FOSS.IN 2007

Talloc:
The Power of C

Contents

Introduction

The Shackles of Allocation

Talloc: Allocation, Stealing, Efficiency

Talloc: More than Freeing

Talloc In Real Life

Thanks

Time Better Spent..

- distcc
  - http://distcc.samba.org
- ccache
  - http://ccache.samba.org
- ccontrol
  - http://ccontrol.ozlabs.org
- mercurial
  - http://www.selenic.com/mercurial

A Word on Interface Design

- An interface must be **hard to misuse**.
  - Easy to use is a nice, but lesser, goal.
- After the first page, coding is all about damage control.

Two Stories About Resources

- jitterbug
  - A CGI in C
- 52 Moore St
  - Moving out of a rental house

The Shackles of Allocation

A standard example:

```
struct kval
{
        const char *key;
        const char *val;
};
struct kval *new_kval(const char *key, const char *val)
{
        struct kval *kv = malloc(sizeof(kv));

        kv->key = key;
        kv->val = val;
        return kv;
}
```

The Shackles of Allocation

It's buggy

It's not general

The Shackles of Allocation

```
struct kval
{
        const char *key;
        const char *val;
};
struct kval *new_kval(const char *key, const char *val)
{
        struct kval *kv = malloc(sizeof(kv));

        kv->key = key;
        kv->val = val;
        return kv;
}
```

The Shackles of Allocation

```
#define new(type) ((type *)(malloc(sizeof(type))))
struct kval
{
        const char *key;
        const char *val;
};
struct kval *new_kval(const char *key, const char *val)
{
        struct kval *kv = new(struct kval);

        kv->key = key;
        kv->val = val;
        return kv;
}
```

The Shackles of Allocation

```
#define new(type) ((type *)(malloc(sizeof(type))))
struct kval
{
        const char *key;
        const char *val;
};
struct kval *new_kval(const char *key, const char *val)
{
        struct kval *kv = new(struct kval);

        kv->key = strdup(key);
        kv->val = strdup(val);
        return kv;
}
```

The Shackles of Allocation

```
#define new(type) ((type *)(malloc(sizeof(type))))
struct kval
{
        const char *key;
        const char *val;
};
struct kval *new_kval(const char *key, const char *val)
{
        struct kval *kv = new(struct kval);

        kv->key = strdup(key);
        kv->val = strdup(val);
        return kv;
}
void free_kval(struct kval *kv)
{
        free((char *)kv->key);
        free((char *)kv->val);
        free(kv);
}
```

The Shackles of Allocation

The Shackles of Allocation

```
@@ -1,3 +1,4 @@
+#define new(type) ((type *)(malloc(sizeof(type))))
 struct kval
 {
        const char *key;
@@ -5,9 +6,15 @@
 struct kval *new_kval(const char *key, const char *val)
 {
-       struct kval *kv = malloc(sizeof(kv));
+       struct kval *kv = new(struct kval);

-       kv->key = key;
-       kv->val = val;
+       kv->key = strdup(key);
+       kv->val = strdup(val);
        return kv;
 }
+void free_kval(struct kval *kv)
+{
+       free((char *)kv->key);
+       free((char *)kv->val);
+       free(kv);
+}
```

Freeing is the Problem

- Programmer must remember free_kval() not free()
  - Fails the “hard to misuse” test.

Freeing is the Problem

- Programmer must remember free_kval() not free()
  - Fails the “hard to misuse” test.
- In general, freeing causes problems:
  - Mainly done in error paths, so untested
  - Not freeing does not cause immediate failure
  - Extra copying required to be safe against freeing

Freeing is the Problem

Consider a function which reads keywords and values from a file descriptor, and puts them into a linked list.

- Without bothering to free memory
- With freeing memory

Freeing is the Problem: No Freeing

```
/* Returns false on error, key == NULL on EOF. */
extern bool read_one(int fd, char **str);

struct list_head *read_kvals(int fd)
{
        struct list_head *list = new(struct list_head);
        char *key, *val;

        while (read_one(fd, &key)) {
                /* EOF?  Done */
                if (!key)
                        return list;
                if (!read_one(fd, &val) || !val)
                        break;
                list_add(list, new_kval(key, val));
        }
        return NULL;
}
// 18 lines long
```

Freeing is the Problem: No Freeing

Freeing is the Problem

```
struct list_head *read_kvals(int fd)
{
        struct list_head *list = new(struct list_head);
        char *key, *val;

        while (read_one(fd, &key)) {
                /* EOF?  Done */
                if (!key)
                        return list;
                if (!read_one(fd, &val) || !val) {
                        free(key);
                        break;
                }
                list_add(list, new_kval(key, val));
                free(key);
                free(val);
        }

        free_list_kvals(list);
        free(list);
        return NULL;
}
```

Freeing is the Problem

```
void free_list_kvals(struct list_head *list)
{
        while (!list_empty(list)) {
                struct kval *kval = list_top(list);
                list_del(&kval->list);
                kval_free(kval);
        }
}
// 34 lines long
```

Seeking Solutions

Just Say No

Seeking Solutions

- Just Say No
- Garbage collection
  - battles badly with C fundamental ideas

Seeking Solutions

- Just Say No
- Garbage collection
  - battles badly with C fundamental ideas
- Malloc override
  - checkpoint & discard

Seeking Solutions

- Just Say No
- Garbage collection
  - battles badly with C fundamental ideas
- Malloc override
  - checkpoint & discard
- Memory pools
  - pass a pool around to allocate from

Seeking Solutions

- Just Say No
- Garbage collection
  - battles badly with C fundamental ideas
- Malloc override
  - checkpoint & discard
- Memory pools
  - pass a pool around to allocate from
- Hanging data off end

Talloc: A Hierarchical Allocator

Every talloc-returned pointer is a pool.

Talloc: A Hierarchical Allocator

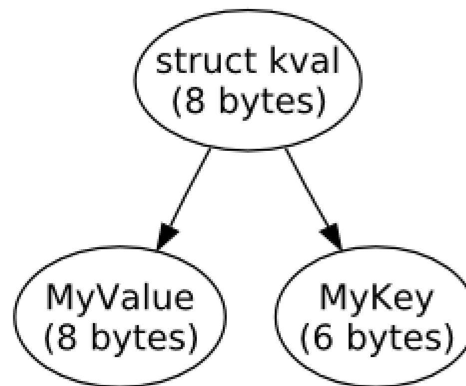
Every talloc-returned pointer is a pool.

```
struct kval
{
        const char *key;
        const char *val;
};
struct kval *new_kval(const char *key, const char *val)
{
        struct kval *kv = talloc(NULL, struct kval);

        kv->key = talloc_strdup(kv, key);
        kv->val = talloc_strdup(kv, val);
        return kv;
}
```

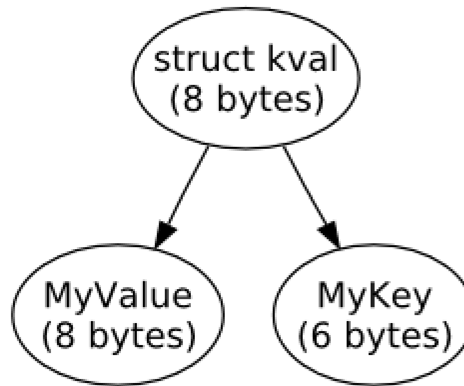
Talloc Hierarchy

Talloc hierarchy:



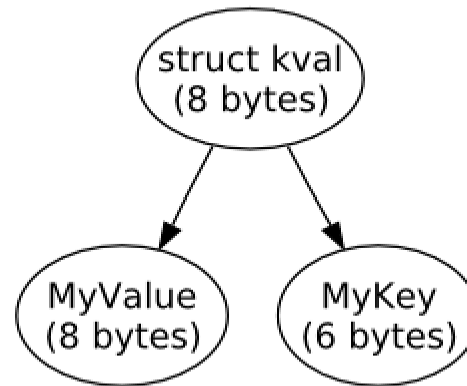
Talloc Hierarchy

Talloc hierarchy:



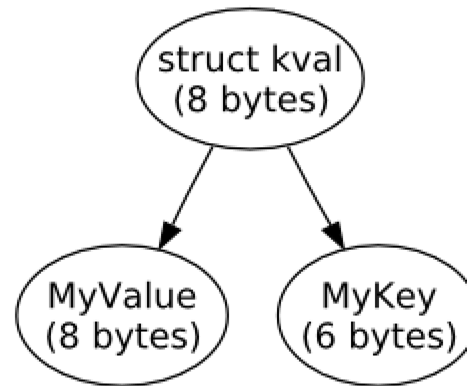
talloc_free() on any child frees that child.

Talloc Hierarchy



- *talloc_free() on parent frees children as well.*
  - One talloc_free(ptr) can clean everything

Talloc Hierarchy



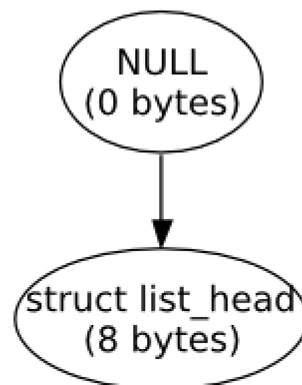
- *talloc_free() on parent frees children as well.*
  - One talloc_free(ptr) can clean everything
- And parents can be reassigned dynamically
  - talloc_steal(newparent, ptr)

Talloc for the List Case

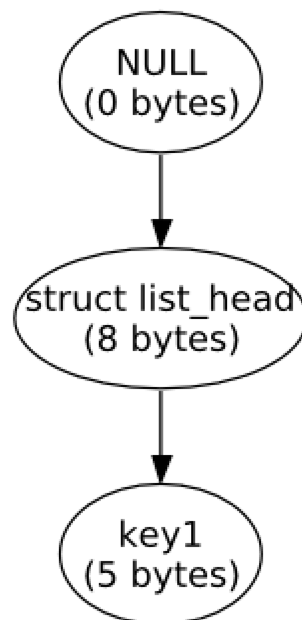
```
 struct list_head *read_kvals(int fd)
 {
-        struct list_head *list = new(struct list_head);
+        struct list_head *list = talloc(NULL,
+                                        struct list_head);
+        struct kval *kv;
         char *key, *val;

         while (read_one(fd, &key)) {
                 /* EOF?  Done */
                 if (!key)
                         return list;
+                talloc_steal(list, key);
                 if (!read_one(fd, &val) || !val)
                         break;
+                talloc_steal(list, val);
-                list_add(list, new_kval(key, val));
+                kv = talloc_steal(list, new_kval(key, val));
+                list_add(list, kv);
         }
+        talloc_free(list);
         return NULL;
 }
```

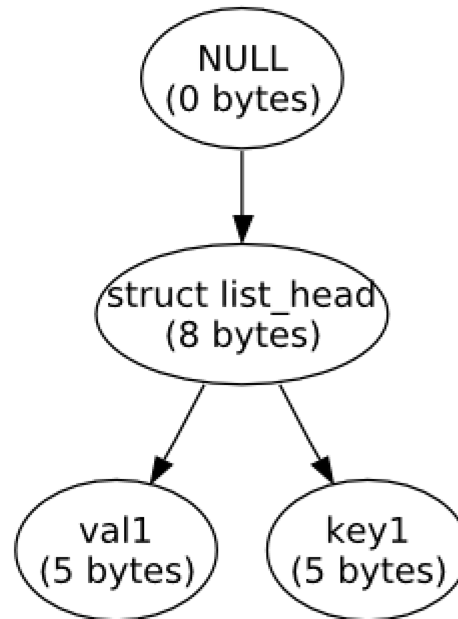
Talloc: Stealing Onto a List



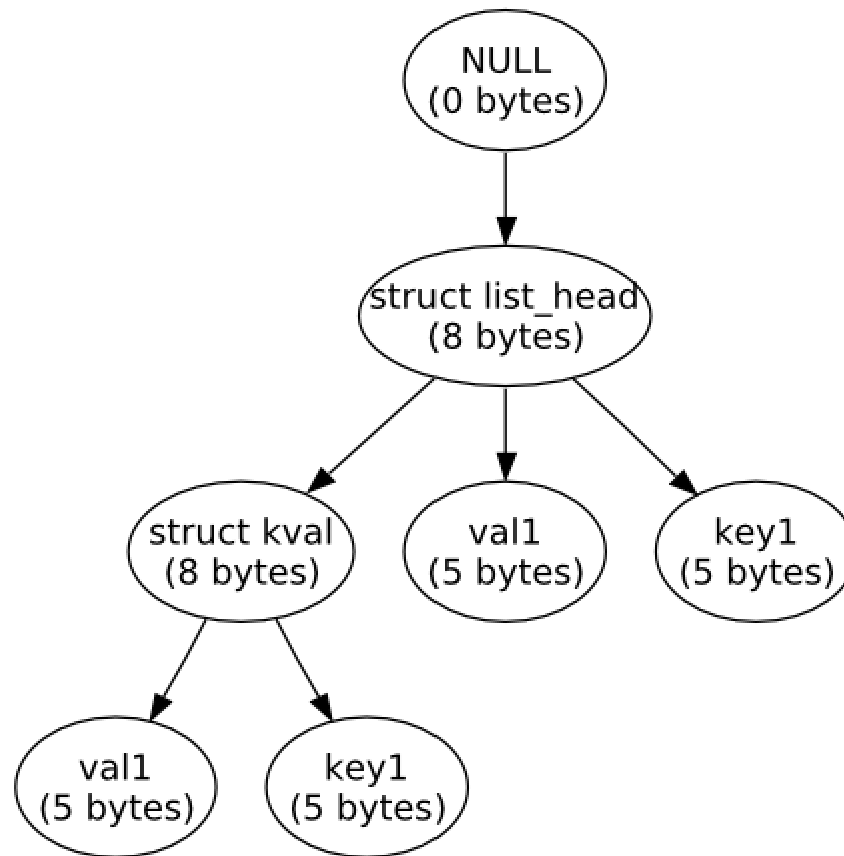
Talloc: Stealing Onto a List



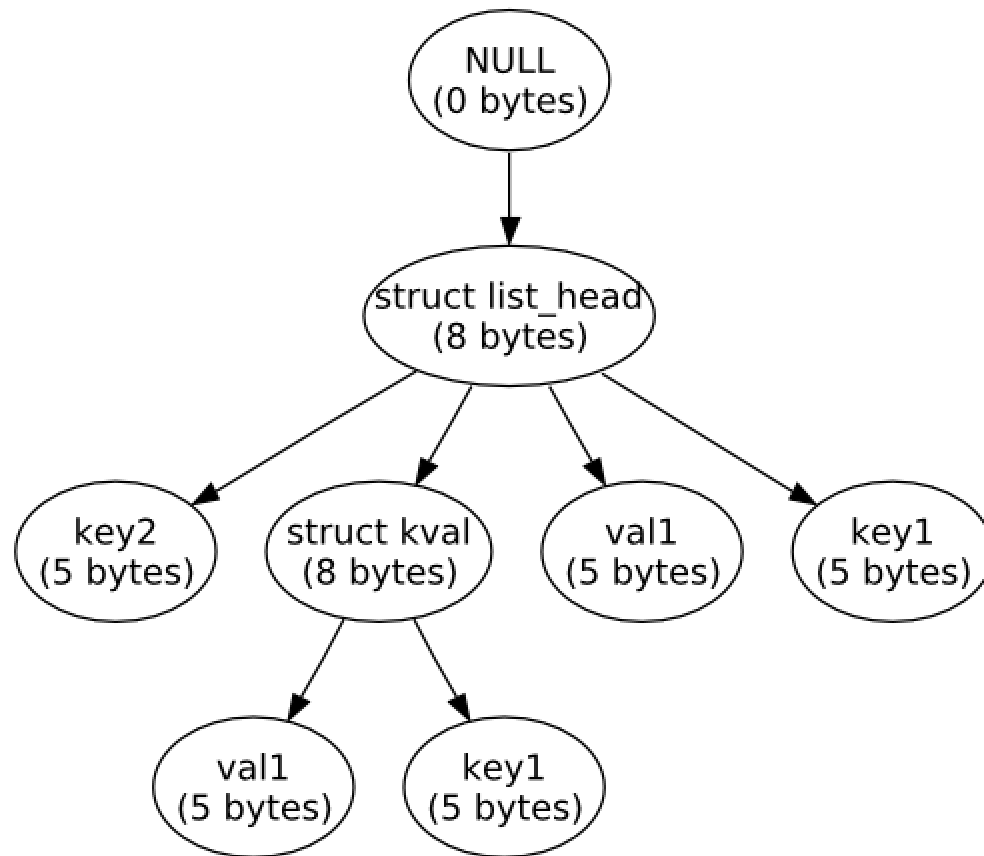
Talloc: Stealing Onto a List



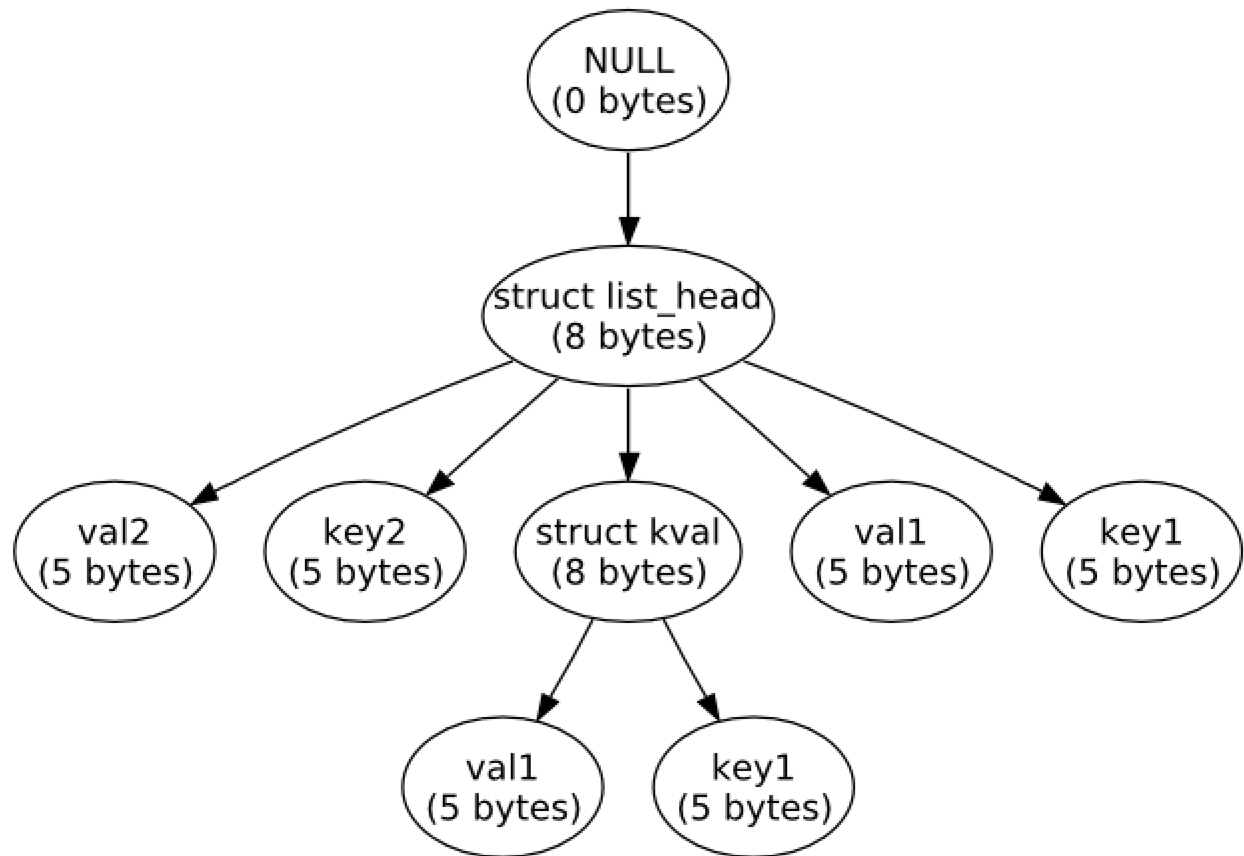
Talloc: Stealing Onto a List



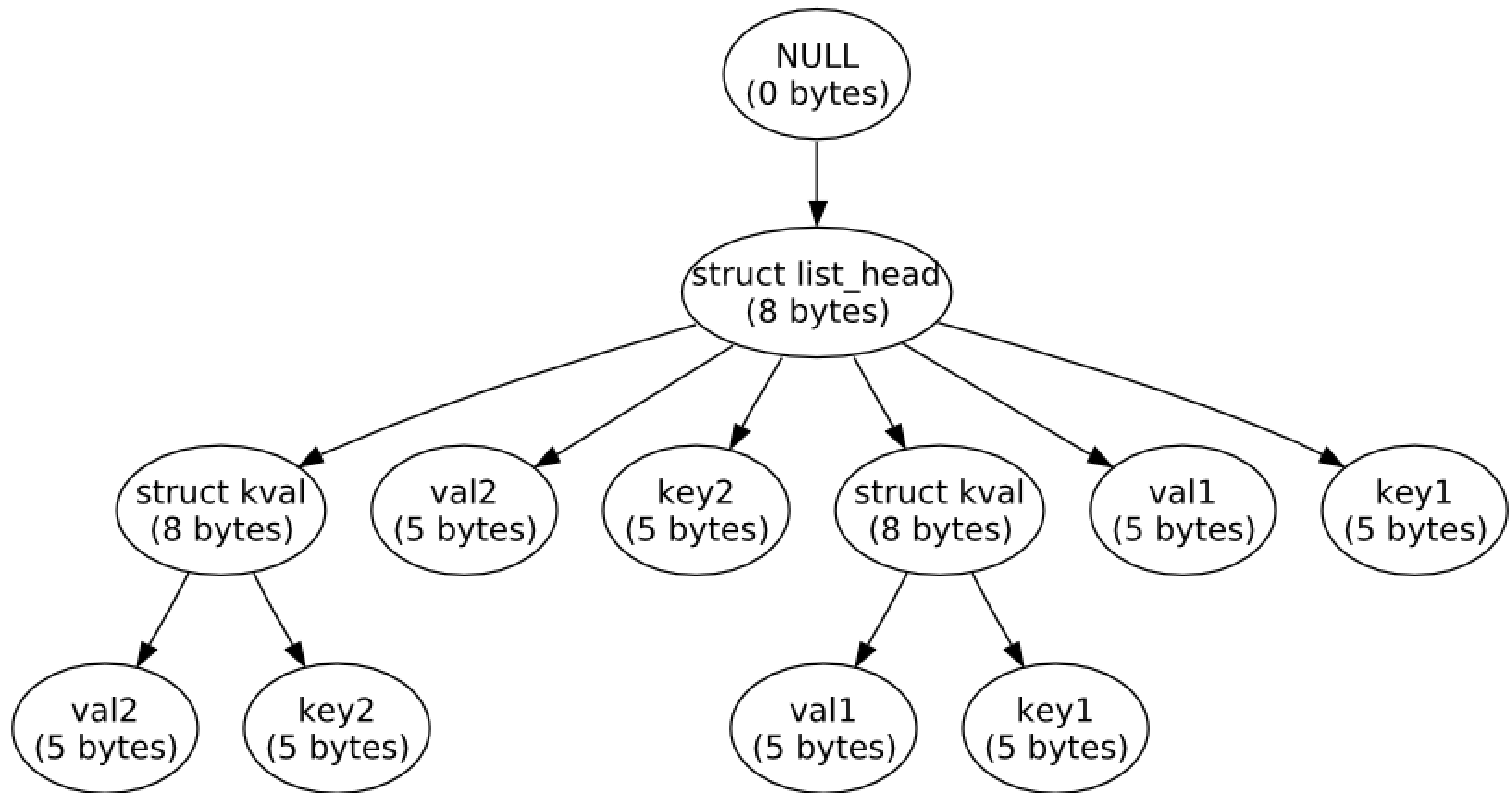
Talloc: Stealing Onto a List



Talloc: Stealing Onto a List



Talloc: Stealing Onto a List



Avoiding the Copy

- We have two copies of keys and values
  - We made this copy so our library would be general without forcing caller to strdup()

Avoiding the Copy

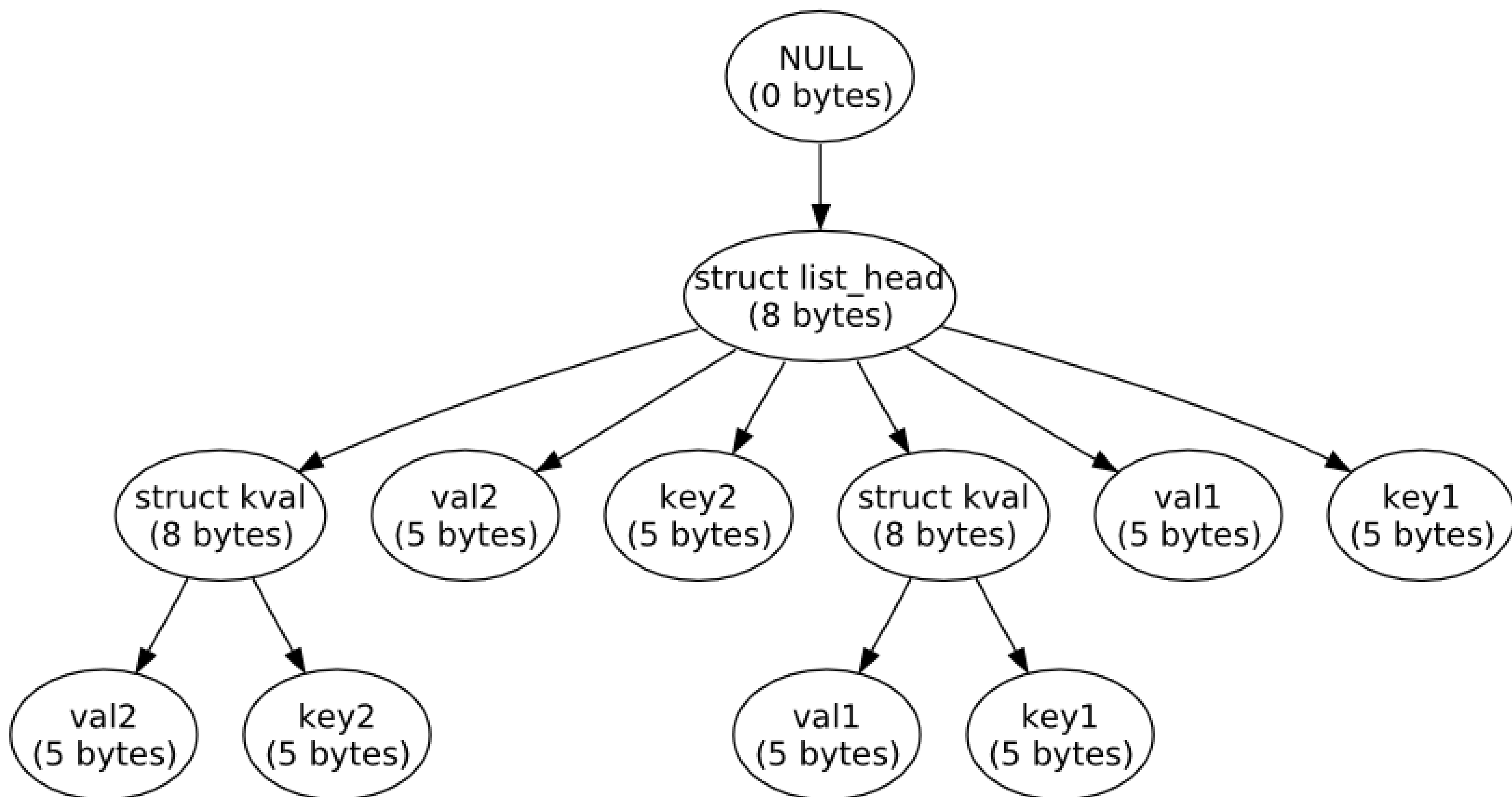
- We have two copies of keys and values
  - We made this copy so our library would be general without forcing caller to strdup()
- Talloc also allows references (ie. additional parents), so we can use that instead.
  - Like talloc_steal, but keeps existing parent(s)

Avoiding the Copy

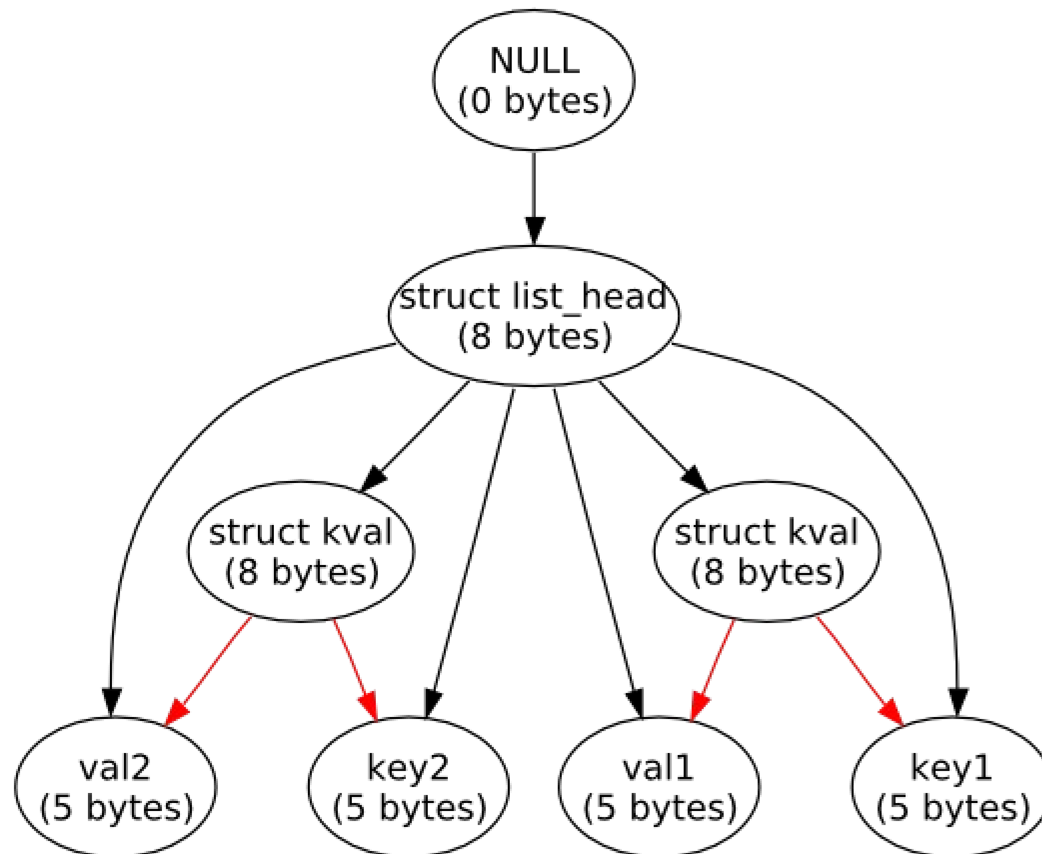
```
--- structure-talloc-multi.c
+++ structure-talloc-multi-ref.c
@@ -13,8 +13,8 @@ struct kval *new_kval(const char *
 {
        struct kval *kv = talloc(NULL, struct kval);

-       kv->key = talloc_strdup(kv, key);
-       kv->val = talloc_strdup(kv, val);
+       kv->key = talloc_reference(kv, key);
+       kv->val = talloc_reference(kv, val);
        return kv;
 }
```

Avoiding the Copy



Avoiding the Copy



More Than Freeing

Imagine we now wanted to handle multiple files.

More Than Freeing

- Imagine we now wanted to handle multiple files.
- Each keyword should now be inserted into a global list as well as the local per-file list.

More Than Freeing

```
@@ -59,7 +43,7 @@
 /* Returns false on error, str == NULL on EOF. */
 extern bool read_one(int fd, char **str);

-struct list_head *read_kvals(int fd)
+struct list_head *read_kvals(int fd,
+                              struct list_head *global)
 {
        struct list_head *list = talloc(NULL, struct list_
        struct kval *kv;
@@ -75,6 +59,7 @@
                kv = talloc_steal(list,
                                  new_kval(key, val));
                list_add(list, kv);
+               list_add(global, kv);
        }
        talloc_free(list);
        return NULL;
```

More Than Freeing

- To get rid of keyword/value pairs from a particular file, we just do:

```
struct list_head *h;
h = read_kvals(fd, &global_list);
...
talloc_free(h);
```

More Than Freeing

- To get rid of keyword/value pairs from a particular file, we just do:

```
struct list_head *h;
h = read_kvals(fd, &global_list);
...
talloc_free(h);
```

  - This will corrupt the global list!

More Than Freeing

- To get rid of keyword/value pairs from a particular file, we just do:

```
struct list_head *h;
h = read_kvals(fd, &global_list);
...
talloc_free(h);
```

  – This will corrupt the global list!

- **Implicit freeing is dangerous without intelligent destruction.**

Talloc Destructors

talloc_set_destructor(type *p, int (*)(type *p));

- Called in pre-order on talloc_free (parents first)
- Return 0

Talloc Destructors

```
@@ -9,12 +9,20 @@
        const char *val;
 };
+
+static int kv_destroy(struct kval *kv)
+{
+       list_del(&global_list, kv);
+       return 0;
+}
 struct kval *new_kval(const char *key, const char *val)
 {
        struct kval *kv = talloc(NULL, struct kval);

        kv->key = talloc_reference(kv, key);
        kv->val = talloc_reference(kv, val);
+       talloc_set_destructor(kv, kv_destroy);
        return kv;
 }
```

Uses of Destructors

Destructors usually remove references to object (linked list removal, etc).

- Object usually know best how to clean up after themselves.
- Otherwise, there should be a talloc_reference...

More Talloc!

- Every talloc alloc function takes a context arg:
  - talloc_array, talloc_zero & talloc_size
  - talloc_realloc & talloc_realloc_size
  - talloc_memdup, talloc_str[n]dup
  - talloc_asprintf, talloc_asprintf_append
- Special talloc calls:
  - talloc_total_size, talloc_total_blocks
  - talloc_autofree_context

Talloc in Real Life

The following programs use talloc:

- talloc
- nfsim
- xenstored
- SAMBA 4

Talloc in Real Life: Talloc

- Every tallocated object has a name
  - Name of type for talloc/talloc_array etc.
  - “filename:line” for untyped objects

Talloc in Real Life: Talloc

- Every tallocated object has a name
  - Name of type for talloc/talloc_array etc.
  - “filename:line” for untyped objects
- Can also be set manually using talloc_set_name & talloc_set_name_const
  - talloc_set_name_const doesn't make a copy
  - talloc_set_name needs to copy

Talloc in Real Life: Talloc

- Every tallocated object has a name
  - Name of type for talloc/talloc_array etc.
  - “filename:line” for untyped objects
- Can also be set manually using talloc_set_name & talloc_set_name_const
  - talloc_set_name_const doesn't make a copy
  - talloc_set_name needs to copy
- Flag to differentiate at free time?

Talloc in Real Life: Talloc

```
void talloc_set_name_const(const void *ptr,
                           const char *name)
{
        struct talloc_chunk *tc = talloc_chunk_from(ptr);
        tc->name = name;
}
```

Talloc in Real Life: Talloc

```
void talloc_set_name_const(const void *ptr,
                           const char *name)
{
        struct talloc_chunk *tc = talloc_chunk_from(ptr);
        tc->name = name;
}

static void talloc_set_name_v(const void *ptr,
                              const char *fmt,
                              va_list ap)
{
        struct talloc_chunk *tc = talloc_chunk_from(ptr);
        tc->name = talloc_vasprintf(ptr, fmt, ap);
        if (tc->name) {
                talloc_set_name_const(tc->name, ".name");
        }
}
```

Talloc in Real Life: nfsim

- nfsim was converted to talloc
  - nfsim is a kernel-like environment which copies netfilter code from the kernel for testing
- Benefits:
  - kmalloc, etc. and talloc_report_full
  - interfaces and implicit routes fixed trivially

Talloc in Real Life: nfsim

Live Talloc Demo

- Allocations used during startup
- Type a command, watch allocations

Talloc in Real Life: xenstored

The Xen Store Daemon relies on talloc:

- Stores device, etc trees for Xen domains

Talloc in Real Life: xenstored

- The Xen Store Daemon relies on talloc:
  - Stores device, etc trees for Xen domains
- Uses destructors more ambitiously:
  - closing fds (talloc_open)

Talloc in Real Life: xenstored

- The Xen Store Daemon relies on talloc:
  - Stores device, etc trees for Xen domains
- Uses destructors more ambitiously:
  - closing fds (talloc_open)
  - destroying in-progress database key/values

Talloc in Real Life: xenstored

The Xen Store Daemon relies on talloc:

– Stores device, etc trees for Xen domains

Uses destructors more ambitiously:

– closing fds (talloc_open)

– destroying in-progress database key/values

– unlinking database file

Talloc in Real Life: xenstored

- The Xen Store Daemon relies on talloc:
  - Stores device, etc trees for Xen domains
- Uses destructors more ambitiously:
  - closing fds (talloc_open)
  - destroying in-progress database key/values
  - unlinking database file
  - tracing

Talloc in Real Life: SAMBA 4

Samba 4 is the most extensive user of talloc

Talloc in Real Life: SAMBA 4

- Samba 4 is the most extensive user of talloc
- Particularly for type-safety:

```
#define talloc_set_type(ptr, type)\
  talloc_set_name_const(ptr, #type)
#define talloc_get_type(ptr, type)\
  (type *)talloc_check_name(ptr, #type)
#define talloc_find_parent_bytype(ptr, type) \
  (type *)talloc_find_parent_byname(ptr, #type)
```

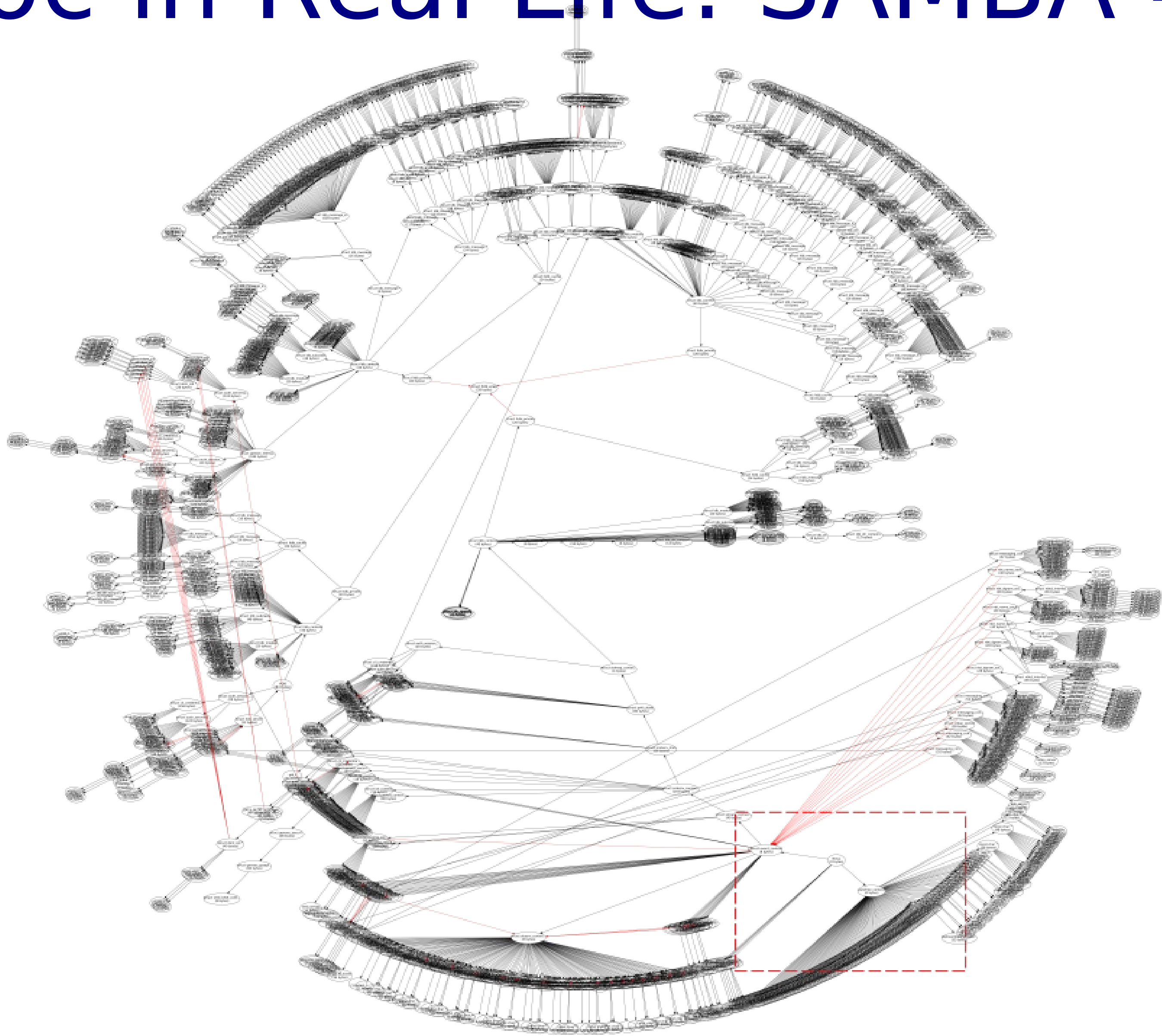
Talloc in Real Life: SAMBA 4

- Samba 4 is the most extensive user of talloc
- Particularly for type-safety:

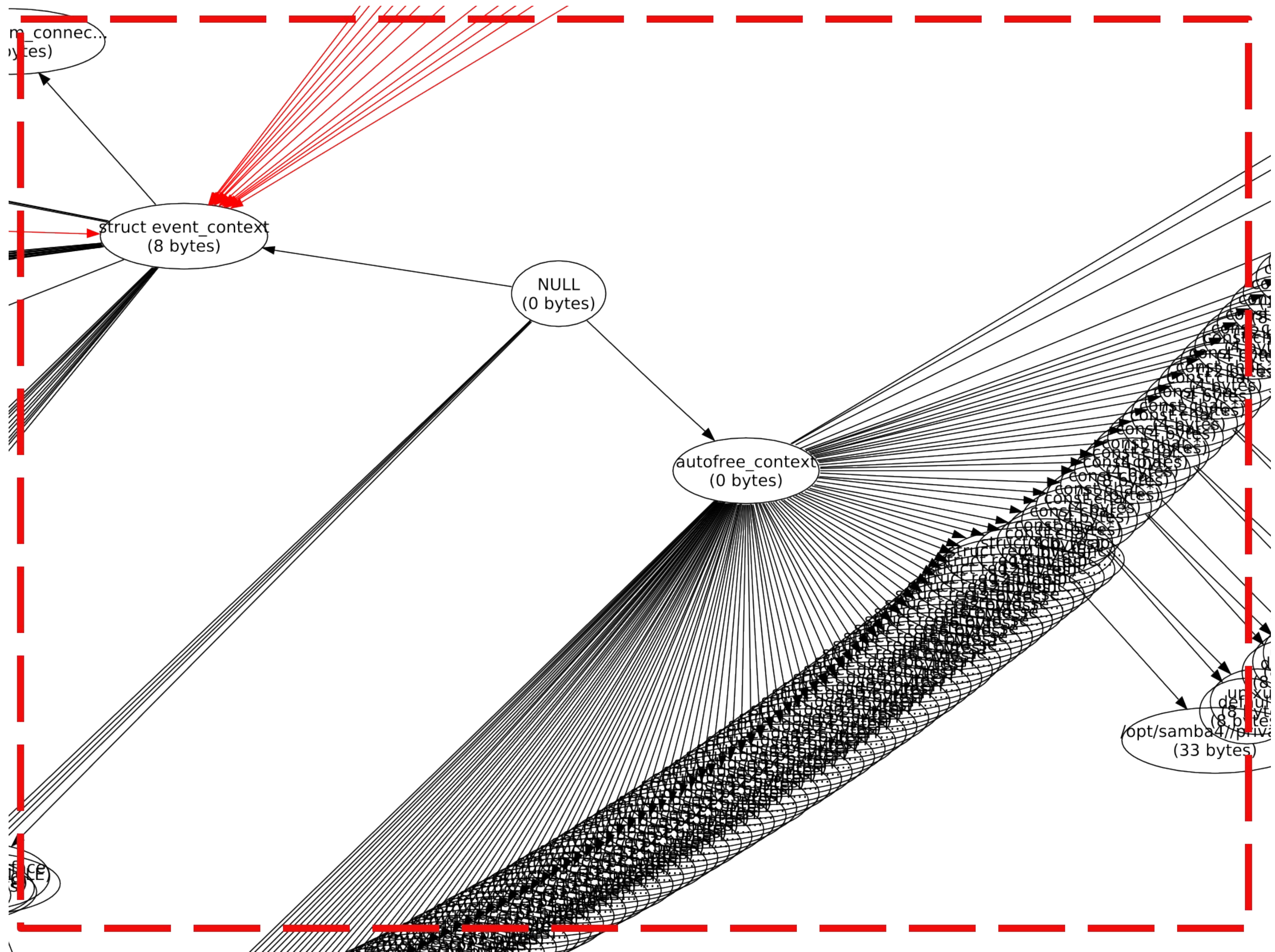
```
#define talloc_set_type(ptr, type)\
  talloc_set_name_const(ptr, #type)
#define talloc_get_type(ptr, type)\
  (type *)talloc_check_name(ptr, #type)
#define talloc_find_parent_bytype(ptr, type) \
  (type *)talloc_find_parent_byname(ptr, #type)
```

- Also “fails” destructors...

Talloc in Real Life: SAMBA 4



Talloc in Real Life: SAMBA 4



Questions?

Thanks to:

- Jeremy Kerr (talloc_report_dot)
- Tony Breeds (SAMBA 4 graph)
- Michael Neuling (GIMP + 1.5G RAM Laptop)
- David Gibson, Michael Ellerman for corrections
- Andrew Tridgell (talloc)