# An Overview of `omalloc`

This document gives a short overview of `omalloc`, the internal memory manager of SINGULAR.

## Requirements of a memory manager for SINGULAR

Most of SINGULAR's computations boil down to primitive polynomial operations like copying, deleting, adding, and multiplying of polynomials. For example, standard bases computations over finite fields spent (on average) 90 per cent of their time realizing the operation `p - m*q` where `m` is a monomial, and `p,q` are polynomials.

SINGULAR uses linked lists of monomials as data structure for representing polynomials. A monomial is represented by a memory block which stores a coefficient and an exponent vector. The size of this memory block varies: its minimum size is three machine words (one word for each, the pointer to the "next" monomial, the coefficient and the exponent vector), it's maximal size is (almost) unlimited, and we can assume that its average size is 4 to 6 machine words (i.e., the exponent vector is 2 to 4 words long).

From a low-level point of view, polynomial operations are operations on linked lists which go as follows:

1. Do something with the exponent vector (e.g., copy, compare, add), and possibly do something with the coefficient (e.g., add, multiply, compare with zero) of a monomial.
2. Advance to the next monomial and/or allocate a new or free an existing monomial.

Assuming that coefficient operations are (inlined) operations on a single machine word (as they are, for example, for coefficients from most finite fields), and observing that the exponent vector operations are linear w.r.t. the length of the exponent vector, we come to the following major conclusion:

> *For most computations, a traversal to the next memory block or a allocation/deallocation of a memory block is, on average, required after every 4 to 6 machine word operations.*

The major requirements of a memory manager for SINGULAR become immediately clear from this conclusion:

**(1) allocation/deallocation of (small) memory blocks must be extremly fast**
: If the allocation/deallocation of a memory blocks requires more than a couple of machine instructions it would become the bottleneck of the computations. In particular, even the overhead of a function call for a memory operation is already too expensive.

**(2) consecutive memory blocks in linked lists must have a high locality of reference**
: Compared with the performed operations on list elements, cache misses (or, even worse, page misses) of memory blocks are extremly expensive: we estimate that one cache miss of a memory block costs at least ten or more list element operations. Hence, if the addresses of consecutive list elements are not physically closed to each other (i.e., if the linked lists have a low locality of reference), then resolving chache (or, page) misses would become the bottleneck of computations.

Furthermore, there a the following more or less obvious requirements on a memory manager for SINGULAR:

**(3) the size overhead to maintain small blocks of memory must be small**
: If managing one memory block requires one or more words (e.g., to store its size) then the total overhead of the memory manager could sum up to 25% of the overall used memory, which is an unacceptable waste of memory.

**(4) the memory manager must have a clean API and it must support debugging**
: The API of the memory manager must be similar to the standard API's of memory managers, otherwise its usability is greatly limited. Furthermore, it needs to provide error checking and debugging support (to detect overwriting, twice freed or not-freed memory blocks, etc.) since errors in the usage of the memory manager are otherwise very hard to find and correct.

**(5) the memory manager must be customizable, tunable, extensible and portable**
: The memory manager should support customizations (like "what to do if no more memory is available"); its parameters (e.g., allocation policies) should be tunable, it should be extensible to allow easy implementations of furthergoing functionality (like overloading of C++ constructors and destructors, etc), and it needs to be portable to all available operating systems.

To the best of our knowledge, there is currently no memory manager available which satisfies these (admittingly in part extreme) requirements. Therefore, we desigend and implemented `omalloc`.

## Principles of omalloc

`omalloc` is a stand-alone, general-purpose memory managment library which can be used by any program. Its major design goals were derived from the memory management requirements of SINGULAR (see above).

`omalloc` does neither provide nor support garbage collection – it is an extension to, and sits on top of, the standard `malloc/realloc/free` routines provided by most operating systems.

`omalloc` is specialized in the management of small memory blocks. The managment of large memory blocks, i.e., of memory blocks which are larger than "one fourth of the page size of the system" (usually appr. 1KByte) are (almost) left entirely to the underlying `malloc/realloc/free` implementation.

`omalloc` manages small blocks of memory on a **per-page basis**. That is, each used page is split up into a page-header and equally-sized memory blocks. The page-header has a size of 6 words (i.e., 24 Byte on a 32 Bit machine), and stores (among others) a pointer to the free-list and a counter of the used memory blocks of this page.

Pages which manage equally-sized memory blocks are collected into so-called "**Bins**", which are (more or less) nothing but linked lists of pages.

On **memory allocation**, an appropriate page (i.e. one which has a non-empty free list of the appropriate block size) is determined based on the used memory allocation mechanism and its arguments (see below). The counter of the page is incremented, and the provided memory block is dequed from the free-list of the page.

On **memory deallocation**, the address of the memory block is used to determine the page (header) which manages the memory block to be freed. The counter of the page is decremented, and the memory block is enqued into the free-list. If decrementing the counter yields zero, i.e., if there are no more used memory blocks in this page, then this page is dequed from its Bin and put back into a global pool of unused pages.

`omalloc`' **API** provides a three-level interface:

1.) Standard function interface level
This level provides the standard allocation/deallocation functions like `malloc, calloc, realloc, free`, etc. No special `#include`'s are required, only linking with `libomalloc` is necessary.

2.) Standard macro interface level
This level provides variants of the standard allocation/deallocation functions and some extensions thereof in form of macros. To use this level, the header file `omalloc.h` needs to be included by an application, and it must be linked with `libomalloc`.

3.) Bin-based macro interface level
This level provides allocation (resp. deallocation) macros whose arguments are not sizes (like those of the standard allocation functions/macros), but directly Bins. Naturally, to use this level, the header the file `omalloc.h` needs to be included by an application, and it must be linked with `libomalloc`.

`omalloc` has a special mode of operation, namely its **debugging mode**. This mode can be activated for the functional interface by setting run-time parameters, and, for the macro-interface, by setting debug-flags on a per-file basis, before `#include`'ing the file `omalloc.h`. Furthermore, the granularity of the debugging can dynamically be set at run-time using various parameters. One special feature of `omalloc` is that memory allocated with a debugging allocation routine can safely be deallocated with a non-debugging deallocation routine (and vice versa). In other words, the debugging and non-debugging mode of omalloc are not mutually exclusive. When in its debugging mode, `omalloc` is set out to catch more than 20 errors in the usage of its API. And, of course, all the common and classical errors are among those which are cought by `omalloc`. Error reporting can either be directe3d to log files or to `stderr`. On some platforms, the error reporting does not only include line number information, but also stack trace (i.e., a description of the routines on the function stack), information.

## Requirement evaluation of `omalloc`

After explaining its princples of `omalloc` let us evaluate `omalloc` w.r.t. the requirments stated above.

**(1) efficiency of allocation/deallocation**
When used with its Bin-based macro interface level, the allocation/deallocation of small memory blocks requires (on average) only a couple (say, 5-10) assembler instructions. We believe that this is optimal, i.e., as fast as one possibly could implement these operations.
The other interface levels incure the additional overhead of the determination

of the appropriate Bin (using a table lookup); and, for the function interface level, the additional overhead of function calls. Compared with existing memory managers, even these other interface levels provide unmatchingly efficient allocation/deallocation routines.

**(2) locality of reference**

Based on its design principles `omalloc` provides an extremly high locality of reference (especially for consecutive elements of linked-list like structures). To realize this, recall that

> consecutively allocated memory blocks are physically almost always from the same memory page (the current memory page of a bin is only changed when it becomes full or empty).
>
> the probabilty that consecutively allocated memory blocks are also physically consecutive (i.e., that their physical addresses are consecutive) is rather high, since memory pages are spooled-back into the pool of unused pages as often as possible, and, hence, are reinitialized as often as possible (notice that allocationg from a freshly initialized page assures that the physical addresses of consecutively allocated memory blocks are also consecutive).

Moreover, the allocation policy of a Bin can be changed at run-time such that it is assured that consecutive allocations from this Bin result in physically consecutive memory blocks. However, the latter can only be achieved by paying for with a slightly higher memory consumption, and should therefore only be used temporarily for long-lived, often traversed-over data.

**(3) small maintainance size overhead**

Since `omalloc` manages small memory blocks on a per-page basis, there is no maintainance-overhead on a per-block basis, but only on a per-page basis. This overhead amounts to 24 Bytes per memory page, i.e., we only need 24 Bytes maintanance overhead per 4096 Bytes, or 0.6%.
For larger memory blocks, this overhead grows since there might be "leftovers" from the division of a memory page into equally-sized chunks of memory blocks. Furthermore, there is the additional (but rather small) overhead of keeping Bins and other "global" maintance structures.
Nevertheless, on average, we observed a maintanance overhead of appr. 1% of the total memory usage, which, to the best of our knowldege, is unsurpassed by any existing, general-purpose, memory manager.

**(4) clean API and debugging support**

Whether or not `omalloc` provides a clean API is certainly a matter of taste and expectation. However, it is intrinsic that the price one has to pay for the speed and flexibiity of a memory manager like `omalloc` is a more complex API than that of the standard `malloc/realloc/free` functions. Nevertheless, as we believe and as our experience has shown, the above outlined layered interface designe provides a practical compromise between simplicity and efficienc/flexibility.

W.r.t. the debugging mode has our experience shown that it is extremly useful to be able to fine-tune the granularity of the debugging under both, a lexical, per-file base scope, and under a dynamic scope set at run-time. It also turned out that it is very convenient that the debugging and non-debuggung mode are not mutually exclusive and that the same library (and, header file) can be used for both modes – i.e., that no recompilations of `omalloc` are required to enable one or the other mode. As a summary `omalloc` debugging features can at least be compared with available memory managment debugging libraries (like dmalloc) – our experience has furthermore showh that the features are suitable and appropriate for most practical purposes. But then again, fittnes is also a matter of taste and expectations and therefore hard to evaluate.

**(5) customizable, tunable, extensible and portable**

## Performance evaluation of `omalloc`

We have no (yet) thouroughly measured the performance of `omalloc` in comparison to other memory managment systems. Nevertheless, our first experiments yielded the following:

SINGULAR's memory consumption is up to 30% lower and its speed is up to 100% faster when used with `omalloc` instead of of a standard memory manager (like the one found in the `libc` library on Linux, or the one underlying `perl`).

We expect that for programs/computations with similar compuational characteristics like those of SINGULAR (e.g., multivariate polynomial computations, sparse matrix computations, etc), the speed increase and memory consumption decrease is comparable to those reported for SINGULAR.

For other applications (like `perl` or `gcc`) we observed a slight speed increase (depending on the input) and an overall decrease in the memory consumptions of up to 25%.

## Status/Copyright of `omalloc`

`omalloc` is a fully-functional, mature memory managment library. It still has a TODO list, with more or less only one major point: Create a detailed technical reference manual (in texinfo), because there is none so far.

`omalloc` was written by Olaf Bachmann who also holds its copyright.