MASARYK UNIVERSITY
FACULTY OF INFORMATICS

# Talloc - a hierarchical memory allocator

BACHELOR THESIS

**Pavel Březina**

Brno, spring 2012

## Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Pavel Březina

**Advisor:** RNDr. Jan Kasprzak

## Acknowledgement

I would like to thank my supervisor for his guidance, Ing. Jakub Hrozek for his great mentoring and RNDr. Anna Jonášová, Bc. Stephen Gallagher and Andreas Schneider for all their help.

# Abstract

The goal of this thesis is to create a comprehensive description of the talloc library, with code samples and a summary of the best practices of its usage. The thesis also aims to create a tutorial that will be published on the library homepage.

## Keywords

talloc, hierarchical, memory, manager, allocator

# Contents

# 1 Introduction

Although the usual amount of the main memory has grown dramatically over the decades, it is still one of the most critical system resources. Therefore, a good memory management is very important, especially in programs that are expected to be running permanently. This no longer applies only to drivers and server applications but also more and more desktop programs are used to be running for the computer uptime (web browsers, instant messaging clients, gaming platforms, etc.). In such programs a proper memory management is very important because even a small memory leak that occurs periodically may result in a much bigger memory consumption after several hours of the application runtime. Even though many popular programming languages contain a garbage collector that takes the memory management off the developer's shoulders, many programs are still written using a language such as C[4] where the heap memory is completely in the hands of the programmer.

## 1.1 Managing memory in C

Managing memory in the C language is a very difficult task. It is mainly due to the fact that the standard library does not provide any tools for simple deallocation of complex structures. The standard approach is to write a specific free function for every custom data type that goes down through the elements tree and frees them one by one from the bottom to the top. The problem is that it takes many lines of code and it is very likely that something is forgotten to be freed. Over the years, few libraries has been developed that aim to make the memory management in C easier and more error proof. Examples of this kind of libraries are memory pools from the APR[1] and halloc[2]. Another library, which is originally based upon halloc[7], is talloc.

Talloc[3] is a very mature and easy to use memory allocator written

1. Apache Portable Runtime: `http://apr.apache.org/`
2. Hierarchical memory allocator: `http://swapped.cc/halloc`
3. Talloc memory allocator: `http://talloc.samba.org`

in C, developed and maintained by the Samba[1] team. Since its first public release in 2009[7] it has become the main memory allocator in a few projects, including Samba, SSSD[2] and nfsim[3]. Unfortunately, the wider expansion of this library is being blocked by the lack of tutorials and a comprehensive description of its properties and characteristics.

## 1.2 Goals of this thesis

The goal of this thesis is to create a comprehensive description of the talloc features with code samples and a summary of the best practices of its usage, which can speed up the initial training of new developers of the SSSD. The thesis also aims to create a tutorial that will be published on the library homepage and that will hopefully help the talloc to spread to many more projects.

1. Windows interoperability suite: `http://www.samba.org`
2. System Security Services Daemon: `https://fedorahosted.org/sssd`
3. Netfilter simulation environment: `http://ozlabs.org/~jk/projects/nfsim`

# 2 Talloc - a hiearchical memory allocator

Talloc is a hierarchical, reference counted memory pool system with destructors. It is built atop the C standard library and it defines a set of utility functions that altogether simplifies allocation and deallocation of data, especially for complex structures that contain many dynamically allocated elements such as strings and arrays.

The main goals of this library are: removing the needs for creating a cleanup function for every complex structure, providing a logical organization of allocated memory blocks and reducing the likelihood of creating memory leaks in long-running applications. All of this is achieved by allocating memory in a hierarchical structure of talloc contexts such that deallocating one context recursively frees all of its descendants as well.

**Main features**

- An open source project
- A hierarchical memory model
- Natural projection of data structures into the memory space
- Simplifies memory management of large data structures
- Automatic execution of a destructor before the memory is freed
- Simulates a dynamic type system
- Implements a transparent memory pool

## 2.1 Prerequisites

This thesis uses talloc version 2.0.6 as a reference.

The current version of the talloc library can be obtained from the project homepage: `http://talloc.samba.org`. On UNIX-like operating systems, the source code may be compiled with the following commands:

```
$ cd <talloc>
$ ./configure
$ make
```

It is also available at least in the Fedora and Ubuntu distributions repositories and can be installed on those systems using:

```
# on Fedora
yum install libtalloc libtalloc-devel

# on Ubuntu
apt-get install libtalloc2 libtalloc-dev
```

Each example in this chapter expects that the `talloc.h` header file is included and the `talloc` library is properly linked against the binary.

Further, the error checking code is skipped in order to make the examples simpler and shorter. The full version of the examples with the error checks can be found on the attached DVD medium.

## 2.2 Talloc context

The talloc context is the most important part of this library for it is responsible for every single feature of this memory allocator. It is a logical unit which represents a memory space managed by talloc.

From the programmer's point of view, the talloc context is completely equivalent to a pointer that would be returned by the memory routines from the C standard library. This means that every context that is returned from the talloc library can be used directly in functions that do not use talloc internally. For example we can do the following:

```c
char *str1 = strdup("I am NOT a talloc context");
char *str2 = talloc_strdup(NULL, "I AM a talloc context");

printf("%d\n", strcmp(str1, str2) == 0);

free(str1);
talloc_free(str2); /* we can not use free() on str2 */
```

This is possible because the context is internally handled as a special fixed-length structure called talloc chunk. Each chunk stores context metadata followed by the memory space requested by the programmer. When a talloc function returns a context (pointer), it in fact returns a pointer to the user space portion of the talloc chunk. And when we want to manipulate with this context using talloc functions, the talloc library transforms the user-space pointer back to the starting address of the chunk. This is also the reason why we were unable to use `free(str2)` in the previous example – because `str2` does not point at the beginning of the allocated block of memory. This is illustrated on Figure 2.1.

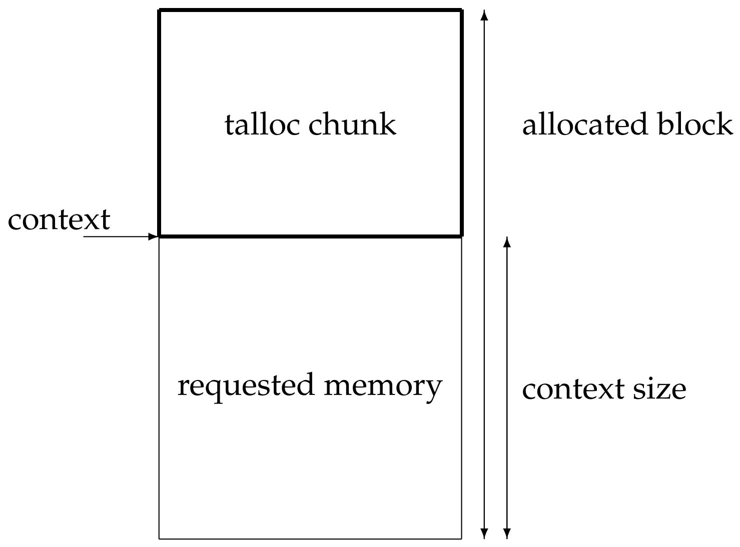

Figure 2.1: Talloc context

The type `TALLOC_CTX` is defined in `talloc.h` to identify a talloc context in function parameters. However, this type is just an alias for `void` and exists only for semantical reasons – thus we can differentiate between `void*` (arbitrary data) and `TALLOC_CTX*` (talloc context).

### Context meta data

Every talloc context carries several pieces of internal information along with the allocated memory:

- name – which is used in reports of context hierarchy (section 2.9) and to simulate a dynamic type system (section 2.6),

- size of the requested memory in bytes – this can be used to determine the number of elements in arrays,
- attached destructor – which is executed just before the memory block is about to be freed (section 2.7),
- references to the context – section 2.2.4,
- children and parent contexts – create the hierarchical view on the memory.

### 2.2.1 Hierarchy of talloc contexts

Every talloc context contains information about its parent and children. Talloc uses this information to create a hierarchical model of memory or to be more precise, it creates an n-ary tree where each node represents a single talloc context. The root node of the tree is referred to as a top level context – a context without any parent.

This approach has several advantages:

- as a consequence of freeing a talloc context, all of its children will be properly deallocated as well,
- the parent of a context can be changed at any time, which results in moving the whole subtree under another node,
- it creates a more natural way of managing data structures.

Let me illustrate this on an example. We have a structure that stores basic information about a user – his/her name, identification number and groups he/she is a member of:

```c
struct user {
  uid_t uid;
  char *username;
  size_t num_groups;
  char **groups;
};
```

Listing 2.1: struct user

Building of such a structure with talloc is very similar to the way it is done using C standard library. We have to allocate memory for every single element of `struct user`. The main difference is that we will specify the parent to which the new talloc context will be attached.

The following listing illustrates how it would be done with the C standard library:

```c
int i;
struct user *user = malloc(sizeof(struct user));
user->uid = 1000;
user->num_groups = N;

user->username = strdup("Test user");
user->groups = malloc(sizeof(char*) * user->num_groups);

for (i = 0; i < user->num_groups; i++) {
  user->groups[i] = asprintf("Test group %d", i);
}
```

Listing 2.2: Building struct user – C standard library

The following sample solves the same issue but using the talloc library. We will create a context tree that is illustrated on Figure 2.2.

```c
/* create new top level context */
struct user *user = talloc(NULL, struct user);

user->uid = 1000;
user->num_groups = N;

/* make user the parent of following contexts */
user->username = talloc_strdup(user, "Test user");
user->groups = talloc_array(user, char*, user->num_groups);

for (i = 0; i < user->num_groups; i++) {
  /* make user->groups the parent of following context */
  user->groups[i] = talloc_asprintf(user->groups,
                                    "Test group %d", i);
}
```

Listing 2.3: Building struct user – talloc library

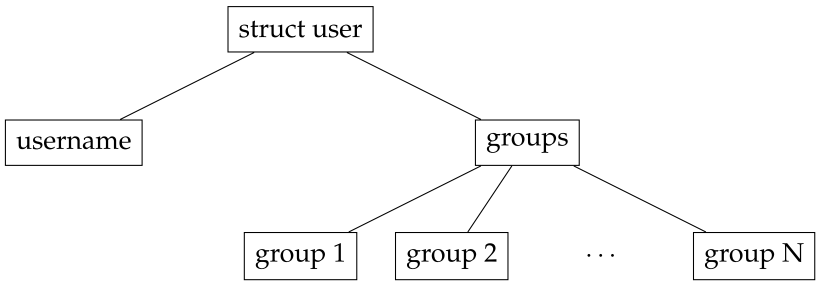

Figure 2.2: Context tree

This way, we have gained a lot of additional capabilities, one of which is very simple deallocation of the structure and all of its elements.

With the C standard library we need first to iterate over the array of groups and free every element separately. Then we must deallocate the array that stores them. Next we deallocate the username and as the last step free the structure itself.

```
int i;

for (i = 0; i < user->num_groups; i++) {
  free(user->groups[i]);
}

free(user->groups);
free(user->username);
free(user);
```

Listing 2.4: Freeing struct user – C standard library

But with talloc, the only operation we need to execute is freeing the structure context. Its descendants will be freed automatically.

```
talloc_free(user);
```

Listing 2.5: Freeing struct user – talloc library

### 2.2.2 Creating a new context

Creating a new talloc context means that we want to create a new talloc chunk (which stores the information about this context – especially its parent and children), allocate the desired amount of the system memory and retrieve a pointer to this memory.

Many functions exist that have the ability to create a new talloc context. This section describes only the most fundamental ones that deal with primitive data types and structures. Functions that are specialized in strings and arrays are described later in this text (sections 2.4 and 2.5).

These functions can be divided into four categories: those that allow us to set the name of the context, functions that create a zero-length context and type-safe and type-unsafe functions.

All of these functions share the following properties:

- they return a new talloc context or `NULL` if the system is out of memory,
- the first parameter is a talloc context which serves as a parent of the new context,
- the parent context can be either an existing context or `NULL`, which creates a new top level context.

### Type-safe functions that create a new context

Type-safe functions take as one of their parameters a data type we want to create. It allocates the size that is necessary for the given type and returns a new, properly-cast pointer. This is useful if we want to rely on the compiler to detect type mismatches.

Another feature of these functions is that they automatically set the name of the context to the name of the data type. We can find this behaviour very handy as it carries the type information even if we cast a variable to some other type. Then we can check the type during the runtime as described in section 2.6: Dynamic type system.

The appropriate functions with this characteristic are:

```
(#type)* talloc(TALLOC_CTX *ctx, #type)
(#type)* talloc_zero(TALLOC_CTX *ctx, #type)
```

The difference between `talloc()` and `talloc_zero()` is that the latter ensures the whole new memory space to be initialized with zeros.

```
struct user *user = talloc(ctx, struct user);

/* initialize to default values */
user->uid = 0;
```

```c
user->name = NULL;
user->num_groups = 0;
user->groups = NULL;

/* or we can achieve the same result with */
struct user *user_zero = talloc_zero(ctx, struct user);
```

Listing 2.6: talloc() and talloc_zero()

### Type-unsafe functions that create a new context

Type-unsafe functions take as a parameter the exact size we want to allocate instead of the type and return a pointer to `void`. The name of the context is set to the location in the source file where the function is invoked.

If we choose to use these functions we lose both the compile-time and the runtime ability to detect a type mismatch. Therefore, we should avoid using these routines unless we really want to retrieve a `void*`. The primary use for these functions is for creating allocation routines for other libraries that allow custom memory allocation. In those cases, we can not know ahead of time what structure will be passed in, so we are limited to using the size.

The following functions are type-unsafe variants of `talloc()` and `talloc_zero()`:

```c
void* talloc_size(TALLOC_CTX *ctx, size_t size)
void* talloc_zero_size(TALLOC_CTX *ctx, size_t size)
```

There is also one very interesting macro `talloc_ptrtype(ctx, ptr)` that may or may not be type-unsafe depending on the compiler. If the compiler is GCC of version greater than or equal to 3, it is type-safe (it uses the `__typeof__` feature of this compiler). Otherwise, it is type-unsafe.

As it is a wrapper around `talloc_size()`, the name of the context will be the location in the source file where the macro is used. This does not depend on whether it will be type-safe or type-unsafe during the compilation.

We can use it to shorten the notation if we do not need to carry the type information in the talloc context:

```c
struct verylongname *x = talloc_ptrtype(ctx, x);
// instead of
struct verylongname *x = talloc(ctx, struct verylongname);
```

Listing 2.7: talloc_ptrtype(ctx, ptr)

### Zero-length contexts

The zero-length context is basically a context without any special semantical meaning. We can use it the same way as any other context. The only difference is that it consists only of the meta data about the context. Therefore, it is strictly of type `TALLOC_CTX*`. It is often used in cases where we want to aggregate several data structures under one parent (zero-length) context, such as a temporary context to contain memory needed within a single function that is not interesting to the caller. Allocating on a zero-length temporary context will make clean-up of the function simpler.

`TALLOC_CTX* talloc_init(const char *fmt, ...)`
: Creates a new top level zero-length context with a custom name.

`TALLOC_CTX* talloc_new(TALLOC_CTX *ctx)`
: Creates a new zero-length context as a child of `ctx`. The name of the context will be the current location in the source file prefixed with `"talloc_new:␣"`.

```c
TALLOC_CTX *ctx = NULL;
struct foo *foo = NULL;
struct bar *bar = NULL;

/* new zero-length top level context */
ctx = talloc_new(NULL);
if (ctx == NULL) {
  return ENOMEM;
}

foo = talloc(ctx, struct foo);
bar = talloc(ctx, struct bar);
```

Listing 2.8: talloc_new()

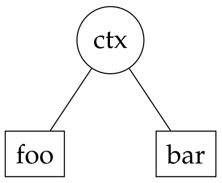

Figure 2.3: Context tree – zero-length context

### Contexts with custom name

A context with a custom name can be created using one of the following functions:

```
void* talloc_named(TALLOC_CTX *ctx, size_t size,
                   const char *fmt, ...)
void* talloc_named_const(TALLOC_CTX *ctx, size_t size,
                         const char *name)
```

However, there is not much practical use for setting a custom name. These two functions serves mainly as a generic background for the previous methods.

The only reasonable usage is to set a name of a context that is supposed to exist for a very long time (possibly for the life time of the application). The name will help us to identify such contexts during debugging.

### 2.2.3 Freeing a context

There are two functions defined that deal with deallocating a context. Both take a talloc context as their argument. If this context is `NULL` then no action is performed.

`int talloc_free(TALLOC_CTX *ctx)`
: Deallocates memory occupied by the context and recursively frees its children as well. The returned value is `0` on success, `-1` if `ctx` is `NULL` or if the destructor[1] attached to this context fails.

`void talloc_free_children(TALLOC_CTX *ctx)`
: Frees only the children of the context.

1. More information on destructors is in section 2.7: Using destructors

Besides these two functions we can find useful a macro that would automatically set the `ctx` to `NULL` to avoid accessing deallocated data. Such macro is already defined in `talloc.h`. The name is `TALLOC_FREE` and it is currently defined as:

```
#define TALLOC_FREE(ctx) do { \
  talloc_free(ctx);          \
  ctx = NULL;                \
} while(0);
```

Listing 2.9: TALLOC_FREE(ctx)

Talloc can automatically fill the memory with some predefined character just before the context is freed. We may want to do this to avoid reaccessing the data after it has been deallocted – for security reasons or to help us with debugging of our application.

This feature is disabled by default. To enable it, we have to set `TALLOC_FREE_FILL` environment variable. It should contain a numeric representation of the character we want to use. For example:

```
setenv("TALLOC_FREE_FILL", "0", 1);

/* the memory occupied by ctx will be filled with '\0' */
talloc_free(ctx);
```

Listing 2.10: Automatically fill the memory

### 2.2.4 Reference counting mechanism

Talloc provides API to create multiple references on a talloc context. This may be sometimes very handy, however the current stage of the implementation breaks the simplicity and hierarchy model of talloc. The SSSD goes so far as to strictly prohibit this feature and creates custom reference counters when needed.

The authors of the library are aware of the problem and the API is very likely going to be changed from the ground[3] up. When I asked about the use cases of a context with multiple references, the answer was: "Do not use it. It is very dangerous." For this reason I have decided not to describe the reference counter part in this thesis. Those who are interested can find more detail about this in the documentation[2] and the source code.

## 2.3 Stealing a context

Talloc has the ability to change the parent of a talloc context to another one. This operation is commonly referred to as stealing and it is one of the most important actions performed with talloc contexts.

Stealing a context is necessary if we want the pointer to outlive the context it is created on. This has many possible use cases, for instance stealing a result of a database search to an in-memory cache context, changing the parent of a field of a generic structure to a more specific one or vice-versa. The most common scenario, at least in SSSD, is to steal output data from a function-specific context to the output context given as an argument of that function – this is more deeply explained as one of the best practices in Section 2.10.2.

`void *talloc_steal(TALLOC_CTX *ctx, const void *ptr)`
: Changes the parent of the `ptr` to `ctx` and returns `ptr`.

`void *talloc_move(TALLOC_CTX *ctx, const void **ptr)`
: Changes the parent of the `ptr` to `ctx` and returns `ptr`. Additionally assigns `NULL` into `ptr`.

In general, the source pointer itself is not changed (it only replaces the parent in the meta data). But the common usage is that the result is assigned to another variable, thus further accessing the pointer from the original variable should be avoided unless it is necessary. In this case `talloc_move()` is the preferred way of stealing a context as it protects the pointer from being accidentally freed and accessed using the old variable after its parent has been changed.

```c
struct foo *foo = talloc_zero(ctx, struct foo);
foo->a1 = talloc_strdup(foo, "a1");
foo->a2 = talloc_strdup(foo, "a2");
foo->a3 = talloc_strdup(foo, "a3");

struct bar *bar = talloc_zero(NULL, struct bar);
/* change parent of foo from ctx to bar */
bar->foo = talloc_steal(bar, foo);

/* or do the same but assign foo = NULL */
bar->foo = talloc_move(bar, &foo);
```

Listing 2.11: talloc_steal() and talloc_move()

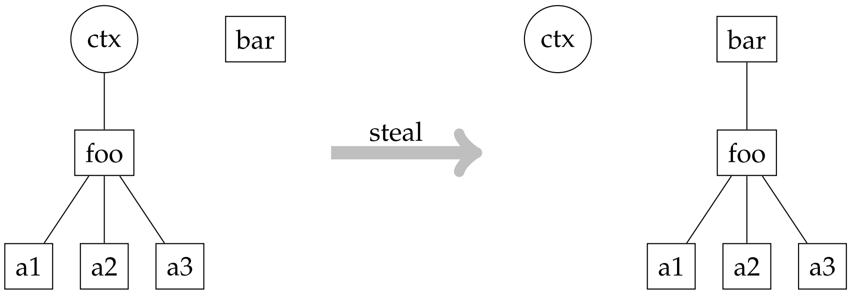

Figure 2.4: Stealing a talloc context

## 2.4 Working with strings

One of the most common issues that a programmer must take care of in the C programming language is the duplication and formatting of strings.

In the C standard library this is usually done with `strdup()` (string duplication), `strndup()` (string duplication with a length limit) and `sprintf()` (stores the formatted string in a preallocated buffer). The GNU extension of the C standard contains in addition a much nicer function for creating a formatted string called `asprintf()` that allocates all the necessary space automatically.

Talloc has adopted its own variants of not only `strdup()` and `strndup()` from the C standard library but also `asprintf()` and its non-variadic equivalent `vasprintf()`. As an addition to these basic routines it adds very useful alternatives that will append the result to the existing string or at the actual end of the talloc context.

All of these functions return a string that is also a valid talloc context which is named identically to the output string. The basic variants take as the first parameter an arbitrary talloc context which will serve as a parent to the newly created context. The extended `_append` variants take a talloc context that has to be a valid string to which the result is appended.

### 2.4.1 Duplicating a string

`char* talloc_strdup(TALLOC_CTX *ctx, const char *str)`
Creates a new talloc context as a child of `ctx` and copies `str` to this context. Returns the new string.

`char* talloc_strdup_append(char *dest, const char *str)`
Reallocates `dest` to the required length and concatenates `str` with the original `dest` string. Returns the new pointer to `dest`. This may change the pointer address of `dest`, so it is unsafe to rely on the old address after calling this function.

`char* talloc_strdup_append_buffer(char *dest, const char *s)`
This is similar to the previous function but it uses the size of the context to determine the `dest` length. The difference is more closer explained in the section 2.4.3.

The `strndup` variants have the same characteristics but they duplicate only `n` characters from `str`.

```
char* talloc_strndup(TALLOC_CTX *ctx, const char *str,
                     size_t n)
char* talloc_strndup_append(char *dest, const char *str,
                            size_t n)
char* talloc_strndup_append_buffer(char *dest, const char
                                   *str, size_t n)
```

### 2.4.2 Formatted string

Functions for formatted strings are very similar to the previous ones. The difference is that they take a printf-style format string and its parameters. Talloc defines both variadic and non-variadic variants.

Variadic:

```
char* talloc_asprintf(TALLOC_CTX *ctx, const char *fmt, ...)
char* talloc_asprintf_append(char *dest, const char *fmt,
                             ...)
char* talloc_asprintf_append_buffer(char *dest, const char
                                    *fmt, ...)
```

Non-variadic:

```
char* talloc_vasprintf(TALLOC_CTX *ctx, const char *fmt,
                       va_list ap)
char* talloc_vasprintf_append(char *dest, const char *fmt,
                              va_list ap)
```

```
char* talloc_vasprintf_append_buffer(char *dest, const char
                                     *fmt, va_list ap)
```

### 2.4.3 Difference between append and append_buffer

The behaviour of `_append` and `_append_buffer` is the same in the majority of cases. The latter one serves as the much more efficient way of appending a string as it does not calculate the length of the destination string. This performance trick takes effect in loops with many iterations. For example, we want to build an LDAP filter to search all users that are member of at least one group from the provided list:

```
char *filter = talloc_strdup(NULL, "(|");
for (i = 0; groups[i] != NULL; i++) {
  filter = talloc_asprintf_append(filter, "(group=%s)",
                                  groups[i]);
}
filter = talloc_strdup_append(filter, ")");
```

Listing 2.12: Appending a string

To simplify the example, consider that each group has a name that consists from $n$ characters and the list contains $m$ groups. What happens is that the length of both `filter` and `groups[i]` is computed every time.

The length of the string is determined by `strlen()`, that means it iterates over each character of the string and increments the character counter until it finds the terminating zero. This gives us (2.1) iterations in `strlen()`. For $n = 10, m = 100$ it is $92.905$ iterations. If we just replace the `_append` with `_append_buffer`, the equation changes to (2.2) and the result decreases to $1803$. This can be a big performance boost in applications that contain a lot pieces of similar code.

$$2 + \sum_{i=1}^{m}(2 + 8i + ni) + 2 + 8m + nm + 1 \tag{2.1}$$

$$2 + \sum_{i=0}^{m}(8 + n) + 1 \tag{2.2}$$

However, a great caution must be paid when `'\0'` is put in the middle of the string. In this case the two functions act differently:

```
char *str_a = talloc_strdup(NULL, "hello world");
char *str_b = talloc_strdup(NULL, "hello world");
str_a[5] = str_b[5] = '\0';

char *app = talloc_strdup_append(str_a, ", hello");
char *buf = talloc_strdup_append_buffer(str_b, ", hello");

printf("%s\n", app); /* hello, hello */
printf("%s\n", buf); /* hello (hello\0world, hello) */
```

Listing 2.13: Zero in the middle of a string

## 2.5 Arrays and talloc

The following sections describe the array interface and special properties that talloc implements. Namely how to create, manage and free an array and how to determine its length from the context size.

### 2.5.1 Creating a new array

We are already able to create an array with the knowledge introduced in the previous text. We can do it in the same way as in the C standard library using `talloc_size()` where the size of the context would be `sizeof(type)* number_of_elements`.

However, this has two disadvantages. First of all we have to use the type-unsafe functions to allocate the requested size which is unreliable. And second, we have to do the multiplication by ourselves. Talloc developers were aware of it and they have created a separate interface for arrays:

```
(#type)* talloc_array(TALLOC_CTX *ctx, #type,
                          unsigned int count)
(#type)* talloc_zero_array(TALLOC_CTX *ctx, #type
                               unsigned int count)
void* talloc_array_size(TALLOC_CTX *ctx, size_t size,
                            unsigned int count)
void* talloc_array_ptrtype(TALLOC_CTX *ctx, #ptrtype,
                               unsigned int count)
```

Or to reallocate the array use:

```
void* talloc_realloc(TALLOC_CTX *parent,
                     TALLOC_CTX *ctx,
                     #type, size_t count)
void* talloc_realloc_size(TALLOC_CTX *parent,
                          TALLOC_CTX *ctx,
                          size_t size)
```

If `ctx` is `NULL` a new context is created. If `count` or `size` is zero, it will free the context. The memory occupied by `ctx` may be moved to another location, therefore always replace it by the returned pointer.

We should always project the natural structure of an array into the talloc context hierarchy. That is, the array should be always used as a parent of its dynamically created elements.

```
struct user **users;
users = talloc_zero_array(NULL, struct user*, N + 1);

for (i = 0; i < N; i++) {
  users[i] = talloc_zero(users, struct user);
  users[i]->num_groups = i;
  users[i]->groups = talloc_zero_array(users[i], char*, i);

  for (j = 0; j < i; j++) {
    users[i]->groups[j] = talloc_asprintf(
                          users[i]->groups,"Group %d",j);
  }
}
```

Listing 2.14: Array of users and context hierarchy

This way we gain a lot of flexibility. We do not have to free every element separately to deallocate the whole array, we can achieve this simply by calling `talloc_free(users)`. Or we can move any part of the array to another parent context to keep it accessible even if we free the array later.

The context tree that graphically illustrates the hierarchy from the previous listing can be find on Figure 2.5.

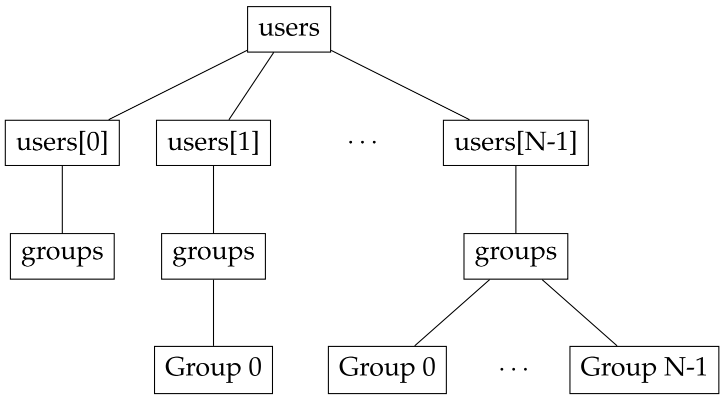

Figure 2.5: Context tree from Listing 2.14

### 2.5.2 Determining the length of an array

We can find ourselves in a situation when we need to iterate through an array but we are not able or we do not want to use a special delimiter to mark the end of the array. We need to store the size of the array in another variable and carry this information with the array.

Talloc makes this easier as every talloc context already contains the information about its size (in bytes). Therefore, instead of carrying the number of the elements separately, we can just divide the context size by the size of the data type the array consists of. Talloc has a macro `talloc_array_length(ctx)` that serves for this purpose.

```
TALLOC_CTX *ctx = talloc_new(NULL);
int *i = talloc_zero(ctx, int);
char *str = talloc_strdup(ctx, "hello world");
int *array = talloc_array(ctx, int, 10);

printf("%lu\n", talloc_array_length(ctx));   /*  0 */
printf("%lu\n", talloc_array_length(i));     /*  1 */
printf("%lu\n", talloc_array_length(str));   /* 12 */
printf("%lu\n", talloc_array_length(array)); /* 10 */

talloc_free(ctx);
```

Listing 2.15: Length of an array

It can be also used as a more efficient way of computing a number of characters in a string that has been created with the talloc library.

However, we need to create our own function (or macro) to do this. Although we must be very careful with the usage as it will misbehave once we put a `'\0'` character inside the string, having that on mind we will gain the constant complexity. For example:

```
size_t talloc_strlen(const char *str)
{
  return talloc_array_length(str) - 1;
}

char *str = talloc_strdup(ctx, "hello world");
char *str2 = talloc_strdup(ctx, "hello world");
str2[5] = '\0';

printf("%lu\n", talloc_strlen(str));    /* 11 */
printf("%lu\n", talloc_strlen(str2));   /* 11 */
printf("%lu\n", strlen(str2));          /* 5! */
```

Listing 2.16: Length of a string

## 2.6 Dynamic type system

Generic programming in the C language is very difficult. There is no inheritance nor templates known from object oriented languages. There is no dynamic type system. Therefore, generic programming in this language is usually done by type-casting a variable to `void*` and transferring it through a generic function to a specialized callback as illustrated on the next listing.

```
void generic_function(callback_fn cb, void *pvt)
{
  /* do some stuff and call the callback */
  cb(pvt);
}

void specific_callback(void *pvt)
{
  struct specific_struct *data;
  data = (struct specific_struct*)pvt;
  /* ... */
}

void specific_function()
```

```
{
  struct specific_struct data;
  generic_function(callback, &data);
}
```

Listing 2.17: Generic programming pattern

Unfortunately, the type information is lost as a result of this type cast. The compiler cannot check the type during the compilation nor are we able to do it at runtime. Providing an invalid data type to the callback will result in unexpected behaviour (not necessarily a crash) of the application. This mistake is usually hard to detect because it is not the first thing on the mind of the developer.

As we already know, every talloc context contains a name. This name is available at any time and it can be used to determine the type of a context even if we lose the type of a variable.

### 2.6.1 Typed context

Although the name of the context can be set to any arbitrary string, the best way of using it to simulate the dynamic type system is to set it directly to the type of the variable.

It is recommended to use one of `talloc()` and `talloc_array()` (or its variants) to create the context as they set its name to the name of the given type automatically.

If we have a context with such a name, we can use two similar functions that do both the type check and the type cast for us:

```
(#type)* talloc_get_type(TALLOC_CTX *ctx, #type)
(#type)* talloc_get_type_abort(TALLOC_CTX *ctx, #type)
```

Both functions take as the first parameter a valid talloc context and as the second parameter the type which the context is supposed to be. If the provided type equals to the context name, they return a properly-cast pointer to this type.

They differ in the other scenario – when the context name is different. The first one simply returns `NULL`. The other one logs this violation into a talloc log (see section 2.9) and immediately calls an abort function. By default it kills the application with `abort()` but this behaviour can be modified by setting our own abort function with:

```
void talloc_set_abort_fn(
  void (*abort_fn)(const char *reason)
)
```

### 2.6.2 Arbitrary name check

If we want to check the context against some custom name, we can do this using:

```
void* talloc_check_name(TALLOC_CTX *ctx, const char *name)
```

The behaviour is the same as of `talloc_get_type()` but it does not perform the type cast of the returned pointer.

### 2.6.3 Examples

Below given is an example of generic programming with talloc – if we provide invalid data to the callback, the program will be aborted. This is a sufficient reaction for such an error in most applications.

```
void foo_callback(void *pvt)
{
  struct foo *data = talloc_get_type_abort(pvt, struct foo);
  /* ... */
}

int do_foo()
{
  struct foo *data = talloc_zero(NULL, struct foo);
  /* ... */
  return generic_function(foo_callback, data);
}
```

Listing 2.18: Dynamic type system #1

But what if we are creating a service application that should be running for the uptime of a server? We may want to abort the application during the development process (to make sure the error is not overlooked) but try to recover from the error in the customer release. This can be achieved by creating a custom abort function with a conditional build.

```c
void my_abort(const char *reason)
{
  fprintf(stderr, "talloc abort: %s\n", reason);
#ifdef ABORT_ON_TYPE_MISMATCH
  abort();
#endif
}

void init()
{
  talloc_set_abort_fn(my_abort);
}
```

Listing 2.19: Custom abort function

The usage of `talloc_get_type_abort()` would be then:

```c
init();

TALLOC_CTX *ctx = talloc_new(NULL);
char *str = talloc_get_type_abort(ctx, char);
if (str == NULL) {
  /* recovery code */
}
/* talloc abort: ../src/main.c:25: Type mismatch:
   name[talloc_new: ../src/main.c:24] expected[char] */
```

Listing 2.20: Sample output

## 2.7 Using destructors

Destructors are well known methods in the world of object oriented programming. A destructor is a method of an object that is automatically run when the object is destroyed. It is usually used to return resources taken by the object back to the system (e.g. closing file descriptors, terminating connection to a database, deallocating memory).

With talloc we can take the advantage of destructors even in C. We can easily attach our own destructor to a talloc context. When the context is freed, the destructor is run automatically.

### 2.7.1 Destructor prototype

Destructor in talloc is a function with following prototype:

```
int destructor(void *ctx)
```

The parameter `ctx` is the talloc context that is about to be freed. The proper type can be retrieved with `talloc_get_type()`. The destructor is type-safe during the compilation if we use `GCC 3` or newer (it uses the `__typeof__` feature of this compiler). With this compiler we can avoid using `void*` and the parameter can be any valid pointer type.

The destructor should return `0` for succes and `-1` on failure. If it returns `-1` then `talloc_free()` fails and the memory of the context and its children is not deallocated.

### 2.7.2 Managing destructors

The following function is used to attach a destructor to a talloc context:

```
void talloc_set_destructor(TALLOC_CTX *ctx,
                           int(*)(void *) destructor)
```

Every context can have one destructor attached as a maximum. If the function is called more than once on the same context, the current destructor is overwritten. To detach the destructor from the context we shall simply set `NULL` as the new destructor.

### 2.7.3 Child contexts with destructors

If we attempt to free a talloc context (`ctx`) that has one or more children with a destructor attached, we may encounter a strange behaviour. If a destructor of `ctx` fails, `talloc_free(ctx)` fails as well and there will be no attempt to free the children of `ctx`.

But if the destructor of `ctx` succeeds, `talloc_free(ctx)` succeeds too and the memory of `ctx` is deallocated even if destructor on one of the children fails.

There has been a discussion on the samba-technical mailing list that addresses this destructor characteristics. The conclusion of this discussion is, that returning `-1` from the destructor means that there is a very good reason to reject deallocation of the context and all of its children[1].

### 2.7.4 Example

Imagine that we have a dynamically created linked list. Before we deallocate an element of the list, we need to make sure that we have successfully removed it from the list. Normally, this would be done by two commands in the exact order: remove it from the list and then free the element. With talloc, we can do this at once by setting a destructor on the element which will remove it from the list and `talloc_free()` will do the rest.

The destructor would be:

```
int list_remove(void *ctx)
{
    struct list_el *el = NULL;
    el = talloc_get_type_abort(ctx, struct list_el);
    /* remove element from the list */
}
```

Listing 2.21: Remove an element from the list – destructor

The GCC 3 or newer can check for the types during the compilation. So if it is our major compiler, we can use a little bit nicer destructor:

```
int list_remove(struct list_el *el)
{
    /* remove element from the list */
}
```

Listing 2.22: Remove an element from the list – type-safe destructor

Now we will assign the destructor to the list element. We can do this directly in the function that inserts it.

```
struct list_el* list_insert(TALLOC_CTX *mem_ctx,
                            struct list_el *where,
                            void *ptr)
{
  struct list_el *el = talloc(mem_ctx, struct list_el);
  el->data = ptr;
  /* insert into list */

  talloc_set_destructor(el, list_remove);
  return el;
}
```

Listing 2.23: Remove an element from the list when freed

Because talloc is a hierarchical memory allocator, we can go a step further and free the data with the element as well:

```
struct list_el* list_insert_free(TALLOC_CTX *mem_ctx,
                                 struct list_el *where,
                                 void *ptr)
{
  struct list_el *el = NULL;
  el = list_insert(mem_ctx, where, ptr);

  talloc_steal(el, ptr);

  return el;
}
```

Listing 2.24: Free the data with the list element

## 2.8 Decreasing number of malloc() calls

Allocation of a new memory is an expensive operation and large programs can contain thousands of calls of `malloc()` for a single computation, where every call allocates only a very small amount of the memory. This can result in an undesirable slowdown of the application. We can avoid this slowdown by decreasing the number of `malloc()` calls by using a memory pool.

A memory pool is a preallocated memory space with a fixed size. If we need to allocate new data we will take the desired amount of the memory from the pool instead of requesting a new memory from the system. This is done by creating a pointer that points inside the preallocated memory. Such a pool must not be reallocated as it would change its location – pointers that were pointing inside the pool would become invalid. Therefore, a memory pool requires a very good estimate of the required memory space.

The talloc library contains its own implementation of a memory pool. It is highly transparent for the programmer. The only thing that needs to be done is an initialization of a new pool context using `talloc_pool()`[1]– which can be used in the same way as any other context.

1. `TALLOC_CTX *talloc_pool(TALLOC_CTX *ctx, size_t size)`

Refactoring of existing code (that uses talloc) to take the advantage of a memory pool is quite simple due to the following properties of the pool context:

- if we are allocating data on a pool context, it takes the desired amount of memory from the pool,
- if the context is a descendant of the pool context, it takes the space from the pool as well,
- if the pool does not have sufficient portion of memory left, it will create a new non-pool context, leaving the pool intact

```
/* allocate 1KiB in a pool */
TALLOC_CTX *pool_ctx = talloc_pool(NULL, 1024);

/* take 512B from the pool, 512B is left there */
void *ptr = talloc_size(pool_ctx, 512);

/* 1024B > 512B, this will create new talloc chunk outside
   the pool */
void *ptr2 = talloc_size(ptr, 1024);

/* the pool still contains 512 free bytes
 * this will take 200B from them */
void *ptr3 = talloc_size(ptr, 200);

/* this will destroy context 'ptr3' but the memory
 * is not freed, the available space in the pool
 * will increase to 512B */
talloc_free(ptr3);

/* this will free memory taken by 'pool_ctx'
 *  and 'ptr2' as well */
talloc_free(pool_ctx);
```

Listing 2.25: Talloc pool

The above given is very convenient, but there is one big issue to be kept in mind. If the parent of a talloc pool child is changed to a parent that is outside of this pool, the whole pool memory will not be freed until the child is freed. For this reason we must be very careful when stealing a descendant of a pool context.

```
TALLOC_CTX *mem_ctx = talloc_new(NULL);
TALLOC_CTX *pool_ctx = talloc_pool(NULL, 1024);
struct foo *foo = talloc(pool_ctx, struct foo);

/* mem_ctx is not in the pool */
talloc_steal(mem_ctx, foo);

/* pool_ctx is marked as freed but the memory is not
   deallocated, accessing the pool_ctx again will cause
   an error */
talloc_free(pool_ctx);

/* now is pool_ctx deallocated */
talloc_free(mem_ctx);
```

Listing 2.26: Stealing from pool context

It may be often better to copy the memory we want to steal to avoid this problem. If we do not need to retain the context name (to keep the type information), we can use `talloc_memdup()` to do this:

```
TALLOC_CTX *talloc_memdup(TALLOC_CTX *parent,
                          TALLOC_CTX *ctx, size_t size)
```

Creates a new context and copies `size` bytes from `ctx` into it. The name of the context will be the location in the source code where is this function called.

Copying the memory out of the pool may, however, discard all the performance boost given by the pool, depending on the size of the copied memory. Therefore, the code should be well profiled before taking this path. In general, the golden rule is: if we need to steal from the pool context, we should not use a pool context.

## 2.9 Debugging options

Although talloc makes memory management significantly easier than the C standard library, developers are still only humans and can make mistakes. Therefore, it can be handy to know some tools for the inspection of talloc memory usage.

### 2.9.1 Talloc log and abort

We have already encountered the abort function in section 2.6. In that case it was used when a type mismatch was detected. However, talloc calls this abort function in several more situations:

- when the provided pointer is not a valid talloc context,
- when the meta data is invalid – probably due to memory corruption,
- and when an access after free is detected.

The third one is probably the most interesting. It can help us with detecting an attempt to double-free a context or any other manipulation with it via talloc functions (using it as a parent, stealing it, etc.).

Before the context is freed talloc sets a flag in the meta data. This is then used to detect the access after free. It basically works on the assumption that the memory stays unchanged (at least for a while) even when it is properly deallocated. This will work even if the memory is filled with the value specified in `TALLOC_FREE_FILL` environment variable, because it fills only the data part and leaves the meta data intact.

Apart from the abort function, talloc uses a log function to provide additional information to the aforementioned violations. To enable logging we shall set the log function with one of:

```
void talloc_set_log_fn(void (*fn)(const char *message))
void talloc_set_log_stderr()
```

The latter one is a shortcut for a common scenario – it will print the `message` in the standard error stream.

Below given is an sample output of accessing a context after it has been freed:

```
talloc_set_log_stderr();
TALLOC_CTX *ctx = talloc_new(NULL);

talloc_free(ctx);
talloc_free(ctx);

results in:
```

```
talloc: access after free error - first free may be at ../
    src/main.c:55
Bad talloc magic value - access after free
```

Listing 2.27: Talloc error – access after free

Another example below is an example of the invalid context:

```
talloc_set_log_stderr();
TALLOC_CTX *ctx = talloc_new(NULL);
char *str = strdup("not a talloc context");
talloc_steal(ctx, str);

results in:
Bad talloc magic value - unknown value
```

Listing 2.28: Talloc error – not a talloc context

### 2.9.2 Memory usage reports

Talloc can print reports of memory usage of specified talloc context to a file (or to `stdout` or `stderr`). The report can be simple or full. The simple report provides information only about the context itself and its direct descendants. The full report goes recursively through the entire context tree.

```
void talloc_report(TALLOC_CTX *ctx, FILE *file)
void talloc_report_full(TALLOC_CTX *ctx, FILE *file)
```

We will use the following code to retrieve the sample report:

```
struct foo {
  char *str;
};

TALLOC_CTX *ctx = talloc_new(NULL);
char *str =  talloc_strdup(ctx, "my string");
struct foo *foo = talloc_zero(ctx, struct foo);
foo->str = talloc_strdup(foo, "I am Foo");
char *str2 = talloc_strdup(foo, "Foo is my parent");

/* print full report */
talloc_report_full(ctx, stdout);
```

Listing 2.29: Full report

It will print a full report of `ctx` to the standard output. The message should be similar to:

```
full talloc report on 'talloc_new: ../src/main.c:82' (total 46 bytes in 5 blocks)
  struct foo contains 34 bytes in 3 blocks (ref 0) 0x1495130
    Foo is my parent contains 17 bytes in 1 blocks (ref 0) 0x1495200
    I am Foo contains 9 bytes in 1 blocks (ref 0) 0x1495190
  my string contains 10 bytes in 1 blocks (ref 0) 0x14950c0
```

Listing 2.30: Sample output of talloc full report

We can notice in this report that something is wrong with the context containing `struct foo`. We know that the structure has only one string element. However, we can see in the report that it has two children. This indicates that we have either violated the memory hierarchy or forgotten to free it as temporary data. Looking into the code, we can see that `"Foo is my parent"` should be attached to `ctx`.

The following functions affect the behaviour of talloc reports. To get correct results, they should be called before any other routine from the talloc library.

**void** `talloc_enable_null_tracking(`**void**`)`
**void** `talloc_disable_null_tracking(`**void**`)`

Enables/disables tracking of talloc contexts without a parent. If it is enabled, we can use `NULL` as a context for the report and it will print the summary of all available contexts.

**void** `talloc_enable_leak_report(`**void**`)`
**void** `talloc_enable_leak_report_full(`**void**`)`

Enables an automatic simple/full report of all contexts at the program exit. The report is printed in `stderr`.

### 2.9.3 Detecting memory leaks

We frequently need to provide a talloc context as a parameter to a function, so this function can use it as a parent context for output values. But sometimes a programmer makes a mistake and allocates something more on the output context than he is supposed to.

This issue can not be usually seen in talloc reports, but fortunately it can be tested automatically with the knowledge of the total size (size of a context and its children) of the parent context.

```
/* creates new talloc context *_foo as a child of mem_ctx */
int struct_foo_init(TALLOC_CTX *mem_ctx, struct foo **_foo);

int ret;
size_t total_size;
struct foo *foo = NULL;
TALLOC_CTX *ctx = talloc_new(NULL);

total_size = talloc_total_size(ctx)
ret = struct_foo_init(test_ctx, &foo);
talloc_free(foo);
if (total_size != talloc_total_size(ctx)) {
  /* struct_foo_init() allocated on 'ctx'
     more than 'foo' */
}
```

Listing 2.31: Memory leaks detection #1

However, such checks are not suitable for an application itself as we have to free `foo` before other usage of the parent context. Therefore, the proper place for this is in your test suit, possibly in unit tests.

The SSSD uses a set of functions[1], the goal of which is to detect memory leaks on a talloc context in the unit tests created with the Check[2] library (but it can be easily modified for other test suites). It does it in a more advanced manner than the previous example introduced. It works on a push/pop basis, where the unit test will fail if a context has after the pop operation a different size than it had during the push operation.

```
/* creates new talloc context *_foo as a child of mem_ctx */
int struct_foo_init(TALLOC_CTX *mem_ctx, struct foo **_foo);

/* create new unit test with Check library */
START_TEST(test_struct_foo_init)
{
  int ret;
  struct foo *foo = NULL;
  TALLOC_CTX *test_ctx = talloc_new(NULL);
  fail_if(test_ctx == NULL); /* if the condition is met,
                                the unit test will fail */

  /* initialize the leak check routines */
```

1. Available on the attached DVD
2. `http://check.sourceforge.net`

```
  leak_check_setup();

  /* remember current test_ctx total size */
  check_leaks_push(test_ctx);

  ret = struct_foo_init(test_ctx, &foo);
  fail_if(ret != EOK)

  /* ... test foo values ... */

  talloc_free(foo);

  /* check for memory leak,
     this will fail the test if struct_foo_init()
     allocated more than 'foo' on test_ctx */
  check_leaks_pop(test_ctx);

  talloc_free(test_ctx);

  /* clean up after the leak check routines */
  leak_check_teardown();[1]
}
```

Listing 2.32: Memory leaks detection #2

## 2.10 Best practices

The following sections contain several best practices and good manners that were found by the Samba and SSSD developers over the years. These will help you to write better code, easier to debug and with as few (hopefully none) memory leaks as possible.

### 2.10.1 Keep the context hierarchy steady

The talloc is a hierarchy memory allocator. The hierarchy nature is what makes the programming more error proof. It makes the memory easier to manage and to free. Therefore, the first thing we should have on our mind is: always project our data structures into the talloc context hierarchy.

1. In the Check library, setup and teardown functions can be run automatically with `tcase_add_checked_fixture()`

That means if we have a structure, we should always use it as a parent context for its elements. This way we will not encounter any troubles when freeing this structure or when changing its parent. The same rule applies for arrays.

For example, the structure `user` which is defined in the Listing 2.1 (page 8) should be created with the context hierarchy illustrated on Figure 2.6.

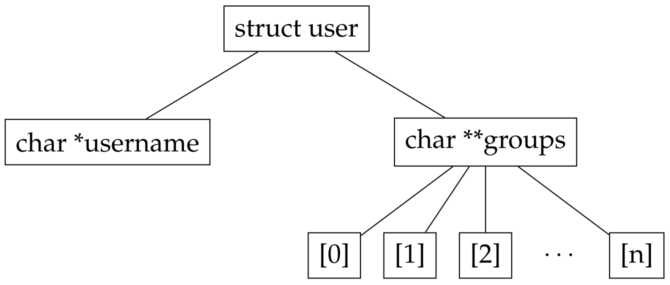

Figure 2.6: Context tree of struct user

### 2.10.2 Every function should use its own context

It is a good practice to create a temporary talloc context at the function beginning and free this context just before the return statement. All the data must be allocated on this context or on its children. This ensures that no memory leaks are created as long as we do not forget to free the temporary context.

This pattern applies to both situations - when a function does not return any dynamically allocated value and when it does. However, it needs a little extension for the latter case.

#### Functions that do not return any dynamically allocated value

If the function does not return any value created on the heap, we will just obey the aforementioned pattern.

```
int bar()
{
  int ret;
  TALLOC_CTX *tmp_ctx = talloc_new(NULL);
  if (tmp_ctx == NULL) {
    ret = ENOMEM;
    goto done;
  }
  /* allocate data on tmp_ctx or on its descendants */
  ret = EOK;
done:
  talloc_free(tmp_ctx);
  return ret;
}
```

Listing 2.33: Temporary context #1

### Functions returning dynamically allocated values

If our function returns any dynamically allocated data, its first parameter should always be the destination talloc context. This context serves as a parent for the output values. But again, we will create the output values as the descendants of the temporary context. If everything goes well, we will change the parent of the output values from the temporary to the destination talloc context.

This pattern ensures that if an error occurs (e.g. I/O error or insufficient amount of the memory), all allocated data is freed and no garbage appears on the destination context.

```
int struct_foo_init(TALLOC_CTX *mem_ctx, struct foo **_foo)
{
  int ret;
  struct foo *foo = NULL;
  TALLOC_CTX *tmp_ctx = talloc_new(NULL);
  if (tmp_ctx == NULL) {
    ret = ENOMEM;
    goto done;
  }
  foo = talloc_zero(tmp_ctx, struct foo);
  /* ... */
  *_foo = talloc_steal(mem_ctx, foo);
  ret = EOK;
done:
  talloc_free(tmp_ctx);
  return ret;
}
```

Listing 2.34: Temporary context #2

### 2.10.3 Allocate temporary contexts on NULL

As it can be seen on the Listing 2.34, instead of allocating the temporary context directly on `mem_ctx`, we created a new top level context using `NULL` as the parameter for `talloc_new()` function. Take a look at the following example:

```
 1 char * create_user_filter(TALLOC_CTX *mem_ctx,
 2                           uid_t uid, const char *username)
 3 {
 4   char *filter = NULL;
 5   char *sanitized_username = NULL;
 6   /* tmp_ctx is a child of mem_ctx */
 7   TALLOC_CTX *tmp_ctx = talloc_new(mem_ctx);
 8   if (tmp_ctx == NULL) {
 9     return NULL;
10   }
11
12   sanitized_username = sanitize_string(tmp_ctx, username);
13   if (sanitized_username == NULL) {
14     talloc_free(tmp_ctx);
15     return NULL;
16   }
17
18   filter = talloc_aprintf(tmp_ctx,"(|(uid=%llu)(uname=%s))",
19                           uid, sanitized_username);
20   if (filter == NULL) {
21     return NULL; /* tmp_ctx is not freed */
22   }
23
24   /* filter becomes a child of mem_ctx */
25   filter = talloc_steal(mem_ctx, filter);
26   talloc_free(tmp_ctx);
27   return filter;
28 }
```

Listing 2.35: Temporary context #3

We forgot to free `tmp_ctx` before the `return` statement on line 21. However, it is created as a child of `mem_ctx` context and as such it will be freed as soon as the `mem_ctx` is freed. Therefore, no detectable memory leak is created.

On the other hand, we do not have any way to access the allocated data and for all we know `mem_ctx` may exist for the lifetime of our application. For these reasons this should be considered as a memory leak. How can we detect if it is unreferenced but still attached to its parent context? The only way is to notice the mistake in the source code.

But if we create the temporary context as a top level context, it will not be freed and memory diagnostic tools (e.g. `valgrind`[1]) are able to do their job.

### 2.10.4 Temporary contexts and the talloc pool

If we want to take the advantage of the talloc pool but also keep to the pattern introduced in the previous section, we are unable to do it directly. The best thing to do is to create a conditional build where we can decide how do we want to create the temporary context. For example, we can create the following macros:

```
#ifdef USE_POOL_CONTEXT
  #define CREATE_POOL_CTX(ctx, size) talloc_pool(ctx, size)
  #define CREATE_TMP_CTX(ctx)        talloc_new(ctx)
#else
  #define CREATE_POOL_CTX(ctx, size) talloc_new(ctx)
  #define CREATE_TMP_CTX(ctx)        talloc_new(NULL)
#endif
```

Listing 2.36: Conditional temporary context macros

Now if our application is under development, we will build it with macro `USE_POOL_CONTEXT` undefined. This way, we can use memory diagnostic utilities to detect memory leaks.

The release version will be compiled with the macro defined. This will enable pool contexts and therefore reduce the `malloc()` calls, which will end up in a little bit faster processing.

```
int struct_foo_init(TALLOC_CTX *mem_ctx, struct foo **_foo)
{
  int ret;
  struct foo *foo = NULL;
  TALLOC_CTX *tmp_ctx = CREATE_TMP_CTX(mem_ctx);
  /* ... */
}

errno_t handle_request(TALLOC_CTX mem_ctx)
{
```

1. `http://valgrind.org`

```
  int ret;
  struct foo *foo = NULL;
  TALLOC_CTX *pool_ctx = CREATE_POOL_CTX(NULL, 1024);
  ret = struct_foo_init(mem_ctx, &foo);
  /* ... */
}
```

Listing 2.37: Conditional temporary context

# 3 Conclusion

Although I have used the available documentation as a starting point, my primary source of knowledge was the talloc source code. This gave me a very deep understanding of the library and it also taught me some nice C and GCC tricks.

As a result of my work I have managed to send several patches to the samba-technical mailing list, which improve the talloc doxygen documentation. Those patches have already become a part of the Samba upstream and the new documentation is available online on the project homepage since 18th April 2012.

I have also created the tutorial for the talloc library. This tutorial is written in the doxygen[1] format, thus it can nicely fit into the existing online documentation. It has been published on the project homepage.

As one of the developers of the SSSD, I have introduced to the team a few interesting features that can give the product a better performance (talloc pools, `buffer` version of string concatenation functions) or that could help us to avoid type mistakes (custom abort function, `talloc_get_type_abort()`).

I have compiled the best practices from my one-year experience with the development of the SSSD and from many discussions with experienced Red Hat developers and Samba team members.

As an open source enthusiast, I am glad that I have been able to contribute to this great open source project and help it to become more accessible.

1. www.doxygen.org

# A Contents of the attached DVD medium

- `/csv` – measured performance in csv format
- `/documentation` – built talloc documentation and tutorial
- `/documents/lshw` – attributes of the workstation where the performance tests were executed
- `/documents/results.ods` – spreadsheet that contains the measured results of the performance tests
- `/examples` – contains the examples and performance tests introduced in this thesis
- `/patches` – contains patches that were applied to the Samba upstream
- `/sssd` – contains memory leak check functions from the SSSD
- `/thesis.pdf` – this document

# B Performance tests results

The official documentation says that talloc is about 4 % slower than malloc[6]. However, it does not specify the testing environment and measuring methodology. Therefore, I was eager to try it myself and I have prepared several test cases: basic memory allocation, array allocation, memory reallocation and string concatenation.

## B.1 Environment

Basic information:

| | |
|---|---|
| CPU: | Intel(R) Xeon(R) CPU W3550 @ 3.07GHz |
| Main memory: | 2GB DDR3 ECC Synchronous 1333 MHz |
| OS: | Fedora 15 x86_64 |
| glibc: | glibc-2.14.1-6.x86_64 |
| libtalloc: | libtalloc-2.0.6-1.fc15.x86_64 |

Dump of `lshw` program is stored on the attached DVD at `/documents /lshw`.

## B.2 Methodology

I have used the system tap probes to retrieve the time needed to perform the requested operations. These probes measure the time span between the enter to and the return from the test function with microsecond precision. Each test case has been performed for a hundred times. I have computed a median from all the measured numbers to eliminate undesirable random fluctuations.

All the tests were originally run in the same binary. However, this approach was generating invalid results, mainly in calloc and realloc tests – they were faster then malloc. And talloc equivalents that were executed at last were faster then these tests. Probably the `glibc` is performing some sort of optimization in these functions. Therefore, I have split the test into separated programs to get more accurate results.

To make the testing automatic and easy to reproduce, I have created several shell scripts. The output of these scripts is in the format of comma separated values. The header line consists of names of the measured functions. The values represent a time span in microseconds that was required to run the function.

## B.3 Test suite

The test suite consists of:

- Memory allocation (malloc)
- Array allocation (calloc)
- Memory reallocation (realloc)
- String concatenation (strcat)

The source codes of these tests can be found on the attached DVD at `<dvd>/examples/src/performance`.

The tests can be run separately using:

```
sudo bash <dvd>/examples/scripts/perftest-malloc.sh \
     loops size pool-guess testcount > output_file.csv

sudo bash <dvd>/examples/scripts/perftest-calloc.sh \
     loops array-length testcount > output_file.csv

sudo bash <dvd>/examples/scripts/perftest-realloc.sh \
     loops increment testcount > output_file.csv

sudo bash <dvd>/examples/scripts/perftest-concat.sh \
     array-length string pool-guess testcount \
     > output_file.csv
```

- `testcount` – how many times is the test binary executed
- `loops` – how many times is the tested function called
- `array-length` – how many elements should be stored in the array
- `pool-guess` – what size should be used for the `talloc_pool()`

- `string` – each element of the array in the concatenation test is a duplication of this string

The tests can be also executed all at once with the same parameters as I have used with the following command:

```
sudo bash <dvd>/examples/scripts/perftest-run.sh PREFIX
```

It stores the measured values in `<PREFIX><testcase>`.

## B.4 Conclusion

The official speed test run on my computer says that the talloc is about 12 % slower than malloc. The results I have measured with my test suite cannot be simply summarized on a few lines. Therefore, I have prepared a spreadsheet that contains a lot of computed data and charts, which I believe are for this case much better than words. The document is located at `<dvd>/documents/results.ods`.

Probably the most surprising part of these results is the speed of `strncat()`. Concatenation of one hundred thousand ten-character strings took this function more than twenty seconds, where `strcat()` managed to do it in three and a half seconds and `memcpy()` made it in less than one thousandth seconds. The talloc append buffer function was very close to `memcpy()` and in some situations suprisingly even little bit faster.

Statistically speaking, the talloc is much slower than the glibc allocator. However, even when the tests use values that can be hardly achieved in most applications, it still makes the difference only in miliseconds precision. Therefore it should not be taken as a point for not picking the talloc for your next project.

# Bibliography

[1] *Discussion of destructor processing*. [online]. URL: https://lists.samba.org/archive/samba-technical/2012-May/083308.html (visited on 05/05/2012).

[2] *Multiple references - documentation*. [online]. URL: http://talloc.samba.org/talloc/doc/html/group__talloc__ref.html (visited on 01/19/2012).

[3] *[RFC] Making talloc_reference() safer*. [online]. URL: https://lists.samba.org/archive/samba-technical/2011-October/080045.html (visited on 01/19/2012).

[4] Samuel P. Harbison, Guy L. Steele. *C: A Reference Manual*. 5th ed. Prentice Hall PTR, 2002. ISBN: 978-0130895929.

[5] Stephen Gallagher. *Why you should use talloc for your next project*. [online]. URL: http://sgallagh.wordpress.com/2010/03/17/why-you-should-use-talloc-for-your-next-project/ (visited on 01/19/2012).

[6] *Talloc - documentation*. [online]. URL: http://talloc.samba.org/talloc/doc/html/index.html (visited on 01/19/2012).

[7] *talloc questions - bachelor thesis*. [online]. URL: http://lists.samba.org/archive/samba-technical/2012-April/082873.html (visited on 05/05/2012).