xmemory module

Nicolas Devillard

May 3, 2002

Abstract

xmemory is a small and efficient module offering memory extension capabilities to ANSI C programs running on POSIX-compliant systems. If offers several useful features such as memory leak detection, protection for free on NULL or unallocated pointers, and virtually unlimited memory space. **xmemory** requires the `mmap()` system call to be implemented in the local C library to function. This module has been ported and tested on a number of Unix flavours (Linux, HPUX, Solaris, BSD derivatives, Dec OSF/1).

1 Introduction

The main issue to face when programming image processing applications is memory handling. As far back as one can look, images have always been much bigger than the affordable amount of memory space. Moore's law ensures that computer performances get an exponential increase but detectors and images follow the same law, yielding an almost constant ratio over the past 40 years. Astronomical image processing faces the same challenges with detector sizes producing gigabyte-sized images by large numbers. People facing the difficult task of processing large amounts of frames (e.g. video editing) are also in need of efficient memory handling schemes.

Since the ratio image size over affordable memory space keeps constant, it is probably worth spending the necessary time to find out a scalable solution that is generic enough to survive the evolutions of technology without having to rewrite large amounts of code too often.

The only way to fit images into the processing space (be it RAM or swap space or wherever the pixels are stored for processing) is to cut down the images into manageable chunks (or tiles). This splitting can be done either by the programmer and included within image processing algorithms, or it can be done invisibly by use of virtual memory space, allowing the programmer to design algorithms believing that all pixels are present in memory at the same moment.

The second kind of method is implemented in the **xmemory** module to respond to a number of constraints. The following document describes its design and implementation.

2 Requirements

The final product shall be a module that allows processing large images on any kind of machine, provided it has a reasonable minimal (bootstrap) amount of memory to be defined.

The module should satisfy the following requirements:

- It must allow the processing of images larger than the amount of memory that can be allocated through the system's `malloc` call or equivalents.
- It must be completely transparent to the programmer.
- Memory allocation functions must keep the same name as before.
- It must not request any extra handling effort for the programmer.
- It should offer easy debugging hooks.
- Performance penalties should remain small.
- It should work on as many platforms as possible, not relying on hardware or OS-specific code.
- The code should remain as small as possible, ideally contained in a single module.
- The code should not rely on other libraries to work.

Let us detail the above points:

The main goal of the module is to give access to a virtually infinite amount of memory. This will allow loading and processing large data sets, typically larger than the machine can normally handle. The usual behaviour of a program when `malloc` returns `NULL` is to die with an error message, this module should allow to go past this point.

The module should remain transparent. Adding extended memory handling should not have the least impact on the code it is supposed to extend.

To avoid a global search/replace operation on all source codes that need extended memory, memory routines should keep the same name (`malloc`, `calloc`, `free`, possibly `realloc`) and prototype. An overloading mechanism must be installed.

The module should remain as invisible as possible, in particular it should be avoided to add any specific call in a code to activate or deactivate the module.

From the above requirements, it can be deduced that the module should install itself in target code at compile-time, overloading the usual memory functions. It should initialize and shut down automatically without need to call specific routines.

The module should also offer easy debugging hooks. Memory handling is particularly tricky in C, installing such a module should not burden even more the programmer's task. Debugging hooks are vital for quality software

development. It is desirable to have the possibility to turn off debugging features in deployed code.

The module should not inflict a too high performance penalty on the host program. Handling large amounts of data is anyway expected to take large amounts of time, adding even more performance issues should be avoided.

The module should ideally not be specialized to any given platform, to preserve its portability and longevity. A dedicated module is likely to have to be thrown away when the target platform becomes obsolete, whereas portable software survives longer than platform-dedicated modules. Note that this does not prevent the inclusion of hardware or OS-specific code in the module, as long as this remains within e.g. `#ifdef`'s.

The module should remain small, to facilitate its exportability. Exporting the module to a large audience (i.e. open-source) will generate more testing and feedback than is possibly achievable by the module author alone. As much as possible, the module should be contained in the least number of files.

Last: the code should be completely stand-alone, making use only of the local C library. Depending on other libraries means applying the above requirements to them, which is not feasible.

3 Analysis

There are not many ways of extending the available memory on a computer. The easiest solution is to buy more RAM chips and install them on the motherboard, but this is expensive, requires manual intervention, and might not even solve the problem when input data sizes are overwhelming (larger than the largest amount of RAM one can possibly install on the motherboard).

Another solution on systems supporting swap filesystems (i.e. practically all modern operating systems) is too create swap space on the local disks. This must unfortunately be done by the superuser and requires some technical knowledge of the OS. This also has the limitation that the input data might still be larger than the largest amount of swap one can define on the machine. This is an alternative solution for building dedicated number-crunching machines cheaply. For an equivalent price, a hard disk has always provided an order of magnitude more storage space than RAM chips. But in the general case of a user wanting to run one data processing program on systems that cannot be customized for this use, this solution is not applicable.

A third solution is to make use of memory-mapped files. This is supported on all POSIX compliant operating systems that have the `mmap()` system call (i.e. all modern Unix flavours). The idea is that `mmap` allows a mapping between a memory pointer and a file on disk. All modifications brought to the pointer are reflected into the file. Since the memory zone is located in a file, it is in effect acting as extra swap buffer added to the system on the fly. The main difference is that mapped files are not swap space thus less efficient, but need no superuser privilege to be created and used.

What are large amounts of memory used for? In general, the first memory-greedy task is the data loader. It is responsible for bringing the potentially

large input flux into a working space accessible to the programmer (RAM or swap space). Whatever the data format used for the input stream, there is a need for a memory allocator to provide space holding the contents of a file read. This is another task for which memory-mapping of files can be of great help. This memory module should offer some facilities related to that task.

Notice that a file/memory mapping primitive is usually available on all modern operating systems. The present module uses the facility provided on POSIX compatible systems, but an equivalent solution (an `mmap()` equivalent) can probably be found on other platforms.

In summary, the issues to solve are:

- How to map the usual allocators to extended allocators at compile-time?
- What is the strategy for creating extra swap files? When are they created? Deleted? When are they necessary? What is needed to handle them?
- When is the module initialized? Shut down?
- How can data loading through file mapping be integrated?

These issues are addressed in the following sections.

3.1 Overloading allocators in C

The easiest way to map allocators to new functions in C is to use the C preprocessor. The following piece of code does the job:

```
#define malloc(s)        xmemory_malloc(s)
#define calloc(n,s)      xmemory_calloc(n,s)
#define strdup(s)        xmemory_strdup(s)
#define free(p)          xmemory_free(p)
```

To include extended memory handling in an application, it is sufficient to include the memory header file to re-direct all memory allocations to other functions. This allows to add extended memory handling to already existing programs without modifying their source (other than adding the include file).

Another solution used in similar memory-debugging tools makes use of dynamic libraries to call the library allocators instead of the standard ones. This has the main drawback of failing portability because it relies on the local dynamic library handling features. It is also not safe to do so because such a mechanism acts upon all programs linked to the C library (e.g. all Unix commands), so any bug in **xmemory** will affect the whole system. On the other hand, it has the advantage of acting without even having to recompile the application. This trick is used by a number of debuggers and profilers on Linux, but does not satisfy the requirements stated for this module.

3.2 Virtual memory files

Using mapped files as external memory is not cheap. Disk accesses are much slower than memory, and using a normal filesystem to do swapping is even slower than normal swap space. Virtual memory files (VM files for short) should only be used when necessary, i.e. when normal memory is exhausted.

To ease the paging task for the operating system, the size of VM files should always be a multiple of the system's memory page size.

VM files should be deleted as soon as the pointer is released (by `free()`). If memory leaks occur, the module should make sure that all extra VM files are deleted when the process ends (with a possible warning about the leak).

The `munmap()` call requires the file handle and the allocated size to perform deallocation of the file. It means that these informations need to be stored together with the allocated pointer for the `free()` function to do its job.

3.3 Initialization and shutdown

The usual way of bringing a module up or down is to provide two functions: one for initialization and one for shutdown. Since we want the module to be as little intrusive as possible, there should be an automatic mechanism to take care of this without forcing the programmer to use such functions.

3.4 Large input file handling

To load a large data file (typically larger than the available memory) into memory, the input has to be split into manageable chunks and the corresponding memory blocks allocated and deallocated. The typical way of loading a file into memory to process it could be described in pseudo-code as:

```
- Allocate a memory buffer of size N (bytes)
- Load the first N bytes of the input file into memory buffer.
- Process the buffer.
- Repeat loading the next N bytes from the input file until
  the whole file has been read and processed.
- Free the buffer, close the file.
```

As for large memory allocations, this process is not very elegant and requires the programmer to think algorithms in advance so that they can be handling the data chunks separately. Cutting an algorithm into smaller parts so that they work on file sections is usually hard to do if not impossible.

Using `mmap()`, this process can be simpler written as:

```
- Map the input file to a memory pointer acting as a buffer.
- Process the buffer.
- Unmap the file.
```

No need to allocate or free any memory, the whole process is convinced that the whole file is loaded into memory and accessible through the pointer

returned by `mmap()`. This way of loading files into memory does not require slicing the input files into manageable chunks.

From the operating system's point of view, data are loaded from the file as they are needed (paged into memory) where they are truly accessed as in a memory buffers, then paged out when not needed. This handling is invisible for the programmer who has to rely on the operating system to perform the paging efficiently.

In essence, `mmap()` acts as an interface to files on disk and allows to write programs working on the whole contents of a file as if it were in memory, even if the file size is bigger than the available amount of true memory.

4 Design

4.1 General architecture

The memory module is expected to run in situations where no more dynamic memory can be obtained through the usual system calls. This implies a static allocation of all required variables, immediately placing some limits on the module usage. Since all relevant variables have to be statically allocated, all sizes have to be known in advance or set to a maximum, ideally with a single `#define` statement. For the current implementation, this design choice has an implication on the maximum number of pointers that can simultaneously be handled by the module.

Let us review what the module should manage. There are three kinds of memory blocks to handle:

- Blocks allocated through the system's `malloc()` or equivalents, freed with `free()`.
- Blocks built from VM files, freed by un-mapping the VM file and deleting it.
- Blocks obtained through memory-mapped files, freed by un-mapping the mapped file if no more pointers are referring to it.

The module infrastructure must be able to remember the type associated to each memory block to perform the correct deallocation. It must also remember the size associated to each pointer, since it is needed for memory blocks in VM or mapped files for the `munmap()` call.

An obvious implementation choice is to statically allocate a table to contain references to all memory allocations. This table should be initialized the first time a memory allocator is called.

4.2 Speed considerations

Every time a pointer deallocation is requested, the module will have to lookup the general allocation table to determine the kind of memory it is linked to, and perform the correct kind of deallocation. Table lookups are notoriously

expensive, so dedicated mechanisms should be provided to avoid impeding performance.

4.3 Operator design

The global algorithm for memory allocation is the following:

```
A request is received for a block of N bytes.
  - Try to allocate it with malloc() and return the block
    if successful.
  - If malloc() failed, try to create a VM file on disk,
    map it using mmap() and return the resulting pointer.
  - If no VM file can be created, fail (exit).
```

The third kind of memory allocation (large file mapping) should be handled by the following algorithm:

```
A request is received for a mapping of a file,
N bytes starting from offset START in the file.
  - If the file is already mapped, return the appropriate
    pointer (file pointer + START bytes).
  - Map the file, return the appropriate pointer or NULL
    if the file cannot be mapped.
```

Deallocating such a pointer means decreasing a counter on the file mapping. If the counter reaches zero, the file can be unmapped.

5 Implementation

5.1 Global allocation table

The global allocation table is a private (static) object, statically allocated inside `xmemory.c`.

The current implementation of this table stores for each pointer:

- The pointer value.
- The size of the associated memory block.
- The name of the source file and the line number where the allocation took place (optional).
- The type of allocated memory.
- If the block is a swapped VM file: a unique swap file ID and file descriptor.
- If the block corresponds to a mapped file: the name of the mapped file, and a reference counter for this file.

Each of these structures is called a *memory cell*.

In addition, the global memory table stores various informations about memory allocation since it was first initialized, e.g. total amount of allocated RAM, of allocated VM files, maximum number of pointers allocated so far, etc.

Storing the name and line number of the calling source file is actually performed using the `__FILE__` and `__LINE__` macros in C. The overloaded calls to memory allocators are the following:

```
#define malloc(s)   xmemory_malloc(s,   __FILE__,__LINE__)
#define calloc(n,s) xmemory_calloc(n,s, __FILE__,__LINE__)
#define free(p)     xmemory_free(p,     __FILE__,__LINE__)
#define strdup(s)   xmemory_strdup(s,   __FILE__,__LINE__)
```

This is not useful for the VM mechanism as such, but is an excellent debugging tool for memory leaks. This mechanism is only activated for a sufficient debug level (see below for description of all possible debug levels).

The swap file descriptor and block size are required to unmap VM files. The swap file ID is a unique number delivered to each VM swap file by the memory module. This number identifies the name used for the file on disk, and is used to build the VM file name when the module needs to delete it. See below for VM file name conventions.

In this implementation, the global memory table is statically allocated to contain 8192 pointers (a value that can be modified by editing `xmemory.c` and changing the definition of `XMEMORY_MAXPTRS`). This represents about 4 megabytes on a Linux PC, which should be reasonably small compared to the expected data sizes.

Increasing the maximal number of handled pointers will increase the initial memory usage of any process linked against this module, so a tradeoff has to be found.

5.2 Initialization and shutdown

The module initializes itself (clears the global memory table, installs signal handling and exit routines) the first time a memory allocator is called.

Shutting down is done by attaching a cleanup function to the process end with `atexit()`. Cleaning up VM files is simple: loop from 1 to the highest file registration number, build up VM file names from this number and try to delete the corresponding file if it exists.

Another case to handle is sudden process interruption. There should be no remaining VM file around once the process is terminated. The easiest way to do that is to install a grabber for a number of interruption signals like `SIGINT`, `SIGTERM`, `SIGBUS` or `SIGSEGV`, and launch the cleanup procedure to remove all VM files before exiting. In the current implementation, the module makes use of the `signal()` system call to catch these signals, print out a message on stderr and call `exit()`.

Attaching the cleanup procedure with `atexit()` also ensures that the filesystem used for VM file creation will be clean after a normal program termination,

even in the case of memory leaks. The only case when a process will leave VM files behind is when it has been killed using the `-KILL` signal (which cannot be caught).

5.3 Virtual Memory files

Virtual memory files are allocated whenever `malloc` starts returning `NULL`. In that case, a new empty file is created on disk with the size rounded up to the next multiple of the memory page size. This file is memory-mapped in public mode, which signals the OS that memory associated to this mapping can be safely dumped to disk at any moment. The mapped pointer is returned to the caller of the allocator.

To avoid name clashes, VM files should all have different names. The current naming scheme is using a base name (`vmswap_`) and concatenates it to the process ID and to a registration number assigned internally by the module. This allows to run several processes making use of VM files simultaneously. Since there cannot be two processes running with the same PID, the name prefix for a process is unique at any given moment.

The file registration number is simply an integer which is incremented for each newly created VM file.

The VM file name is necessary to delete the file when it is deallocated, but the whole file name does not need to be stored in the memory cells. Since the naming scheme for VM files is constant for a given version of the module, it is enough to remember only the file registration number (VM file ID) and build up the file name from it when necessary.

It might be judged necessary to change the VM file naming scheme at some later stage, so programs using the module should never rely on the scheme itself, only on VM file IDs.

VM files are created in a temporary area, which should be large enough to fit the memory requirements. Setting up an automated system to find a suitable place is out of the scope of this module. The default directory used to store VM files is '.' (the current directory). This can be changed at any moment by using `xmemory_settmpdir()` and read by using `xmemory_gettmpdir()` (see module reference below).

5.4 Large input file handling

xmemory offers an additional convenience to the programmer: if the `mmap()` mechanism is unified with the usual memory-allocation operators, pointers can be used without having to know if they are referring to memory-mapped files or true memory allocated through the normal means. Only the `free()` function has to know the origins of the pointer, and apply the correct deallocator: the system's own `free()` function, virtual memory file deletion or file unmapping.

This module offers a convenient mmap-like function that maps a file to a memory pointer that can later be freed using `free()`, so the programmer does not need to know the origins of the pointer.

This function has the following prototype:

```
char * falloc(char * name, size_t offs, size_t * size);

name        Name of the file to map into memory
offs        Offset from the beginning of the file in bytes
size        Returned mapped size in bytes.
```

Here is an example: the following program dumps the contents of the file which name is passed as first argument in reverse order onto `stdout`. Notice that the mapped file could be of any size.

```
/* Error handling omitted for readability */
#include <stdio.h>
#include "xmemory.h"

int main(void)
{
    char * contents ;
    size_t size ;
    int    i ;

    contents = falloc(argv[1], 0, &size);
    printf("file size is %ld\n", (long)size);
    for (i=size-1 ; i>=0 ; i--) {
        printf("%c", contents[i]);
    }
    free(contents);
    return 0 ;
}
```

The arguments passed to `falloc()` are the file name (possibly including a complete path), an offset in bytes from which the file will be mapped, and a pointer to a `size_t` value which is updated to contain the size of the mapped buffer in bytes.

To implement this behaviour, the `falloc()` call is performing the requested memory-mapping and stores the pointer returned by `mmap()` into the general memory allocation table defined in **xmemory** , together with the additional information needed to unmap the file when required.

Notice that some Operating Systems (e.g. HPUX) do not allow a process to map the same file twice in memory. This is protected in the `falloc()` call: if the module identifies that the file has already been requested for mapping, it will simply return the already associated pointer and increase a reference counter for that file. When `free()` is called on that pointer, the reference counter is decreased until it reaches zero, at which point the file is unmapped.

5.5 Memory leak detector

Since every allocation call is logged in **xmemory** , it is easy to use the module as a memory leak detector. The way to do it is to call the `xmemory_status()`

function before the program exits, at a point where no memory block should remain allocated. This function will remain silent if all memory has been deallocated, or print out a list of still allocated pointers, where they have been allocated, their size and type. Example:

```
#include <stdio.h>
#include "xmemory.h"

int main(void)
{
    int * ip ;

    ip = malloc(4 * sizeof(int));
    xmemory_status();
    return 0 ;
}
```

In xmemory version 2.0 and later, the output of this program looks like:

```
#----- memory status called from a.c (9) --------
#- ALL status
ALL_npointers       1
ALL_size            16
ALL_maxalloc_kb     0
#- RAM status
RAM_alloc           16
#- pointer details
(0x804bdc8) from a.c (8) in RAM  has 16 bytes
```

If you add a `free(ip)` statement before the end, `xmemory_status()` will not report anything.

The `xmemory_status()` function may print out diagnostics in addition to the memory status if the debug level is high enough.

5.6 Debug levels

xmemory features a single debug level:

```
/**
  This symbol sets the debug level for the xmemory module.
  Debug levels are defined as follows:

  0   no debugging.
  1   add tracing capabilities for memory leak detection
  2   add diagnostics in xmemory_status
  3   add debug messages
*/
#ifndef XMEMORY_DEBUG
```

```
#define XMEMORY_DEBUG       0
#endif
```

As mentioned in the doc, most users will want to compile the module without defining the `XMEMORY_DEBUG` symbol in the Makefile (through the `-D` preprocessor option) which will reduce the size of the global allocation table.

If you suspect you have memory leaks, it is a good idea to switch the debug level to 1, which will help you by reporting for each allocated block the kind of memory that was allocated (real, virtual, or mapped file), and most importantly the name of the file and line number where the allocation took place.

Setting the debug level to 2 will change the behaviour of `xmemory_status()`, which will then systematically print out diagnostics about your memory use, in addition to the potential memory status.

A debug level of 3 and above will activate debug messages informing you whenever an allocation or deallocation takes place, about the internal working of the module, etc. It is not recommended to activate such a debug level unless you want to debug the **xmemory** module itself.

The above definition of `XMEMORY_DEBUG` allows to change the value at compile-time without modifying the **xmemory** source code. Example:

```
Compile without debug:
% cc -c xmemory.c
%
Compile with highest debug level:
% cc -c xmemory.c -DXMEMORY_DEBUG=3
```

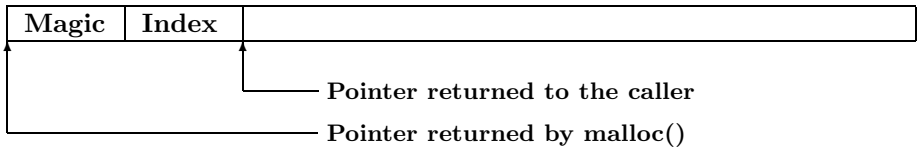
5.7 Fast pointer lookups

The main performance issue with this module is the time spent in pointer lookups in the main memory table every time a pointer is deallocated. A linear search (as implemented in **xmemory** until version 2.0) is likely to slow the module down too much. Binary searches or equivalent algorithms would unfortunately require the memory table to be sorted by increasing or decreasing pointer values, which also involves time-consuming tasks. The solution implemented here avoids table lookups completely.

The aim is to be able to retrieve the place where a pointer is stored in the main memory table in the shortest possible time. The idea is to store the index in the main memory table *together with the pointer*, i.e. in the memory block itself.

When a request for an allocation of N bytes is received, the allocator tries to allocate N bytes plus the size for two integers. If the allocation succeeds, the first extra int is set to a magic number, the pointer is added to the global memory table and the pointer index in this table is stored in the second extra int allocated in the block.

This is illustrated here:



When the deallocator receives a pointer to free, it goes two ints back before the pointer and checks the value of the two integer numbers in memory. If the first int corresponds to the **xmemory** magic value, the second int is taken as the index in the global memory table. Otherwise, a normal linear search is made through the table.

The trick of allocating a little bit more memory than requested to store extra information is not completely safe. If a normal pointer (one that does not contain these extra informations) is passed to the deallocator, the module may be unlucky enough to encounter the magic number and believe the following int is an index. This is covered by extra tests to ensure that if the deallocation is not consistent with the table, a linear search is launched anyway.

This memory marking can unfortunately not be used for VM files or mapped input files, but this kind of memory involves slow disk accesses anyway so a linear table search is not likely to be noticed.

6 System-specific features

6.1 HPUX

On HPUX it is impossible for a userspace process to query the maximum allocatable amount of memory for a single process. This is actually not an issue with this module, since it will start using VM files when system memory is exhausted.

Swapping mapped files proves to be extremely unefficient on HPUX, but this probably also depends on the kind of hard disks available on the machine. Only benchmarking can give a reasonable estimate for HP performances.

6.2 Solaris

Solaris has not shown any system-specific issue with memory handling.

6.3 Linux

The main problem in Linux is memory over-commitment. Linux has this very specific feature that it will promise more dynamically allocated memory than it can actually honour. A simple program can show that:

```
#define ONEMEG  (1024*1024)
int main(void)
{
    size_t  total=0 ;
    while (1) {
        if (malloc(ONEMEG)==NULL) {
```

```
            printf("stopped at total=%ld\n", (long)total);
            return -1 ;
        }
        total += ONEMEG ;
    }
    return 0 ;
}
```

This program will happily allocate as many memory pointers as it can count (on a 32-bit processor that amounts to 2 or 4 gigabytes) whereas on a non-Linux machine, it will fail after having exhausted all of the available memory.

This has the nasty side-effect that `malloc()` on Linux never returns NULL, but a seemingly valid pointer. When pointers start being used and filled up with data, the machine will start allocating all of the available RAM, then fill up the swap, then processes will start dying because of memory allocation failures.

This often has the very bad effect of killing daemons, which sometimes ends up in a total freeze of the machine, needing a manual reboot. The problem has been acknowledged by the Linux memory-management team, but no satisfactory solution is currently foreseen. The newer VM allocation system installed since Linux 2.4.11 does not seem to have solved this issue.

As a workaround, it is enough to "touch" memory regions immediately after they have been allocated. By doing this, the OS realizes that the allocated memory pages will truly be used by the program. This ends up in having a `malloc()` call that truly returns `NULL` when memory is exhausted, achieving the desired effect. To touch a region in the fastest way, the module will write one zero at every multiple of the memory page size within the block. The memory module will thus be slower to allocate large memory blocks on Linux than on other systems.

7 Module usage

The module has been completely implemented in a single source file called `xmemory.c`, associated to its header file `xmemory.h`. Users who want to add extended memory functionalities to their programs only have to include `xmemory.h`:

```
#include "xmemory.h"
```

The module appears to compile and run as expected on at least Linux, HPUX, Solaris, BSD and derivatives, AIX and OSF/1.

Notice that using this module, you will *never* get a NULL value from a `malloc`, `calloc` or `strdup` call. This comes from the fact that `xmemory_malloc` is configured to exit the current process (with `exit()`) if it cannot return a valid memory block of the requested size. This makes sense, because this module will already give you far more memory than your machine can in theory handle. If you really get out of memory using this module, you have probably run out of pointers or something just as tragic. Better quit the current program than try to make do.

This being said, if you are worried about that you can deactivate the exit statement in `xmemory_malloc` to make it return NULL in case it runs out of memory too.

The main advantage of a trustful `malloc()` is that it allows you to get rid of all the usual tests done to avoid writing to a NULL pointer. The usual code looks like this:

```
if ((p=malloc(10 * sizeof(int)))==NULL) {
    fprintf(stderr, "out of memory: exiting\n");
    return -1 ;
}
```

Whereas the code without error-checking looks like that:

```
p = malloc(10 * sizeof(int));
```

If you are using lots of objects (in the OO sense) defined as structs which are allocated using malloc, you can also clean out all object constructor calls, because if an object is allocated using malloc, it either returns a valid object or it dies in the process before it even returns to the caller.

The **xmemory** module will print out its error messages (few) on the `stderr` stream. If that is a problem for your application or if you would like to hook up your own error message routine, you will need to replace all calls to `fprintf(stderr,...)` by your own function. There are not so many error messages in `xmemory.c`, especially if the debug level is zero.

8 Limitations

Gathering implementation features from the above, we can state the following limitations for this module:

- The machine running the module should have enough memory to store statically the global allocation table. In the current implementation, this amounts to about 4 megabytes.
- The module will run out of memory when all possible pointer values are exhausted on the local machine. On a 32-bit processor, this usually translates to a 2 or 4 gigabyte limit in memory allocation, no matter how much disk space is available. This limit should anyway be raised on 64-bit machines to insanely high values.
- The module will fail if more than the expected maximum number of allocated pointers is reached. If you have a program that frequently allocates thousands of pointers simultaneously, you may want to increase the size of the static allocation table.
- The module will be (comparatively) slower on Linux machines due to the fact that memory blocks have to be touched after allocation.

9 Module reference

Programmers using the library should be aware of the following:

Including extended memory handling is done by:

```
#include "xmemory.h"
```

9.1 Symbols

The **xmemory** module defines a symbol called `_XMEMORY_` to allow testing for the presence of the module at compile-time. This allows e.g.:

```
#ifdef _XMEMORY_
    printf("xmemory is present!\n");
#endif
```

9.2 Memory allocators

Calls to `malloc`, `calloc`, `strdup` and `free` are re-directed to the **xmemory** functions.

`realloc` is not re-directed for several reasons which would make it unportable. If you *really* need to re-allocate a block of memory, it is preferrable to do it yourself, i.e. allocate a new block, copy the contents of the previous block into the new one, deallocate the old block.

9.3 Environment

You can get and set the name of the temporary directory used to create VM files using `xmemory_settmpdir()` and `xmemory_gettmpdir()`. See the documentation in `xmemory.h` about these two functions.

9.4 Status and diagnostics

You can get a complete memory status at any moment in your code by calling `xmemory_status()`. This function will summarize memory usage on `stderr`, giving the total allocated amount of memory of each kind (system memory and VM files), the number of opened VM files, and a list of all currently allocated memory blocks. This function is useful for debugging purposes. If you want to make sure you do not have memory leaks, it is recommended to call it as the last command in your program.

If the debug level is set to any value greater than 2, `xmemory_status()` will also print out diagnostics informations.

10 Testing the module

To test this module on your machine, you probably want to avoid overflooding memory. The simplest way is to use the shell's `limit` (csh, tcsh) or `ulimit` (sh, bash, ksh) command. This command allows to set a number of limits

for the processes launched by the current shell (i.e. only the window you are typing commands in, it does not affect other shell sessions). The syntax of the parameters you can change depends on your shell and platform, but the one reserved for memory usage is usually called `datasize`.

11 Frequently Asked Questions

11.1 Does malloc() really never returns NULL?

Yes! If an allocation cannot take place in true memory or in swap files, the module will call `exit()` and never return. You can safely call `malloc()` without fear of getting a NULL pointer back.

Do not forget that the module extends the amount of allocatable memory to the magic barrier of 2Gb or 4Gb on a 32-bit machine. When did you last need that much memory? `:-)`

11.2 malloc() never returns NULL, is it a good thing?

Yes and no. Neglecting a return value in C is usually a source of errors, and doing it systematically on `malloc()` is actually introducing a dependency in your code on this module.

On the other hand, if you really have a need for very large amounts of memory, this module is not going to replace a careful analysis of the hardware you have. You will have to go buy RAM chips or add swap space to your machine. This is not due to the library, but to the fact that you know you are going to need large amounts of memory and should spend the money on the machine in consequence. The VM module is there as a safety net so that the process can run on virtually any machine, it does not replace efficient hardware.

If you have more RAM than you need, this module will actually never create swap files and only be transferring calls to the system's memory functions.

11.3 Is there a way to signal when VM files start being used?

There used to be such a warning, it has been disabled. Notice that the upper limit for allocatable memory is rarely the amount of RAM + swap available on the machine. A first limit is set in the kernel, and a second one is set by the shell using 'limit' or 'ulimit'. Standard system tools can tell you when a process has reached this limit.

11.4 What about cleaning up if xmemory exits the program?

You can always make use of the `atexit()` system call to add cleaning functions, in case the process is terminated by **xmemory** and not by your own code.

11.5 Why is realloc not supported?

The main reason is that `realloc()` is rarely used in C programs. Since the cost of implementing such an operator is not negligible, I'd prefer to avoid it

altogether.

There are also some portability problems. Some platforms do not allow reallocation of a NULL pointer, which is sometimes useful to allocate growing lists without programming the first iteration specifically. To keep your program portable, do not use this feature. Another use of realloc is to free() a pointer, this is also not uniformly implemented.

It seems that the realloc() implementation is simply doing a `malloc+memcpy` on many platforms. See with your C library implementation, but the general perception in the C programming community is that this function is very often implemented as an alias for "allocate and copy".

Anyway, a `realloc()` call could be implemented in this module if the need arises. It is just that most module users could live without it until now, at least.

12 Further references

A good reference about `mmap()` is simply the source code for the Linux or BSD kernels. You will find there everything you always wanted to know about memory mappings and memory allocation. Having a look inside a C library to see how `malloc` and friends are implemented is also a good idea.

Good luck using **xmemory** , do not hesitate to send feedback or report bugs to the author.

`N. Devillard - May 2002`