
#define MORECORE_CONTIGUOUS 1
#endif  /* MORECORE_CONTIGUOUS */
#endif  /* HAVE_MORECORE */
#ifndef DEFAULT_GRANULARITY
#if (MORECORE_CONTIGUOUS || defined(WIN32))
#define DEFAULT_GRANULARITY (0)  /* 0 means to compute in init_mparams */
#else   /* MORECORE_CONTIGUOUS */
#define DEFAULT_GRANULARITY ((size_t)64U * (size_t)1024U)
#endif  /* MORECORE_CONTIGUOUS */
#endif  /* DEFAULT_GRANULARITY */
#ifndef DEFAULT_TRIM_THRESHOLD
#ifndef MORECORE_CANNOT_TRIM
#define DEFAULT_TRIM_THRESHOLD ((size_t)2U * (size_t)1024U * (size_t)1024U)
#else   /* MORECORE_CANNOT_TRIM */
#define DEFAULT_TRIM_THRESHOLD MAX_SIZE_T
#endif  /* MORECORE_CANNOT_TRIM */
#endif  /* DEFAULT_TRIM_THRESHOLD */
#ifndef DEFAULT_MMAP_THRESHOLD
#if HAVE_MMAP
#define DEFAULT_MMAP_THRESHOLD ((size_t)256U * (size_t)1024U)
#else   /* HAVE_MMAP */
#define DEFAULT_MMAP_THRESHOLD MAX_SIZE_T
#endif  /* HAVE_MMAP */
#endif  /* DEFAULT_MMAP_THRESHOLD */
#ifndef MAX_RELEASE_CHECK_RATE
#if HAVE_MMAP
#define MAX_RELEASE_CHECK_RATE 4095
#else
#define MAX_RELEASE_CHECK_RATE MAX_SIZE_T
#endif /* HAVE_MMAP */
#endif /* MAX_RELEASE_CHECK_RATE */
#ifndef USE_BUILTIN_FFS
#define USE_BUILTIN_FFS 0
#endif  /* USE_BUILTIN_FFS */
#ifndef USE_DEV_RANDOM
#define USE_DEV_RANDOM 0
#endif  /* USE_DEV_RANDOM */
#ifndef NO_MALLINFO
#define NO_MALLINFO 0
#endif  /* NO_MALLINFO */
#ifndef MALLINFO_FIELD_TYPE
#define MALLINFO_FIELD_TYPE size_t
#endif  /* MALLINFO_FIELD_TYPE */
#ifndef NO_SEGMENT_TRAVERSAL
#define NO_SEGMENT_TRAVERSAL 0
#endif /* NO_SEGMENT_TRAVERSAL */

/*
  mallopt tuning options.  SVID/XPG defines four standard parameter
  numbers for mallopt, normally defined in malloc.h.  None of these
  are used in this malloc, so setting them has no effect. But this
  malloc does support the following options.
*/

#define M_TRIM_THRESHOLD     (-1)
#define M_GRANULARITY        (-2)
#define M_MMAP_THRESHOLD     (-3)
```

Chapter 3

Mallinfo declarations

```
#if !NO_MALLINFO
/*
  This version of malloc supports the standard SVID/XPG mallinfo
  routine that returns a struct containing usage properties and
  statistics. It should work on any system that has a
  /usr/include/malloc.h defining struct mallinfo.  The main
  declaration needed is the mallinfo struct that is returned (by-copy)
  by mallinfo().  The malloinfo struct contains a bunch of fields that
  are not even meaningful in this version of malloc.  These fields are
  are instead filled by mallinfo() with other numbers that might be of
  interest.

  HAVE_USR_INCLUDE_MALLOC_H should be set if you have a
  /usr/include/malloc.h file that includes a declaration of struct
  mallinfo.  If so, it is included; else a compliant version is
  declared below.  These must be precisely the same for mallinfo() to
  work.  The original SVID version of this struct, defined on most
  systems with mallinfo, declares all fields as ints. But some others
  define as unsigned long. If your system defines the fields using a
  type of different width than listed here, you MUST #include your
  system version and #define HAVE_USR_INCLUDE_MALLOC_H.
*/

/* #define HAVE_USR_INCLUDE_MALLOC_H */

#ifdef HAVE_USR_INCLUDE_MALLOC_H
#include "/usr/include/malloc.h"
#else /* HAVE_USR_INCLUDE_MALLOC_H */
#ifndef STRUCT_MALLINFO_DECLARED
#define STRUCT_MALLINFO_DECLARED 1
struct mallinfo {
  MALLINFO_FIELD_TYPE arena;    /* non-mmapped space allocated from system */
  MALLINFO_FIELD_TYPE ordblks;  /* number of free chunks */
  MALLINFO_FIELD_TYPE smblks;   /* always 0 */
  MALLINFO_FIELD_TYPE hblks;    /* always 0 */
  MALLINFO_FIELD_TYPE hblkhd;   /* space in mmapped regions */
  MALLINFO_FIELD_TYPE usmblks;  /* maximum total allocated space */
  MALLINFO_FIELD_TYPE fsmblks;  /* always 0 */
```



```
  MALLINFO_FIELD_TYPE uordblks; /* total allocated space */
  MALLINFO_FIELD_TYPE fordblks; /* total free space */
  MALLINFO_FIELD_TYPE keepcost; /* releasable (via malloc_trim) space */
};
#endif /* STRUCT_MALLINFO_DECLARED */
#endif /* HAVE_USR_INCLUDE_MALLOC_H */
#endif /* NO_MALLINFO */

/*
  Try to persuade compilers to inline. The most critical functions for
  inlining are defined as macros, so these aren't used for them.
*/

#ifndef FORCEINLINE
  #if defined(__GNUC__)
#define FORCEINLINE __inline __attribute__ ((always_inline))
  #elif defined(_MSC_VER)
    #define FORCEINLINE __forceinline
  #endif
#endif
#ifndef NOINLINE
  #if defined(__GNUC__)
    #define NOINLINE __attribute__ ((noinline))
  #elif defined(_MSC_VER)
    #define NOINLINE __declspec(noinline)
  #else
    #define NOINLINE
  #endif
#endif

#ifdef __cplusplus
extern "C" {
#ifndef FORCEINLINE
 #define FORCEINLINE inline
#endif
#endif /* __cplusplus */
#ifndef FORCEINLINE
 #define FORCEINLINE
#endif

#if !ONLY_MSPACES
```

Chapter 4

Declarations of public routines

```
#ifndef USE_DL_PREFIX
#define dlcalloc               calloc
#define dlfree                 free
#define dlmalloc               malloc
#define dlmemalign             memalign
#define dlrealloc              realloc
#define dlvalloc               valloc
#define dlpvalloc              pvalloc
#define dlmallinfo             mallinfo
#define dlmallopt              mallopt
#define dlmalloc_trim          malloc_trim
#define dlmalloc_stats         malloc_stats
#define dlmalloc_usable_size   malloc_usable_size
#define dlmalloc_footprint     malloc_footprint
#define dlmalloc_max_footprint malloc_max_footprint
#define dlindependent_calloc   independent_calloc
#define dlindependent_comalloc independent_comalloc
#endif /* USE_DL_PREFIX */


/*
  malloc(size_t n)
  Returns a pointer to a newly allocated chunk of at least n bytes, or
  null if no space is available, in which case errno is set to ENOMEM
  on ANSI C systems.

  If n is zero, malloc returns a minimum-sized chunk. (The minimum
  size is 16 bytes on most 32bit systems, and 32 bytes on 64bit
  systems.)  Note that size_t is an unsigned type, so calls with
  arguments that would be negative if signed are interpreted as
  requests for huge amounts of space, which will often fail. The
  maximum supported value of n differs across systems, but is in all
  cases less than the maximum representable value of a size_t.
*/
void* dlmalloc(size_t);

/*
  free(void* p)
```



```
  Releases the chunk of memory pointed to by p, that had been previously
  allocated using malloc or a related routine such as realloc.
  It has no effect if p is null. If p was not malloced or already
  freed, free(p) will by default cause the current program to abort.
*/
void  dlfree(void*);

/*
  calloc(size_t n_elements, size_t element_size);
  Returns a pointer to n_elements * element_size bytes, with all locations
  set to zero.
*/
void* dlcalloc(size_t, size_t);

/*
  realloc(void* p, size_t n)
  Returns a pointer to a chunk of size n that contains the same data
  as does chunk p up to the minimum of (n, p's size) bytes, or null
  if no space is available.

  The returned pointer may or may not be the same as p. The algorithm
  prefers extending p in most cases when possible, otherwise it
  employs the equivalent of a malloc-copy-free sequence.

  If p is null, realloc is equivalent to malloc.

  If space is not available, realloc returns null, errno is set (if on
  ANSI) and p is NOT freed.

  if n is for fewer bytes than already held by p, the newly unused
  space is lopped off and freed if possible.  realloc with a size
  argument of zero (re)allocates a minimum-sized chunk.

  The old unix realloc convention of allowing the last-free'd chunk
  to be used as an argument to realloc is not supported.
*/

void* dlrealloc(void*, size_t);

/*
  memalign(size_t alignment, size_t n);
  Returns a pointer to a newly allocated chunk of n bytes, aligned
  in accord with the alignment argument.

  The alignment argument should be a power of two. If the argument is
  not a power of two, the nearest greater power is used.
  8-byte alignment is guaranteed by normal malloc calls, so don't
  bother calling memalign with an argument of 8 or less.

  Overreliance on memalign is a sure way to fragment space.
*/
void* dlmemalign(size_t, size_t);
```

```
/*
  valloc(size_t n);
  Equivalent to memalign(pagesize, n), where pagesize is the page
  size of the system. If the pagesize is unknown, 4096 is used.
*/
void* dlvalloc(size_t);

/*
  mallopt(int parameter_number, int parameter_value)
  Sets tunable parameters The format is to provide a
  (parameter-number, parameter-value) pair.  mallopt then sets the
  corresponding parameter to the argument value if it can (i.e., so
  long as the value is meaningful), and returns 1 if successful else
  0.  To workaround the fact that mallopt is specified to use int,
  not size_t parameters, the value -1 is specially treated as the
  maximum unsigned size_t value.

  SVID/XPG/ANSI defines four standard param numbers for mallopt,
  normally defined in malloc.h.  None of these are use in this malloc,
  so setting them has no effect. But this malloc also supports other
  options in mallopt. See below for details.  Briefly, supported
  parameters are as follows (listed defaults are for "typical"
  configurations).

  Symbol            param #  default    allowed param values
  M_TRIM_THRESHOLD     -1   2*1024*1024   any   (-1 disables)
  M_GRANULARITY        -2     page size   any power of 2 >= page size
  M_MMAP_THRESHOLD     -3      256*1024   any   (or 0 if no MMAP support)
*/
int dlmallopt(int, int);

/*
  malloc_footprint();
  Returns the number of bytes obtained from the system.  The total
  number of bytes allocated by malloc, realloc etc., is less than this
  value. Unlike mallinfo, this function returns only a precomputed
  result, so can be called frequently to monitor memory consumption.
  Even if locks are otherwise defined, this function does not use them,
  so results might not be up to date.
*/
size_t dlmalloc_footprint(void);

/*
  malloc_max_footprint();
  Returns the maximum number of bytes obtained from the system. This
  value will be greater than current footprint if deallocated space
  has been reclaimed by the system. The peak number of bytes allocated
  by malloc, realloc etc., is less than this value. Unlike mallinfo,
  this function returns only a precomputed result, so can be called
  frequently to monitor memory consumption.  Even if locks are
  otherwise defined, this function does not use them, so results might
```



```
  not be up to date.
*/
size_t dlmalloc_max_footprint(void);

#if !NO_MALLINFO
/*
  mallinfo()
  Returns (by copy) a struct containing various summary statistics:

  arena:     current total non-mmapped bytes allocated from system
  ordblks:   the number of free chunks
  smblks:    always zero.
  hblks:     current number of mmapped regions
  hblkhd:    total bytes held in mmapped regions
  usmblks:   the maximum total allocated space. This will be greater
                than current total if trimming has occurred.
  fsmblks:   always zero
  uordblks:  current total allocated space (normal or mmapped)
  fordblks:  total free space
  keepcost:  the maximum number of bytes that could ideally be released
               back to system via malloc_trim. ("ideally" means that
               it ignores page restrictions etc.)

  Because these fields are ints, but internal bookkeeping may
  be kept as longs, the reported values may wrap around zero and
  thus be inaccurate.
*/
struct mallinfo dlmallinfo(void);
#endif /* NO_MALLINFO */

/*
  independent_calloc(size_t n_elements, size_t element_size, void* chunks[]);

  independent_calloc is similar to calloc, but instead of returning a
  single cleared space, it returns an array of pointers to n_elements
  independent elements that can hold contents of size elem_size, each
  of which starts out cleared, and can be independently freed,
  realloc'ed etc. The elements are guaranteed to be adjacently
  allocated (this is not guaranteed to occur with multiple callocs or
  mallocs), which may also improve cache locality in some
  applications.

  The "chunks" argument is optional (i.e., may be null, which is
  probably the most typical usage). If it is null, the returned array
  is itself dynamically allocated and should also be freed when it is
  no longer needed. Otherwise, the chunks array must be of at least
  n_elements in length. It is filled in with the pointers to the
  chunks.

  In either case, independent_calloc returns this pointer array, or
  null if the allocation failed.  If n_elements is zero and "chunks"
  is null, it returns a chunk representing an array with zero elements
```

```
  (which should be freed if not wanted).

  Each element must be individually freed when it is no longer
  needed. If you'd like to instead be able to free all at once, you
  should instead use regular calloc and assign pointers into this
  space to represent elements.  (In this case though, you cannot
  independently free elements.)

  independent_calloc simplifies and speeds up implementations of many
  kinds of pools.  It may also be useful when constructing large data
  structures that initially have a fixed number of fixed-sized nodes,
  but the number is not known at compile time, and some of the nodes
  may later need to be freed. For example:

  struct Node { int item; struct Node* next; };

  struct Node* build_list() {
    struct Node** pool;
    int n = read_number_of_nodes_needed();
    if (n <= 0) return 0;
    pool = (struct Node**)(independent_calloc(n, sizeof(struct Node), 0);
    if (pool == 0) die();
    // organize into a linked list...
    struct Node* first = pool[0];
    for (i = 0; i < n-1; ++i)
      pool[i]->next = pool[i+1];
    free(pool);     // Can now free the array (or not, if it is needed later)
    return first;
  }
*/
void** dlindependent_calloc(size_t, size_t, void**);

/*
  independent_comalloc(size_t n_elements, size_t sizes[], void* chunks[]);

  independent_comalloc allocates, all at once, a set of n_elements
  chunks with sizes indicated in the "sizes" array.    It returns
  an array of pointers to these elements, each of which can be
  independently freed, realloc'ed etc. The elements are guaranteed to
  be adjacently allocated (this is not guaranteed to occur with
  multiple callocs or mallocs), which may also improve cache locality
  in some applications.

  The "chunks" argument is optional (i.e., may be null). If it is null
  the returned array is itself dynamically allocated and should also
  be freed when it is no longer needed. Otherwise, the chunks array
  must be of at least n_elements in length. It is filled in with the
  pointers to the chunks.

  In either case, independent_comalloc returns this pointer array, or
  null if the allocation failed.  If n_elements is zero and chunks is
  null, it returns a chunk representing an array with zero elements
```



```
  (which should be freed if not wanted).

  Each element must be individually freed when it is no longer
  needed. If you'd like to instead be able to free all at once, you
  should instead use a single regular malloc, and assign pointers at
  particular offsets in the aggregate space. (In this case though, you
  cannot independently free elements.)

  independent_comallac differs from independent_calloc in that each
  element may have a different size, and also that it does not
  automatically clear elements.

  independent_comalloc can be used to speed up allocation in cases
  where several structs or objects must always be allocated at the
  same time.  For example:

  struct Head { ... }
  struct Foot { ... }

  void send_message(char* msg) {
    int msglen = strlen(msg);
    size_t sizes[3] = { sizeof(struct Head), msglen, sizeof(struct Foot) };
    void* chunks[3];
    if (independent_comalloc(3, sizes, chunks) == 0)
      die();
    struct Head* head = (struct Head*)(chunks[0]);
    char*       body = (char*)(chunks[1]);
    struct Foot* foot = (struct Foot*)(chunks[2]);
    // ...
  }

  In general though, independent_comalloc is worth using only for
  larger values of n_elements. For small values, you probably won't
  detect enough difference from series of malloc calls to bother.

  Overuse of independent_comalloc can increase overall memory usage,
  since it cannot reuse existing noncontiguous small chunks that
  might be available for some of the elements.
*/
void** dlindependent_comalloc(size_t, size_t*, void**);


/*
  pvalloc(size_t n);
  Equivalent to valloc(minimum-page-that-holds(n)), that is,
  round up n to nearest pagesize.
 */
void*  dlpvalloc(size_t);

/*
  malloc_trim(size_t pad);
```

```
  If possible, gives memory back to the system (via negative arguments
  to sbrk) if there is unused memory at the ‘high’ end of the malloc
  pool or in unused MMAP segments. You can call this after freeing
  large blocks of memory to potentially reduce the system-level memory
  requirements of a program. However, it cannot guarantee to reduce
  memory. Under some allocation patterns, some large free blocks of
  memory will be locked between two used chunks, so they cannot be
  given back to the system.

  The ‘pad’ argument to malloc_trim represents the amount of free
  trailing space to leave untrimmed. If this argument is zero, only
  the minimum amount of memory to maintain internal data structures
  will be left. Non-zero arguments can be supplied to maintain enough
  trailing space to service future expected allocations without having
  to re-obtain memory from the system.

  Malloc_trim returns 1 if it actually released any memory, else 0.
*/
int  dlmalloc_trim(size_t);

/*
  malloc_stats();
  Prints on stderr the amount of space obtained from the system (both
  via sbrk and mmap), the maximum amount (which may be more than
  current if malloc_trim and/or munmap got called), and the current
  number of bytes allocated via malloc (or realloc, etc) but not yet
  freed. Note that this is the number of bytes allocated, not the
  number requested. It will be larger than the number requested
  because of alignment and bookkeeping overhead. Because it includes
  alignment wastage as being in use, this figure may be greater than
  zero even when no user-level chunks are allocated.

  The reported current and maximum system memory can be inaccurate if
  a program makes other calls to system memory allocation functions
  (normally sbrk) outside of malloc.

  malloc_stats prints only the most commonly interesting statistics.
  More information can be obtained by calling mallinfo.
*/
void  dlmalloc_stats(void);

#endif /* ONLY_MSPACES */

/*
  malloc_usable_size(void* p);

  Returns the number of bytes you can actually use in
  an allocated chunk, which may be more than you requested (although
  often not) due to alignment and minimum size constraints.
  You can use this many bytes without worrying about
  overwriting other allocated objects. This is not a particularly great
  programming practice. malloc_usable_size can be more useful in
```



```
  debugging and assertions, for example:

  p = malloc(n);
  assert(malloc_usable_size(p) >= 256);
*/
size_t dlmalloc_usable_size(void*);


#if MSPACES

/*
  mspace is an opaque type representing an independent
  region of space that supports mspace_malloc, etc.
*/
typedef void* mspace;

/*
  create_mspace creates and returns a new independent space with the
  given initial capacity, or, if 0, the default granularity size.  It
  returns null if there is no system memory available to create the
  space.  If argument locked is non-zero, the space uses a separate
  lock to control access. The capacity of the space will grow
  dynamically as needed to service mspace_malloc requests.  You can
  control the sizes of incremental increases of this space by
  compiling with a different DEFAULT_GRANULARITY or dynamically
  setting with mallopt(M_GRANULARITY, value).
*/
mspace create_mspace(size_t capacity, int locked);

/*
  destroy_mspace destroys the given space, and attempts to return all
  of its memory back to the system, returning the total number of
  bytes freed. After destruction, the results of access to all memory
  used by the space become undefined.
*/
size_t destroy_mspace(mspace msp);

/*
  create_mspace_with_base uses the memory supplied as the initial base
  of a new mspace. Part (less than 128*sizeof(size_t) bytes) of this
  space is used for bookkeeping, so the capacity must be at least this
  large. (Otherwise 0 is returned.) When this initial space is
  exhausted, additional memory will be obtained from the system.
  Destroying this space will deallocate all additionally allocated
  space (if possible) but not the initial base.
*/
mspace create_mspace_with_base(void* base, size_t capacity, int locked);

/*
  mspace_track_large_chunks controls whether requests for large chunks
  are allocated in their own untracked mmapped regions, separate from
  others in this mspace. By default large chunks are not tracked,
```

```
  which reduces fragmentation. However, such chunks are not
  necessarily released to the system upon destroy_mspace.  Enabling
  tracking by setting to true may increase fragmentation, but avoids
  leakage when relying on destroy_mspace to release all memory
  allocated using this space.  The function returns the previous
  setting.
*/
int mspace_track_large_chunks(mspace msp, int enable);


/*
  mspace_malloc behaves as malloc, but operates within
  the given space.
*/
void* mspace_malloc(mspace msp, size_t bytes);

/*
  mspace_free behaves as free, but operates within
  the given space.

  If compiled with FOOTERS==1, mspace_free is not actually needed.
  free may be called instead of mspace_free because freed chunks from
  any space are handled by their originating spaces.
*/
void mspace_free(mspace msp, void* mem);

/*
  mspace_realloc behaves as realloc, but operates within
  the given space.

  If compiled with FOOTERS==1, mspace_realloc is not actually
  needed.  realloc may be called instead of mspace_realloc because
  realloced chunks from any space are handled by their originating
  spaces.
*/
void* mspace_realloc(mspace msp, void* mem, size_t newsize);

/*
  mspace_calloc behaves as calloc, but operates within
  the given space.
*/
void* mspace_calloc(mspace msp, size_t n_elements, size_t elem_size);

/*
  mspace_memalign behaves as memalign, but operates within
  the given space.
*/
void* mspace_memalign(mspace msp, size_t alignment, size_t bytes);

/*
  mspace_independent_calloc behaves as independent_calloc, but
  operates within the given space.
```



```
*/
void** mspace_independent_calloc(mspace msp, size_t n_elements,
                                 size_t elem_size, void* chunks[]);

/*
  mspace_independent_comalloc behaves as independent_comalloc, but
  operates within the given space.
*/
void** mspace_independent_comalloc(mspace msp, size_t n_elements,
                                   size_t sizes[], void* chunks[]);

/*
  mspace_footprint() returns the number of bytes obtained from the
  system for this space.
*/
size_t mspace_footprint(mspace msp);

/*
  mspace_max_footprint() returns the peak number of bytes obtained from the
  system for this space.
*/
size_t mspace_max_footprint(mspace msp);


#if !NO_MALLINFO
/*
  mspace_mallinfo behaves as mallinfo, but reports properties of
  the given space.
*/
struct mallinfo mspace_mallinfo(mspace msp);
#endif /* NO_MALLINFO */

/*
  malloc_usable_size(void* p) behaves the same as malloc_usable_size;
*/
  size_t mspace_usable_size(void* mem);

/*
  mspace_malloc_stats behaves as malloc_stats, but reports
  properties of the given space.
*/
void mspace_malloc_stats(mspace msp);

/*
  mspace_trim behaves as malloc_trim, but
  operates within the given space.
*/
int mspace_trim(mspace msp, size_t pad);

/*
  An alias for mallopt.
*/
```

```
int mspace_mallopt(int, int);

#endif /* MSPACES */

#ifdef __cplusplus
};  /* end of extern "C" */
#endif /* __cplusplus */

/*
  ========================================================================
  To make a fully customizable malloc.h header file, cut everything
  above this line, put into file malloc.h, edit to suit, and #include it
  on the next line, as well as in programs that use this malloc.
  ========================================================================
*/

/* #include "malloc.h" */
```

Chapter 5

Internal #includes

```
#ifdef WIN32
#pragma warning( disable : 4146 ) /* no "unsigned" warnings */
#endif /* WIN32 */

#include <stdio.h>       /* for printing in malloc_stats */

#ifndef LACKS_ERRNO_H
#include <errno.h>       /* for MALLOC_FAILURE_ACTION */
#endif /* LACKS_ERRNO_H */
#if FOOTERS || DEBUG
#include <time.h>        /* for magic initialization */
#endif /* FOOTERS */
#ifndef LACKS_STDLIB_H
#include <stdlib.h>      /* for abort() */
#endif /* LACKS_STDLIB_H */
#ifdef DEBUG
#if ABORT_ON_ASSERT_FAILURE
#undef assert
#define assert(x) if(!(x)) ABORT
#else /* ABORT_ON_ASSERT_FAILURE */
#include <assert.h>
#endif /* ABORT_ON_ASSERT_FAILURE */
#else  /* DEBUG */
#ifndef assert
#define assert(x)
#endif
#define DEBUG 0
#endif /* DEBUG */
#ifndef LACKS_STRING_H
#include <string.h>      /* for memset etc */
#endif  /* LACKS_STRING_H */
#if USE_BUILTIN_FFS
#ifndef LACKS_STRINGS_H
#include <strings.h>     /* for ffs */
#endif /* LACKS_STRINGS_H */
#endif /* USE_BUILTIN_FFS */
#if HAVE_MMAP
#ifndef LACKS_SYS_MMAN_H
/* On some versions of linux, mremap decl in mman.h needs __USE_GNU set */
```

```
#if (defined(linux) && !defined(__USE_GNU))
#define __USE_GNU 1
#include <sys/mman.h>    /* for mmap */
#undef __USE_GNU
#else
#include <sys/mman.h>    /* for mmap */
#endif /* linux */
#endif /* LACKS_SYS_MMAN_H */
#ifndef LACKS_FCNTL_H
#include <fcntl.h>
#endif /* LACKS_FCNTL_H */
#endif /* HAVE_MMAP */
#ifndef LACKS_UNISTD_H
#include <unistd.h>     /* for sbrk, sysconf */
#else /* LACKS_UNISTD_H */
#if !defined(__FreeBSD__) && !defined(__OpenBSD__) && !defined(__NetBSD__)
extern void*     sbrk(ptrdiff_t);
#endif /* FreeBSD etc */
#endif /* LACKS_UNISTD_H */

/* Declarations for locking */
#if USE_LOCKS
#ifndef WIN32
#include <pthread.h>
#if defined (__SVR4) && defined (__sun)  /* solaris */
#include <thread.h>
#endif /* solaris */
#else
#ifndef _M_AMD64
/* These are already defined on AMD64 builds */
#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */
LONG __cdecl _InterlockedCompareExchange(LONG volatile *Dest, \
                                                LONG Exchange, LONG Comp);
LONG __cdecl _InterlockedExchange(LONG volatile *Target, LONG Value);
#ifdef __cplusplus
}
#endif /* __cplusplus */
#endif /* _M_AMD64 */
#pragma intrinsic (_InterlockedCompareExchange)
#pragma intrinsic (_InterlockedExchange)
#define interlockedcompareexchange _InterlockedCompareExchange
#define interlockedexchange _InterlockedExchange
#endif /* Win32 */
#endif /* USE_LOCKS */

/* Declarations for bit scanning on win32 */
#if defined(_MSC_VER) && _MSC_VER>=1300
#ifndef BitScanForward  /* Try to avoid pulling in WinNT.h */
#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */
unsigned char _BitScanForward(unsigned long *index, unsigned long mask);
unsigned char _BitScanReverse(unsigned long *index, unsigned long mask);
#ifdef __cplusplus
}
#endif /* __cplusplus */

#define BitScanForward _BitScanForward
#define BitScanReverse _BitScanReverse
#pragma intrinsic(_BitScanForward)
#pragma intrinsic(_BitScanReverse)
#endif /* BitScanForward */
#endif /* defined(_MSC_VER) && _MSC_VER>=1300 */

#ifndef WIN32
#ifndef malloc_getpagesize
#  ifdef _SC_PAGESIZE         /* some SVR4 systems omit an underscore */
#    ifndef _SC_PAGE_SIZE
#      define _SC_PAGE_SIZE _SC_PAGESIZE
#    endif
#  endif
#  ifdef _SC_PAGE_SIZE
#    define malloc_getpagesize sysconf(_SC_PAGE_SIZE)
#  else
#    if defined(BSD) || defined(DGUX) || defined(HAVE_GETPAGESIZE)
       extern size_t getpagesize();
#      define malloc_getpagesize getpagesize()
#    else
#      ifdef WIN32 /* use supplied emulation of getpagesize */
#        define malloc_getpagesize getpagesize()
#      else
#        ifndef LACKS_SYS_PARAM_H
#          include <sys/param.h>
#        endif
#        ifdef EXEC_PAGESIZE
#          define malloc_getpagesize EXEC_PAGESIZE
#        else
#          ifdef NBPG
#            ifndef CLSIZE
#              define malloc_getpagesize NBPG
#            else
#              define malloc_getpagesize (NBPG * CLSIZE)
#            endif
#          else
#            ifdef NBPC
#              define malloc_getpagesize NBPC
#            else
#              ifdef PAGESIZE
#                define malloc_getpagesize PAGESIZE
#              else /* just guess */
#                define malloc_getpagesize ((size_t)4096U)
#              endif
```

```
#            endif
#          endif
#        endif
#      endif
#    endif
#  endif
#endif
#endif
```

Chapter 6

size_t and alignment properties

```
/* The byte and bit size of a size_t */
#define SIZE_T_SIZE         (sizeof(size_t))
#define SIZE_T_BITSIZE      (sizeof(size_t) << 3)

/* Some constants coerced to size_t */
/* Annoying but necessary to avoid errors on some platforms */
#define SIZE_T_ZERO         ((size_t)0)
#define SIZE_T_ONE          ((size_t)1)
#define SIZE_T_TWO          ((size_t)2)
#define SIZE_T_FOUR         ((size_t)4)
#define TWO_SIZE_T_SIZES    (SIZE_T_SIZE<<1)
#define FOUR_SIZE_T_SIZES   (SIZE_T_SIZE<<2)
#define SIX_SIZE_T_SIZES    (FOUR_SIZE_T_SIZES+TWO_SIZE_T_SIZES)
#define HALF_MAX_SIZE_T     (MAX_SIZE_T / 2U)

/* The bit mask value corresponding to MALLOC_ALIGNMENT */
#define CHUNK_ALIGN_MASK    (MALLOC_ALIGNMENT - SIZE_T_ONE)

/* True if address a has acceptable alignment */
#define is_aligned(A)       (((size_t)((A)) & (CHUNK_ALIGN_MASK)) == 0)

/* the number of bytes to offset an address to align it */
#define align_offset(A)\
 ((((size_t)(A) & CHUNK_ALIGN_MASK) == 0)? 0 :\
  ((MALLOC_ALIGNMENT - ((size_t)(A) & CHUNK_ALIGN_MASK)) & CHUNK_ALIGN_MASK))
```

Chapter 7

MMAP preliminaries

```
/*
   If HAVE_MORECORE or HAVE_MMAP are false, we just define calls and
   checks to fail so compiler optimizer can delete code rather than
   using so many "#if"s.
*/


/* MORECORE and MMAP must return MFAIL on failure */
#define MFAIL                ((void*)(MAX_SIZE_T))
#define CMFAIL               ((char*)(MFAIL)) /* defined for convenience */

#if HAVE_MMAP

#ifndef WIN32
#define MUNMAP_DEFAULT(a, s)  munmap((a), (s))
#define MMAP_PROT            (PROT_READ|PROT_WRITE)
#if !defined(MAP_ANONYMOUS) && defined(MAP_ANON)
#define MAP_ANONYMOUS        MAP_ANON
#endif /* MAP_ANON */
#ifdef MAP_ANONYMOUS
#define MMAP_FLAGS           (MAP_PRIVATE|MAP_ANONYMOUS)
#define MMAP_DEFAULT(s)       mmap(0, (s), MMAP_PROT, MMAP_FLAGS, -1, 0)
#else /* MAP_ANONYMOUS */
/*
   Nearly all versions of mmap support MAP_ANONYMOUS, so the following
   is unlikely to be needed, but is supplied just in case.
*/
#define MMAP_FLAGS           (MAP_PRIVATE)
static int dev_zero_fd = -1; /* Cached file descriptor for /dev/zero. */
#define MMAP_DEFAULT(s) ((dev_zero_fd < 0) ? \
           (dev_zero_fd = open("/dev/zero", O_RDWR), \
            mmap(0, (s), MMAP_PROT, MMAP_FLAGS, dev_zero_fd, 0)) : \
            mmap(0, (s), MMAP_PROT, MMAP_FLAGS, dev_zero_fd, 0))
#endif /* MAP_ANONYMOUS */

#define DIRECT_MMAP_DEFAULT(s) MMAP_DEFAULT(s)

#else /* WIN32 */
```



```
/* Win32 MMAP via VirtualAlloc */
static FORCEINLINE void* win32mmap(size_t size) {
  void* ptr = VirtualAlloc(0, size, MEM_RESERVE|MEM_COMMIT, PAGE_READWRITE);
  return (ptr != 0)? ptr: MFAIL;
}

/* For direct MMAP, use MEM_TOP_DOWN to minimize interference */
static FORCEINLINE void* win32direct_mmap(size_t size) {
  void* ptr = VirtualAlloc(0, size, MEM_RESERVE|MEM_COMMIT|MEM_TOP_DOWN,
                           PAGE_READWRITE);
  return (ptr != 0)? ptr: MFAIL;
}

/* This function supports releasing coalesed segments */
static FORCEINLINE int win32munmap(void* ptr, size_t size) {
  MEMORY_BASIC_INFORMATION minfo;
  char* cptr = (char*)ptr;
  while (size) {
    if (VirtualQuery(cptr, &minfo, sizeof(minfo)) == 0)
      return -1;
    if (minfo.BaseAddress != cptr || minfo.AllocationBase != cptr ||
        minfo.State != MEM_COMMIT || minfo.RegionSize > size)
      return -1;
    if (VirtualFree(cptr, 0, MEM_RELEASE) == 0)
      return -1;
    cptr += minfo.RegionSize;
    size -= minfo.RegionSize;
  }
  return 0;
}

#define MMAP_DEFAULT(s)             win32mmap(s)
#define MUNMAP_DEFAULT(a, s)        win32munmap((a), (s))
#define DIRECT_MMAP_DEFAULT(s)      win32direct_mmap(s)
#endif /* WIN32 */
#endif /* HAVE_MMAP */

#if HAVE_MREMAP
#ifndef WIN32
#define MREMAP_DEFAULT(addr, osz, nsz, mv) mremap((addr), (osz), (nsz), (mv))
#endif /* WIN32 */
#endif /* HAVE_MREMAP */


/**
 * Define CALL_MORECORE
 */
#if HAVE_MORECORE
    #ifdef MORECORE
        #define CALL_MORECORE(S)    MORECORE(S)
    #else  /* MORECORE */
        #define CALL_MORECORE(S)    MORECORE_DEFAULT(S)
```

```c
    #endif /* MORECORE */
#else  /* HAVE_MORECORE */
    #define CALL_MORECORE(S)        MFAIL
#endif /* HAVE_MORECORE */

/**
 * Define CALL_MMAP/CALL_MUNMAP/CALL_DIRECT_MMAP
 */
#if HAVE_MMAP
    #define USE_MMAP_BIT            (SIZE_T_ONE)

    #ifdef MMAP
        #define CALL_MMAP(s)        MMAP(s)
    #else /* MMAP */
        #define CALL_MMAP(s)        MMAP_DEFAULT(s)
    #endif /* MMAP */
    #ifdef MUNMAP
        #define CALL_MUNMAP(a, s)   MUNMAP((a), (s))
    #else /* MUNMAP */
        #define CALL_MUNMAP(a, s)   MUNMAP_DEFAULT((a), (s))
    #endif /* MUNMAP */
    #ifdef DIRECT_MMAP
        #define CALL_DIRECT_MMAP(s) DIRECT_MMAP(s)
    #else /* DIRECT_MMAP */
        #define CALL_DIRECT_MMAP(s) DIRECT_MMAP_DEFAULT(s)
    #endif /* DIRECT_MMAP */
#else  /* HAVE_MMAP */
    #define USE_MMAP_BIT            (SIZE_T_ZERO)

    #define MMAP(s)                 MFAIL
    #define MUNMAP(a, s)            (-1)
    #define DIRECT_MMAP(s)          MFAIL
    #define CALL_DIRECT_MMAP(s)     DIRECT_MMAP(s)
    #define CALL_MMAP(s)            MMAP(s)
    #define CALL_MUNMAP(a, s)       MUNMAP((a), (s))
#endif /* HAVE_MMAP */

/**
 * Define CALL_MREMAP
 */
#if HAVE_MMAP && HAVE_MREMAP
    #ifdef MREMAP
        #define CALL_MREMAP(addr, osz, nsz, mv) MREMAP((addr), (osz), (nsz), (mv))
    #else /* MREMAP */
        #define CALL_MREMAP(addr, osz, nsz, mv) \
                          MREMAP_DEFAULT((addr), (osz), (nsz), (mv))
    #endif /* MREMAP */
#else  /* HAVE_MMAP && HAVE_MREMAP */
    #define CALL_MREMAP(addr, osz, nsz, mv)     MFAIL
#endif /* HAVE_MMAP && HAVE_MREMAP */

/* mstate bit set if continguous morecore disabled or failed */
```



```c
#define USE_NONCONTIGUOUS_BIT (4U)

/* segment bit set in create_mspace_with_base */
#define EXTERN_BIT            (8U)
```

Chapter 8

Locks

8.1 Lock preliminaries

```
/*
  When locks are defined, there is one global lock, plus
  one per-mspace lock.

  The global lock_ensures that mparams.magic and other unique
  mparams values are initialized only once. It also protects
  sequences of calls to MORECORE.  In many cases sys_alloc requires
  two calls, that should not be interleaved with calls by other
  threads.  This does not protect against direct calls to MORECORE
  by other threads not using this lock, so there is still code to
  cope the best we can on interference.

  Per-mspace locks surround calls to malloc, free, etc.  To enable use
  in layered extensions, per-mspace locks are reentrant.

  Because lock-protected regions generally have bounded times, it is
  OK to use the supplied simple spinlocks in the custom versions for
  x86. Spinlocks are likely to improve performance for lightly
  contended applications, but worsen performance under heavy
  contention.

  If USE_LOCKS is > 1, the definitions of lock routines here are
  bypassed, in which case you will need to define the type MLOCK_T,
  and at least INITIAL_LOCK, ACQUIRE_LOCK, RELEASE_LOCK and possibly
  TRY_LOCK (which is not used in this malloc, but commonly needed in
  extensions.)  You must also declare a
    static MLOCK_T malloc_global_mutex = { initialization values };.

*/

#if USE_LOCKS == 1

#if USE_SPIN_LOCKS && SPIN_LOCKS_AVAILABLE
#ifndef WIN32

/* Custom pthread-style spin locks on x86 and x64 for gcc */
```



```
struct pthread_mlock_t {
  volatile unsigned int l;
  unsigned int c;
  pthread_t threadid;
};
#define MLOCK_T               struct pthread_mlock_t
#define CURRENT_THREAD        pthread_self()
#define INITIAL_LOCK(sl)      ((sl)->threadid = 0, (sl)->l = (sl)->c = 0, 0)
#define ACQUIRE_LOCK(sl)      pthread_acquire_lock(sl)
#define RELEASE_LOCK(sl)      pthread_release_lock(sl)
#define TRY_LOCK(sl)          pthread_try_lock(sl)
#define SPINS_PER_YIELD       63

static MLOCK_T malloc_global_mutex = { 0, 0, 0};

static FORCEINLINE int pthread_acquire_lock (MLOCK_T *sl) {
  int spins = 0;
  volatile unsigned int* lp = &sl->l;
  for (;;) {
    if (*lp != 0) {
      if (sl->threadid == CURRENT_THREAD) {
        ++sl->c;
        return 0;
      }
    }
    else {
      /* place args to cmpxchgl in locals to evade oddities in some gccs */
      int cmp = 0;
      int val = 1;
      int ret;
      __asm__ __volatile__  ("lock; cmpxchgl %1, %2"
                             : "=a" (ret)
                             : "r" (val), "m" (*(lp)), "0"(cmp)
                             : "memory", "cc");
      if (!ret) {
        assert(!sl->threadid);
        sl->threadid = CURRENT_THREAD;
        sl->c = 1;
        return 0;
      }
    }
    if ((++spins & SPINS_PER_YIELD) == 0) {
#if defined (__SVR4) && defined (__sun) /* solaris */
      thr_yield();
#else
#if defined(__linux__) || defined(__FreeBSD__) || defined(__APPLE__)
      sched_yield();
#else  /* no-op yield on unknown systems */
      ;
#endif /* __linux__ || __FreeBSD__ || __APPLE__ */
#endif /* solaris */
    }
```

```
  }
}

static FORCEINLINE void pthread_release_lock (MLOCK_T *sl) {
  volatile unsigned int* lp = &sl->l;
  assert(*lp != 0);
  assert(sl->threadid == CURRENT_THREAD);
  if (--sl->c == 0) {
    sl->threadid = 0;
    int prev = 0;
    int ret;
    __asm__ __volatile__ ("lock; xchgl %0, %1"
                          : "=r" (ret)
                          : "m" (*(lp)), "0"(prev)
                          : "memory");
  }
}

static FORCEINLINE int pthread_try_lock (MLOCK_T *sl) {
  volatile unsigned int* lp = &sl->l;
  if (*lp != 0) {
    if (sl->threadid == CURRENT_THREAD) {
      ++sl->c;
      return 1;
    }
  }
  else {
    int cmp = 0;
    int val = 1;
    int ret;
    __asm__ __volatile__  ("lock; cmpxchgl %1, %2"
                           : "=a" (ret)
                           : "r" (val), "m" (*(lp)), "0"(cmp)
                           : "memory", "cc");
    if (!ret) {
      assert(!sl->threadid);
      sl->threadid = CURRENT_THREAD;
      sl->c = 1;
      return 1;
    }
  }
  return 0;
}


#else /* WIN32 */
/* Custom win32-style spin locks on x86 and x64 for MSC */
struct win32_mlock_t {
  volatile long l;
  unsigned int c;
  long threadid;
};
```



```
#define MLOCK_T               struct win32_mlock_t
#define CURRENT_THREAD        GetCurrentThreadId()
#define INITIAL_LOCK(sl)      ((sl)->threadid = 0, (sl)->l = (sl)->c = 0, 0)
#define ACQUIRE_LOCK(sl)      win32_acquire_lock(sl)
#define RELEASE_LOCK(sl)      win32_release_lock(sl)
#define TRY_LOCK(sl)          win32_try_lock(sl)
#define SPINS_PER_YIELD       63

static MLOCK_T malloc_global_mutex = { 0, 0, 0};

static FORCEINLINE int win32_acquire_lock (MLOCK_T *sl) {
  int spins = 0;
  for (;;) {
    if (sl->l != 0) {
      if (sl->threadid == CURRENT_THREAD) {
        ++sl->c;
        return 0;
      }
    }
    else {
      if (!interlockedexchange(&sl->l, 1)) {
        assert(!sl->threadid);
        sl->threadid = CURRENT_THREAD;
        sl->c = 1;
        return 0;
      }
    }
    if ((++spins & SPINS_PER_YIELD) == 0)
      SleepEx(0, FALSE);
  }
}

static FORCEINLINE void win32_release_lock (MLOCK_T *sl) {
  assert(sl->threadid == CURRENT_THREAD);
  assert(sl->l != 0);
  if (--sl->c == 0) {
    sl->threadid = 0;
    interlockedexchange (&sl->l, 0);
  }
}

static FORCEINLINE int win32_try_lock (MLOCK_T *sl) {
  if (sl->l != 0) {
    if (sl->threadid == CURRENT_THREAD) {
      ++sl->c;
      return 1;
    }
  }
  else {
    if (!interlockedexchange(&sl->l, 1)){
      assert(!sl->threadid);
```

```
      sl->threadid = CURRENT_THREAD;
      sl->c = 1;
      return 1;
    }
  }
  return 0;
}

#endif /* WIN32 */
#else /* USE_SPIN_LOCKS */

#ifndef WIN32
/* pthreads-based locks */

#define MLOCK_T               pthread_mutex_t
#define CURRENT_THREAD        pthread_self()
#define INITIAL_LOCK(sl)      pthread_init_lock(sl)
#define ACQUIRE_LOCK(sl)      pthread_mutex_lock(sl)
#define RELEASE_LOCK(sl)      pthread_mutex_unlock(sl)
#define TRY_LOCK(sl)          (!pthread_mutex_trylock(sl))

static MLOCK_T malloc_global_mutex = PTHREAD_MUTEX_INITIALIZER;

/* Cope with old-style linux recursive lock initialization by adding */
/* skipped internal declaration from pthread.h */
#ifdef linux
#ifndef PTHREAD_MUTEX_RECURSIVE
extern int pthread_mutexattr_setkind_np __P ((pthread_mutexattr_t *__attr,
                                              int __kind));
#define PTHREAD_MUTEX_RECURSIVE PTHREAD_MUTEX_RECURSIVE_NP
#define pthread_mutexattr_settype(x,y) pthread_mutexattr_setkind_np(x,y)
#endif
#endif

static int pthread_init_lock (MLOCK_T *sl) {
  pthread_mutexattr_t attr;
  if (pthread_mutexattr_init(&attr)) return 1;
  if (pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE)) return 1;
  if (pthread_mutex_init(sl, &attr)) return 1;
  if (pthread_mutexattr_destroy(&attr)) return 1;
  return 0;
}

#else /* WIN32 */
/* Win32 critical sections */
#define MLOCK_T               CRITICAL_SECTION
#define CURRENT_THREAD        GetCurrentThreadId()
#define INITIAL_LOCK(s)       (!InitializeCriticalSectionAndSpinCount((s), \
                                                  0x80000000|4000))
#define ACQUIRE_LOCK(s)       (EnterCriticalSection(sl), 0)
#define RELEASE_LOCK(s)       LeaveCriticalSection(sl)
#define TRY_LOCK(s)           TryEnterCriticalSection(sl)
```



```
#define NEED_GLOBAL_LOCK_INIT

static MLOCK_T malloc_global_mutex;
static volatile long malloc_global_mutex_status;

/* Use spin loop to initialize global lock */
static void init_malloc_global_mutex() {
  for (;;) {
    long stat = malloc_global_mutex_status;
    if (stat > 0)
      return;
    /* transition to < 0 while initializing, then to > 0) */
    if (stat == 0 &&
        interlockedcompareexchange(&malloc_global_mutex_status, -1, 0) == 0) {
      InitializeCriticalSection(&malloc_global_mutex);
      interlockedexchange(&malloc_global_mutex_status,1);
      return;
    }
    SleepEx(0, FALSE);
  }
}

#endif /* WIN32 */
#endif /* USE_SPIN_LOCKS */
#endif /* USE_LOCKS == 1 */
```

8.2 User-defined locks

```
#if USE_LOCKS > 1
/* Define your own lock implementation here */
/* #define INITIAL_LOCK(sl)  ... */
/* #define ACQUIRE_LOCK(sl)  ... */
/* #define RELEASE_LOCK(sl)  ... */
/* #define TRY_LOCK(sl) ... */
/* static MLOCK_T malloc_global_mutex = ... */
#endif /* USE_LOCKS > 1 */
```

8.3 Lock-based state

```
#if USE_LOCKS
#define USE_LOCK_BIT               (2U)
#else  /* USE_LOCKS */
#define USE_LOCK_BIT               (0U)
#define INITIAL_LOCK(l)
#endif /* USE_LOCKS */

#if USE_LOCKS
#ifndef ACQUIRE_MALLOC_GLOBAL_LOCK
```

```
#define ACQUIRE_MALLOC_GLOBAL_LOCK()  ACQUIRE_LOCK(&malloc_global_mutex);
#endif
#ifndef RELEASE_MALLOC_GLOBAL_LOCK
#define RELEASE_MALLOC_GLOBAL_LOCK()  RELEASE_LOCK(&malloc_global_mutex);
#endif
#else  /* USE_LOCKS */
#define ACQUIRE_MALLOC_GLOBAL_LOCK()
#define RELEASE_MALLOC_GLOBAL_LOCK()
#endif /* USE_LOCKS */
```

Chapter 9

Chunks

9.1 Chunk representations

```
/*
  (The following includes lightly edited explanations by Colin Plumb.)

  The malloc_chunk declaration below is misleading (but accurate and
  necessary).  It declares a "view" into memory allowing access to
  necessary fields at known offsets from a given base.

  Chunks of memory are maintained using a ‘boundary tag’ method as
  originally described by Knuth.  (See the paper by Paul Wilson
  ftp://ftp.cs.utexas.edu/pub/garbage/allocsrv.ps for a survey of such
  techniques.)  Sizes of free chunks are stored both in the front of
  each chunk and at the end.  This makes consolidating fragmented
  chunks into bigger chunks fast.  The head fields also hold bits
  representing whether chunks are free or in use.

  Here are some pictures to make it clearer.  They are "exploded" to
  show that the state of a chunk can be thought of as extending from
  the high 31 bits of the head field of its header through the
  prev_foot and PINUSE_BIT bit of the following chunk header.

  A chunk that’s in use looks like:

   chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
           | Size of previous chunk (if P = 0)                             |
           +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
         +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+ |P|
         | Size of this chunk                                         1| +-+
   mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
         |                                                               |
         +-                                                             -+
         |                                                               |
         +-                                                             -+
         |                                                               :
         +-      size - sizeof(size_t) available payload bytes          -+
         :                                                               |
 chunk-> +-                                                             -+
```

```
            |                                                               |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
          +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+ |1|
          | Size of next chunk (may or may not be in use)               | +-+
    mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

   And if it's free, it looks like this:

    chunk-> +-                                                              -+
              | User payload (must be in use, or we would have merged!)       |
              +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+ |P|
            | Size of this chunk                                         0| +-+
      mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            | Next pointer                                                  |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            | Prev pointer                                                  |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |                                                               :
            +-      size - sizeof(struct chunk) unused bytes               -+
            :                                                               |
    chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            | Size of this chunk                                            |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
          +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+ |0|
          | Size of next chunk (must be in use, or we would have merged)| +-+
    mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
          |                                                               :
          +- User payload                                                -+
          :                                                               |
          +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
                                                                        |0|
                                                                        +-+
  Note that since we always merge adjacent free chunks, the chunks
  adjacent to a free chunk must be in use.

  Given a pointer to a chunk (which can be derived trivially from the
  payload pointer) we can, in O(1) time, find out whether the adjacent
  chunks are free, and if so, unlink them from the lists that they
  are on and merge them with the current chunk.

  Chunks always begin on even word boundaries, so the mem portion
  (which is returned to the user) is also on an even word boundary, and
  thus at least double-word aligned.

  The P (PINUSE_BIT) bit, stored in the unused low-order bit of the
  chunk size (which is always a multiple of two words), is an in-use
  bit for the *previous* chunk.  If that bit is *clear*, then the
  word before the current chunk size contains the previous chunk
  size, and can be used to find the front of the previous chunk.
  The very first chunk allocated always has this bit set, preventing
  access to non-existent (or non-owned) memory. If pinuse is set for
```



```
  any given chunk, then you CANNOT determine the size of the
  previous chunk, and might even get a memory addressing fault when
  trying to do so.

  The C (CINUSE_BIT) bit, stored in the unused second-lowest bit of
  the chunk size redundantly records whether the current chunk is
  inuse (unless the chunk is mmapped). This redundancy enables usage
  checks within free and realloc, and reduces indirection when freeing
  and consolidating chunks.

  Each freshly allocated chunk must have both cinuse and pinuse set.
  That is, each allocated chunk borders either a previously allocated
  and still in-use chunk, or the base of its memory arena. This is
  ensured by making all allocations from the the `lowest' part of any
  found chunk.  Further, no free chunk physically borders another one,
  so each free chunk is known to be preceded and followed by either
  inuse chunks or the ends of memory.

  Note that the `foot' of the current chunk is actually represented
  as the prev_foot of the NEXT chunk. This makes it easier to
  deal with alignments etc but can be very confusing when trying
  to extend or adapt this code.

  The exceptions to all this are

     1. The special chunk `top' is the top-most available chunk (i.e.,
        the one bordering the end of available memory). It is treated
        specially.  Top is never included in any bin, is used only if
        no other chunk is available, and is released back to the
        system if it is very large (see M_TRIM_THRESHOLD).  In effect,
        the top chunk is treated as larger (and thus less well
        fitting) than any other available chunk.  The top chunk
        doesn't update its trailing size field since there is no next
        contiguous chunk that would have to index off it. However,
        space is still allocated for it (TOP_FOOT_SIZE) to enable
        separation or merging when space is extended.

     3. Chunks allocated via mmap, have both cinuse and pinuse bits
        cleared in their head fields.  Because they are allocated
        one-by-one, each must carry its own prev_foot field, which is
        also used to hold the offset this chunk has within its mmapped
        region, which is needed to preserve alignment. Each mmapped
        chunk is trailed by the first two fields of a fake next-chunk
        for sake of usage checks.

*/

struct malloc_chunk {
  size_t               prev_foot;  /* Size of previous chunk (if free).  */
  size_t               head;       /* Size and inuse bits. */
  struct malloc_chunk* fd;         /* double links -- used only if free. */
  struct malloc_chunk* bk;
```

```
};

typedef struct malloc_chunk  mchunk;
typedef struct malloc_chunk* mchunkptr;
typedef struct malloc_chunk* sbinptr;  /* The type of bins of chunks */
typedef unsigned int bindex_t;         /* Described below */
typedef unsigned int binmap_t;         /* Described below */
typedef unsigned int flag_t;           /* The type of various bit flag sets */
```

9.2 Chunks sizes and alignments

```
#define MCHUNK_SIZE         (sizeof(mchunk))

#if FOOTERS
#define CHUNK_OVERHEAD      (TWO_SIZE_T_SIZES)
#else /* FOOTERS */
#define CHUNK_OVERHEAD      (SIZE_T_SIZE)
#endif /* FOOTERS */

/* MMapped chunks need a second word of overhead ... */
#define MMAP_CHUNK_OVERHEAD (TWO_SIZE_T_SIZES)
/* ... and additional padding for fake next-chunk at foot */
#define MMAP_FOOT_PAD       (FOUR_SIZE_T_SIZES)

/* The smallest size we can malloc is an aligned minimal chunk */
#define MIN_CHUNK_SIZE\
  ((MCHUNK_SIZE + CHUNK_ALIGN_MASK) & ~CHUNK_ALIGN_MASK)

/* conversion from malloc headers to user pointers, and back */
#define chunk2mem(p)        ((void*)((char*)(p)       + TWO_SIZE_T_SIZES))
#define mem2chunk(mem)      ((mchunkptr)((char*)(mem) - TWO_SIZE_T_SIZES))
/* chunk associated with aligned address A */
#define align_as_chunk(A)   (mchunkptr)((A) + align_offset(chunk2mem(A)))

/* Bounds on request (not chunk) sizes. */
#define MAX_REQUEST         ((-MIN_CHUNK_SIZE) << 2)
#define MIN_REQUEST         (MIN_CHUNK_SIZE - CHUNK_OVERHEAD - SIZE_T_ONE)

/* pad request bytes into a usable size */
#define pad_request(req) \
   (((req) + CHUNK_OVERHEAD + CHUNK_ALIGN_MASK) & ~CHUNK_ALIGN_MASK)

/* pad request, checking for minimum (but not maximum) */
#define request2size(req) \
  (((req) < MIN_REQUEST)? MIN_CHUNK_SIZE : pad_request(req))
```


9.3 Operations on head and foot fields

```
/*
  The head field of a chunk is or'ed with PINUSE_BIT when previous
  adjacent chunk in use, and or'ed with CINUSE_BIT if this chunk is in
  use, unless mmapped, in which case both bits are cleared.

  FLAG4_BIT is not used by this malloc, but might be useful in extensions.
*/

#define PINUSE_BIT          (SIZE_T_ONE)
#define CINUSE_BIT          (SIZE_T_TWO)
#define FLAG4_BIT           (SIZE_T_FOUR)
#define INUSE_BITS          (PINUSE_BIT|CINUSE_BIT)
#define FLAG_BITS           (PINUSE_BIT|CINUSE_BIT|FLAG4_BIT)

/* Head value for fenceposts */
#define FENCEPOST_HEAD      (INUSE_BITS|SIZE_T_SIZE)

/* extraction of fields from head words */
#define cinuse(p)           ((p)->head & CINUSE_BIT)
#define pinuse(p)           ((p)->head & PINUSE_BIT)
#define is_inuse(p)         (((p)->head & INUSE_BITS) != PINUSE_BIT)
#define is_mmapped(p)       (((p)->head & INUSE_BITS) == 0)

#define chunksize(p)        ((p)->head & ~(FLAG_BITS))

#define clear_pinuse(p)     ((p)->head &= ~PINUSE_BIT)

/* Treat space at ptr +/- offset as a chunk */
#define chunk_plus_offset(p, s)  ((mchunkptr)(((char*)(p)) + (s)))
#define chunk_minus_offset(p, s) ((mchunkptr)(((char*)(p)) - (s)))

/* Ptr to next or previous physical malloc_chunk. */
#define next_chunk(p) ((mchunkptr)( ((char*)(p)) + ((p)->head & ~FLAG_BITS)))
#define prev_chunk(p) ((mchunkptr)( ((char*)(p)) - ((p)->prev_foot) ))

/* extract next chunk's pinuse bit */
#define next_pinuse(p)  ((next_chunk(p)->head) & PINUSE_BIT)

/* Get/set size at footer */
#define get_foot(p, s)  (((mchunkptr)((char*)(p) + (s)))->prev_foot)
#define set_foot(p, s)  (((mchunkptr)((char*)(p) + (s)))->prev_foot = (s))

/* Set size, pinuse bit, and foot */
#define set_size_and_pinuse_of_free_chunk(p, s)\
  ((p)->head = (s|PINUSE_BIT), set_foot(p, s))

/* Set size, pinuse bit, foot, and clear next pinuse */
#define set_free_with_pinuse(p, s, n)\
```

```
  (clear_pinuse(n), set_size_and_pinuse_of_free_chunk(p, s))

/* Get the internal overhead associated with chunk p */
#define overhead_for(p)\
 (is_mmapped(p)? MMAP_CHUNK_OVERHEAD : CHUNK_OVERHEAD)

/* Return true if malloced space is not necessarily cleared */
#if MMAP_CLEARS
#define calloc_must_clear(p) (!is_mmapped(p))
#else /* MMAP_CLEARS */
#define calloc_must_clear(p) (1)
#endif /* MMAP_CLEARS */
```

9.4 Overlaid data structures

```
/*
  When chunks are not in use, they are treated as nodes of either
  lists or trees.

  "Small"  chunks are stored in circular doubly-linked lists, and look
  like this:

    chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of previous chunk                            |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    `head:' |             Size of chunk, in bytes                         |P|
      mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Forward pointer to next chunk in list             |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Back pointer to previous chunk in list            |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Unused space (may be 0 bytes long)                .
            .                                                               .
            .                                                               |
nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    `foot:' |             Size of chunk, in bytes                           |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

  Larger chunks are kept in a form of bitwise digital trees (aka
  tries) keyed on chunksizes.  Because malloc_tree_chunks are only for
  free chunks greater than 256 bytes, their size doesn't impose any
  constraints on user chunk sizes.  Each node looks like:

    chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of previous chunk                            |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    `head:' |             Size of chunk, in bytes                         |P|
      mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Forward pointer to next chunk of same size        |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Back pointer to previous chunk of same size       |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```



```
            |             Pointer to left child (child[0])                  |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Pointer to right child (child[1])                 |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Pointer to parent                                 |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             bin index of this chunk                           |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Unused space                                      .
            .                                                               |
nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    `foot:' |             Size of chunk, in bytes                           |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

  Each tree holding treenodes is a tree of unique chunk sizes.  Chunks
  of the same size are arranged in a circularly-linked list, with only
  the oldest chunk (the next to be used, in our FIFO ordering)
  actually in the tree.  (Tree members are distinguished by a non-null
  parent pointer.)  If a chunk with the same size an an existing node
  is inserted, it is linked off the existing node using pointers that
  work in the same way as fd/bk pointers of small chunks.

  Each tree contains a power of 2 sized range of chunk sizes (the
  smallest is 0x100 <= x < 0x180), which is is divided in half at each
  tree level, with the chunks in the smaller half of the range (0x100
  <= x < 0x140 for the top nose) in the left subtree and the larger
  half (0x140 <= x < 0x180) in the right subtree.  This is, of course,
  done by inspecting individual bits.

  Using these rules, each node's left subtree contains all smaller
  sizes than its right subtree.  However, the node at the root of each
  subtree has no particular ordering relationship to either.  (The
  dividing line between the subtree sizes is based on trie relation.)
  If we remove the last chunk of a given size from the interior of the
  tree, we need to replace it with a leaf node.  The tree ordering
  rules permit a node to be replaced by any leaf below it.

  The smallest chunk in a tree (a common operation in a best-fit
  allocator) can be found by walking a path to the leftmost leaf in
  the tree.  Unlike a usual binary tree, where we follow left child
  pointers until we reach a null, here we follow the right child
  pointer any time the left one is null, until we reach a leaf with
  both child pointers null. The smallest chunk in the tree will be
  somewhere along that path.

  The worst case number of steps to add, find, or remove a node is
  bounded by the number of bits differentiating chunks within
  bins. Under current bin calculations, this ranges from 6 up to 21
  (for 32 bit sizes) or up to 53 (for 64 bit sizes). The typical case
  is of course much better.
*/
```

```
struct malloc_tree_chunk {
  /* The first four fields must be compatible with malloc_chunk */
  size_t                    prev_foot;
  size_t                    head;
  struct malloc_tree_chunk* fd;
  struct malloc_tree_chunk* bk;

  struct malloc_tree_chunk* child[2];
  struct malloc_tree_chunk* parent;
  bindex_t                  index;
};

typedef struct malloc_tree_chunk  tchunk;
typedef struct malloc_tree_chunk* tchunkptr;
typedef struct malloc_tree_chunk* tbinptr; /* The type of bins of trees */

/* A little helper macro for trees */
#define leftmost_child(t) ((t)->child[0] != 0? (t)->child[0] : (t)->child[1])
```

Chapter 10

Segments

```
/*
  Each malloc space may include non-contiguous segments, held in a
  list headed by an embedded malloc_segment record representing the
  top-most space. Segments also include flags holding properties of
  the space. Large chunks that are directly allocated by mmap are not
  included in this list. They are instead independently created and
  destroyed without otherwise keeping track of them.

  Segment management mainly comes into play for spaces allocated by
  MMAP.  Any call to MMAP might or might not return memory that is
  adjacent to an existing segment.  MORECORE normally contiguously
  extends the current space, so this space is almost always adjacent,
  which is simpler and faster to deal with. (This is why MORECORE is
  used preferentially to MMAP when both are available -- see
  sys_alloc.)  When allocating using MMAP, we don't use any of the
  hinting mechanisms (inconsistently) supported in various
  implementations of unix mmap, or distinguish reserving from
  committing memory. Instead, we just ask for space, and exploit
  contiguity when we get it.  It is probably possible to do
  better than this on some systems, but no general scheme seems
  to be significantly better.

  Management entails a simpler variant of the consolidation scheme
  used for chunks to reduce fragmentation -- new adjacent memory is
  normally prepended or appended to an existing segment. However,
  there are limitations compared to chunk consolidation that mostly
  reflect the fact that segment processing is relatively infrequent
  (occurring only when getting memory from system) and that we
  don't expect to have huge numbers of segments:

  * Segments are not indexed, so traversal requires linear scans.  (It
    would be possible to index these, but is not worth the extra
    overhead and complexity for most programs on most platforms.)
  * New segments are only appended to old ones when holding top-most
    memory; if they cannot be prepended to others, they are held in
    different segments.

  Except for the top-most segment of an mstate, each segment record
  is kept at the tail of its segment. Segments are added by pushing
```

```
  segment records onto the list headed by &mstate.seg for the
  containing mstate.

  Segment flags control allocation/merge/deallocation policies:
  * If EXTERN_BIT set, then we did not allocate this segment,
    and so should not try to deallocate or merge with others.
    (This currently holds only for the initial segment passed
    into create_mspace_with_base.)
  * If USE_MMAP_BIT set, the segment may be merged with
    other surrounding mmapped segments and trimmed/de-allocated
    using munmap.
  * If neither bit is set, then the segment was obtained using
    MORECORE so can be merged with surrounding MORECORE'd segments
    and deallocated/trimmed using MORECORE with negative arguments.
*/

struct malloc_segment {
  char*        base;             /* base address */
  size_t       size;             /* allocated size */
  struct malloc_segment* next;   /* ptr to next segment */
  flag_t       sflags;           /* mmap and extern flag */
};

#define is_mmapped_segment(S)  ((S)->sflags & USE_MMAP_BIT)
#define is_extern_segment(S)   ((S)->sflags & EXTERN_BIT)

typedef struct malloc_segment  msegment;
typedef struct malloc_segment* msegmentptr;
```

Chapter 11

State

11.1 malloc_state

```
/*
  A malloc_state holds all of the bookkeeping for a space.
  The main fields are:

 Top
   The topmost chunk of the currently active segment. Its size is
   cached in topsize.  The actual size of topmost space is
   topsize+TOP_FOOT_SIZE, which includes space reserved for adding
   fenceposts and segment records if necessary when getting more
   space from the system.  The size at which to autotrim top is
   cached from mparams in trim_check, except that it is disabled if
   an autotrim fails.

 Designated victim (dv)
   This is the preferred chunk for servicing small requests that
   don't have exact fits.  It is normally the chunk split off most
   recently to service another small request.  Its size is cached in
   dvsize. The link fields of this chunk are not maintained since it
   is not kept in a bin.

 SmallBins
   An array of bin headers for free chunks.  These bins hold chunks
   with sizes less than MIN_LARGE_SIZE bytes. Each bin contains
   chunks of all the same size, spaced 8 bytes apart.  To simplify
   use in double-linked lists, each bin header acts as a malloc_chunk
   pointing to the real first node, if it exists (else pointing to
   itself).  This avoids special-casing for headers.  But to avoid
   waste, we allocate only the fd/bk pointers of bins, and then use
   repositioning tricks to treat these as the fields of a chunk.

 TreeBins
   Treebins are pointers to the roots of trees holding a range of
   sizes. There are 2 equally spaced treebins for each power of two
   from TREE_SHIFT to TREE_SHIFT+16. The last bin holds anything
   larger.
```

```
  Bin maps
    There is one bit map for small bins ("smallmap") and one for
    treebins ("treemap).  Each bin sets its bit when non-empty, and
    clears the bit when empty.  Bit operations are then used to avoid
    bin-by-bin searching -- nearly all "search" is done without ever
    looking at bins that won't be selected.  The bit maps
    conservatively use 32 bits per map word, even if on 64bit system.
    For a good description of some of the bit-based techniques used
    here, see Henry S. Warren Jr's book "Hacker's Delight" (and
    supplement at http://hackersdelight.org/). Many of these are
    intended to reduce the branchiness of paths through malloc etc, as
    well as to reduce the number of memory locations read or written.

  Segments
    A list of segments headed by an embedded malloc_segment record
    representing the initial space.

  Address check support
    The least_addr field is the least address ever obtained from
    MORECORE or MMAP. Attempted frees and reallocs of any address less
    than this are trapped (unless INSECURE is defined).

  Magic tag
    A cross-check field that should always hold same value as mparams.magic.

  Flags
    Bits recording whether to use MMAP, locks, or contiguous MORECORE

  Statistics
    Each space keeps track of current and maximum system memory
    obtained via MORECORE or MMAP.

  Trim support
    Fields holding the amount of unused topmost memory that should trigger
    timming, and a counter to force periodic scanning to release unused
    non-topmost segments.

  Locking
    If USE_LOCKS is defined, the "mutex" lock is acquired and released
    around every public call using this mspace.

  Extension support
    A void* pointer and a size_t field that can be used to help implement
    extensions to this malloc.
*/

/* Bin types, widths and sizes */
#define NSMALLBINS        (32U)
#define NTREEBINS         (32U)
#define SMALLBIN_SHIFT    (3U)
#define SMALLBIN_WIDTH    (SIZE_T_ONE << SMALLBIN_SHIFT)
#define TREEBIN_SHIFT     (8U)
```



```
#define MIN_LARGE_SIZE    (SIZE_T_ONE << TREEBIN_SHIFT)
#define MAX_SMALL_SIZE    (MIN_LARGE_SIZE - SIZE_T_ONE)
#define MAX_SMALL_REQUEST (MAX_SMALL_SIZE - CHUNK_ALIGN_MASK - CHUNK_OVERHEAD)

struct malloc_state {
  binmap_t   smallmap;
  binmap_t   treemap;
  size_t     dvsize;
  size_t     topsize;
  char*      least_addr;
  mchunkptr  dv;
  mchunkptr  top;
  size_t     trim_check;
  size_t     release_checks;
  size_t     magic;
  mchunkptr  smallbins[(NSMALLBINS+1)*2];
  tbinptr    treebins[NTREEBINS];
  size_t     footprint;
  size_t     max_footprint;
  flag_t     mflags;
#if USE_LOCKS
  MLOCK_T    mutex;     /* locate lock among fields that rarely change */
#endif /* USE_LOCKS */
  msegment   seg;
  void*      extp;      /* Unused but available for extensions */
  size_t     exts;
};

typedef struct malloc_state*    mstate;
```

11.2 Global malloc_state and malloc_params

```
/*
  malloc_params holds global properties, including those that can be
  dynamically set using mallopt. There is a single instance, mparams,
  initialized in init_mparams. Note that the non-zeroness of "magic"
  also serves as an initialization flag.
*/

struct malloc_params {
  volatile size_t magic;
  size_t page_size;
  size_t granularity;
  size_t mmap_threshold;
  size_t trim_threshold;
  flag_t default_mflags;
};

static struct malloc_params mparams;
```

```
/* Ensure mparams initialized */
#define ensure_initialization() (void)(mparams.magic != 0 || init_mparams())

#if !ONLY_MSPACES

/* The global malloc_state used for all non-"mspace" calls */
static struct malloc_state _gm_;
#define gm                 (&_gm_)
#define is_global(M)       ((M) == &_gm_)

#endif /* !ONLY_MSPACES */

#define is_initialized(M)  ((M)->top != 0)
```

Chapter 12

System alloc setup

```
/* Operations on mflags */

#define use_lock(M)           ((M)->mflags &   USE_LOCK_BIT)
#define enable_lock(M)        ((M)->mflags |=  USE_LOCK_BIT)
#define disable_lock(M)       ((M)->mflags &= ~USE_LOCK_BIT)

#define use_mmap(M)           ((M)->mflags &   USE_MMAP_BIT)
#define enable_mmap(M)        ((M)->mflags |=  USE_MMAP_BIT)
#define disable_mmap(M)       ((M)->mflags &= ~USE_MMAP_BIT)

#define use_noncontiguous(M)  ((M)->mflags &   USE_NONCONTIGUOUS_BIT)
#define disable_contiguous(M) ((M)->mflags |=  USE_NONCONTIGUOUS_BIT)

#define set_lock(M,L)\
 ((M)->mflags = (L)?\
  ((M)->mflags | USE_LOCK_BIT) :\
  ((M)->mflags & ~USE_LOCK_BIT))

/* page-align a size */
#define page_align(S)\
 (((S) + (mparams.page_size - SIZE_T_ONE)) & ~(mparams.page_size - SIZE_T_ONE))

/* granularity-align a size */
#define granularity_align(S)\
  (((S) + (mparams.granularity - SIZE_T_ONE))\
   & ~(mparams.granularity - SIZE_T_ONE))


/* For mmap, use granularity alignment on windows, else page-align */
#ifdef WIN32
#define mmap_align(S) granularity_align(S)
#else
#define mmap_align(S) page_align(S)
#endif

/* For sys_alloc, enough padding to ensure can malloc request on success */
#define SYS_ALLOC_PADDING (TOP_FOOT_SIZE + MALLOC_ALIGNMENT)

#define is_page_aligned(S)\
```

```
   (((size_t)(S) & (mparams.page_size - SIZE_T_ONE)) == 0)
#define is_granularity_aligned(S)\
   (((size_t)(S) & (mparams.granularity - SIZE_T_ONE)) == 0)

/*  True if segment S holds address A */
#define segment_holds(S, A)\
  ((char*)(A) >= S->base && (char*)(A) < S->base + S->size)

/* Return segment holding given address */
static msegmentptr segment_holding(mstate m, char* addr) {
  msegmentptr sp = &m->seg;
  for (;;) {
    if (addr >= sp->base && addr < sp->base + sp->size)
      return sp;
    if ((sp = sp->next) == 0)
      return 0;
  }
}

/* Return true if segment contains a segment link */
static int has_segment_link(mstate m, msegmentptr ss) {
  msegmentptr sp = &m->seg;
  for (;;) {
    if ((char*)sp >= ss->base && (char*)sp < ss->base + ss->size)
      return 1;
    if ((sp = sp->next) == 0)
      return 0;
  }
}

#ifndef MORECORE_CANNOT_TRIM
#define should_trim(M,s)  ((s) > (M)->trim_check)
#else  /* MORECORE_CANNOT_TRIM */
#define should_trim(M,s)  (0)
#endif /* MORECORE_CANNOT_TRIM */

/*
  TOP_FOOT_SIZE is padding at the end of a segment, including space
  that may be needed to place segment records and fenceposts when new
  noncontiguous segments are added.
*/
#define TOP_FOOT_SIZE\
  (align_offset(chunk2mem(0))+pad_request(sizeof(struct malloc_segment))+ \
                                                            MIN_CHUNK_SIZE)
```

Chapter 13

Hooks

```
/*
  PREACTION should be defined to return 0 on success, and nonzero on
  failure. If you are not using locking, you can redefine these to do
  anything you like.
*/

#if USE_LOCKS

#define PREACTION(M)  ((use_lock(M))? ACQUIRE_LOCK(&(M)->mutex) : 0)
#define POSTACTION(M) { if (use_lock(M)) RELEASE_LOCK(&(M)->mutex); }
#else /* USE_LOCKS */

#ifndef PREACTION
#define PREACTION(M) (0)
#endif  /* PREACTION */

#ifndef POSTACTION
#define POSTACTION(M)
#endif  /* POSTACTION */

#endif /* USE_LOCKS */

/*
  CORRUPTION_ERROR_ACTION is triggered upon detected bad addresses.
  USAGE_ERROR_ACTION is triggered on detected bad frees and
  reallocs. The argument p is an address that might have triggered the
  fault. It is ignored by the two predefined actions, but might be
  useful in custom actions that try to help diagnose errors.
*/

#if PROCEED_ON_ERROR

/* A count of the number of corruption errors causing resets */
int malloc_corruption_error_count;

/* default corruption action */
static void reset_on_error(mstate m);

#define CORRUPTION_ERROR_ACTION(m)  reset_on_error(m)
```

```
#define USAGE_ERROR_ACTION(m, p)

#else /* PROCEED_ON_ERROR */

#ifndef CORRUPTION_ERROR_ACTION
#define CORRUPTION_ERROR_ACTION(m) ABORT
#endif /* CORRUPTION_ERROR_ACTION */

#ifndef USAGE_ERROR_ACTION
#define USAGE_ERROR_ACTION(m,p) ABORT
#endif /* USAGE_ERROR_ACTION */

#endif /* PROCEED_ON_ERROR */
```

Chapter 14

Debugging setup

```
#if ! DEBUG

#define check_free_chunk(M,P)
#define check_inuse_chunk(M,P)
#define check_malloced_chunk(M,P,N)
#define check_mmapped_chunk(M,P)
#define check_malloc_state(M)
#define check_top_chunk(M,P)

#else /* DEBUG */
#define check_free_chunk(M,P)       do_check_free_chunk(M,P)
#define check_inuse_chunk(M,P)      do_check_inuse_chunk(M,P)
#define check_top_chunk(M,P)        do_check_top_chunk(M,P)
#define check_malloced_chunk(M,P,N) do_check_malloced_chunk(M,P,N)
#define check_mmapped_chunk(M,P)    do_check_mmapped_chunk(M,P)
#define check_malloc_state(M)       do_check_malloc_state(M)

static void   do_check_any_chunk(mstate m, mchunkptr p);
static void   do_check_top_chunk(mstate m, mchunkptr p);
static void   do_check_mmapped_chunk(mstate m, mchunkptr p);
static void   do_check_inuse_chunk(mstate m, mchunkptr p);
static void   do_check_free_chunk(mstate m, mchunkptr p);
static void   do_check_malloced_chunk(mstate m, void* mem, size_t s);
static void   do_check_tree(mstate m, tchunkptr t);
static void   do_check_treebin(mstate m, bindex_t i);
static void   do_check_smallbin(mstate m, bindex_t i);
static void   do_check_malloc_state(mstate m);
static int    bin_find(mstate m, mchunkptr x);
static size_t traverse_and_check(mstate m);
#endif /* DEBUG */
```

Chapter 15

Bins

15.1 Indexing Bins

```
#define is_small(s)         (((s) >> SMALLBIN_SHIFT) < NSMALLBINS)
#define small_index(s)      ((s)  >> SMALLBIN_SHIFT)
#define small_index2size(i) ((i)  << SMALLBIN_SHIFT)
#define MIN_SMALL_INDEX     (small_index(MIN_CHUNK_SIZE))

/* addressing by index. See above about smallbin repositioning */
#define smallbin_at(M, i)   ((sbinptr)((char*)&((M)->smallbins[(i)<<1])))
#define treebin_at(M,i)     (&((M)->treebins[i]))

/* assign tree index for size S to variable I. Use x86 asm if possible  */
#if defined(__GNUC__) && (defined(__i386__) || defined(__x86_64__))
#define compute_tree_index(S, I)\
{\
  unsigned int X = S >> TREEBIN_SHIFT;\
  if (X == 0)\
    I = 0;\
  else if (X > 0xFFFF)\
    I = NTREEBINS-1;\
  else {\
    unsigned int K;\
    __asm__("bsrl\t%1, %0\n\t" : "=r" (K) : "g"  (X));\
    I =  (bindex_t)((K << 1) + ((S >> (K + (TREEBIN_SHIFT-1)) & 1)));\
  }\
}

#elif defined (__INTEL_COMPILER)
#define compute_tree_index(S, I)\
{\
  size_t X = S >> TREEBIN_SHIFT;\
  if (X == 0)\
    I = 0;\
  else if (X > 0xFFFF)\
    I = NTREEBINS-1;\
  else {\
    unsigned int K = _bit_scan_reverse (X); \
```



```
    I =  (bindex_t)((K << 1) + ((S >> (K + (TREEBIN_SHIFT-1)) & 1)));\
  }\
}

#elif defined(_MSC_VER) && _MSC_VER>=1300
#define compute_tree_index(S, I)\
{\
  size_t X = S >> TREEBIN_SHIFT;\
  if (X == 0)\
    I = 0;\
  else if (X > 0xFFFF)\
    I = NTREEBINS-1;\
  else {\
    unsigned int K;\
    _BitScanReverse((DWORD *) &K, X);\
    I =  (bindex_t)((K << 1) + ((S >> (K + (TREEBIN_SHIFT-1)) & 1)));\
  }\
}

#else /* GNUC */
#define compute_tree_index(S, I)\
{\
  size_t X = S >> TREEBIN_SHIFT;\
  if (X == 0)\
    I = 0;\
  else if (X > 0xFFFF)\
    I = NTREEBINS-1;\
  else {\
    unsigned int Y = (unsigned int)X;\
    unsigned int N = ((Y - 0x100) >> 16) & 8;\
    unsigned int K = (((Y <<= N) - 0x1000) >> 16) & 4;\
    N += K;\
    N += K = (((Y <<= K) - 0x4000) >> 16) & 2;\
    K = 14 - N + ((Y <<= K) >> 15);\
    I = (K << 1) + ((S >> (K + (TREEBIN_SHIFT-1)) & 1));\
  }\
}
#endif /* GNUC */

/* Bit representing maximum resolved size in a treebin at i */
#define bit_for_tree_index(i) \
   (i == NTREEBINS-1)? (SIZE_T_BITSIZE-1) : (((i) >> 1) + TREEBIN_SHIFT - 2)

/* Shift placing maximum resolved bit in a treebin at i as sign bit */
#define leftshift_for_tree_index(i) \
   ((i == NTREEBINS-1)? 0 : \
    ((SIZE_T_BITSIZE-SIZE_T_ONE) - (((i) >> 1) + TREEBIN_SHIFT - 2)))

/* The size of the smallest chunk held in bin with index i */
#define minsize_for_tree_index(i) \
   ((SIZE_T_ONE << (((i) >> 1) + TREEBIN_SHIFT)) |  \
   (((size_t)((i) & SIZE_T_ONE)) << (((i) >> 1) + TREEBIN_SHIFT - 1)))
```

15.2 Operations on bin maps

```
/* bit corresponding to given index */
#define idx2bit(i)              ((binmap_t)(1) << (i))

/* Mark/Clear bits with given index */
#define mark_smallmap(M,i)      ((M)->smallmap |=  idx2bit(i))
#define clear_smallmap(M,i)     ((M)->smallmap &= ~idx2bit(i))
#define smallmap_is_marked(M,i) ((M)->smallmap &   idx2bit(i))

#define mark_treemap(M,i)       ((M)->treemap  |=  idx2bit(i))
#define clear_treemap(M,i)      ((M)->treemap  &= ~idx2bit(i))
#define treemap_is_marked(M,i)  ((M)->treemap  &   idx2bit(i))

/* isolate the least set bit of a bitmap */
#define least_bit(x)         ((x) & -(x))

/* mask with all bits to left of least bit of x on */
#define left_bits(x)         ((x<<1) | -(x<<1))

/* mask with all bits to left of or equal to least bit of x on */
#define same_or_left_bits(x) ((x) | -(x))

/* index corresponding to given bit. Use x86 asm if possible */

#if defined(__GNUC__) && (defined(__i386__) || defined(__x86_64__))
#define compute_bit2idx(X, I)\
{\
  unsigned int J;\
  __asm__("bsfl\t%1, %0\n\t" : "=r" (J) : "g" (X));\
  I = (bindex_t)J;\
}

#elif defined (__INTEL_COMPILER)
#define compute_bit2idx(X, I)\
{\
  unsigned int J;\
  J = _bit_scan_forward (X); \
  I = (bindex_t)J;\
}

#elif defined(_MSC_VER) && _MSC_VER>=1300
#define compute_bit2idx(X, I)\
{\
  unsigned int J;\
  _BitScanForward((DWORD *) &J, X);\
  I = (bindex_t)J;\
}
```



```
#elif USE_BUILTIN_FFS
#define compute_bit2idx(X, I) I = ffs(X)-1

#else
#define compute_bit2idx(X, I)\
{\
  unsigned int Y = X - 1;\
  unsigned int K = Y >> (16-4) & 16;\
  unsigned int N = K;        Y >>= K;\
  N += K = Y >> (8-3) &  8;  Y >>= K;\
  N += K = Y >> (4-2) &  4;  Y >>= K;\
  N += K = Y >> (2-1) &  2;  Y >>= K;\
  N += K = Y >> (1-0) &  1;  Y >>= K;\
  I = (bindex_t)(N + Y);\
}
#endif /* GNUC */
```

Chapter 16

Runtime Check Support

```
/*
  For security, the main invariant is that malloc/free/etc never
  writes to a static address other than malloc_state, unless static
  malloc_state itself has been corrupted, which cannot occur via
  malloc (because of these checks). In essence this means that we
  believe all pointers, sizes, maps etc held in malloc_state, but
  check all of those linked or offsetted from other embedded data
  structures.  These checks are interspersed with main code in a way
  that tends to minimize their run-time cost.

  When FOOTERS is defined, in addition to range checking, we also
  verify footer fields of inuse chunks, which can be used guarantee
  that the mstate controlling malloc/free is intact.  This is a
  streamlined version of the approach described by William Robertson
  et al in "Run-time Detection of Heap-based Overflows" LISA'03
  http://www.usenix.org/events/lisa03/tech/robertson.html The footer
  of an inuse chunk holds the xor of its mstate and a random seed,
  that is checked upon calls to free() and realloc().  This is
  (probablistically) unguessable from outside the program, but can be
  computed by any code successfully malloc'ing any chunk, so does not
  itself provide protection against code that has already broken
  security through some other means.  Unlike Robertson et al, we
  always dynamically check addresses of all offset chunks (previous,
  next, etc). This turns out to be cheaper than relying on hashes.
*/

#if !INSECURE
/* Check if address a is at least as high as any from MORECORE or MMAP */
#define ok_address(M, a) ((char*)(a) >= (M)->least_addr)
/* Check if address of next chunk n is higher than base chunk p */
#define ok_next(p, n)    ((char*)(p) < (char*)(n))
/* Check if p has inuse status */
#define ok_inuse(p)     is_inuse(p)
/* Check if p has its pinuse bit on */
#define ok_pinuse(p)     pinuse(p)

#else /* !INSECURE */
#define ok_address(M, a) (1)
#define ok_next(b, n)    (1)
```



```
#define ok_inuse(p)      (1)
#define ok_pinuse(p)     (1)
#endif /* !INSECURE */

#if (FOOTERS && !INSECURE)
/* Check if (alleged) mstate m has expected magic field */
#define ok_magic(M)      ((M)->magic == mparams.magic)
#else  /* (FOOTERS && !INSECURE) */
#define ok_magic(M)      (1)
#endif /* (FOOTERS && !INSECURE) */


/* In gcc, use __builtin_expect to minimize impact of checks */
#if !INSECURE
#if defined(__GNUC__) && __GNUC__ >= 3
#define RTCHECK(e)  __builtin_expect(e, 1)
#else /* GNUC */
#define RTCHECK(e)  (e)
#endif /* GNUC */
#else /* !INSECURE */
#define RTCHECK(e)  (1)
#endif /* !INSECURE */

/* macros to set up inuse chunks with or without footers */

#if !FOOTERS

#define mark_inuse_foot(M,p,s)

/* Macros for setting head/foot of non-mmapped chunks */

/* Set cinuse bit and pinuse bit of next chunk */
#define set_inuse(M,p,s)\
  ((p)->head = (((p)->head & PINUSE_BIT)|s|CINUSE_BIT),\
  ((mchunkptr)(((char*)(p)) + (s)))->head |= PINUSE_BIT)

/* Set cinuse and pinuse of this chunk and pinuse of next chunk */
#define set_inuse_and_pinuse(M,p,s)\
  ((p)->head = (s|PINUSE_BIT|CINUSE_BIT),\
  ((mchunkptr)(((char*)(p)) + (s)))->head |= PINUSE_BIT)

/* Set size, cinuse and pinuse bit of this chunk */
#define set_size_and_pinuse_of_inuse_chunk(M, p, s)\
  ((p)->head = (s|PINUSE_BIT|CINUSE_BIT))

#else /* FOOTERS */

/* Set foot of inuse chunk to be xor of mstate and seed */
#define mark_inuse_foot(M,p,s)\
  (((mchunkptr)((char*)(p) + (s)))->prev_foot = ((size_t)(M) ^ mparams.magic))

#define get_mstate_for(p)\
```

```
  ((mstate)(((mchunkptr)((char*)(p) +\
    (chunksize(p))))->prev_foot ^ mparams.magic))

#define set_inuse(M,p,s)\
  ((p)->head = (((p)->head & PINUSE_BIT)|s|CINUSE_BIT),\
  (((mchunkptr)(((char*)(p)) + (s)))->head |= PINUSE_BIT), \
  mark_inuse_foot(M,p,s))

#define set_inuse_and_pinuse(M,p,s)\
  ((p)->head = (s|PINUSE_BIT|CINUSE_BIT),\
  (((mchunkptr)(((char*)(p)) + (s)))->head |= PINUSE_BIT),\
 mark_inuse_foot(M,p,s))

#define set_size_and_pinuse_of_inuse_chunk(M, p, s)\
  ((p)->head = (s|PINUSE_BIT|CINUSE_BIT),\
  mark_inuse_foot(M, p, s))

#endif /* !FOOTERS */
```

Chapter 17

Setting mparams

```
/* Initialize mparams */
static int init_mparams(void) {
#ifdef NEED_GLOBAL_LOCK_INIT
  if (malloc_global_mutex_status <= 0)
    init_malloc_global_mutex();
#endif

  ACQUIRE_MALLOC_GLOBAL_LOCK();
  if (mparams.magic == 0) {
    size_t magic;
    size_t psize;
    size_t gsize;

#ifndef WIN32
    psize = malloc_getpagesize;
    gsize = ((DEFAULT_GRANULARITY != 0)? DEFAULT_GRANULARITY : psize);
#else /* WIN32 */
    {
      SYSTEM_INFO system_info;
      GetSystemInfo(&system_info);
      psize = system_info.dwPageSize;
      gsize = ((DEFAULT_GRANULARITY != 0)?
               DEFAULT_GRANULARITY : system_info.dwAllocationGranularity);
    }
#endif /* WIN32 */

    /* Sanity-check configuration:
       size_t must be unsigned and as wide as pointer type.
       ints must be at least 4 bytes.
       alignment must be at least 8.
       Alignment, min chunk size, and page size must all be powers of 2.
    */
    if ((sizeof(size_t) != sizeof(char*)) ||
        (MAX_SIZE_T < MIN_CHUNK_SIZE)  ||
        (sizeof(int) < 4)  ||
        (MALLOC_ALIGNMENT < (size_t)8U) ||
        ((MALLOC_ALIGNMENT & (MALLOC_ALIGNMENT-SIZE_T_ONE)) != 0) ||
        ((MCHUNK_SIZE      & (MCHUNK_SIZE-SIZE_T_ONE))      != 0) ||
        ((gsize            & (gsize-SIZE_T_ONE))            != 0) ||
```

```
        ((psize            & (psize-SIZE_T_ONE))            != 0))
      ABORT;

    mparams.granularity = gsize;
    mparams.page_size = psize;
    mparams.mmap_threshold = DEFAULT_MMAP_THRESHOLD;
    mparams.trim_threshold = DEFAULT_TRIM_THRESHOLD;
#if MORECORE_CONTIGUOUS
    mparams.default_mflags = USE_LOCK_BIT|USE_MMAP_BIT;
#else  /* MORECORE_CONTIGUOUS */
    mparams.default_mflags = USE_LOCK_BIT|USE_MMAP_BIT|USE_NONCONTIGUOUS_BIT;
#endif /* MORECORE_CONTIGUOUS */

#if !ONLY_MSPACES
    /* Set up lock for main malloc area */
    gm->mflags = mparams.default_mflags;
    INITIAL_LOCK(&gm->mutex);
#endif

    {
#if USE_DEV_RANDOM
      int fd;
      unsigned char buf[sizeof(size_t)];
      /* Try to use /dev/urandom, else fall back on using time */
      if ((fd = open("/dev/urandom", O_RDONLY)) >= 0 &&
          read(fd, buf, sizeof(buf)) == sizeof(buf)) {
        magic = *((size_t *) buf);
        close(fd);
      }
      else
#endif /* USE_DEV_RANDOM */
#ifdef WIN32
        magic = (size_t)(GetTickCount() ^ (size_t)0x55555555U);
#else
        magic = (size_t)(time(0) ^ (size_t)0x55555555U);
#endif
      magic |= (size_t)8U;    /* ensure nonzero */
      magic &= ~(size_t)7U;   /* improve chances of fault for bad values */
      mparams.magic = magic;
    }
  }

  RELEASE_MALLOC_GLOBAL_LOCK();
  return 1;
}

/* support for mallopt */
static int change_mparam(int param_number, int value) {
  size_t val;
  ensure_initialization();
  val = (value == -1)? MAX_SIZE_T : (size_t)value;
  switch(param_number) {
  case M_TRIM_THRESHOLD:
    mparams.trim_threshold = val;
    return 1;
  case M_GRANULARITY:
    if (val >= mparams.page_size && ((val & (val-1)) == 0)) {
      mparams.granularity = val;
      return 1;
    }
    else
      return 0;
  case M_MMAP_THRESHOLD:
    mparams.mmap_threshold = val;
    return 1;
  default:
    return 0;
  }
}
```

Chapter 18

Debugging Support

```
#if DEBUG

/* Check properties of any chunk, whether free, inuse, mmapped etc  */
static void do_check_any_chunk(mstate m, mchunkptr p) {
  assert((is_aligned(chunk2mem(p))) || (p->head == FENCEPOST_HEAD));
  assert(ok_address(m, p));
}

/* Check properties of top chunk */
static void do_check_top_chunk(mstate m, mchunkptr p) {
  msegmentptr sp = segment_holding(m, (char*)p);
  size_t  sz = p->head & ~INUSE_BITS; /* third-lowest bit can be set! */
  assert(sp != 0);
  assert((is_aligned(chunk2mem(p))) || (p->head == FENCEPOST_HEAD));
  assert(ok_address(m, p));
  assert(sz == m->topsize);
  assert(sz > 0);
  assert(sz == ((sp->base + sp->size) - (char*)p) - TOP_FOOT_SIZE);
  assert(pinuse(p));
  assert(!pinuse(chunk_plus_offset(p, sz)));
}

/* Check properties of (inuse) mmapped chunks */
static void do_check_mmapped_chunk(mstate m, mchunkptr p) {
  size_t  sz = chunksize(p);
  size_t len = (sz + (p->prev_foot) + MMAP_FOOT_PAD);
  assert(is_mmapped(p));
  assert(use_mmap(m));
  assert((is_aligned(chunk2mem(p))) || (p->head == FENCEPOST_HEAD));
  assert(ok_address(m, p));
  assert(!is_small(sz));
  assert((len & (mparams.page_size-SIZE_T_ONE)) == 0);
  assert(chunk_plus_offset(p, sz)->head == FENCEPOST_HEAD);
  assert(chunk_plus_offset(p, sz+SIZE_T_SIZE)->head == 0);
}

/* Check properties of inuse chunks */
static void do_check_inuse_chunk(mstate m, mchunkptr p) {
  do_check_any_chunk(m, p);
```



```
  assert(is_inuse(p));
  assert(next_pinuse(p));
  /* If not pinuse and not mmapped, previous chunk has OK offset */
  assert(is_mmapped(p) || pinuse(p) || next_chunk(prev_chunk(p)) == p);
  if (is_mmapped(p))
    do_check_mmapped_chunk(m, p);
}

/* Check properties of free chunks */
static void do_check_free_chunk(mstate m, mchunkptr p) {
  size_t sz = chunksize(p);
  mchunkptr next = chunk_plus_offset(p, sz);
  do_check_any_chunk(m, p);
  assert(!is_inuse(p));
  assert(!next_pinuse(p));
  assert (!is_mmapped(p));
  if (p != m->dv && p != m->top) {
    if (sz >= MIN_CHUNK_SIZE) {
      assert((sz & CHUNK_ALIGN_MASK) == 0);
      assert(is_aligned(chunk2mem(p)));
      assert(next->prev_foot == sz);
      assert(pinuse(p));
      assert (next == m->top || is_inuse(next));
      assert(p->fd->bk == p);
      assert(p->bk->fd == p);
    }
    else  /* markers are always of size SIZE_T_SIZE */
      assert(sz == SIZE_T_SIZE);
  }
}

/* Check properties of malloced chunks at the point they are malloced */
static void do_check_malloced_chunk(mstate m, void* mem, size_t s) {
  if (mem != 0) {
    mchunkptr p = mem2chunk(mem);
    size_t sz = p->head & ~INUSE_BITS;
    do_check_inuse_chunk(m, p);
    assert((sz & CHUNK_ALIGN_MASK) == 0);
    assert(sz >= MIN_CHUNK_SIZE);
    assert(sz >= s);
    /* unless mmapped, size is less than MIN_CHUNK_SIZE more than request */
    assert(is_mmapped(p) || sz < (s + MIN_CHUNK_SIZE));
  }
}

/* Check a tree and its subtrees.  */
static void do_check_tree(mstate m, tchunkptr t) {
  tchunkptr head = 0;
  tchunkptr u = t;
  bindex_t tindex = t->index;
  size_t tsize = chunksize(t);
  bindex_t idx;
```

```
  compute_tree_index(tsize, idx);
  assert(tindex == idx);
  assert(tsize >= MIN_LARGE_SIZE);
  assert(tsize >= minsize_for_tree_index(idx));
  assert((idx == NTREEBINS-1) || (tsize < minsize_for_tree_index((idx+1))));

  do { /* traverse through chain of same-sized nodes */
    do_check_any_chunk(m, ((mchunkptr)u));
    assert(u->index == tindex);
    assert(chunksize(u) == tsize);
    assert(!is_inuse(u));
    assert(!next_pinuse(u));
    assert(u->fd->bk == u);
    assert(u->bk->fd == u);
    if (u->parent == 0) {
      assert(u->child[0] == 0);
      assert(u->child[1] == 0);
    }
    else {
      assert(head == 0); /* only one node on chain has parent */
      head = u;
      assert(u->parent != u);
      assert (u->parent->child[0] == u ||
              u->parent->child[1] == u ||
              *((tbinptr*)(u->parent)) == u);
      if (u->child[0] != 0) {
        assert(u->child[0]->parent == u);
        assert(u->child[0] != u);
        do_check_tree(m, u->child[0]);
      }
      if (u->child[1] != 0) {
        assert(u->child[1]->parent == u);
        assert(u->child[1] != u);
        do_check_tree(m, u->child[1]);
      }
      if (u->child[0] != 0 && u->child[1] != 0) {
        assert(chunksize(u->child[0]) < chunksize(u->child[1]));
      }
    }
    u = u->fd;
  } while (u != t);
  assert(head != 0);
}

/*  Check all the chunks in a treebin.  */
static void do_check_treebin(mstate m, bindex_t i) {
  tbinptr* tb = treebin_at(m, i);
  tchunkptr t = *tb;
  int empty = (m->treemap & (1U << i)) == 0;
  if (t == 0)
    assert(empty);
  if (!empty)
    do_check_tree(m, t);
}

/*  Check all the chunks in a smallbin.  */
static void do_check_smallbin(mstate m, bindex_t i) {
  sbinptr b = smallbin_at(m, i);
  mchunkptr p = b->bk;
  unsigned int empty = (m->smallmap & (1U << i)) == 0;
  if (p == b)
    assert(empty);
  if (!empty) {
    for (; p != b; p = p->bk) {
      size_t size = chunksize(p);
      mchunkptr q;
      /* each chunk claims to be free */
      do_check_free_chunk(m, p);
      /* chunk belongs in bin */
      assert(small_index(size) == i);
      assert(p->bk == b || chunksize(p->bk) == chunksize(p));
      /* chunk is followed by an inuse chunk */
      q = next_chunk(p);
      if (q->head != FENCEPOST_HEAD)
        do_check_inuse_chunk(m, q);
    }
  }
}

/* Find x in a bin. Used in other check functions. */
static int bin_find(mstate m, mchunkptr x) {
  size_t size = chunksize(x);
  if (is_small(size)) {
    bindex_t sidx = small_index(size);
    sbinptr b = smallbin_at(m, sidx);
    if (smallmap_is_marked(m, sidx)) {
      mchunkptr p = b;
      do {
        if (p == x)
          return 1;
      } while ((p = p->fd) != b);
    }
  }
  else {
    bindex_t tidx;
    compute_tree_index(size, tidx);
    if (treemap_is_marked(m, tidx)) {
      tchunkptr t = *treebin_at(m, tidx);
      size_t sizebits = size << leftshift_for_tree_index(tidx);
      while (t != 0 && chunksize(t) != size) {
        t = t->child[(sizebits >> (SIZE_T_BITSIZE-SIZE_T_ONE)) & 1];
        sizebits <<= 1;
      }
      if (t != 0) {
```

```
        tchunkptr u = t;
        do {
          if (u == (tchunkptr)x)
            return 1;
        } while ((u = u->fd) != t);
      }
    }
  }
  return 0;
}

/* Traverse each chunk and check it; return total */
static size_t traverse_and_check(mstate m) {
  size_t sum = 0;
  if (is_initialized(m)) {
    msegmentptr s = &m->seg;
    sum += m->topsize + TOP_FOOT_SIZE;
    while (s != 0) {
      mchunkptr q = align_as_chunk(s->base);
      mchunkptr lastq = 0;
      assert(pinuse(q));
      while (segment_holds(s, q) &&
             q != m->top && q->head != FENCEPOST_HEAD) {
        sum += chunksize(q);
        if (is_inuse(q)) {
          assert(!bin_find(m, q));
          do_check_inuse_chunk(m, q);
        }
        else {
          assert(q == m->dv || bin_find(m, q));
          assert(lastq == 0 || is_inuse(lastq)); /* Not 2 consecutive free */
          do_check_free_chunk(m, q);
        }
        lastq = q;
        q = next_chunk(q);
      }
      s = s->next;
    }
  }
  return sum;
}

/* Check all properties of malloc_state. */
static void do_check_malloc_state(mstate m) {
  bindex_t i;
  size_t total;
  /* check bins */
  for (i = 0; i < NSMALLBINS; ++i)
    do_check_smallbin(m, i);
  for (i = 0; i < NTREEBINS; ++i)
    do_check_treebin(m, i);

  if (m->dvsize != 0) { /* check dv chunk */
    do_check_any_chunk(m, m->dv);
    assert(m->dvsize == chunksize(m->dv));
    assert(m->dvsize >= MIN_CHUNK_SIZE);
    assert(bin_find(m, m->dv) == 0);
  }

  if (m->top != 0) {   /* check top chunk */
    do_check_top_chunk(m, m->top);
    /*assert(m->topsize == chunksize(m->top)); redundant */
    assert(m->topsize > 0);
    assert(bin_find(m, m->top) == 0);
  }

  total = traverse_and_check(m);
  assert(total <= m->footprint);
  assert(m->footprint <= m->max_footprint);
}
#endif /* DEBUG */
```

Chapter 19

Statistics

```
#if !NO_MALLINFO
static struct mallinfo internal_mallinfo(mstate m) {
  struct mallinfo nm = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
  ensure_initialization();
  if (!PREACTION(m)) {
    check_malloc_state(m);
    if (is_initialized(m)) {
      size_t nfree = SIZE_T_ONE; /* top always free */
      size_t mfree = m->topsize + TOP_FOOT_SIZE;
      size_t sum = mfree;
      msegmentptr s = &m->seg;
      while (s != 0) {
        mchunkptr q = align_as_chunk(s->base);
        while (segment_holds(s, q) &&
               q != m->top && q->head != FENCEPOST_HEAD) {
          size_t sz = chunksize(q);
          sum += sz;
          if (!is_inuse(q)) {
            mfree += sz;
            ++nfree;
          }
          q = next_chunk(q);
        }
        s = s->next;
      }

      nm.arena    = sum;
      nm.ordblks  = nfree;
      nm.hblkhd   = m->footprint - sum;
      nm.usmblks  = m->max_footprint;
      nm.uordblks = m->footprint - mfree;
      nm.fordblks = mfree;
      nm.keepcost = m->topsize;
    }

    POSTACTION(m);
  }
  return nm;
}
```



```
#endif /* !NO_MALLINFO */

static void internal_malloc_stats(mstate m) {
  ensure_initialization();
  if (!PREACTION(m)) {
    size_t maxfp = 0;
    size_t fp = 0;
    size_t used = 0;
    check_malloc_state(m);
    if (is_initialized(m)) {
      msegmentptr s = &m->seg;
      maxfp = m->max_footprint;
      fp = m->footprint;
      used = fp - (m->topsize + TOP_FOOT_SIZE);

      while (s != 0) {
        mchunkptr q = align_as_chunk(s->base);
        while (segment_holds(s, q) &&
               q != m->top && q->head != FENCEPOST_HEAD) {
          if (!is_inuse(q))
            used -= chunksize(q);
          q = next_chunk(q);
        }
        s = s->next;
      }
    }

    fprintf(stderr, "max system bytes = %10lu\n", (unsigned long)(maxfp));
    fprintf(stderr, "system bytes     = %10lu\n", (unsigned long)(fp));
    fprintf(stderr, "in use bytes     = %10lu\n", (unsigned long)(used));

    POSTACTION(m);
  }
}
```

Chapter 20

Operations on smallbins and trees

```
/*
  Various forms of linking and unlinking are defined as macros.  Even
  the ones for trees, which are very long but have very short typical
  paths.  This is ugly but reduces reliance on inlining support of
  compilers.
*/
```

20.1 Operations on smallbins

```
/* Link a free chunk into a smallbin  */
#define insert_small_chunk(M, P, S) {\
  bindex_t I  = small_index(S);\
  mchunkptr B = smallbin_at(M, I);\
  mchunkptr F = B;\
  assert(S >= MIN_CHUNK_SIZE);\
  if (!smallmap_is_marked(M, I))\
    mark_smallmap(M, I);\
  else if (RTCHECK(ok_address(M, B->fd)))\
    F = B->fd;\
  else {\
    CORRUPTION_ERROR_ACTION(M);\
  }\
  B->fd = P;\
  F->bk = P;\
  P->fd = F;\
  P->bk = B;\
}

/* Unlink a chunk from a smallbin  */
#define unlink_small_chunk(M, P, S) {\
  mchunkptr F = P->fd;\
  mchunkptr B = P->bk;\
  bindex_t I = small_index(S);\
  assert(P != B);\
  assert(P != F);\
  assert(chunksize(P) == small_index2size(I));\
  if (F == B)\
    clear_smallmap(M, I);\
  else if (RTCHECK((F == smallbin_at(M,I) || ok_address(M, F)) &&\
```



```
                   (B == smallbin_at(M,I) || ok_address(M, B)))) {\
    F->bk = B;\
    B->fd = F;\
  }\
  else {\
    CORRUPTION_ERROR_ACTION(M);\
  }\
}

/* Unlink the first chunk from a smallbin */
#define unlink_first_small_chunk(M, B, P, I) {\
  mchunkptr F = P->fd;\
  assert(P != B);\
  assert(P != F);\
  assert(chunksize(P) == small_index2size(I));\
  if (B == F)\
    clear_smallmap(M, I);\
  else if (RTCHECK(ok_address(M, F))) {\
    B->fd = F;\
    F->bk = B;\
  }\
  else {\
    CORRUPTION_ERROR_ACTION(M);\
  }\
}



/* Replace dv node, binning the old one */
/* Used only when dvsize known to be small */
#define replace_dv(M, P, S) {\
  size_t DVS = M->dvsize;\
  if (DVS != 0) {\
    mchunkptr DV = M->dv;\
    assert(is_small(DVS));\
    insert_small_chunk(M, DV, DVS);\
  }\
  M->dvsize = S;\
  M->dv = P;\
}
```

20.2 Operations on trees

```
/* Insert chunk into tree */
#define insert_large_chunk(M, X, S) {\
  tbinptr* H;\
  bindex_t I;\
  compute_tree_index(S, I);\
  H = treebin_at(M, I);\
  X->index = I;\
  X->child[0] = X->child[1] = 0;\
```

```
  if (!treemap_is_marked(M, I)) {\
    mark_treemap(M, I);\
    *H = X;\
    X->parent = (tchunkptr)H;\
    X->fd = X->bk = X;\
  }\
  else {\
    tchunkptr T = *H;\
    size_t K = S << leftshift_for_tree_index(I);\
    for (;;) {\
      if (chunksize(T) != S) {\
        tchunkptr* C = &(T->child[(K >> (SIZE_T_BITSIZE-SIZE_T_ONE)) & 1]);\
        K <<= 1;\
        if (*C != 0)\
          T = *C;\
        else if (RTCHECK(ok_address(M, C))) {\
          *C = X;\
          X->parent = T;\
          X->fd = X->bk = X;\
          break;\
        }\
        else {\
          CORRUPTION_ERROR_ACTION(M);\
          break;\
        }\
      }\
      else {\
        tchunkptr F = T->fd;\
        if (RTCHECK(ok_address(M, T) && ok_address(M, F))) {\
          T->fd = F->bk = X;\
          X->fd = F;\
          X->bk = T;\
          X->parent = 0;\
          break;\
        }\
        else {\
          CORRUPTION_ERROR_ACTION(M);\
          break;\
        }\
      }\
    }\
  }\
}

/*
  Unlink steps:

  1. If x is a chained node, unlink it from its same-sized fd/bk links
     and choose its bk node as its replacement.
  2. If x was the last node of its size, but not a leaf node, it must
     be replaced with a leaf node (not merely one with an open left or
     right), to make sure that lefts and rights of descendents
```



```
     correspond properly to bit masks.  We use the rightmost descendent
     of x.  We could use any other leaf, but this is easy to locate and
     tends to counteract removal of leftmosts elsewhere, and so keeps
     paths shorter than minimally guaranteed.  This doesn't loop much
     because on average a node in a tree is near the bottom.
  3. If x is the base of a chain (i.e., has parent links) relink
     x's parent and children to x's replacement (or null if none).
*/

#define unlink_large_chunk(M, X) {\
  tchunkptr XP = X->parent;\
  tchunkptr R;\
  if (X->bk != X) {\
    tchunkptr F = X->fd;\
    R = X->bk;\
    if (RTCHECK(ok_address(M, F))) {\
      F->bk = R;\
      R->fd = F;\
    }\
    else {\
      CORRUPTION_ERROR_ACTION(M);\
    }\
  }\
  else {\
    tchunkptr* RP;\
    if (((R = *(RP = &(X->child[1]))) != 0) ||\
        ((R = *(RP = &(X->child[0]))) != 0)) {\
      tchunkptr* CP;\
      while ((*(CP = &(R->child[1])) != 0) ||\
             (*(CP = &(R->child[0])) != 0)) {\
        R = *(RP = CP);\
      }\
      if (RTCHECK(ok_address(M, RP)))\
        *RP = 0;\
      else {\
        CORRUPTION_ERROR_ACTION(M);\
      }\
    }\
  }\
  if (XP != 0) {\
    tbinptr* H = treebin_at(M, X->index);\
    if (X == *H) {\
      if ((*H = R) == 0) \
        clear_treemap(M, X->index);\
    }\
    else if (RTCHECK(ok_address(M, XP))) {\
      if (XP->child[0] == X) \
        XP->child[0] = R;\
      else \
        XP->child[1] = R;\
    }\
    else\
```

```
      CORRUPTION_ERROR_ACTION(M);\
    if (R != 0) {\
      if (RTCHECK(ok_address(M, R))) {\
        tchunkptr C0, C1;\
        R->parent = XP;\
        if ((C0 = X->child[0]) != 0) {\
          if (RTCHECK(ok_address(M, C0))) {\
            R->child[0] = C0;\
            C0->parent = R;\
          }\
          else\
            CORRUPTION_ERROR_ACTION(M);\
        }\
        if ((C1 = X->child[1]) != 0) {\
          if (RTCHECK(ok_address(M, C1))) {\
            R->child[1] = C1;\
            C1->parent = R;\
          }\
          else\
            CORRUPTION_ERROR_ACTION(M);\
        }\
      }\
      else\
        CORRUPTION_ERROR_ACTION(M);\
    }\
  }\
}

/* Relays to large vs small bin operations */

#define insert_chunk(M, P, S)\
  if (is_small(S)) insert_small_chunk(M, P, S)\
  else { tchunkptr TP = (tchunkptr)(P); insert_large_chunk(M, TP, S); }

#define unlink_chunk(M, P, S)\
  if (is_small(S)) unlink_small_chunk(M, P, S)\
  else { tchunkptr TP = (tchunkptr)(P); unlink_large_chunk(M, TP); }


/* Relays to internal calls to malloc/free from realloc, memalign etc */

#if ONLY_MSPACES
#define internal_malloc(m, b) mspace_malloc(m, b)
#define internal_free(m, mem) mspace_free(m,mem);
#else /* ONLY_MSPACES */
#if MSPACES
#define internal_malloc(m, b)\
   (m == gm)? dlmalloc(b) : mspace_malloc(m, b)
#define internal_free(m, mem)\
   if (m == gm) dlfree(mem); else mspace_free(m,mem);
#else /* MSPACES */
#define internal_malloc(m, b) dlmalloc(b)
```



```
#define internal_free(m, mem) dlfree(mem)
#endif /* MSPACES */
#endif /* ONLY_MSPACES */
```

Chapter 21

Direct-mmapping chunks

```
/*
  Directly mmapped chunks are set up with an offset to the start of
  the mmapped region stored in the prev_foot field of the chunk. This
  allows reconstruction of the required argument to MUNMAP when freed,
  and also allows adjustment of the returned chunk to meet alignment
  requirements (especially in memalign).
*/

/* Malloc using mmap */
static void* mmap_alloc(mstate m, size_t nb) {
  size_t mmsize = mmap_align(nb + SIX_SIZE_T_SIZES + CHUNK_ALIGN_MASK);
  if (mmsize > nb) {     /* Check for wrap around 0 */
    char* mm = (char*)(CALL_DIRECT_MMAP(mmsize));
    if (mm != CMFAIL) {
      size_t offset = align_offset(chunk2mem(mm));
      size_t psize = mmsize - offset - MMAP_FOOT_PAD;
      mchunkptr p = (mchunkptr)(mm + offset);
      p->prev_foot = offset;
      p->head = psize;
      mark_inuse_foot(m, p, psize);
      chunk_plus_offset(p, psize)->head = FENCEPOST_HEAD;
      chunk_plus_offset(p, psize+SIZE_T_SIZE)->head = 0;

      if (m->least_addr == 0 || mm < m->least_addr)
        m->least_addr = mm;
      if ((m->footprint += mmsize) > m->max_footprint)
        m->max_footprint = m->footprint;
      assert(is_aligned(chunk2mem(p)));
      check_mmapped_chunk(m, p);
      return chunk2mem(p);
    }
  }
  return 0;
}

/* Realloc using mmap */
static mchunkptr mmap_resize(mstate m, mchunkptr oldp, size_t nb) {
  size_t oldsize = chunksize(oldp);
  if (is_small(nb)) /* Can't shrink mmap regions below small size */
```



```
    return 0;
  /* Keep old chunk if big enough but not too big */
  if (oldsize >= nb + SIZE_T_SIZE &&
      (oldsize - nb) <= (mparams.granularity << 1))
    return oldp;
  else {
    size_t offset = oldp->prev_foot;
    size_t oldmmsize = oldsize + offset + MMAP_FOOT_PAD;
    size_t newmmsize = mmap_align(nb + SIX_SIZE_T_SIZES + CHUNK_ALIGN_MASK);
    char* cp = (char*)CALL_MREMAP((char*)oldp - offset,
                                  oldmmsize, newmmsize, 1);
    if (cp != CMFAIL) {
      mchunkptr newp = (mchunkptr)(cp + offset);
      size_t psize = newmmsize - offset - MMAP_FOOT_PAD;
      newp->head = psize;
      mark_inuse_foot(m, newp, psize);
      chunk_plus_offset(newp, psize)->head = FENCEPOST_HEAD;
      chunk_plus_offset(newp, psize+SIZE_T_SIZE)->head = 0;

      if (cp < m->least_addr)
        m->least_addr = cp;
      if ((m->footprint += newmmsize - oldmmsize) > m->max_footprint)
        m->max_footprint = m->footprint;
      check_mmapped_chunk(m, newp);
      return newp;
    }
  }
  return 0;
}
```

Chapter 22

mspace management

```
/* Initialize top chunk and its size */
static void init_top(mstate m, mchunkptr p, size_t psize) {
  /* Ensure alignment */
  size_t offset = align_offset(chunk2mem(p));
  p = (mchunkptr)((char*)p + offset);
  psize -= offset;

  m->top = p;
  m->topsize = psize;
  p->head = psize | PINUSE_BIT;
  /* set size of fake trailing chunk holding overhead space only once */
  chunk_plus_offset(p, psize)->head = TOP_FOOT_SIZE;
  m->trim_check = mparams.trim_threshold; /* reset on each update */
}

/* Initialize bins for a new mstate that is otherwise zeroed out */
static void init_bins(mstate m) {
  /* Establish circular links for smallbins */
  bindex_t i;
  for (i = 0; i < NSMALLBINS; ++i) {
    sbinptr bin = smallbin_at(m,i);
    bin->fd = bin->bk = bin;
  }
}

#if PROCEED_ON_ERROR

/* default corruption action */
static void reset_on_error(mstate m) {
  int i;
  ++malloc_corruption_error_count;
  /* Reinitialize fields to forget about all memory */
  m->smallbins = m->treebins = 0;
  m->dvsize = m->topsize = 0;
  m->seg.base = 0;
  m->seg.size = 0;
  m->seg.next = 0;
  m->top = m->dv = 0;
  for (i = 0; i < NTREEBINS; ++i)
```



```
    *treebin_at(m, i) = 0;
  init_bins(m);
}
#endif /* PROCEED_ON_ERROR */

/* Allocate chunk and prepend remainder with chunk in successor base. */
static void* prepend_alloc(mstate m, char* newbase, char* oldbase,
                           size_t nb) {
  mchunkptr p = align_as_chunk(newbase);
  mchunkptr oldfirst = align_as_chunk(oldbase);
  size_t psize = (char*)oldfirst - (char*)p;
  mchunkptr q = chunk_plus_offset(p, nb);
  size_t qsize = psize - nb;
  set_size_and_pinuse_of_inuse_chunk(m, p, nb);

  assert((char*)oldfirst > (char*)q);
  assert(pinuse(oldfirst));
  assert(qsize >= MIN_CHUNK_SIZE);

  /* consolidate remainder with first chunk of old base */
  if (oldfirst == m->top) {
    size_t tsize = m->topsize += qsize;
    m->top = q;
    q->head = tsize | PINUSE_BIT;
    check_top_chunk(m, q);
  }
  else if (oldfirst == m->dv) {
    size_t dsize = m->dvsize += qsize;
    m->dv = q;
    set_size_and_pinuse_of_free_chunk(q, dsize);
  }
  else {
    if (!is_inuse(oldfirst)) {
      size_t nsize = chunksize(oldfirst);
      unlink_chunk(m, oldfirst, nsize);
      oldfirst = chunk_plus_offset(oldfirst, nsize);
      qsize += nsize;
    }
    set_free_with_pinuse(q, qsize, oldfirst);
    insert_chunk(m, q, qsize);
    check_free_chunk(m, q);
  }

  check_malloced_chunk(m, chunk2mem(p), nb);
  return chunk2mem(p);
}

/* Add a segment to hold a new noncontiguous region */
static void add_segment(mstate m, char* tbase, size_t tsize, flag_t mmapped) {
  /* Determine locations and sizes of segment, fenceposts, old top */
  char* old_top = (char*)m->top;
  msegmentptr oldsp = segment_holding(m, old_top);
```

```
  char* old_end = oldsp->base + oldsp->size;
  size_t ssize = pad_request(sizeof(struct malloc_segment));
  char* rawsp = old_end - (ssize + FOUR_SIZE_T_SIZES + CHUNK_ALIGN_MASK);
  size_t offset = align_offset(chunk2mem(rawsp));
  char* asp = rawsp + offset;
  char* csp = (asp < (old_top + MIN_CHUNK_SIZE))? old_top : asp;
  mchunkptr sp = (mchunkptr)csp;
  msegmentptr ss = (msegmentptr)(chunk2mem(sp));
  mchunkptr tnext = chunk_plus_offset(sp, ssize);
  mchunkptr p = tnext;
  int nfences = 0;

  /* reset top to new space */
  init_top(m, (mchunkptr)tbase, tsize - TOP_FOOT_SIZE);

  /* Set up segment record */
  assert(is_aligned(ss));
  set_size_and_pinuse_of_inuse_chunk(m, sp, ssize);
  *ss = m->seg; /* Push current record */
  m->seg.base = tbase;
  m->seg.size = tsize;
  m->seg.sflags = mmapped;
  m->seg.next = ss;

  /* Insert trailing fenceposts */
  for (;;) {
    mchunkptr nextp = chunk_plus_offset(p, SIZE_T_SIZE);
    p->head = FENCEPOST_HEAD;
    ++nfences;
    if ((char*)(&(nextp->head)) < old_end)
      p = nextp;
    else
      break;
  }
  assert(nfences >= 2);

  /* Insert the rest of old top into a bin as an ordinary free chunk */
  if (csp != old_top) {
    mchunkptr q = (mchunkptr)old_top;
    size_t psize = csp - old_top;
    mchunkptr tn = chunk_plus_offset(q, psize);
    set_free_with_pinuse(q, psize, tn);
    insert_chunk(m, q, psize);
  }

  check_top_chunk(m, m->top);
}
```

Chapter 23

System allocation and deallocation

23.1 System allocation

```
/* Get memory from system using MORECORE or MMAP */
static void* sys_alloc(mstate m, size_t nb) {
  char* tbase = CMFAIL;
  size_t tsize = 0;
  flag_t mmap_flag = 0;

  ensure_initialization();

  /* Directly map large chunks, but only if already initialized */
  if (use_mmap(m) && nb >= mparams.mmap_threshold && m->topsize != 0) {
    void* mem = mmap_alloc(m, nb);
    if (mem != 0)
      return mem;
  }

  /*
    Try getting memory in any of three ways (in most-preferred to
    least-preferred order):
    1. A call to MORECORE that can normally contiguously extend memory.
       (disabled if not MORECORE_CONTIGUOUS or not HAVE_MORECORE or
       or main space is mmapped or a previous contiguous call failed)
    2. A call to MMAP new space (disabled if not HAVE_MMAP).
       Note that under the default settings, if MORECORE is unable to
       fulfill a request, and HAVE_MMAP is true, then mmap is
       used as a noncontiguous system allocator. This is a useful backup
       strategy for systems with holes in address spaces -- in this case
       sbrk cannot contiguously expand the heap, but mmap may be able to
       find space.
    3. A call to MORECORE that cannot usually contiguously extend memory.
       (disabled if not HAVE_MORECORE)

   In all cases, we need to request enough bytes from system to ensure
   we can malloc nb bytes upon success, so pad with enough space for
   top_foot, plus alignment-pad to make sure we don't lose bytes if
   not on boundary, and round this up to a granularity unit.
  */
```

```
    if (MORECORE_CONTIGUOUS && !use_noncontiguous(m)) {
      char* br = CMFAIL;
      msegmentptr ss = (m->top == 0)? 0 : segment_holding(m, (char*)m->top);
      size_t asize = 0;
      ACQUIRE_MALLOC_GLOBAL_LOCK();

      if (ss == 0) {  /* First time through or recovery */
        char* base = (char*)CALL_MORECORE(0);
        if (base != CMFAIL) {
          asize = granularity_align(nb + SYS_ALLOC_PADDING);
          /* Adjust to end on a page boundary */
          if (!is_page_aligned(base))
            asize += (page_align((size_t)base) - (size_t)base);
          /* Can't call MORECORE if size is negative when treated as signed */
          if (asize < HALF_MAX_SIZE_T &&
              (br = (char*)(CALL_MORECORE(asize))) == base) {
            tbase = base;
            tsize = asize;
          }
        }
      }
      else {
        /* Subtract out existing available top space from MORECORE request. */
        asize = granularity_align(nb - m->topsize + SYS_ALLOC_PADDING);
        /* Use mem here only if it did continuously extend old space */
        if (asize < HALF_MAX_SIZE_T &&
            (br = (char*)(CALL_MORECORE(asize))) == ss->base+ss->size) {
          tbase = br;
          tsize = asize;
        }
      }

      if (tbase == CMFAIL) {    /* Cope with partial failure */
        if (br != CMFAIL) {    /* Try to use/extend the space we did get */
          if (asize < HALF_MAX_SIZE_T &&
              asize < nb + SYS_ALLOC_PADDING) {
            size_t esize = granularity_align(nb + SYS_ALLOC_PADDING - asize);
            if (esize < HALF_MAX_SIZE_T) {
              char* end = (char*)CALL_MORECORE(esize);
              if (end != CMFAIL)
                asize += esize;
              else {            /* Can't use; try to release */
                (void) CALL_MORECORE(-asize);
                br = CMFAIL;
              }
            }
          }
        }
        if (br != CMFAIL) {    /* Use the space we did get */
          tbase = br;
          tsize = asize;
```



```
        }
        else
          disable_contiguous(m); /* Don't try contiguous path in the future */
      }

      RELEASE_MALLOC_GLOBAL_LOCK();
    }

    if (HAVE_MMAP && tbase == CMFAIL) {  /* Try MMAP */
      size_t rsize = granularity_align(nb + SYS_ALLOC_PADDING);
      if (rsize > nb) { /* Fail if wraps around zero */
        char* mp = (char*)(CALL_MMAP(rsize));
        if (mp != CMFAIL) {
          tbase = mp;
          tsize = rsize;
          mmap_flag = USE_MMAP_BIT;
        }
      }
    }

    if (HAVE_MORECORE && tbase == CMFAIL) { /* Try noncontiguous MORECORE */
      size_t asize = granularity_align(nb + SYS_ALLOC_PADDING);
      if (asize < HALF_MAX_SIZE_T) {
        char* br = CMFAIL;
        char* end = CMFAIL;
        ACQUIRE_MALLOC_GLOBAL_LOCK();
        br = (char*)(CALL_MORECORE(asize));
        end = (char*)(CALL_MORECORE(0));
        RELEASE_MALLOC_GLOBAL_LOCK();
        if (br != CMFAIL && end != CMFAIL && br < end) {
          size_t ssize = end - br;
          if (ssize > nb + TOP_FOOT_SIZE) {
            tbase = br;
            tsize = ssize;
          }
        }
      }
    }

    if (tbase != CMFAIL) {

      if ((m->footprint += tsize) > m->max_footprint)
        m->max_footprint = m->footprint;

      if (!is_initialized(m)) { /* first-time initialization */
        if (m->least_addr == 0 || tbase < m->least_addr)
          m->least_addr = tbase;
        m->seg.base = tbase;
        m->seg.size = tsize;
        m->seg.sflags = mmap_flag;
        m->magic = mparams.magic;
        m->release_checks = MAX_RELEASE_CHECK_RATE;
```

```
      init_bins(m);
#if !ONLY_MSPACES
      if (is_global(m))
        init_top(m, (mchunkptr)tbase, tsize - TOP_FOOT_SIZE);
      else
#endif
      {
        /* Offset top by embedded malloc_state */
        mchunkptr mn = next_chunk(mem2chunk(m));
        init_top(m, mn, (size_t)((tbase + tsize) - (char*)mn) -TOP_FOOT_SIZE);
      }
    }

    else {
      /* Try to merge with an existing segment */
      msegmentptr sp = &m->seg;
      /* Only consider most recent segment if traversal suppressed */
      while (sp != 0 && tbase != sp->base + sp->size)
        sp = (NO_SEGMENT_TRAVERSAL) ? 0 : sp->next;
      if (sp != 0 &&
          !is_extern_segment(sp) &&
          (sp->sflags & USE_MMAP_BIT) == mmap_flag &&
          segment_holds(sp, m->top)) { /* append */
        sp->size += tsize;
        init_top(m, m->top, m->topsize + tsize);
      }
      else {
        if (tbase < m->least_addr)
          m->least_addr = tbase;
        sp = &m->seg;
        while (sp != 0 && sp->base != tbase + tsize)
          sp = (NO_SEGMENT_TRAVERSAL) ? 0 : sp->next;
        if (sp != 0 &&
            !is_extern_segment(sp) &&
            (sp->sflags & USE_MMAP_BIT) == mmap_flag) {
          char* oldbase = sp->base;
          sp->base = tbase;
          sp->size += tsize;
          return prepend_alloc(m, tbase, oldbase, nb);
        }
        else
          add_segment(m, tbase, tsize, mmap_flag);
      }
    }

    if (nb < m->topsize) { /* Allocate from new or extended top space */
      size_t rsize = m->topsize -= nb;
      mchunkptr p = m->top;
      mchunkptr r = m->top = chunk_plus_offset(p, nb);
      r->head = rsize | PINUSE_BIT;
      set_size_and_pinuse_of_inuse_chunk(m, p, nb);
      check_top_chunk(m, m->top);
```



```
      check_malloced_chunk(m, chunk2mem(p), nb);
      return chunk2mem(p);
    }
  }

  MALLOC_FAILURE_ACTION;
  return 0;
}
```

23.2 System deallocation

```
/* Unmap and unlink any mmapped segments that don't contain used chunks */
static size_t release_unused_segments(mstate m) {
  size_t released = 0;
  int nsegs = 0;
  msegmentptr pred = &m->seg;
  msegmentptr sp = pred->next;
  while (sp != 0) {
    char* base = sp->base;
    size_t size = sp->size;
    msegmentptr next = sp->next;
    ++nsegs;
    if (is_mmapped_segment(sp) && !is_extern_segment(sp)) {
      mchunkptr p = align_as_chunk(base);
      size_t psize = chunksize(p);
      /* Can unmap if first chunk holds entire segment and not pinned */
      if (!is_inuse(p) && (char*)p + psize >= base + size - TOP_FOOT_SIZE) {
        tchunkptr tp = (tchunkptr)p;
        assert(segment_holds(sp, (char*)sp));
        if (p == m->dv) {
          m->dv = 0;
          m->dvsize = 0;
        }
        else {
          unlink_large_chunk(m, tp);
        }
        if (CALL_MUNMAP(base, size) == 0) {
          released += size;
          m->footprint -= size;
          /* unlink obsoleted record */
          sp = pred;
          sp->next = next;
        }
        else { /* back out if cannot unmap */
          insert_large_chunk(m, tp, psize);
        }
      }
    }
    if (NO_SEGMENT_TRAVERSAL) /* scan only first segment */
      break;
    pred = sp;
```

```
    sp = next;
  }
  /* Reset check counter */
  m->release_checks = ((nsegs > MAX_RELEASE_CHECK_RATE)?
                       nsegs : MAX_RELEASE_CHECK_RATE);
  return released;
}

static int sys_trim(mstate m, size_t pad) {
  size_t released = 0;
  ensure_initialization();
  if (pad < MAX_REQUEST && is_initialized(m)) {
    pad += TOP_FOOT_SIZE; /* ensure enough room for segment overhead */

    if (m->topsize > pad) {
      /* Shrink top space in granularity-size units, keeping at least one */
      size_t unit = mparams.granularity;
      size_t extra = ((m->topsize - pad + (unit - SIZE_T_ONE)) / unit -
                      SIZE_T_ONE) * unit;
      msegmentptr sp = segment_holding(m, (char*)m->top);

      if (!is_extern_segment(sp)) {
        if (is_mmapped_segment(sp)) {
          if (HAVE_MMAP &&
              sp->size >= extra &&
              !has_segment_link(m, sp)) { /* can't shrink if pinned */
            size_t newsize = sp->size - extra;
            /* Prefer mremap, fall back to munmap */
            if ((CALL_MREMAP(sp->base, sp->size, newsize, 0) != MFAIL) ||
                (CALL_MUNMAP(sp->base + newsize, extra) == 0)) {
              released = extra;
            }
          }
        }
        else if (HAVE_MORECORE) {
          if (extra >= HALF_MAX_SIZE_T) /* Avoid wrapping negative */
            extra = (HALF_MAX_SIZE_T) + SIZE_T_ONE - unit;
          ACQUIRE_MALLOC_GLOBAL_LOCK();
          {
            /* Make sure end of memory is where we last set it. */
            char* old_br = (char*)(CALL_MORECORE(0));
            if (old_br == sp->base + sp->size) {
              char* rel_br = (char*)(CALL_MORECORE(-extra));
              char* new_br = (char*)(CALL_MORECORE(0));
              if (rel_br != CMFAIL && new_br < old_br)
                released = old_br - new_br;
            }
          }
          RELEASE_MALLOC_GLOBAL_LOCK();
        }
      }
```



```
      if (released != 0) {
        sp->size -= released;
        m->footprint -= released;
        init_top(m, m->top, m->topsize - released);
        check_top_chunk(m, m->top);
      }
    }

    /* Unmap any unused mmapped segments */
    if (HAVE_MMAP)
      released += release_unused_segments(m);

    /* On failure, disable autotrim to avoid repeated failed future calls */
    if (released == 0 && m->topsize > m->trim_check)
      m->trim_check = MAX_SIZE_T;
  }

  return (released != 0)? 1 : 0;
}
```

Chapter 24

Support for public routines

24.1 malloc support

```
/* allocate a large request from the best fitting chunk in a treebin */
static void* tmalloc_large(mstate m, size_t nb) {
  tchunkptr v = 0;
  size_t rsize = -nb; /* Unsigned negation */
  tchunkptr t;
  bindex_t idx;
  compute_tree_index(nb, idx);
  if ((t = *treebin_at(m, idx)) != 0) {
    /* Traverse tree for this bin looking for node with size == nb */
    size_t sizebits = nb << leftshift_for_tree_index(idx);
    tchunkptr rst = 0;  /* The deepest untaken right subtree */
    for (;;) {
      tchunkptr rt;
      size_t trem = chunksize(t) - nb;
      if (trem < rsize) {
        v = t;
        if ((rsize = trem) == 0)
          break;
      }
      rt = t->child[1];
      t = t->child[(sizebits >> (SIZE_T_BITSIZE-SIZE_T_ONE)) & 1];
      if (rt != 0 && rt != t)
        rst = rt;
      if (t == 0) {
        t = rst; /* set t to least subtree holding sizes > nb */
        break;
      }
      sizebits <<= 1;
    }
  }
  if (t == 0 && v == 0) { /* set t to root of next non-empty treebin */
    binmap_t leftbits = left_bits(idx2bit(idx)) & m->treemap;
    if (leftbits != 0) {
      bindex_t i;
      binmap_t leastbit = least_bit(leftbits);
      compute_bit2idx(leastbit, i);
```



```
      t = *treebin_at(m, i);
    }
  }

  while (t != 0) { /* find smallest of tree or subtree */
    size_t trem = chunksize(t) - nb;
    if (trem < rsize) {
      rsize = trem;
      v = t;
    }
    t = leftmost_child(t);
  }

  /*  If dv is a better fit, return 0 so malloc will use it */
  if (v != 0 && rsize < (size_t)(m->dvsize - nb)) {
    if (RTCHECK(ok_address(m, v))) { /* split */
      mchunkptr r = chunk_plus_offset(v, nb);
      assert(chunksize(v) == rsize + nb);
      if (RTCHECK(ok_next(v, r))) {
        unlink_large_chunk(m, v);
        if (rsize < MIN_CHUNK_SIZE)
          set_inuse_and_pinuse(m, v, (rsize + nb));
        else {
          set_size_and_pinuse_of_inuse_chunk(m, v, nb);
          set_size_and_pinuse_of_free_chunk(r, rsize);
          insert_chunk(m, r, rsize);
        }
        return chunk2mem(v);
      }
    }
    CORRUPTION_ERROR_ACTION(m);
  }
  return 0;
}

/* allocate a small request from the best fitting chunk in a treebin */
static void* tmalloc_small(mstate m, size_t nb) {
  tchunkptr t, v;
  size_t rsize;
  bindex_t i;
  binmap_t leastbit = least_bit(m->treemap);
  compute_bit2idx(leastbit, i);
  v = t = *treebin_at(m, i);
  rsize = chunksize(t) - nb;

  while ((t = leftmost_child(t)) != 0) {
    size_t trem = chunksize(t) - nb;
    if (trem < rsize) {
      rsize = trem;
      v = t;
    }
  }
```

```c
  if (RTCHECK(ok_address(m, v))) {
    mchunkptr r = chunk_plus_offset(v, nb);
    assert(chunksize(v) == rsize + nb);
    if (RTCHECK(ok_next(v, r))) {
      unlink_large_chunk(m, v);
      if (rsize < MIN_CHUNK_SIZE)
        set_inuse_and_pinuse(m, v, (rsize + nb));
      else {
        set_size_and_pinuse_of_inuse_chunk(m, v, nb);
        set_size_and_pinuse_of_free_chunk(r, rsize);
        replace_dv(m, r, rsize);
      }
      return chunk2mem(v);
    }
  }

  CORRUPTION_ERROR_ACTION(m);
  return 0;
}
```

24.2 realloc support

```c
static void* internal_realloc(mstate m, void* oldmem, size_t bytes) {
  if (bytes >= MAX_REQUEST) {
    MALLOC_FAILURE_ACTION;
    return 0;
  }
  if (!PREACTION(m)) {
    mchunkptr oldp = mem2chunk(oldmem);
    size_t oldsize = chunksize(oldp);
    mchunkptr next = chunk_plus_offset(oldp, oldsize);
    mchunkptr newp = 0;
    void* extra = 0;

    /* Try to either shrink or extend into top. Else malloc-copy-free */

    if (RTCHECK(ok_address(m, oldp) && ok_inuse(oldp) &&
                ok_next(oldp, next) && ok_pinuse(next))) {
      size_t nb = request2size(bytes);
      if (is_mmapped(oldp))
        newp = mmap_resize(m, oldp, nb);
      else if (oldsize >= nb) { /* already big enough */
        size_t rsize = oldsize - nb;
        newp = oldp;
        if (rsize >= MIN_CHUNK_SIZE) {
          mchunkptr remainder = chunk_plus_offset(newp, nb);
          set_inuse(m, newp, nb);
          set_inuse_and_pinuse(m, remainder, rsize);
          extra = chunk2mem(remainder);
        }
```



```c
      }
      else if (next == m->top && oldsize + m->topsize > nb) {
        /* Expand into top */
        size_t newsize = oldsize + m->topsize;
        size_t newtopsize = newsize - nb;
        mchunkptr newtop = chunk_plus_offset(oldp, nb);
        set_inuse(m, oldp, nb);
        newtop->head = newtopsize |PINUSE_BIT;
        m->top = newtop;
        m->topsize = newtopsize;
        newp = oldp;
      }
    }
    else {
      USAGE_ERROR_ACTION(m, oldmem);
      POSTACTION(m);
      return 0;
    }
#if DEBUG
    if (newp != 0) {
      check_inuse_chunk(m, newp); /* Check requires lock */
    }
#endif

    POSTACTION(m);

    if (newp != 0) {
      if (extra != 0) {
        internal_free(m, extra);
      }
      return chunk2mem(newp);
    }
    else {
      void* newmem = internal_malloc(m, bytes);
      if (newmem != 0) {
        size_t oc = oldsize - overhead_for(oldp);
        memcpy(newmem, oldmem, (oc < bytes)? oc : bytes);
        internal_free(m, oldmem);
      }
      return newmem;
    }
  }
  return 0;
}
```

24.3 memalign support

```c
static void* internal_memalign(mstate m, size_t alignment, size_t bytes) {
  if (alignment <= MALLOC_ALIGNMENT)    /* Can just use malloc */
    return internal_malloc(m, bytes);
```

```
    if (alignment <  MIN_CHUNK_SIZE) /* must be at least a minimum chunk size */
      alignment = MIN_CHUNK_SIZE;
    if ((alignment & (alignment-SIZE_T_ONE)) != 0) {/* Ensure a power of 2 */
      size_t a = MALLOC_ALIGNMENT << 1;
      while (a < alignment) a <<= 1;
      alignment = a;
    }

    if (bytes >= MAX_REQUEST - alignment) {
      if (m != 0)  { /* Test isn't needed but avoids compiler warning */
        MALLOC_FAILURE_ACTION;
      }
    }
    else {
      size_t nb = request2size(bytes);
      size_t req = nb + alignment + MIN_CHUNK_SIZE - CHUNK_OVERHEAD;
      char* mem = (char*)internal_malloc(m, req);
      if (mem != 0) {
        void* leader = 0;
        void* trailer = 0;
        mchunkptr p = mem2chunk(mem);

        if (PREACTION(m)) return 0;
        if ((((size_t)(mem)) % alignment) != 0) { /* misaligned */
          /*
            Find an aligned spot inside chunk.  Since we need to give
            back leading space in a chunk of at least MIN_CHUNK_SIZE, if
            the first calculation places us at a spot with less than
            MIN_CHUNK_SIZE leader, we can move to the next aligned spot.
            We've allocated enough total room so that this is always
            possible.
          */
          char* br = (char*)mem2chunk((size_t)(((size_t)(mem +
                                                         alignment -
                                                         SIZE_T_ONE)) &
                                               -alignment));
          char* pos = ((size_t)(br - (char*)(p)) >= MIN_CHUNK_SIZE)?
            br : br+alignment;
          mchunkptr newp = (mchunkptr)pos;
          size_t leadsize = pos - (char*)(p);
          size_t newsize = chunksize(p) - leadsize;

          if (is_mmapped(p)) { /* For mmapped chunks, just adjust offset */
            newp->prev_foot = p->prev_foot + leadsize;
            newp->head = newsize;
          }
          else { /* Otherwise, give back leader, use the rest */
            set_inuse(m, newp, newsize);
            set_inuse(m, p, leadsize);
            leader = chunk2mem(p);
          }
          p = newp;
```



```
        }

        /* Give back spare room at the end */
        if (!is_mmapped(p)) {
          size_t size = chunksize(p);
          if (size > nb + MIN_CHUNK_SIZE) {
            size_t remainder_size = size - nb;
            mchunkptr remainder = chunk_plus_offset(p, nb);
            set_inuse(m, p, nb);
            set_inuse(m, remainder, remainder_size);
            trailer = chunk2mem(remainder);
          }
        }

        assert (chunksize(p) >= nb);
        assert((((size_t)(chunk2mem(p))) % alignment) == 0);
        check_inuse_chunk(m, p);
        POSTACTION(m);
        if (leader != 0) {
          internal_free(m, leader);
        }
        if (trailer != 0) {
          internal_free(m, trailer);
        }
        return chunk2mem(p);
      }
    }
    return 0;
  }
```

24.4 comalloc/coalloc support

```
  static void** ialloc(mstate m,
                       size_t n_elements,
                       size_t* sizes,
                       int opts,
                       void* chunks[]) {
    /*
      This provides common support for independent_X routines, handling
      all of the combinations that can result.

      The opts arg has:
      bit 0 set if all elements are same size (using sizes[0])
      bit 1 set if elements should be zeroed
    */

    size_t    element_size;   /* chunksize of each element, if all same */
    size_t    contents_size;  /* total size of elements */
    size_t    array_size;     /* request size of pointer array */
    void*     mem;            /* malloced aggregate space */
    mchunkptr p;              /* corresponding chunk */
```

```
  size_t    remainder_size; /* remaining bytes while splitting */
  void**    marray;         /* either "chunks" or malloced ptr array */
  mchunkptr array_chunk;    /* chunk for malloced ptr array */
  flag_t    was_enabled;    /* to disable mmap */
  size_t    size;
  size_t    i;

  ensure_initialization();
  /* compute array length, if needed */
  if (chunks != 0) {
    if (n_elements == 0)
      return chunks; /* nothing to do */
    marray = chunks;
    array_size = 0;
  }
  else {
    /* if empty req, must still return chunk representing empty array */
    if (n_elements == 0)
      return (void**)internal_malloc(m, 0);
    marray = 0;
    array_size = request2size(n_elements * (sizeof(void*)));
  }

  /* compute total element size */
  if (opts & 0x1) { /* all-same-size */
    element_size = request2size(*sizes);
    contents_size = n_elements * element_size;
  }
  else { /* add up all the sizes */
    element_size = 0;
    contents_size = 0;
    for (i = 0; i != n_elements; ++i)
      contents_size += request2size(sizes[i]);
  }

  size = contents_size + array_size;

  /*
     Allocate the aggregate chunk.  First disable direct-mmapping so
     malloc won't use it, since we would not be able to later
     free/realloc space internal to a segregated mmap region.
  */
  was_enabled = use_mmap(m);
  disable_mmap(m);
  mem = internal_malloc(m, size - CHUNK_OVERHEAD);
  if (was_enabled)
    enable_mmap(m);
  if (mem == 0)
    return 0;

  if (PREACTION(m)) return 0;
  p = mem2chunk(mem);
```



```
  remainder_size = chunksize(p);

  assert(!is_mmapped(p));

  if (opts & 0x2) {       /* optionally clear the elements */
    memset((size_t*)mem, 0, remainder_size - SIZE_T_SIZE - array_size);
  }

  /* If not provided, allocate the pointer array as final part of chunk */
  if (marray == 0) {
    size_t  array_chunk_size;
    array_chunk = chunk_plus_offset(p, contents_size);
    array_chunk_size = remainder_size - contents_size;
    marray = (void**) (chunk2mem(array_chunk));
    set_size_and_pinuse_of_inuse_chunk(m, array_chunk, array_chunk_size);
    remainder_size = contents_size;
  }

  /* split out elements */
  for (i = 0; ; ++i) {
    marray[i] = chunk2mem(p);
    if (i != n_elements-1) {
      if (element_size != 0)
        size = element_size;
      else
        size = request2size(sizes[i]);
      remainder_size -= size;
      set_size_and_pinuse_of_inuse_chunk(m, p, size);
      p = chunk_plus_offset(p, size);
    }
    else { /* the final element absorbs any overallocation slop */
      set_size_and_pinuse_of_inuse_chunk(m, p, remainder_size);
      break;
    }
  }

#if DEBUG
  if (marray != chunks) {
    /* final element must have exactly exhausted chunk */
    if (element_size != 0) {
      assert(remainder_size == element_size);
    }
    else {
      assert(remainder_size == request2size(sizes[i]));
    }
    check_inuse_chunk(m, mem2chunk(marray));
  }
  for (i = 0; i != n_elements; ++i)
    check_inuse_chunk(m, mem2chunk(marray[i]));

#endif /* DEBUG */
```

```
    POSTACTION(m);
    return marray;
}
```

Chapter 25

Public routines

```
#if !ONLY_MSPACES

void* dlmalloc(size_t bytes) {
  /*
     Basic algorithm:
     If a small request (< 256 bytes minus per-chunk overhead):
       1. If one exists, use a remainderless chunk in associated smallbin.
          (Remainderless means that there are too few excess bytes to
          represent as a chunk.)
       2. If it is big enough, use the dv chunk, which is normally the
          chunk adjacent to the one used for the most recent small request.
       3. If one exists, split the smallest available chunk in a bin,
          saving remainder in dv.
       4. If it is big enough, use the top chunk.
       5. If available, get memory from system and use it
     Otherwise, for a large request:
       1. Find the smallest available binned chunk that fits, and use it
          if it is better fitting than dv chunk, splitting if necessary.
       2. If better fitting than any binned chunk, use the dv chunk.
       3. If it is big enough, use the top chunk.
       4. If request size >= mmap threshold, try to directly mmap this chunk.
       5. If available, get memory from system and use it

     The ugly goto's here ensure that postaction occurs along all paths.
  */

#if USE_LOCKS
  ensure_initialization(); /* initialize in sys_alloc if not using locks */
#endif

  if (!PREACTION(gm)) {
    void* mem;
    size_t nb;
    if (bytes <= MAX_SMALL_REQUEST) {
      bindex_t idx;
      binmap_t smallbits;
      nb = (bytes < MIN_REQUEST)? MIN_CHUNK_SIZE : pad_request(bytes);
      idx = small_index(nb);
      smallbits = gm->smallmap >> idx;
```

```
      if ((smallbits & 0x3U) != 0) { /* Remainderless fit to a smallbin. */
        mchunkptr b, p;
        idx += ~smallbits & 1;       /* Uses next bin if idx empty */
        b = smallbin_at(gm, idx);
        p = b->fd;
        assert(chunksize(p) == small_index2size(idx));
        unlink_first_small_chunk(gm, b, p, idx);
        set_inuse_and_pinuse(gm, p, small_index2size(idx));
        mem = chunk2mem(p);
        check_malloced_chunk(gm, mem, nb);
        goto postaction;
      }

      else if (nb > gm->dvsize) {
        if (smallbits != 0) { /* Use chunk in next nonempty smallbin */
          mchunkptr b, p, r;
          size_t rsize;
          bindex_t i;
          binmap_t leftbits = (smallbits << idx) & left_bits(idx2bit(idx));
          binmap_t leastbit = least_bit(leftbits);
          compute_bit2idx(leastbit, i);
          b = smallbin_at(gm, i);
          p = b->fd;
          assert(chunksize(p) == small_index2size(i));
          unlink_first_small_chunk(gm, b, p, i);
          rsize = small_index2size(i) - nb;
          /* Fit here cannot be remainderless if 4byte sizes */
          if (SIZE_T_SIZE != 4 && rsize < MIN_CHUNK_SIZE)
            set_inuse_and_pinuse(gm, p, small_index2size(i));
          else {
            set_size_and_pinuse_of_inuse_chunk(gm, p, nb);
            r = chunk_plus_offset(p, nb);
            set_size_and_pinuse_of_free_chunk(r, rsize);
            replace_dv(gm, r, rsize);
          }
          mem = chunk2mem(p);
          check_malloced_chunk(gm, mem, nb);
          goto postaction;
        }

        else if (gm->treemap != 0 && (mem = tmalloc_small(gm, nb)) != 0) {
          check_malloced_chunk(gm, mem, nb);
          goto postaction;
        }
      }
    }
    else if (bytes >= MAX_REQUEST)
      nb = MAX_SIZE_T; /* Too big to allocate. Force failure (in sys alloc) */
    else {
      nb = pad_request(bytes);
      if (gm->treemap != 0 && (mem = tmalloc_large(gm, nb)) != 0) {
        check_malloced_chunk(gm, mem, nb);
        goto postaction;
      }
    }

    if (nb <= gm->dvsize) {
      size_t rsize = gm->dvsize - nb;
      mchunkptr p = gm->dv;
      if (rsize >= MIN_CHUNK_SIZE) { /* split dv */
        mchunkptr r = gm->dv = chunk_plus_offset(p, nb);
        gm->dvsize = rsize;
        set_size_and_pinuse_of_free_chunk(r, rsize);
        set_size_and_pinuse_of_inuse_chunk(gm, p, nb);
      }
      else { /* exhaust dv */
        size_t dvs = gm->dvsize;
        gm->dvsize = 0;
        gm->dv = 0;
        set_inuse_and_pinuse(gm, p, dvs);
      }
      mem = chunk2mem(p);
      check_malloced_chunk(gm, mem, nb);
      goto postaction;
    }

    else if (nb < gm->topsize) { /* Split top */
      size_t rsize = gm->topsize -= nb;
      mchunkptr p = gm->top;
      mchunkptr r = gm->top = chunk_plus_offset(p, nb);
      r->head = rsize | PINUSE_BIT;
      set_size_and_pinuse_of_inuse_chunk(gm, p, nb);
      mem = chunk2mem(p);
      check_top_chunk(gm, gm->top);
      check_malloced_chunk(gm, mem, nb);
      goto postaction;
    }

    mem = sys_alloc(gm, nb);

  postaction:
    POSTACTION(gm);
    return mem;
  }

  return 0;
}

void dlfree(void* mem) {
  /*
     Consolidate freed chunks with preceeding or succeeding bordering
     free chunks, if they exist, and then place in a bin.  Intermixed
     with special cases for top, dv, mmapped chunks, and usage errors.
```

```
  */

  if (mem != 0) {
    mchunkptr p  = mem2chunk(mem);
#if FOOTERS
    mstate fm = get_mstate_for(p);
    if (!ok_magic(fm)) {
      USAGE_ERROR_ACTION(fm, p);
      return;
    }
#else /* FOOTERS */
#define fm gm
#endif /* FOOTERS */
    if (!PREACTION(fm)) {
      check_inuse_chunk(fm, p);
      if (RTCHECK(ok_address(fm, p) && ok_inuse(p))) {
        size_t psize = chunksize(p);
        mchunkptr next = chunk_plus_offset(p, psize);
        if (!pinuse(p)) {
          size_t prevsize = p->prev_foot;
          if (is_mmapped(p)) {
            psize += prevsize + MMAP_FOOT_PAD;
            if (CALL_MUNMAP((char*)p - prevsize, psize) == 0)
              fm->footprint -= psize;
            goto postaction;
          }
          else {
            mchunkptr prev = chunk_minus_offset(p, prevsize);
            psize += prevsize;
            p = prev;
            if (RTCHECK(ok_address(fm, prev))) { /* consolidate backward */
              if (p != fm->dv) {
                unlink_chunk(fm, p, prevsize);
              }
              else if ((next->head & INUSE_BITS) == INUSE_BITS) {
                fm->dvsize = psize;
                set_free_with_pinuse(p, psize, next);
                goto postaction;
              }
            }
            else
              goto erroraction;
          }
        }

        if (RTCHECK(ok_next(p, next) && ok_pinuse(next))) {
          if (!cinuse(next)) {  /* consolidate forward */
            if (next == fm->top) {
              size_t tsize = fm->topsize += psize;
              fm->top = p;
              p->head = tsize | PINUSE_BIT;
              if (p == fm->dv) {
                fm->dv = 0;
                fm->dvsize = 0;
              }
              if (should_trim(fm, tsize))
                sys_trim(fm, 0);
              goto postaction;
            }
            else if (next == fm->dv) {
              size_t dsize = fm->dvsize += psize;
              fm->dv = p;
              set_size_and_pinuse_of_free_chunk(p, dsize);
              goto postaction;
            }
            else {
              size_t nsize = chunksize(next);
              psize += nsize;
              unlink_chunk(fm, next, nsize);
              set_size_and_pinuse_of_free_chunk(p, psize);
              if (p == fm->dv) {
                fm->dvsize = psize;
                goto postaction;
              }
            }
          }
          else
            set_free_with_pinuse(p, psize, next);

          if (is_small(psize)) {
            insert_small_chunk(fm, p, psize);
            check_free_chunk(fm, p);
          }
          else {
            tchunkptr tp = (tchunkptr)p;
            insert_large_chunk(fm, tp, psize);
            check_free_chunk(fm, p);
            if (--fm->release_checks == 0)
              release_unused_segments(fm);
          }
          goto postaction;
        }
      }
    erroraction:
      USAGE_ERROR_ACTION(fm, p);
    postaction:
      POSTACTION(fm);
    }
  }
#if !FOOTERS
#undef fm
#endif /* FOOTERS */
}
```

```
void* dlcalloc(size_t n_elements, size_t elem_size) {
  void* mem;
  size_t req = 0;
  if (n_elements != 0) {
    req = n_elements * elem_size;
    if (((n_elements | elem_size) & ~(size_t)0xffff) &&
        (req / n_elements != elem_size))
      req = MAX_SIZE_T; /* force downstream failure on overflow */
  }
  mem = dlmalloc(req);
  if (mem != 0 && calloc_must_clear(mem2chunk(mem)))
    memset(mem, 0, req);
  return mem;
}

void* dlrealloc(void* oldmem, size_t bytes) {
  if (oldmem == 0)
    return dlmalloc(bytes);
#ifdef REALLOC_ZERO_BYTES_FREES
  if (bytes == 0) {
    dlfree(oldmem);
    return 0;
  }
#endif /* REALLOC_ZERO_BYTES_FREES */
  else {
#if ! FOOTERS
    mstate m = gm;
#else /* FOOTERS */
    mstate m = get_mstate_for(mem2chunk(oldmem));
    if (!ok_magic(m)) {
      USAGE_ERROR_ACTION(m, oldmem);
      return 0;
    }
#endif /* FOOTERS */
    return internal_realloc(m, oldmem, bytes);
  }
}

void* dlmemalign(size_t alignment, size_t bytes) {
  return internal_memalign(gm, alignment, bytes);
}

void** dlindependent_calloc(size_t n_elements, size_t elem_size,
                                 void* chunks[]) {
  size_t sz = elem_size; /* serves as 1-element array */
  return ialloc(gm, n_elements, &sz, 3, chunks);
}

void** dlindependent_comalloc(size_t n_elements, size_t sizes[],
                                   void* chunks[]) {
  return ialloc(gm, n_elements, sizes, 0, chunks);
}

void* dlvalloc(size_t bytes) {
  size_t pagesz;
  ensure_initialization();
  pagesz = mparams.page_size;
  return dlmemalign(pagesz, bytes);
}

void* dlpvalloc(size_t bytes) {
  size_t pagesz;
  ensure_initialization();
  pagesz = mparams.page_size;
  return dlmemalign(pagesz, (bytes + pagesz - SIZE_T_ONE) & ~(pagesz - SIZE_T_ONE));
}

int dlmalloc_trim(size_t pad) {
  int result = 0;
  ensure_initialization();
  if (!PREACTION(gm)) {
    result = sys_trim(gm, pad);
    POSTACTION(gm);
  }
  return result;
}

size_t dlmalloc_footprint(void) {
  return gm->footprint;
}

size_t dlmalloc_max_footprint(void) {
  return gm->max_footprint;
}

#if !NO_MALLINFO
struct mallinfo dlmallinfo(void) {
  return internal_mallinfo(gm);
}
#endif /* NO_MALLINFO */

void dlmalloc_stats() {
  internal_malloc_stats(gm);
}

int dlmallopt(int param_number, int value) {
  return change_mparam(param_number, value);
}

#endif /* !ONLY_MSPACES */

size_t dlmalloc_usable_size(void* mem) {
  if (mem != 0) {
    mchunkptr p = mem2chunk(mem);
```

```
    if (is_inuse(p))
      return chunksize(p) - overhead_for(p);
  }
  return 0;
}
```

Chapter 26

User mspaces

```
#if MSPACES

static mstate init_user_mstate(char* tbase, size_t tsize) {
  size_t msize = pad_request(sizeof(struct malloc_state));
  mchunkptr mn;
  mchunkptr msp = align_as_chunk(tbase);
  mstate m = (mstate)(chunk2mem(msp));
  memset(m, 0, msize);
  INITIAL_LOCK(&m->mutex);
  msp->head = (msize|INUSE_BITS);
  m->seg.base = m->least_addr = tbase;
  m->seg.size = m->footprint = m->max_footprint = tsize;
  m->magic = mparams.magic;
  m->release_checks = MAX_RELEASE_CHECK_RATE;
  m->mflags = mparams.default_mflags;
  m->extp = 0;
  m->exts = 0;
  disable_contiguous(m);
  init_bins(m);
  mn = next_chunk(mem2chunk(m));
  init_top(m, mn, (size_t)((tbase + tsize) - (char*)mn) - TOP_FOOT_SIZE);
  check_top_chunk(m, m->top);
  return m;
}

mspace create_mspace(size_t capacity, int locked) {
  mstate m = 0;
  size_t msize;
  ensure_initialization();
  msize = pad_request(sizeof(struct malloc_state));
  if (capacity < (size_t) -(msize + TOP_FOOT_SIZE + mparams.page_size)) {
    size_t rs = ((capacity == 0)? mparams.granularity :
                 (capacity + TOP_FOOT_SIZE + msize));
    size_t tsize = granularity_align(rs);
    char* tbase = (char*)(CALL_MMAP(tsize));
    if (tbase != CMFAIL) {
      m = init_user_mstate(tbase, tsize);
      m->seg.sflags = USE_MMAP_BIT;
      set_lock(m, locked);
```

```
    }
  }
  return (mspace)m;
}

mspace create_mspace_with_base(void* base, size_t capacity, int locked) {
  mstate m = 0;
  size_t msize;
  ensure_initialization();
  msize = pad_request(sizeof(struct malloc_state));
  if (capacity > msize + TOP_FOOT_SIZE &&
      capacity < (size_t) -(msize + TOP_FOOT_SIZE + mparams.page_size)) {
    m = init_user_mstate((char*)base, capacity);
    m->seg.sflags = EXTERN_BIT;
    set_lock(m, locked);
  }
  return (mspace)m;
}

int mspace_track_large_chunks(mspace msp, int enable) {
  int ret = 0;
  mstate ms = (mstate)msp;
  if (!PREACTION(ms)) {
    if (!use_mmap(ms))
      ret = 1;
    if (!enable)
      enable_mmap(ms);
    else
      disable_mmap(ms);
    POSTACTION(ms);
  }
  return ret;
}

size_t destroy_mspace(mspace msp) {
  size_t freed = 0;
  mstate ms = (mstate)msp;
  if (ok_magic(ms)) {
    msegmentptr sp = &ms->seg;
    while (sp != 0) {
      char* base = sp->base;
      size_t size = sp->size;
      flag_t flag = sp->sflags;
      sp = sp->next;
      if ((flag & USE_MMAP_BIT) && !(flag & EXTERN_BIT) &&
          CALL_MUNMAP(base, size) == 0)
        freed += size;
    }
  }
  else {
    USAGE_ERROR_ACTION(ms,ms);
  }
```



```
  return freed;
}

/*
  mspace versions of routines are near-clones of the global
  versions. This is not so nice but better than the alternatives.
*/

void* mspace_malloc(mspace msp, size_t bytes) {
  mstate ms = (mstate)msp;
  if (!ok_magic(ms)) {
    USAGE_ERROR_ACTION(ms,ms);
    return 0;
  }
  if (!PREACTION(ms)) {
    void* mem;
    size_t nb;
    if (bytes <= MAX_SMALL_REQUEST) {
      bindex_t idx;
      binmap_t smallbits;
      nb = (bytes < MIN_REQUEST)? MIN_CHUNK_SIZE : pad_request(bytes);
      idx = small_index(nb);
      smallbits = ms->smallmap >> idx;

      if ((smallbits & 0x3U) != 0) { /* Remainderless fit to a smallbin. */
        mchunkptr b, p;
        idx += ~smallbits & 1;       /* Uses next bin if idx empty */
        b = smallbin_at(ms, idx);
        p = b->fd;
        assert(chunksize(p) == small_index2size(idx));
        unlink_first_small_chunk(ms, b, p, idx);
        set_inuse_and_pinuse(ms, p, small_index2size(idx));
        mem = chunk2mem(p);
        check_malloced_chunk(ms, mem, nb);
        goto postaction;
      }

      else if (nb > ms->dvsize) {
        if (smallbits != 0) { /* Use chunk in next nonempty smallbin */
          mchunkptr b, p, r;
          size_t rsize;
          bindex_t i;
          binmap_t leftbits = (smallbits << idx) & left_bits(idx2bit(idx));
          binmap_t leastbit = least_bit(leftbits);
          compute_bit2idx(leastbit, i);
          b = smallbin_at(ms, i);
          p = b->fd;
          assert(chunksize(p) == small_index2size(i));
          unlink_first_small_chunk(ms, b, p, i);
          rsize = small_index2size(i) - nb;
          /* Fit here cannot be remainderless if 4byte sizes */
          if (SIZE_T_SIZE != 4 && rsize < MIN_CHUNK_SIZE)
```

```
            set_inuse_and_pinuse(ms, p, small_index2size(i));
          else {
            set_size_and_pinuse_of_inuse_chunk(ms, p, nb);
            r = chunk_plus_offset(p, nb);
            set_size_and_pinuse_of_free_chunk(r, rsize);
            replace_dv(ms, r, rsize);
          }
          mem = chunk2mem(p);
          check_malloced_chunk(ms, mem, nb);
          goto postaction;
        }

        else if (ms->treemap != 0 && (mem = tmalloc_small(ms, nb)) != 0) {
          check_malloced_chunk(ms, mem, nb);
          goto postaction;
        }
      }
    }
    else if (bytes >= MAX_REQUEST)
      nb = MAX_SIZE_T; /* Too big to allocate. Force failure (in sys alloc) */
    else {
      nb = pad_request(bytes);
      if (ms->treemap != 0 && (mem = tmalloc_large(ms, nb)) != 0) {
        check_malloced_chunk(ms, mem, nb);
        goto postaction;
      }
    }

    if (nb <= ms->dvsize) {
      size_t rsize = ms->dvsize - nb;
      mchunkptr p = ms->dv;
      if (rsize >= MIN_CHUNK_SIZE) { /* split dv */
        mchunkptr r = ms->dv = chunk_plus_offset(p, nb);
        ms->dvsize = rsize;
        set_size_and_pinuse_of_free_chunk(r, rsize);
        set_size_and_pinuse_of_inuse_chunk(ms, p, nb);
      }
      else { /* exhaust dv */
        size_t dvs = ms->dvsize;
        ms->dvsize = 0;
        ms->dv = 0;
        set_inuse_and_pinuse(ms, p, dvs);
      }
      mem = chunk2mem(p);
      check_malloced_chunk(ms, mem, nb);
      goto postaction;
    }

    else if (nb < ms->topsize) { /* Split top */
      size_t rsize = ms->topsize -= nb;
      mchunkptr p = ms->top;
      mchunkptr r = ms->top = chunk_plus_offset(p, nb);
```



```
      r->head = rsize | PINUSE_BIT;
      set_size_and_pinuse_of_inuse_chunk(ms, p, nb);
      mem = chunk2mem(p);
      check_top_chunk(ms, ms->top);
      check_malloced_chunk(ms, mem, nb);
      goto postaction;
    }

    mem = sys_alloc(ms, nb);

  postaction:
    POSTACTION(ms);
    return mem;
  }

  return 0;
}

void mspace_free(mspace msp, void* mem) {
  if (mem != 0) {
    mchunkptr p  = mem2chunk(mem);
#if FOOTERS
    mstate fm = get_mstate_for(p);
    msp = msp; /* placate people compiling -Wunused */
#else /* FOOTERS */
    mstate fm = (mstate)msp;
#endif /* FOOTERS */
    if (!ok_magic(fm)) {
      USAGE_ERROR_ACTION(fm, p);
      return;
    }
    if (!PREACTION(fm)) {
      check_inuse_chunk(fm, p);
      if (RTCHECK(ok_address(fm, p) && ok_inuse(p))) {
        size_t psize = chunksize(p);
        mchunkptr next = chunk_plus_offset(p, psize);
        if (!pinuse(p)) {
          size_t prevsize = p->prev_foot;
          if (is_mmapped(p)) {
            psize += prevsize + MMAP_FOOT_PAD;
            if (CALL_MUNMAP((char*)p - prevsize, psize) == 0)
              fm->footprint -= psize;
            goto postaction;
          }
          else {
            mchunkptr prev = chunk_minus_offset(p, prevsize);
            psize += prevsize;
            p = prev;
            if (RTCHECK(ok_address(fm, prev))) { /* consolidate backward */
              if (p != fm->dv) {
                unlink_chunk(fm, p, prevsize);
              }
```

```
            else if ((next->head & INUSE_BITS) == INUSE_BITS) {
              fm->dvsize = psize;
              set_free_with_pinuse(p, psize, next);
              goto postaction;
            }
          }
          else
            goto erroraction;
        }
      }

      if (RTCHECK(ok_next(p, next) && ok_pinuse(next))) {
        if (!cinuse(next)) {  /* consolidate forward */
          if (next == fm->top) {
            size_t tsize = fm->topsize += psize;
            fm->top = p;
            p->head = tsize | PINUSE_BIT;
            if (p == fm->dv) {
              fm->dv = 0;
              fm->dvsize = 0;
            }
            if (should_trim(fm, tsize))
              sys_trim(fm, 0);
            goto postaction;
          }
          else if (next == fm->dv) {
            size_t dsize = fm->dvsize += psize;
            fm->dv = p;
            set_size_and_pinuse_of_free_chunk(p, dsize);
            goto postaction;
          }
          else {
            size_t nsize = chunksize(next);
            psize += nsize;
            unlink_chunk(fm, next, nsize);
            set_size_and_pinuse_of_free_chunk(p, psize);
            if (p == fm->dv) {
              fm->dvsize = psize;
              goto postaction;
            }
          }
        }
        else
          set_free_with_pinuse(p, psize, next);

        if (is_small(psize)) {
          insert_small_chunk(fm, p, psize);
          check_free_chunk(fm, p);
        }
        else {
          tchunkptr tp = (tchunkptr)p;
          insert_large_chunk(fm, tp, psize);
            check_free_chunk(fm, p);
            if (--fm->release_checks == 0)
              release_unused_segments(fm);
          }
          goto postaction;
        }
      }
    erroraction:
      USAGE_ERROR_ACTION(fm, p);
    postaction:
      POSTACTION(fm);
    }
  }
}

void* mspace_calloc(mspace msp, size_t n_elements, size_t elem_size) {
  void* mem;
  size_t req = 0;
  mstate ms = (mstate)msp;
  if (!ok_magic(ms)) {
    USAGE_ERROR_ACTION(ms,ms);
    return 0;
  }
  if (n_elements != 0) {
    req = n_elements * elem_size;
    if (((n_elements | elem_size) & ~(size_t)0xffff) &&
        (req / n_elements != elem_size))
      req = MAX_SIZE_T; /* force downstream failure on overflow */
  }
  mem = internal_malloc(ms, req);
  if (mem != 0 && calloc_must_clear(mem2chunk(mem)))
    memset(mem, 0, req);
  return mem;
}

void* mspace_realloc(mspace msp, void* oldmem, size_t bytes) {
  if (oldmem == 0)
    return mspace_malloc(msp, bytes);
#ifdef REALLOC_ZERO_BYTES_FREES
  if (bytes == 0) {
    mspace_free(msp, oldmem);
    return 0;
  }
#endif /* REALLOC_ZERO_BYTES_FREES */
  else {
#if FOOTERS
    mchunkptr p  = mem2chunk(oldmem);
    mstate ms = get_mstate_for(p);
#else /* FOOTERS */
    mstate ms = (mstate)msp;
#endif /* FOOTERS */
    if (!ok_magic(ms)) {
```

```
      USAGE_ERROR_ACTION(ms,ms);
      return 0;
    }
    return internal_realloc(ms, oldmem, bytes);
  }
}

void* mspace_memalign(mspace msp, size_t alignment, size_t bytes) {
  mstate ms = (mstate)msp;
  if (!ok_magic(ms)) {
    USAGE_ERROR_ACTION(ms,ms);
    return 0;
  }
  return internal_memalign(ms, alignment, bytes);
}

void** mspace_independent_calloc(mspace msp, size_t n_elements,
                                 size_t elem_size, void* chunks[]) {
  size_t sz = elem_size; /* serves as 1-element array */
  mstate ms = (mstate)msp;
  if (!ok_magic(ms)) {
    USAGE_ERROR_ACTION(ms,ms);
    return 0;
  }
  return ialloc(ms, n_elements, &sz, 3, chunks);
}

void** mspace_independent_comalloc(mspace msp, size_t n_elements,
                                   size_t sizes[], void* chunks[]) {
  mstate ms = (mstate)msp;
  if (!ok_magic(ms)) {
    USAGE_ERROR_ACTION(ms,ms);
    return 0;
  }
  return ialloc(ms, n_elements, sizes, 0, chunks);
}

int mspace_trim(mspace msp, size_t pad) {
  int result = 0;
  mstate ms = (mstate)msp;
  if (ok_magic(ms)) {
    if (!PREACTION(ms)) {
      result = sys_trim(ms, pad);
      POSTACTION(ms);
    }
  }
  else {
    USAGE_ERROR_ACTION(ms,ms);
  }
  return result;
}
```

```
void mspace_malloc_stats(mspace msp) {
  mstate ms = (mstate)msp;
  if (ok_magic(ms)) {
    internal_malloc_stats(ms);
  }
  else {
    USAGE_ERROR_ACTION(ms,ms);
  }
}

size_t mspace_footprint(mspace msp) {
  size_t result = 0;
  mstate ms = (mstate)msp;
  if (ok_magic(ms)) {
    result = ms->footprint;
  }
  else {
    USAGE_ERROR_ACTION(ms,ms);
  }
  return result;
}


size_t mspace_max_footprint(mspace msp) {
  size_t result = 0;
  mstate ms = (mstate)msp;
  if (ok_magic(ms)) {
    result = ms->max_footprint;
  }
  else {
    USAGE_ERROR_ACTION(ms,ms);
  }
  return result;
}


#if !NO_MALLINFO
struct mallinfo mspace_mallinfo(mspace msp) {
  mstate ms = (mstate)msp;
  if (!ok_magic(ms)) {
    USAGE_ERROR_ACTION(ms,ms);
  }
  return internal_mallinfo(ms);
}
#endif /* NO_MALLINFO */

size_t mspace_usable_size(void* mem) {
  if (mem != 0) {
    mchunkptr p = mem2chunk(mem);
    if (is_inuse(p))
      return chunksize(p) - overhead_for(p);
  }
```

```
  return 0;
}

int mspace_mallopt(int param_number, int value) {
  return change_mparam(param_number, value);
}

#endif /* MSPACES */
```

Chapter 27

Postscript

27.1 Alternative MORECORE functions

```
/*
  Guidelines for creating a custom version of MORECORE:

  * For best performance, MORECORE should allocate in multiples of pagesize.
  * MORECORE may allocate more memory than requested. (Or even less,
      but this will usually result in a malloc failure.)
  * MORECORE must not allocate memory when given argument zero, but
      instead return one past the end address of memory from previous
      nonzero call.
  * For best performance, consecutive calls to MORECORE with positive
      arguments should return increasing addresses, indicating that
      space has been contiguously extended.
  * Even though consecutive calls to MORECORE need not return contiguous
      addresses, it must be OK for malloc'ed chunks to span multiple
      regions in those cases where they do happen to be contiguous.
  * MORECORE need not handle negative arguments -- it may instead
      just return MFAIL when given negative arguments.
      Negative arguments are always multiples of pagesize. MORECORE
      must not misinterpret negative args as large positive unsigned
      args. You can suppress all such calls from even occurring by defining
      MORECORE_CANNOT_TRIM,

  As an example alternative MORECORE, here is a custom allocator
  kindly contributed for pre-OSX macOS.  It uses virtually but not
  necessarily physically contiguous non-paged memory (locked in,
  present and won't get swapped out).  You can use it by uncommenting
  this section, adding some #includes, and setting up the appropriate
  defines above:

      #define MORECORE osMoreCore

  There is also a shutdown routine that should somehow be called for
  cleanup upon program exit.

  #define MAX_POOL_ENTRIES 100
  #define MINIMUM_MORECORE_SIZE  (64 * 1024U)
```

```
  static int next_os_pool;
  void *our_os_pools[MAX_POOL_ENTRIES];

  void *osMoreCore(int size)
  {
    void *ptr = 0;
    static void *sbrk_top = 0;

    if (size > 0)
    {
      if (size < MINIMUM_MORECORE_SIZE)
         size = MINIMUM_MORECORE_SIZE;
      if (CurrentExecutionLevel() == kTaskLevel)
         ptr = PoolAllocateResident(size + RM_PAGE_SIZE, 0);
      if (ptr == 0)
      {
        return (void *) MFAIL;
      }
      // save ptrs so they can be freed during cleanup
      our_os_pools[next_os_pool] = ptr;
      next_os_pool++;
      ptr = (void *) ((((size_t) ptr) + RM_PAGE_MASK) & ~RM_PAGE_MASK);
      sbrk_top = (char *) ptr + size;
      return ptr;
    }
    else if (size < 0)
    {
      // we don't currently support shrink behavior
      return (void *) MFAIL;
    }
    else
    {
      return sbrk_top;
    }
  }

  // cleanup any allocated memory pools
  // called as last thing before shutting down driver

  void osCleanupMem(void)
  {
    void **ptr;

    for (ptr = our_os_pools; ptr < &our_os_pools[MAX_POOL_ENTRIES]; ptr++)
      if (*ptr)
      {
         PoolDeallocate(*ptr);
         *ptr = 0;
      }
  }

*/
```


27.2 History

```
/*
    V2.8.4 Wed May 27 09:56:23 2009  Doug Lea  (dl at gee)
      * Use zeros instead of prev foot for is_mmapped
      * Add mspace_track_large_chunks; thanks to Jean Brouwers
      * Fix set_inuse in internal_realloc; thanks to Jean Brouwers
      * Fix insufficient sys_alloc padding when using 16byte alignment
      * Fix bad error check in mspace_footprint
      * Adaptations for ptmalloc; thanks to Wolfram Gloger.
      * Reentrant spin locks; thanks to Earl Chew and others
      * Win32 improvements; thanks to Niall Douglas and Earl Chew
      * Add NO_SEGMENT_TRAVERSAL and MAX_RELEASE_CHECK_RATE options
      * Extension hook in malloc_state
      * Various small adjustments to reduce warnings on some compilers
      * Various configuration extensions/changes for more platforms. Thanks
         to all who contributed these.

    V2.8.3 Thu Sep 22 11:16:32 2005  Doug Lea  (dl at gee)
      * Add max_footprint functions
      * Ensure all appropriate literals are size_t
      * Fix conditional compilation problem for some #define settings
      * Avoid concatenating segments with the one provided
        in create_mspace_with_base
      * Rename some variables to avoid compiler shadowing warnings
      * Use explicit lock initialization.
      * Better handling of sbrk interference.
      * Simplify and fix segment insertion, trimming and mspace_destroy
      * Reinstate REALLOC_ZERO_BYTES_FREES option from 2.7.x
      * Thanks especially to Dennis Flanagan for help on these.

    V2.8.2 Sun Jun 12 16:01:10 2005  Doug Lea  (dl at gee)
      * Fix memalign brace error.

    V2.8.1 Wed Jun  8 16:11:46 2005  Doug Lea  (dl at gee)
      * Fix improper #endif nesting in C++
      * Add explicit casts needed for C++

    V2.8.0 Mon May 30 14:09:02 2005  Doug Lea  (dl at gee)
      * Use trees for large bins
      * Support mspaces
      * Use segments to unify sbrk-based and mmap-based system allocation,
        removing need for emulation on most platforms without sbrk.
      * Default safety checks
      * Optional footer checks. Thanks to William Robertson for the idea.
      * Internal code refactoring
      * Incorporate suggestions and platform-specific changes.
        Thanks to Dennis Flanagan, Colin Plumb, Niall Douglas,
        Aaron Bachmann,  Emery Berger, and others.
      * Speed up non-fastbin processing enough to remove fastbins.
      * Remove useless cfree() to avoid conflicts with other apps.
```

```
    * Remove internal memcpy, memset. Compilers handle builtins better.
    * Remove some options that no one ever used and rename others.

  V2.7.2 Sat Aug 17 09:07:30 2002  Doug Lea  (dl at gee)
    * Fix malloc_state bitmap array misdeclaration

  V2.7.1 Thu Jul 25 10:58:03 2002  Doug Lea  (dl at gee)
    * Allow tuning of FIRST_SORTED_BIN_SIZE
    * Use PTR_UINT as type for all ptr->int casts. Thanks to John Belmonte.
    * Better detection and support for non-contiguousness of MORECORE.
      Thanks to Andreas Mueller, Conal Walsh, and Wolfram Gloger
    * Bypass most of malloc if no frees. Thanks To Emery Berger.
    * Fix freeing of old top non-contiguous chunk im sysmalloc.
    * Raised default trim and map thresholds to 256K.
    * Fix mmap-related #defines. Thanks to Lubos Lunak.
    * Fix copy macros; added LACKS_FCNTL_H. Thanks to Neal Walfield.
    * Branch-free bin calculation
    * Default trim and mmap thresholds now 256K.

  V2.7.0 Sun Mar 11 14:14:06 2001  Doug Lea  (dl at gee)
    * Introduce independent_comalloc and independent_calloc.
      Thanks to Michael Pachos for motivation and help.
    * Make optional .h file available
    * Allow > 2GB requests on 32bit systems.
    * new WIN32 sbrk, mmap, munmap, lock code from <Walter@GeNeSys-e.de>.
      Thanks also to Andreas Mueller <a.mueller at paradatec.de>,
      and Anonymous.
    * Allow override of MALLOC_ALIGNMENT (Thanks to Ruud Waij for
      helping test this.)
    * memalign: check alignment arg
    * realloc: don't try to shift chunks backwards, since this
      leads to  more fragmentation in some programs and doesn't
      seem to help in any others.
    * Collect all cases in malloc requiring system memory into sysmalloc
    * Use mmap as backup to sbrk
    * Place all internal state in malloc_state
    * Introduce fastbins (although similar to 2.5.1)
    * Many minor tunings and cosmetic improvements
    * Introduce USE_PUBLIC_MALLOC_WRAPPERS, USE_MALLOC_LOCK
    * Introduce MALLOC_FAILURE_ACTION, MORECORE_CONTIGUOUS
      Thanks to Tony E. Bennett <tbennett@nvidia.com> and others.
    * Include errno.h to support default failure action.

  V2.6.6 Sun Dec  5 07:42:19 1999  Doug Lea  (dl at gee)
    * return null for negative arguments
    * Added Several WIN32 cleanups from Martin C. Fong <mcfong at yahoo.com>
       * Add 'LACKS_SYS_PARAM_H' for those systems without 'sys/param.h'
        (e.g. WIN32 platforms)
       * Cleanup header file inclusion for WIN32 platforms
       * Cleanup code to avoid Microsoft Visual C++ compiler complaints
       * Add 'USE_DL_PREFIX' to quickly allow co-existence with existing
         memory allocation routines
```



```
       * Set 'malloc_getpagesize' for WIN32 platforms (needs more work)
       * Use 'assert' rather than 'ASSERT' in WIN32 code to conform to
         usage of 'assert' in non-WIN32 code
       * Improve WIN32 'sbrk()' emulation's 'findRegion()' routine to
         avoid infinite loop
    * Always call 'fREe()' rather than 'free()'

  V2.6.5 Wed Jun 17 15:57:31 1998  Doug Lea  (dl at gee)
    * Fixed ordering problem with boundary-stamping

  V2.6.3 Sun May 19 08:17:58 1996  Doug Lea  (dl at gee)
    * Added pvalloc, as recommended by H.J. Liu
    * Added 64bit pointer support mainly from Wolfram Gloger
    * Added anonymously donated WIN32 sbrk emulation
    * Malloc, calloc, getpagesize: add optimizations from Raymond Nijssen
    * malloc_extend_top: fix mask error that caused wastage after
      foreign sbrks
    * Add linux mremap support code from HJ Liu

  V2.6.2 Tue Dec  5 06:52:55 1995  Doug Lea  (dl at gee)
    * Integrated most documentation with the code.
    * Add support for mmap, with help from
      Wolfram Gloger (Gloger@lrz.uni-muenchen.de).
    * Use last_remainder in more cases.
    * Pack bins using idea from  colin@nyx10.cs.du.edu
    * Use ordered bins instead of best-fit threshhold
    * Eliminate block-local decls to simplify tracing and debugging.
    * Support another case of realloc via move into top
    * Fix error occuring when initial sbrk_base not word-aligned.
    * Rely on page size for units instead of SBRK_UNIT to
      avoid surprises about sbrk alignment conventions.
    * Add mallinfo, mallopt. Thanks to Raymond Nijssen
      (raymond@es.ele.tue.nl) for the suggestion.
    * Add `pad' argument to malloc_trim and top_pad mallopt parameter.
    * More precautions for cases where other routines call sbrk,
      courtesy of Wolfram Gloger (Gloger@lrz.uni-muenchen.de).
    * Added macros etc., allowing use in linux libc from
      H.J. Lu (hjl@gnu.ai.mit.edu)
    * Inverted this history list

  V2.6.1 Sat Dec  2 14:10:57 1995  Doug Lea  (dl at gee)
    * Re-tuned and fixed to behave more nicely with V2.6.0 changes.
    * Removed all preallocation code since under current scheme
      the work required to undo bad preallocations exceeds
      the work saved in good cases for most test programs.
    * No longer use return list or unconsolidated bins since
      no scheme using them consistently outperforms those that don't
      given above changes.
    * Use best fit for very large chunks to prevent some worst-cases.
    * Added some support for debugging

  V2.6.0 Sat Nov  4 07:05:23 1995  Doug Lea  (dl at gee)
```

```
      * Removed footers when chunks are in use. Thanks to
        Paul Wilson (wilson@cs.texas.edu) for the suggestion.

    V2.5.4 Wed Nov  1 07:54:51 1995  Doug Lea  (dl at gee)
      * Added malloc_trim, with help from Wolfram Gloger
        (wmglo@Dent.MED.Uni-Muenchen.DE).

    V2.5.3 Tue Apr 26 10:16:01 1994  Doug Lea  (dl at g)

    V2.5.2 Tue Apr  5 16:20:40 1994  Doug Lea  (dl at g)
      * realloc: try to expand in both directions
      * malloc: swap order of clean-bin strategy;
      * realloc: only conditionally expand backwards
      * Try not to scavenge used bins
      * Use bin counts as a guide to preallocation
      * Occasionally bin return list chunks in first scan
      * Add a few optimizations from colin@nyx10.cs.du.edu

    V2.5.1 Sat Aug 14 15:40:43 1993  Doug Lea  (dl at g)
      * faster bin computation & slightly different binning
      * merged all consolidations to one part of malloc proper
         (eliminating old malloc_find_space & malloc_clean_bin)
      * Scan 2 returns chunks (not just 1)
      * Propagate failure in realloc if malloc returns 0
      * Add stuff to allow compilation on non-ANSI compilers
          from kpv@research.att.com

    V2.5 Sat Aug  7 07:41:59 1993  Doug Lea  (dl at g.oswego.edu)
      * removed potential for odd address access in prev_chunk
      * removed dependency on getpagesize.h
      * misc cosmetics and a bit more internal documentation
      * anticosmetics: mangled names in macros to evade debugger strangeness
      * tested on sparc, hp-700, dec-mips, rs6000
          with gcc & native cc (hp, dec only) allowing
          Detlefs & Zorn comparison study (in SIGPLAN Notices.)

    Trial version Fri Aug 28 13:14:29 1992  Doug Lea  (dl at g.oswego.edu)
      * Based loosely on libg++-1.2X malloc. (It retains some of the overall
         structure of old version,  but most details differ.)
*/
```