LSAN and ASAN: tips, tricks, and hacks

Aleksei Veselovskii

Sometimes sanitizer logs are hard to read or interpret.

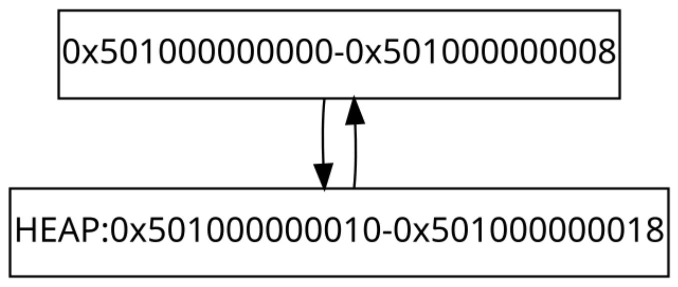
For example, it is hard to see that this:

For example, it is hard to see that this:

```
Indirect leak of 8 byte(s) in 1 object(s) allocated from:
    #0 0x7af05a816222 in operator new(unsigned long)
../../../../src/libsanitizer/lsan/lsan_interceptors.cpp:248
    #1 0x59c4b2c2b173 in main (/home/valexey/Projects/lsan_parser/lsan_parser/a.out+0x1173)
(BuildId: 562193a929b97a8c7383e721cd1082cb1a01166e)
    #2 0x7af05a02a1c9 in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58
    #3 0x7af05a02a28a in __libc_start_main_impl ../csu/libc-start.c:360
    #4 0x59c4b2c2b084 in _start (/home/valexey/Projects/lsan_parser/lsan_parser/a.out+0x1084)
(BuildId: 562193a929b97a8c7383e721cd1082cb1a01166e)

Indirect leak of 8 byte(s) in 1 object(s) allocated from:
    #0 0x7af05a816222 in operator new(unsigned long)
../../../../src/libsanitizer/lsan/lsan_interceptors.cpp:248
    #1 0x59c4b2c2b15e in main (/home/valexey/Projects/lsan_parser/lsan_parser/a.out+0x115e)
(BuildId: 562193a929b97a8c7383e721cd1082cb1a01166e)
    #2 0x7af05a02a1c9 in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58
    #3 0x7af05a02a28a in __libc_start_main_impl ../csu/libc-start.c:360
    #4 0x59c4b2c2b084 in _start (/home/valexey/Projects/lsan_parser/lsan_parser/a.out+0x1084)
(BuildId: 562193a929b97a8c7383e721cd1082cb1a01166e)
```

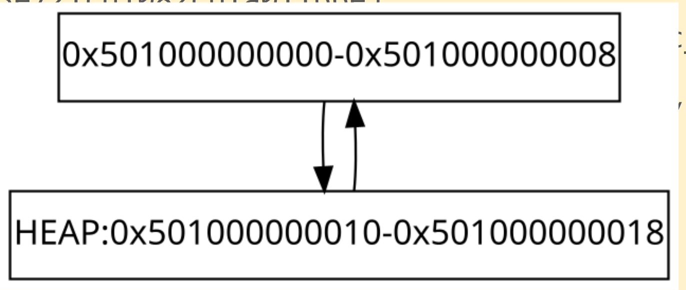
Is this:

```
Indirect leak of 8 byte(s) in 1 object(s) allocated from:
    #0 0x7af05a816222 in operator new(unsigned long)
../../../../src/libsanitizer/lsan/lsan_interceptors.cpp:248
    #1 0x59c4b2c2b173 in main (/home/valexey/Projects/lsan_parser/lsan_parser/a.out+0x1173)
(BuildId: 562193a929b97a8c7383e721cd1082cb1a01166e)
    #2 0x7af05a02a1c9 in __li                                  _start_call_main.h:58
    #3 0x7af05a02a28a in __li                                  360
    #4 0x59c4b2c2b084 in _sta                                  lsan_parser/a.out+0x1084)
(BuildId: 562193a929b97a8c738

Indirect leak of 8 byte(s) in
    #0 0x7af05a816222 in opera
../../../../src/libsanitizer/lsan/lsan_interceptors.cpp:248
    #1 0x59c4b2c2b15e in main (/home/valexey/Projects/lsan_parser/lsan_parser/a.out+0x115e)
(BuildId: 562193a929b97a8c7383e721cd1082cb1a01166e)
    #2 0x7af05a02a1c9 in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58
    #3 0x7af05a02a28a in __libc_start_main_impl ../csu/libc-start.c:360
    #4 0x59c4b2c2b084 in _start (/home/valexey/Projects/lsan_parser/lsan_parser/a.out+0x1084)
(BuildId: 562193a929b97a8c7383e721cd1082cb1a01166e)
```



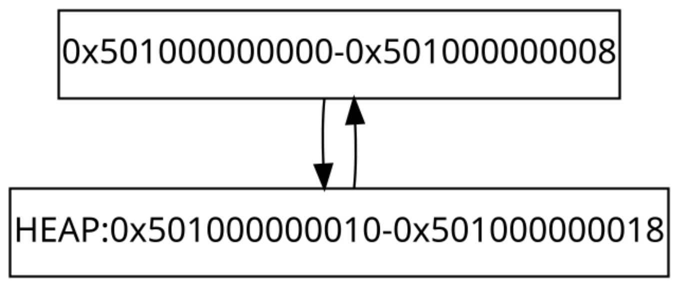
Is this – cyclical reference

```
Indirect leak of 8 byte(s) in 1 object(s) allocated from:
    #0 0x7af05a816222 in operator new(unsigned long)
../../../../src/libsanitizer/lsan/lsan_interceptors.cpp:248
    #1 0x59c4b2c2b173 in main (/home/valexey/Projects/lsan_parser/lsan_parser/a.out+0x1173)
(BuildId: 562193a929b97a8c7383e721cd1082cb1a01166e)
    #2 0x7af05a02a1c9 in __li                                 _start_call_main.h:58
    #3 0x7af05a02a28a in __li                                 360
    #4 0x59c4b2c2b084 in _sta                                 /lsan_parser/a.out+0x1084)
(BuildId: 562193a929b97a8c738

Indirect leak of 8 byte(s) in
    #0 0x7af05a816222 in opera
../../../../src/libsanitizer/lsan/lsan_interceptors.cpp:248
    #1 0x59c4b2c2b15e in main (/home/valexey/Projects/lsan_parser/lsan_parser/a.out+0x115e)
(BuildId: 562193a929b97a8c7383e721cd1082cb1a01166e)
    #2 0x7af05a02a1c9 in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58
    #3 0x7af05a02a28a in __libc_start_main_impl ../csu/libc-start.c:360
    #4 0x59c4b2c2b084 in _start (/home/valexey/Projects/lsan_parser/lsan_parser/a.out+0x1084)
(BuildId: 562193a929b97a8c7383e721cd1082cb1a01166e)
```



For reading sanitizer logs it is good to know how it works.

```
Indirect leak of 8 byte(s) in 1 object(s) allocated from:
    #0 0x7af05a816222 in operator new(unsigned long)
../../../../src/libsanitizer/lsan/lsan_interceptors.cpp:248
    #1 0x59c4b2c2b173 in main (/home/valexey/Projects/lsan_parser/lsan_parser/a.out+0x1173)
(BuildId: 562193a929b97a8c7383e721cd1082cb1a01166e)
    #2 0x7af05a02a1c9 in __lil                                    _start_call_main.h:58
    #3 0x7af05a02a28a in __lil                                    360
    #4 0x59c4b2c2b084 in _star                                    lsan_parser/a.out+0x1084)
(BuildId: 562193a929b97a8c738

Indirect leak of 8 byte(s) in
    #0 0x7af05a816222 in opera
../../../../src/libsanitizer/lsan/lsan_interceptors.cpp:248
    #1 0x59c4b2c2b15e in main (/home/valexey/Projects/lsan_parser/lsan_parser/a.out+0x115e)
(BuildId: 562193a929b97a8c7383e721cd1082cb1a01166e)
    #2 0x7af05a02a1c9 in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58
    #3 0x7af05a02a28a in __libc_start_main_impl ../csu/libc-start.c:360
    #4 0x59c4b2c2b084 in _start (/home/valexey/Projects/lsan_parser/lsan_parser/a.out+0x1084)
(BuildId: 562193a929b97a8c7383e721cd1082cb1a01166e)
```



What we’ll need to know?

What we'll need to know?

Leak Sanitizer Algorithm

What we'll need to know?

Leak Sanitizer Algorithm
Memory Manager

What we'll need to know?

Leak Sanitizer Algorithm
Memory Manager
Interceptors

What we'll need to know?

Leak Sanitizer Algorithm

Memory Manager

Interceptors

+Some sanitizer options

Plan for today:

1. Interceptors
2. Sanitizer Memory manager
3. Leak Sanitizer Algorithm
4. Leaked objects graph

Interceptors. Why?

Interceptors. Why?

To replace some known (system) functions by sanitizer versions

Interceptors. Why?

To replace some known (system) functions by sanitizer versions

… for example Memory Manager functions (**malloc/free/new/delete**)

Interceptors. Why?

To replace some known (system) functions by sanitizer versions

… for example Memory Manager functions (**malloc**/**free**/**new**/**delete**)

(ASAN/LSAN)

Interceptors. Why?

To replace some known (system) functions by sanitizer versions

… for example Memory Manager functions (**malloc/free/new/delete**)

(ASAN/LSAN)

… or pthread functions in TSAN (**pthread_mutex_create, …**)

Interceptors. Why?

To replace some known (system) functions by sanitizer versions

… for example Memory Manager functions (**malloc/free/new/delete**)

(ASAN/LSAN)

… or pthread functions in TSAN (**pthread_mutex_create, …**)

To wrap some known (system) functions

Interceptors. Why?

To replace some known (system) functions by sanitizer versions

… for example Memory Manager functions (**malloc/free/new/delete**) (ASAN/LSAN)

… or pthread functions in TSAN (**pthread_mutex_create, …**)

To wrap some known (system) functions

… for example **strcpy**, **strlen**… in ASAN

Interceptors. Why?

To replace some known (system) functions by sanitizer versions

… for example Memory Manager functions (**malloc/free/new/delete**) (ASAN/LSAN)

… or pthread functions in TSAN (**pthread_mutex_create, …**)

To wrap some known (system) functions

… for example **strcpy**, **strlen**… in ASAN

```
ReturnType interceptor_for_Func(ArgType arg) {
   // do some sanitizer specific things
   ...
   return REAL(Func)(arg);
}
```

Interceptors. Requirements.

Interceptors. Requirements.

1. Able to replace some library function by our implementation

Interceptors. Requirements.

1. Able to replace some library function by our implementation
2. Able to call original function from our implementation

Interceptors. Requirements.

1. Able to replace some library function by our implementation
2. Able to call original function from our implementation
3. We don’t want to hardcode anything to compiler

Interceptors. How?

1. Able to replace some library function by our implementation
2. Able to call original function from our implementation
3. We don’t want to hardcode anything to compiler
4. ???

Interceptors. How?

1. Able to replace some library function by our implementation
2. Able to call original function from our implementation
3. We don’t want to hardcode anything to compiler
4. ???
5. **Something during link time**

Interceptors. How?

1. Able to replace some library function by our implementation
2. Able to call original function from our implementation
3. We don’t want to hardcode anything to compiler
4. ???
5. **Something during link time**

Linux x86_64 only

Linker… things

Linker doesn’t know much about functions. Mainly it works with symbols.

Linker… things

[Click to watch](#)

How Linux Elf Symbols Work and How They Are Used in C++ and C Programming - Anders Schau Knatten

Linker: Sections and Symbols

Object file is list of headers and sections

Linker: Sections and Symbols

Object file is a list of headers and sections
Object file is...

Linker: Sections and Symbols

Object file is a list of headers and sections

Object file is…

compiled CU

```
$ cat m.c
void foo() {}
$ clang -c m.c
$ ls
m.c m.o
```

Linker: Sections and Symbols

Object file is a list of headers and sections

Object file is…

compiled CU

```
$ cat m.c
void foo() {}
$ clang -c m.c
$ ls
m.c m.o
```

linked shared object

```
$ clang -shared m.c -o m.so
$ ls
m.so
```

Linker: Sections and Symbols

Object file is a list of headers and sections

Object file is…

- compiled CU

```
$ cat m.c
void foo() {}
$ clang -c m.c
$ ls
m.c m.o
```

- linked shared object

```
$ clang -shared m.c -o m.so
$ ls
m.so
```

- linked executable

```
$ cat m.c
void main() {}
$ clang m.c
$ ls
a.out
```

Linker: Sections and Symbols

Object file is a list of headers and sections: let's look inside…

```
$ readelf -aW m.o
```

Linker: Sections and Symbols

There are a lot of sections here

```
$ readelf -aW m.o
ELF Header:
…
Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        0000000000000000 000040 00000b 00  AX  0   0  1
  [ 2] .data             PROGBITS        0000000000000000 00004b 000000 00  WA  0   0  1
  [ 3] .bss              NOBITS          0000000000000000 00004b 000000 00  WA  0   0  1
  [ 4] .comment          PROGBITS        0000000000000000 00004b 000027 01  MS  0   0  1
  [ 5] .note.GNU-stack   PROGBITS        0000000000000000 000072 000000 00      0   0  1
  [ 6] .note.gnu.property NOTE            0000000000000000 000078 000020 00   A  0   0  8
  [ 7] .eh_frame         PROGBITS        0000000000000000 000098 000038 00   A  0   0  8
  [ 8] .rela.eh_frame    RELA            0000000000000000 000140 000018 18   I  9   7  8
  [ 9] .symtab           SYMTAB          0000000000000000 0000d0 000060 18     10   3  8
  [10] .strtab           STRTAB          0000000000000000 000130 000009 00      0   0  1
  [11] .shstrtab         STRTAB          0000000000000000 000158 000067 00      0   0  1
```

Linker: Sections and Symbols

But we’ll consider only few of them: .symtab

```
$ readelf -aW m.o
ELF Header:
…
Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        0000000000000000 000040 00000b 00  AX  0   0  1
  [ 2] .data             PROGBITS        0000000000000000 00004b 000000 00  WA  0   0  1
  [ 3] .bss              NOBITS          0000000000000000 00004b 000000 00  WA  0   0  1
  [ 4] .comment          PROGBITS        0000000000000000 00004b 000027 01  MS  0   0  1
  [ 5] .note.GNU-stack   PROGBITS        0000000000000000 000072 000000 00      0   0  1
  [ 6] .note.gnu.property NOTE            0000000000000000 000078 000020 00   A  0   0  8
  [ 7] .eh_frame         PROGBITS        0000000000000000 000098 000038 00   A  0   0  8
  [ 8] .rela.eh_frame    RELA            0000000000000000 000140 000018 18   I  9   7  8
  [ 9] .symtab           SYMTAB          0000000000000000 0000d0 000060 18     10   3  8
  [10] .strtab           STRTAB          0000000000000000 000130 000009 00      0   0  1
  [11] .shstrtab         STRTAB          0000000000000000 000158 000067 00      0   0  1
```

Linker: Sections and Symbols

But we'll consider only few of them: .symtab, .text

```
$ readelf -aW m.o
ELF Header:
…
Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        0000000000000000 000040 00000b 00  AX  0   0  1
  [ 2] .data             PROGBITS        0000000000000000 00004b 000000 00  WA  0   0  1
  [ 3] .bss              NOBITS          0000000000000000 00004b 000000 00  WA  0   0  1
  [ 4] .comment          PROGBITS        0000000000000000 00004b 000027 01  MS  0   0  1
  [ 5] .note.GNU-stack   PROGBITS        0000000000000000 000072 000000 00      0   0  1
  [ 6] .note.gnu.property NOTE            0000000000000000 000078 000020 00   A  0   0  8
  [ 7] .eh_frame         PROGBITS        0000000000000000 000098 000038 00   A  0   0  8
  [ 8] .rela.eh_frame    RELA            0000000000000000 000140 000018 18   I  9   7  8
  [ 9] .symtab           SYMTAB          0000000000000000 0000d0 000060 18     10   3  8
  [10] .strtab           STRTAB          0000000000000000 000130 000009 00      0   0  1
  [11] .shstrtab         STRTAB          0000000000000000 000158 000067 00      0   0  1
```

Linker: Sections and Symbols

But we'll consider only few of them: .symtab, .text, .data

```
$ readelf -aW m.o
ELF Header:
…
Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        0000000000000000 000040 00000b 00  AX  0   0  1
  [ 2] .data             PROGBITS        0000000000000000 00004b 000000 00  WA  0   0  1
  [ 3] .bss              NOBITS          0000000000000000 00004b 000000 00  WA  0   0  1
  [ 4] .comment          PROGBITS        0000000000000000 00004b 000027 01  MS  0   0  1
  [ 5] .note.GNU-stack   PROGBITS        0000000000000000 000072 000000 00      0   0  1
  [ 6] .note.gnu.property NOTE            0000000000000000 000078 000020 00   A  0   0  8
  [ 7] .eh_frame         PROGBITS        0000000000000000 000098 000038 00   A  0   0  8
  [ 8] .rela.eh_frame    RELA            0000000000000000 000140 000018 18   I  9   7  8
  [ 9] .symtab           SYMTAB          0000000000000000 0000d0 000060 18     10   3  8
  [10] .strtab           STRTAB          0000000000000000 000130 000009 00      0   0  1
  [11] .shstrtab         STRTAB          0000000000000000 000158 000067 00      0   0  1
```

Linker: Sections and Symbols

A simple example: 2 funcs, 2 vars

```
$ cat m.c
int global_var = 42;
static int static_var = 42;

void global_func() {}
static void static_func(){}
```

Linker: Sections and Symbols

Compile…

```
$ cat m.c
int global_var = 42;
static int static_var = 42;

void global_func() {}
static void static_func(){}


$ clang -c m.c
```

Linker: Sections and Symbols

Explore

```
$ readelf -sW m.o

Symbol table '.symtab' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND 
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000004     4 OBJECT  LOCAL  DEFAULT    2 static_var
     4: 000000000000000b    11 FUNC    LOCAL  DEFAULT    1 static_func
     5: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    2 global_var
     6: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
```

Linker: Sections and Symbols

Each symbol has a name

```
$ readelf -sW m.o

Symbol table '.symtab' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND 
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000004     4 OBJECT  LOCAL  DEFAULT    2 static_var
     4: 000000000000000b    11 FUNC    LOCAL  DEFAULT    1 static_func
     5: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    2 global_var
     6: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
```

Linker: Sections and Symbols

And a value

```
$ readelf -sW m.o

Symbol table '.symtab' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000004     4 OBJECT  LOCAL  DEFAULT    2 static_var
     4: 000000000000000b    11 FUNC    LOCAL  DEFAULT    1 static_func
     5: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    2 global_var
     6: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
```

Linker: Sections and Symbols

Two rows have the same value. Why?

```
$ readelf -sW m.o

Symbol table '.symtab' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND 
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000004     4 OBJECT  LOCAL  DEFAULT    2 static_var
     4: 000000000000000b    11 FUNC    LOCAL  DEFAULT    1 static_func
     5: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    2 global_var
     6: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
```

Linker: Sections and Symbols

Two rows have the same value. Why? They are located in the different sections!

```
$ readelf -sW m.o

Symbol table '.symtab' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000004     4 OBJECT  LOCAL  DEFAULT    2 static_var
     4: 000000000000000b    11 FUNC    LOCAL  DEFAULT    1 static_func
     5: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    2 global_var
     6: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
```

```
$ readelf -aW m.o
ELF Header:
…
Section Headers:
  [Nr] Name
  [ 0]
  [ 1] .text
  [ 2] .data
  [ 3] .bss
  [ 4] .comment
  [ 5] .note.GNU-stack
  [ 6] .note.gnu.property
  [ 7] .eh_frame
  [ 8] .rela.eh_frame
  [ 9] .symtab
  [10] .strtab
  [11] .shstrtab
```

Linker: Sections and Symbols

Function in a .text section

```
$ readelf -sW m.o

Symbol table '.symtab' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000004     4 OBJECT  LOCAL  DEFAULT    2 static_var
     4: 000000000000000b    11 FUNC    LOCAL  DEFAULT    1 static_func
     5: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    2 global_var
     6: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
```

```
$ readelf -aW m.o
ELF Header:
…
Section Headers:
  [Nr] Name
  [ 0]
  [ 1] .text
  [ 2] .data
  [ 3] .bss
  [ 4] .comment
  [ 5] .note.GNU-stack
  [ 6] .note.gnu.property
  [ 7] .eh_frame
  [ 8] .rela.eh_frame
  [ 9] .symtab
  [10] .strtab
  [11] .shstrtab
```

Linker: Sections and Symbols

Variable in a .data section

```
$ readelf -sW m.o

Symbol table '.symtab' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000004     4 OBJECT  LOCAL  DEFAULT    2 static_var
     4: 000000000000000b    11 FUNC    LOCAL  DEFAULT    1 static_func
     5: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    2 global_var
     6: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
```

```
$ readelf -aW m.o
ELF Header:
…
Section Headers:
  [Nr] Name
  [ 0]
  [ 1] .text
  [ 2] .data
  [ 3] .bss
  [ 4] .comment
  [ 5] .note.GNU-stack
  [ 6] .note.gnu.property
  [ 7] .eh_frame
  [ 8] .rela.eh_frame
  [ 9] .symtab
  [10] .strtab
  [11] .shstrtab
```

Linker: Sections and Symbols

Two functions

```
$ readelf -sW m.o

Symbol table '.symtab' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000004     4 OBJECT  LOCAL  DEFAULT    2 static_var
     4: 000000000000000b    11 FUNC    LOCAL  DEFAULT    1 static_func
     5: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    2 global_var
     6: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
```

```
$ readelf -aW m.o
ELF Header:
…
Section Headers:
  [Nr] Name
  [ 0]
  [ 1] .text
  [ 2] .data
  [ 3] .bss
  [ 4] .comment
  [ 5] .note.GNU-stack
  [ 6] .note.gnu.property
  [ 7] .eh_frame
  [ 8] .rela.eh_frame
  [ 9] .symtab
  [10] .strtab
  [11] .shstrtab
```

Linker: Sections and Symbols

Two functions with a different bindings

```
$ readelf -sW m.o

Symbol table '.symtab' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000004     4 OBJECT  LOCAL  DEFAULT    2 static_var
     4: 000000000000000b    11 FUNC    LOCAL  DEFAULT    1 static_func
     5: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    2 global_var
     6: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
```

```
$ readelf -aW m.o
ELF Header:
…
Section Headers:
  [Nr] Name
  [ 0]
  [ 1] .text
  [ 2] .data
  [ 3] .bss
  [ 4] .comment
  [ 5] .note.GNU-stack
  [ 6] .note.gnu.property
  [ 7] .eh_frame
  [ 8] .rela.eh_frame
  [ 9] .symtab
  [10] .strtab
  [11] .shstrtab
```

Linker: Sections and Symbols

Values are different

```
$ readelf -sW m.o

Symbol table '.symtab' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000004     4 OBJECT  LOCAL  DEFAULT    2 static_var
     4: 000000000000000b    11 FUNC    LOCAL  DEFAULT    1 static_func
     5: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    2 global_var
     6: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
```

```
$ readelf -aW m.o
ELF Header:
…
Section Headers:
  [Nr] Name
  [ 0]
  [ 1] .text
  [ 2] .data
  [ 3] .bss
  [ 4] .comment
  [ 5] .note.GNU-stack
  [ 6] .note.gnu.property
  [ 7] .eh_frame
  [ 8] .rela.eh_frame
  [ 9] .symtab
  [10] .strtab
  [11] .shstrtab
```

Linker: Sections and Symbols

What can we do with this table?

```
$ readelf -sW m.o

Symbol table '.symtab' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000004     4 OBJECT  LOCAL  DEFAULT    2 static_var
     4: 000000000000000b    11 FUNC    LOCAL  DEFAULT    1 static_func
     5: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    2 global_var
     6: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
```

Linker: Sections and Symbols

Two (or multiple) rows with same value but different names

```
$ readelf -sW m.o

Symbol table '.symtab' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND 
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000004     4 OBJECT  LOCAL  DEFAULT    2 static_var
     4: 000000000000000b    11 FUNC    LOCAL  DEFAULT    1 static_func
     5: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    2 global_var
     6: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
```

Linker: Sections and Symbols

Two (or multiple) rows with same value but different names

```
$ readelf -sW m.o

Symbol table '.symtab' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND 
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000004     4 OBJECT  LOCAL  DEFAULT    2 static_var
     4: 000000000000000b    11 FUNC    LOCAL  DEFAULT    1 static_func
     5: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    2 global_var
     6: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
     7: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func_2
```

Linker: Sections and Symbols

Let’s do it in code

```
$
```

Linker: Sections and Symbols

Let’s do it in code. It is not C or C++! It is an **extension**.

```
$ cat m.c
int global_var = 42;
static int static_var = 42;

void global_func() {}
void __attribute__((alias("global_func"))) global_func_2();

static void static_func(){}


$ clang -c m.c
```

Linker: Sections and Symbols

Result:

```
$ readelf -sW m.o

Symbol table '.symtab' contains 8 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND 
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000004     4 OBJECT  LOCAL  DEFAULT    2 static_var
     4: 000000000000000b    11 FUNC    LOCAL  DEFAULT    1 static_func
     5: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    2 global_var
     6: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
     7: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func_2
```

Linker: Sections and Symbols

What about two values with one name?

```
$
```

Linker: Sections and Symbols

What about two values with one name?
Compiler doesn't let us to define it in one compilation unit

```
$
```

Linker: Sections and Symbols

What about two values with one name?
Compiler doesn’t let us to define it in one compilation unit
But in multiple…

```
$
```

Linker: Sections and Symbols

During linking linker merges .symtabs from multiple object files to one.

```
$
```

Linker: Sections and Symbols

During linking linker merges .symtabs from multiple object files to one.
If there are two symbols with the same name generally it is an error:

```
$ cat m.c
void global_func() {}
$ clang -c m.c
```

Linker: Sections and Symbols

During linking linker merges .symtabs from multiple object files to one.
If there are two symbols with the same name generally it is an error:

```
$ cat m.c
void global_func() {}
$ clang -c m.c

$ cat main.c
void global_func() {}
void main() {}
$ clang -c main.c
```

Linker: Sections and Symbols

During linking linker merges .symtabs from multiple object files to one.
If there are two symbols with the same name generally it is an error:

```
$ cat m.c
void global_func() {}
$ clang -c m.c

$ cat main.c
void global_func() {}
void main() {}
$ clang -c main.c
```

```
$ clang m.o main.o
/usr/bin/ld: main.o: in function `global_func':
main.c:(.text+0x0): multiple definition of `global_func';
m.o:m.c:(.text+0x0): first defined here
collect2: error: ld returned 1 exit status
```

Linker: Sections and Symbols

During linking linker merges .symtabs from multiple object files to one.
If there are two symbols with the same name generally it is an error:

```
main.c:(.text+0x0): multiple definition of `global_func';
m.o:m.c:(.text+0x0): first defined here
```

```
$ cat m.c
void global_func() {}
$ clang -c m.c

$ cat main.c
void global_func() {}
void main() {}
$ clang -c main.c
```

```
$ readelf -sW m.o
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
```

Linker: Sections and Symbols

During linking linker merges .symtabs from multiple object files to one.
If there are two symbols with the same name generally it is an error:

```
main.c:(.text+0x0): multiple definition of `global_func';
m.o:m.c:(.text+0x0): first defined here
```

```
$ cat m.c
void global_func() {}
$ clang -c m.c

$ cat main.c
void global_func() {}
void main() {}
$ clang -c main.c
```

```
$ readelf -sW m.o
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
$ readelf -sW main.o
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS main.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
     4: 000000000000000b    11 FUNC    GLOBAL DEFAULT    1 main
```

Linker: Sections and Symbols

But if we define function with **weak** attribute…

```
$ cat m.c
void __attribute__((weak))
global_func() {}
$ clang -c m.c

$ cat main.c
void global_func() {}
void main() {}
$ clang -c main.c
```

Linker: Sections and Symbols

But if we define function with **weak** attribute…
Linking will be ok

```
$ cat m.c
void __attribute__((weak))
global_func() {}
$ clang -c m.c

$ cat main.c
void global_func() {}
void main() {}
$ clang -c main.c
$ clang m.o main.o
```

```
$ readelf -sW m.o
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000000    11 FUNC    WEAK   DEFAULT    1 global_func
$ readelf -sW main.o
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS main.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
     4: 000000000000000b    11 FUNC    GLOBAL DEFAULT    1 main
```

Linker: Sections and Symbols

But if we define function with **weak** attribute…
Linking will be ok
Executable also contains .symtab

```
$ cat m.c
void __attribute__((weak))
global_func() {}
$ clang -c m.c

$ cat main.c
void global_func() {}
void main() {}
$ clang -c main.c
$ clang m.o main.o
```

```
$ readelf -sW a.out | grep global_func
36: 0000000000001134    11 FUNC    GLOBAL DEFAULT   14 global_func
```

Linker: Sections and Symbols

What if we remove global_func from main.c?

```
$ cat m.c
void __attribute__((weak))
global_func() {}
$ clang -c m.c

$ cat main.c
//void global_func() {}
void main() {}
$ clang -c main.c
$ clang m.o main.o
```

```
$
```

Linker: Sections and Symbols

What if we remove global_func from main.c?

Question to audience: will the result be the same?

```
$ cat m.c
void __attribute__((weak))
global_func() {}
$ clang -c m.c

$ cat main.c
//void global_func() {}
void main() {}
$ clang -c main.c
$ clang m.o main.o
```

```
$ readelf -sW a.out | grep global
…
```

Linker: Sections and Symbols

What if we remove global_func from main.c?

Question to audience: will the result be the same?

No. Linker really copies symbols as is.

```
$ cat m.c
void __attribute__((weak))
global_func() {}
$ clang -c m.c

$ cat main.c
//void global_func() {}
void main() {}
$ clang -c main.c
$ clang m.o main.o
```

```
$ readelf -sW a.out | grep global
…
36: 0000000000001129    11 FUNC    WEAK   DEFAULT   14 global_func
```

Linker: Sections and Symbols

STRONG (global) symbols always prevail over **WEAK** symbols

```
$ cat m.c
void __attribute__((weak))
global_func() {}
$ clang -c m.c

$ cat main.c
//void global_func() {}
void main() {}
$ clang -c main.c
$ clang m.o main.o
```

```
$ readelf -sW a.out | grep global
…
36: 0000000000001129    11 FUNC    WEAK   DEFAULT   14 global_func
```

Linker: Sections and Symbols

Aliases and Weak symbols are independent. That’s why we can combine them:
Create weak alias to some function.

```
$ cat m.c
void global_func() {}
void __attribute__((weak ,alias("global_func"))) global_func_2();
$ clang -c m.c
```

Linker: Sections and Symbols

Aliases and Weak symbols are independent. That’s why we can combine them.
Create weak alias to some function.

```
$ cat m.c
void global_func() {}
void __attribute__((weak, alias("global_func"))) global_func_2();
$ clang -c m.c
$ readelf -sW m.o
Num:    Value          Size Type    Bind   Vis      Ndx Name
  0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
  1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
  2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
  3: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
  4: 0000000000000000    11 FUNC    WEAK   DEFAULT    1 global_func_2
```

Linker: Sections and Symbols

Aliases and Weak symbols are independent. That's why we can combine them.
Question to the audience:

Can we create a **STRONG** alias to a **WEAK** function?

```
$ cat m.c
void __attribute__((weak)) global_func() {}
void __attribute__((alias("global_func"))) global_func_2();
?
```

Linker: Sections and Symbols

Aliases and Weak symbols are independent. That’s why we can combine them.

Question to the audience:

Can we create a **STRONG** alias to a **WEAK** function?

Yep. We can! All records in .symtab are equal. There is no primary record.

```
$ cat m.c
void __attribute__((weak)) global_func() {}
void __attribute__((alias("global_func"))) global_func_2();
$ clang -c m.c
$ readelf -sW m.o
Num:    Value          Size Type    Bind   Vis      Ndx Name
  0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
  1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
  2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
  3: 0000000000000000    11 FUNC    WEAK   DEFAULT    1 global_func
  4: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func_2
```

Weak symbols as extension points

Weak symbols as extension points

Sanitizers define special weak symbols to let a user define some sanitizer behavior

```
$
```

Weak symbols as extension points

Sanitizers define special weak symbols to let a user define some sanitizer behavior
For example, ASAN and LSAN defines `__lsan_default_options` and `__asan_default_options`

```
$
```

Weak symbols as extension points

Sanitizers define special weak symbols to let a user define some sanitizer behavior

For example, ASAN and LSAN defines `__lsan_default_options` and `__asan_default_options`

You can redefine it:

```
$ cat my_lib.c
const char* __lsan_default_options() {
    return "log_pointers=1:leak_check_at_exit=0";
}
```

Dynamic linking

clang uses **static** ASAN runtime (by default)
gcc uses **dynamic** ASAN runtime (by default)

Dynamic linker works differently

Dynamic linking

Dynamic linker works differently

Dynamic linker is much more lazier than static one.

Dynamic linking

Seems like dynamic linker works differently

Dynamic linker is much more lazier than static one.

Static linker checks all symbols with the same name.

Dynamic linking

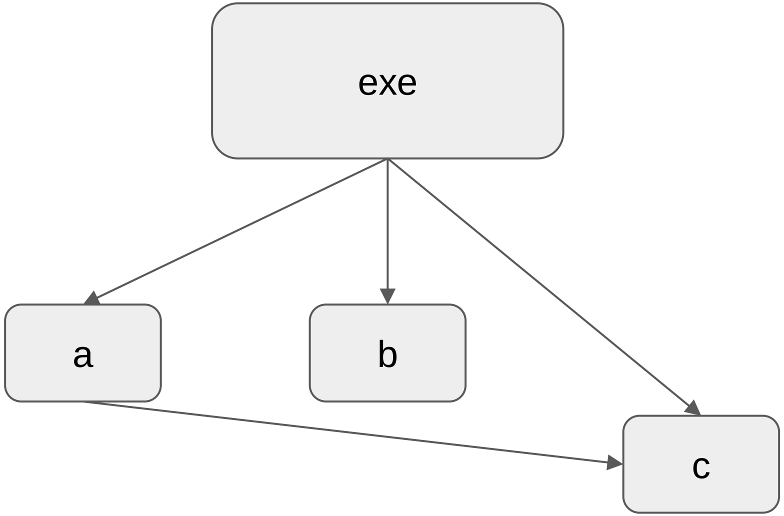
Seems like dynamic linker works differently

Dynamic linker is much more lazier than static one.

Static linker checks all symbols with the same name.
Dynamic linker just uses first found symbol

Dynamic linking

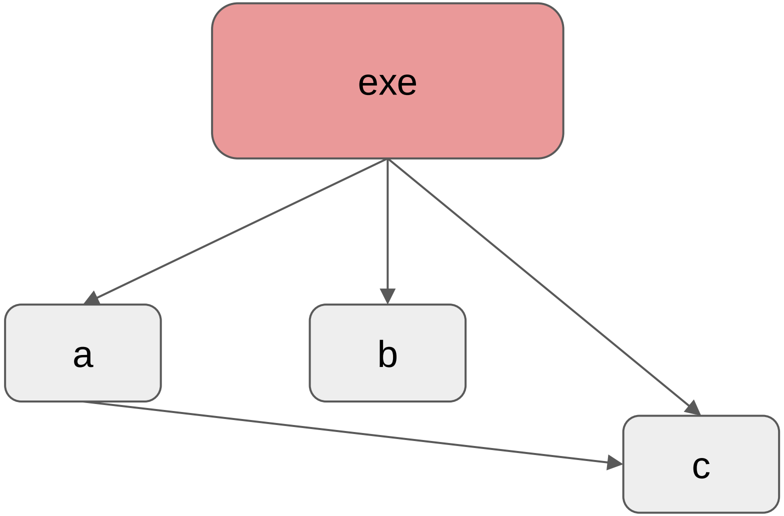
We have some executable with some dynamic dependencies. And dependencies have their own dependencies. The symbol search will be the following:



Dynamic linking

We have some executable with some dynamic dependencies. And dependencies have their own dependencies. The symbol search will be the following:

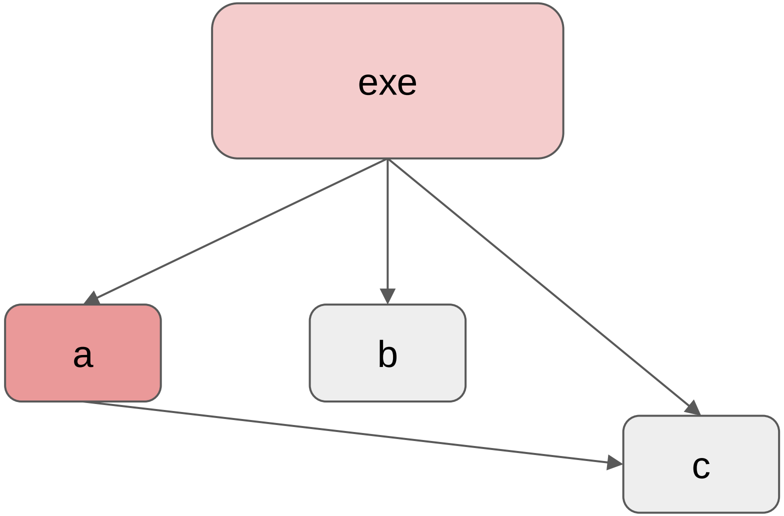
1. executable



Dynamic linking

We have some executable with some dynamic dependencies. And dependencies have their own dependencies. The symbol search will be the following:

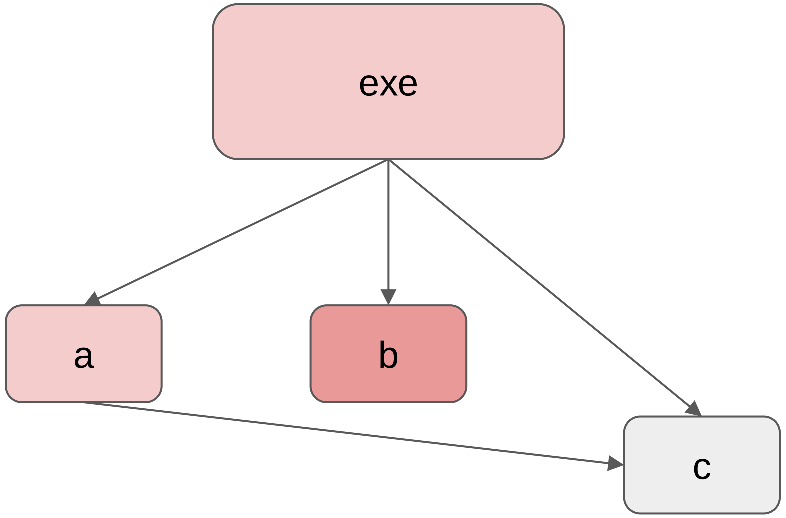
1. executable
2. a



Dynamic linking

We have some executable with some dynamic dependencies. And dependencies have their own dependencies. The symbol search will be the following:

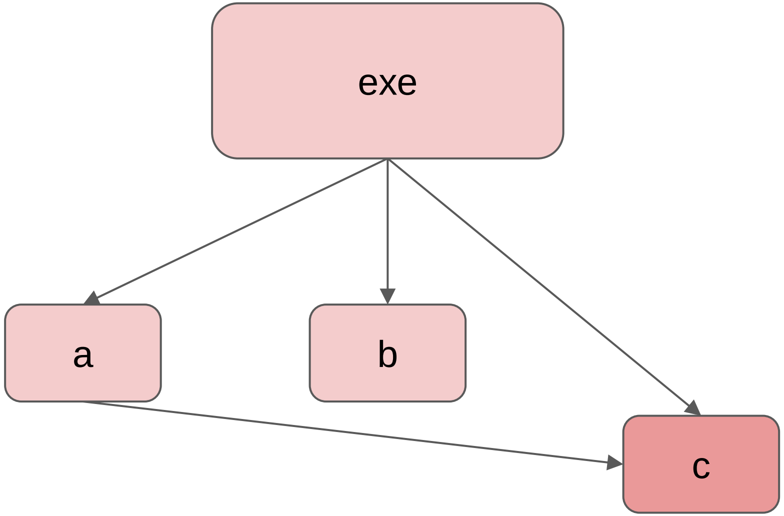
1. executable
2. a
3. b



Dynamic linking

We have some executable with some dynamic dependencies. And dependencies have their own dependencies. The symbol search will be the following:

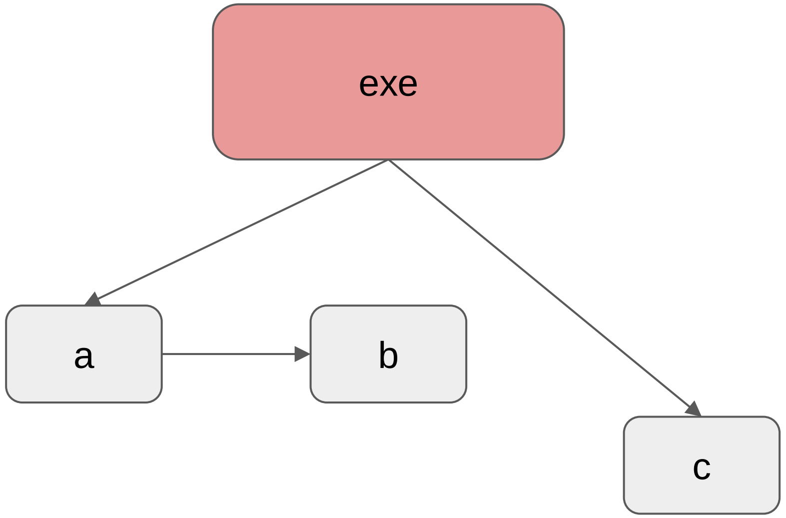
1. executable
2. a
3. b
4. c



Dynamic linking

We have some executable with some dynamic dependencies. And dependencies have their own dependencies. The symbol search will be the following:

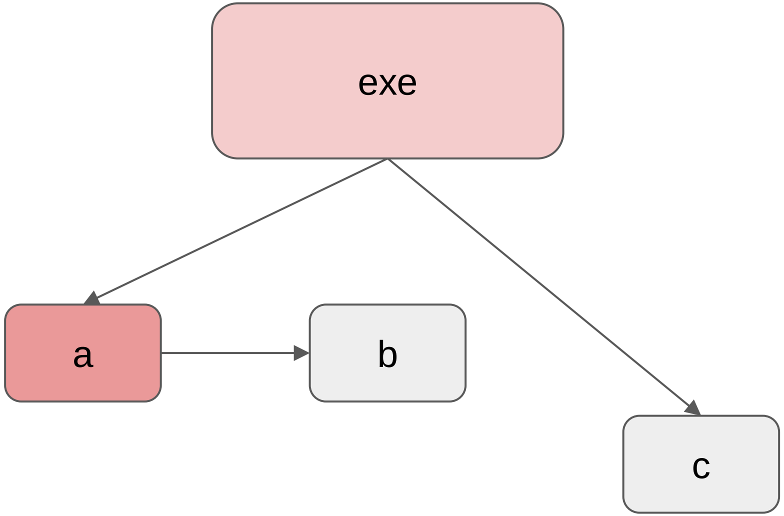
1. executable



Dynamic linking

We have some executable with some dynamic dependencies. And dependencies have their own dependencies. The symbol search will be the following:

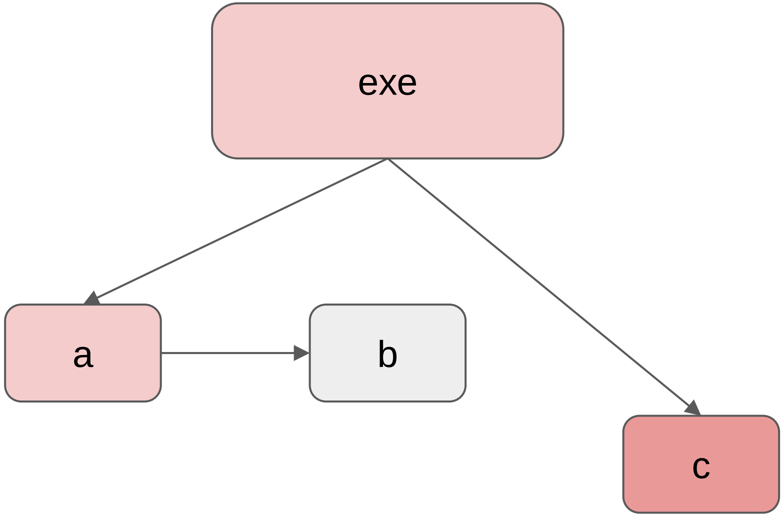
1. executable
2. a



Dynamic linking

We have some executable with some dynamic dependencies. And dependencies have their own dependencies. The symbol search will be the following:

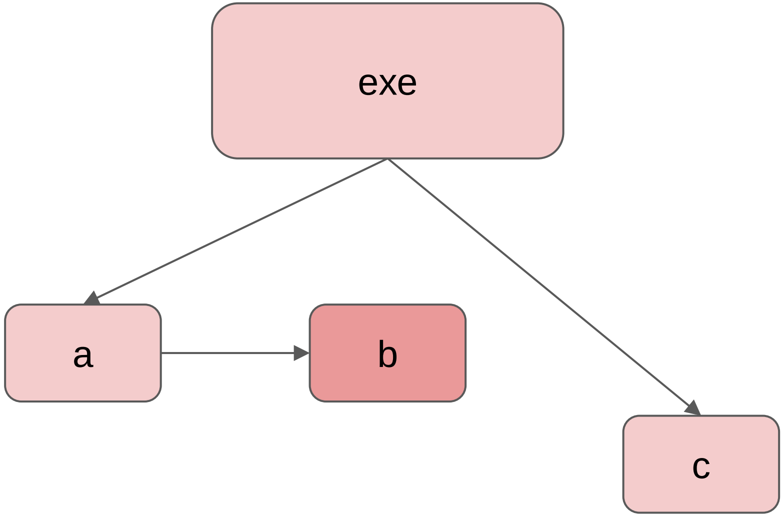
1. executable
2. a
3. c



Dynamic linking

We have some executable with some dynamic dependencies. And dependencies have their own dependencies. The symbol search will be the following:

1. executable
2. a
3. c
4. b



Dynamic linking

Let’s check how dynamic linker works with a simple app. What dependencies etc…

```
$ cat main.c
#include <stdlib.h>
int main() {
   int* a = malloc(sizeof(int));
}

$ gcc -fsanitize=address main.c
```

Dynamic linking

Let’s check dynamic dependencies.

```
$ gcc -fsanitize=address main.c
$ ldd ./a.out
        linux-vdso.so.1 (0x00007ffdd0bfd000)
        libasan.so.8 => /lib/x86_64-linux-gnu/libasan.so.8 (0x00007fc054e00000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fc054a00000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007fc055545000)
        libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007fc055518000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fc055651000)
```

Dynamic linking

We can see that ASAN runtime is the **first** dependency (**vdso** is virtual shared object, `man vdso`)

```
$ gcc -fsanitize=address main.c
$ ldd ./a.out
        linux-vdso.so.1 (0x00007ffdd0bfd000)
        libasan.so.8 => /lib/x86_64-linux-gnu/libasan.so.8 (0x00007fc054e00000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fc054a00000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007fc055545000)
        libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007fc055518000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fc055651000)
```

Dynamic linking

That’s how it intercepts all necessary functions.

```
$ gcc -fsanitize=address main.c
$ ldd ./a.out
        linux-vdso.so.1 (0x00007ffdd0bfd000)
        libasan.so.8 => /lib/x86_64-linux-gnu/libasan.so.8 (0x00007fc054e00000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fc054a00000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007fc055545000)
        libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007fc055518000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fc055651000)
```

Dynamic linking

Let’s check how `malloc` search is performed

```
$ cat main.c
#include <stdlib.h>
int main() {
   int* a = malloc(sizeof(int));
}
$ gcc -fsanitize=address main.c
```

Dynamic linking

Firstly, it is searched in executable (without success)
Secondly in lib**asan**. And it has been found in libasan.

```
$ cat main.c
#include <stdlib.h>
int main() {
   int* a = malloc(sizeof(int));
}
$ gcc -fsanitize=address main.c
$ LD_DEBUG=symbols 2>&1 ./a.out | grep malloc
    319140:  symbol=malloc;  lookup in file=./a.out [0]
    319140:  symbol=malloc;  lookup in file=/lib/x86_64-linux-gnu/libasan.so.8 [0]
```

Dynamic linking

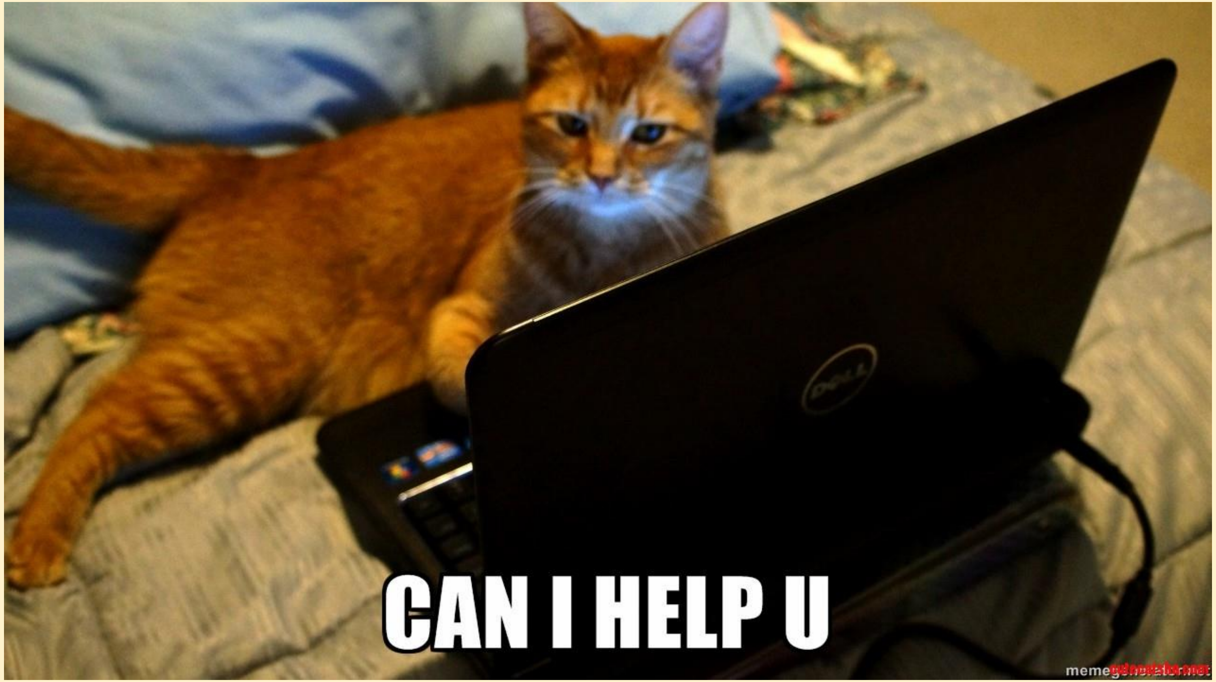
LD_DEBUG – very useful tool

```
$ LD_DEBUG=help cat
```

Dynamic linking

LD_DEBUG – very useful tool

```
$ LD_DEBUG=help cat
```



Dynamic linking

LD_DEBUG – very useful tool

```
$ LD_DEBUG=help cat
Valid options for the LD_DEBUG environment variable are:

  libs        display library search paths
  reloc       display relocation processing
  files       display progress for input file
  symbols     display symbol table processing
  bindings    display information about symbol binding
  versions    display version dependencies
  scopes      display scope information
  all         all previous options combined
  statistics  display relocation statistics
  unused      determined unused DSOs
  help        display this help message and exit

To direct the debugging output into a file instead of standard output
a filename can be specified using the LD_DEBUG_OUTPUT environment variable.
```

Dynamic linking

LD_DEBUG - why is a cat able to talk?

```
$ LD_DEBUG=help cat
Valid options for the LD_DEBUG environment variable are:

  libs        display library search paths
  reloc       display relocation processing
  files       display progress for input file
  symbols     display symbol table processing
  bindings    display information about symbol binding
  versions    display version dependencies
  scopes      display scope information
  all         all previous options combined
  statistics  display relocation statistics
  unused      determined unused DSOs
  help        display this help message and exit

To direct the debugging output into a file instead of standard output
a filename can be specified using the LD_DEBUG_OUTPUT environment variable.
```

Dynamic linking

LD_DEBUG - why is a cat able to talk?

```
$ man ld.so
NAME
          ld.so, ld-linux.so - dynamic linker/loader

SYNOPSIS
          The dynamic linker can be run either indirectly by running some dynami-
          cally  linked  program  or shared object (in which case no command-line
          options to the dynamic linker can be passed and, in the ELF  case,  the
          dynamic linker which is stored in the .interp section of the program is
          executed) or directly by running:

          /lib/ld-linux.so.*  [OPTIONS] [PROGRAM [ARGUMENTS]]

DESCRIPTION
          The  programs  ld.so  and ld-linux.so* find and load the shared objects
          (shared libraries) needed by a program, prepare the program to run, and
          then run it.
```

Dynamic linking

LD_DEBUG, LD_PRELOAD, LD_LIBRARY_PATH, etc – are env vars for **ld.so**

```
$ man ld.so
NAME
          ld.so, ld-linux.so - dynamic linker/loader

SYNOPSIS
          The dynamic linker can be run either indirectly by running some dynami-
          cally  linked  program  or shared object (in which case no command-line
          options to the dynamic linker can be passed and, in the ELF  case,  the
          dynamic linker which is stored in the .interp section of the program is
          executed) or directly by running:

          /lib/ld-linux.so.*  [OPTIONS] [PROGRAM [ARGUMENTS]]

DESCRIPTION
          The  programs  ld.so  and ld-linux.so* find and load the shared objects
          (shared libraries) needed by a program, prepare the program to run, and
          then run it.
```

Let’s use it!

Let’s use it!

ASAN public API has useful function

Let’s use it!

ASAN public API has useful function

```
// Prints stack traces for all live heap allocations ordered by total
// allocation size until top_percent of total live heap is shown. top_percent
// should be between 1 and 100. At most max_number_of_contexts contexts
// (stack traces) are printed.
// Experimental feature currently available only with ASan on Linux/x86_64.
void __sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts);
```

Let’s use it!

ASAN public API has useful function

```
// Prints stack traces for all live heap allocations ordered by total
// allocation size until top_percent of total live heap is shown. top_percent
// should be between 1 and 100. At most max_number_of_contexts contexts
// (stack traces) are printed.
// Experimental feature currently available only with ASan on Linux/x86_64.
void __sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts);
```

```
Live Heap Allocations: 475809 bytes in 6 chunks; quarantined: 0 bytes in 0 chunks; 17747 other chunks; total chunks:
17753; showing top 100% (at most 1000 unique contexts)

402000 byte(s) (84%) in 1 allocation(s)
    #0 0x6508136538d1 in operator new[](unsigned long) (/tmp/test/a.out+0x1058d1)
    #1 0x650813655b48 in main (/tmp/test/a.out+0x107b48)
    #2 0x7241ba62a1c9 in __libc_start_call_main csu/../sysdeps/nptl/libc_start_call_main.h:58:16
    #3 0x7241ba62a28a in __libc_start_main csu/../csu/libc-start.c:360:3
    #4 0x65081357a344 in _start (/tmp/test/a.out+0x2c344)

73728 byte(s) (15%) in 1 allocation(s)
    #0 0x650813615193 in malloc (/tmp/test/a.out+0xc7193)
…
```

Let’s use it!

ASAN public API has useful function

```
void __sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts);
```

We want call this function if ASAN runtime is available

Let’s use it!

ASAN public API has useful function

```
void __sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts);
```

We want to call this function if ASAN runtime is available

We want to avoid recompilation. That’s why we can’t use conditional compilation.

Let's use it!

ASAN public API has useful function

```
void __sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts);
```

We want to call this function if ASAN runtime is available

We want to avoid recompilation. That's why we can't use conditional compilation.

```
$ LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libasan.so.6 ./myCoolApp
…
$ ./myCoolApp
```

Let’s use it!

```
void __sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts);
```

We want to call this function if ASAN runtime is available

We want to avoid recompilation. That’s why we can’t use conditional compilation.
Also it should work when our app was built and linked statically with ASAN runtime

```
$ LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libasan.so.6 ./myCoolApp
…
$ ./myCoolApp
$ clang++ -fsanitize=address myCoolApp.cpp
```

Let’s use it!

We should cover all 3 cases:

1. Static ASAN runtime
2. Dynamic ASAN runtime
3. No ASAN runtime

```
$ LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libasan.so.6 ./myCoolApp
…
$ ./myCoolApp
$ clang++ -fsanitize=address -o myCoolApp myCoolApp.cpp
$ ./myCoolApp
```

Let’s use it for print_memory_profile

The solution is simple – create empty function in a separate .so

```
$
```

Let’s use it for print_memory_profile

The solution is simple – create empty function in a separate .so

```
$ cat fake_asan_profile.c
#include <stdlib.h>
void __sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts) {}
```

Let’s use it for print_memory_profile

The solution is simple – create empty function in a separate .so

```
$ cat fake_asan_profile.c
#include <stdlib.h>
void __sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts) {}
$ cat main.c
#include <sanitizer/asan_interface.h>
int main() {
   __sanitizer_print_memory_profile(100, 1000);
}
```

Let’s use it for print_memory_profile

The solution is simple – create empty function in a separate .so

```
$ cat fake_asan_profile.c
#include <stdlib.h>
void __sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts) {}
$ cat main.c
#include <sanitizer/asan_interface.h>
int main() {
   __sanitizer_print_memory_profile(100, 1000);
}
$ gcc -shared asan_profile.c -o libfake_asan_profile.so
$ gcc main.c -L. -lfake_asan_profile
```

Let’s use it for print_memory_profile

The solution is simple – create empty function in a separate .so

```
$ cat fake_asan_profile.c
#include <stdlib.h>
void __sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts) {}
$ cat main.c
#include <sanitizer/asan_interface.h>
int main() {
   __sanitizer_print_memory_profile(100, 1000);
}
$ gcc -shared asan_profile.c -o libfake_asan_profile.so
$ gcc main.c -L. -lfake_asan_profile
$ ./a.out
$
```

Let’s use it for print_memory_profile

The solution is simple – create empty function in a separate .so

```
$ gcc main.c -L. -lfake_asan_profile
$ ./a.out
$
$ ldd a.out
linux-vdso.so.1 (0x00007fffc1953000)
libfake_asan_profile.so => ./libfake_asan_profile.so (0x00007c73c5d20000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007c73c5a00000)
/lib64/ld-linux-x86-64.so.2 (0x00007c73c5d2c000)
```

Let’s use it for print_memory_profile

The solution is simple – create empty function in a separate .so
Works as expected (without ASAN runtime)

```
$ gcc main.c -L. -lfake_asan_profile
$ ./a.out
$
$ ldd a.out
linux-vdso.so.1 (0x00007fffc1953000)
libfake_asan_profile.so => ./libfake_asan_profile.so (0x00007c73c5d20000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007c73c5a00000)
/lib64/ld-linux-x86-64.so.2 (0x00007c73c5d2c000)
$ LD_DEBUG=symbols 2>&1 ./a.out | grep __sanitizer_print_memory_profile
323573:     symbol=__sanitizer_print_memory_profile;  lookup in file=./a.out [0]
323573:     symbol=__sanitizer_print_memory_profile;  lookup in file=./libfake_asan_profile.so [0]
```

Let’s use it for print_memory_profile

Let’s check it with ASAN runtime

```
$ gcc -fsanitize=address main.c -L. -lfake_asan_profile
$ ./a.out
```

Let’s use it for print_memory_profile

Let’s check it with ASAN runtime.
It works!

```
$ gcc -fsanitize=address main.c -L. -lfake_asan_profile
$ ./a.out
Live Heap Allocations: 92 bytes in 2 chunks; quarantined: 0 bytes in 0 chunks; 1948 other chunks;
total chunks: 1950; showing top 100% (at most 1000 unique contexts)
68 byte(s) (73%) in 1 allocation(s)
    #0 0x7dcf952fbb37 in malloc ../../../../src/libsanitizer/asan/asan_malloc_linux.cpp:69
    #1 0x7dcf95936db1 in malloc ../include/rtld-malloc.h:56
    #2 0x7dcf95936db1 in __GI__dl_exception_create_format elf/dl-exception.c:157
    #3 0x7dcf9593e641 in _dl_lookup_symbol_x elf/dl-lookup.c:809
    #4 0x7dcf94f8521c in do_sym elf/dl-sym.c:146
...
```

Let’s use it for print_memory_profile

Let’s check it with ASAN runtime.
libasan.so has higher priority. So we’ve created “default” implementation for this func.

```
$ gcc -fsanitize=address main.c -L. -lfake_asan_profile
$ ldd a.out
linux-vdso.so.1 (0x00007fff2018e000)
libasan.so.8 => /lib/x86_64-linux-gnu/libasan.so.8 (0x0000746a29200000)
libfake_asan_profile.so => ./libfake_asan_profile.so (0x0000746a29a31000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x0000746a28e00000)
libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x0000746a29948000)
libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x0000746a2991b000)
/lib64/ld-linux-x86-64.so.2 (0x0000746a29a59000)
```

Let’s use it for print_memory_profile

Let’s check it with ASAN runtime.
Symbol search check via LD_DEBUG:

```
$ gcc -fsanitize=address main.c -L. -lfake_asan_profile
$ LD_DEBUG=symbols 2>&1 ./a.out | grep __sanitizer_print_memory_profile
324222:    symbol=__sanitizer_print_memory_profile;  lookup in file=./a.out [0]
324222:    symbol=__sanitizer_print_memory_profile;  lookup in file=/lib/x86_64-linux-gnu/libasan.so.8 [0]
```

Let's use it for print_memory_profile

With and without ASAN runtime

```
$ gcc -fsanitize=address main.c -L. -lfake_asan_profile
$ LD_DEBUG=symbols 2>&1 ./a.out | …
lookup in file=./a.out [0]
lookup in file=/lib/x86_64-linux-gnu/libasan.so.8 [0]
```

```
$ gcc main.c -L. -lfake_asan_profile
$ LD_DEBUG=symbols 2>&1 ./a.out | …
lookup in file=./a.out [0]
lookup in file=./libfake_asan_profile.so [0]
```

Let's use it for print_memory_profile

How will it work for all 3 cases:

Static ASAN runtime: function version from main executable (**static ASAN runtime**)

Dynamic ASAN runtime: function version from first .so (**ASAN runtime**) will be used

No ASAN runtime: function version from our .so will be used

Sum up techniques

To create default implementation which could be replaced by another one you can:

For static linking: create WEAK symbol

Executable	default weak **foo**()
Executable	Possible **strong** overload for **foo**()

Sum up techniques

To create default implementation which could be replaced by another one you can:

For static linking: create WEAK symbol

For dynamic linking: move your symbol to dynamic lib. Load it AFTER possible overload.

Executable	default weak **foo**()
Executable	Possible **strong** overload for **foo**()

.so	possible overload for **boo**()

.so	default **boo**()

Sum up techniques

To create default implementation which could be replaced by another one you can:

For static linking: create WEAK symbol

For dynamic linking: move your symbol to dynamic lib. Load it AFTER possible overload.

To redefine symbol define youth version in executable or in dynamic lib which will be loaded first.

Executable	default weak **foo**()
Executable	Possible **strong** overload for **foo**()
Executable	Possible overload for **boo**()

.so	possible overload for **boo**()

.so	default **boo**()

How to call redefined function?

How to call redefined function?

Just use **dlsym** call!

How to call redefined function?

Just use **dlsym** call!

```
$ cat my_malloc.c
#include <unistd.h>
#include <dlfcn.h>
typedef void*(*real_malloc)(size_t)
void* malloc(size_t size) {
   write(1, "my malloc\n", 11);
   void* func = dlsym(RTDL_NEXT, "malloc");
   real_malloc real_malloc = func;
   return real_malloc(size);
}
```

How to call redefined function?

Just use **dlsym** call!

dlsym RTDL_NEXT finds a **next** given symbol

```
$ cat my_malloc.c
#include <unistd.h>
#include <dlfcn.h>
typedef void*(*real_malloc)(size_t)
void* malloc(size_t size) {
   write(1, "my malloc\n", 11);
   void* func = dlsym(RTDL_NEXT, "malloc");
   real_malloc real_malloc = func;
   return real_malloc(size);
}
```

How to call redefined function?

Let's try it!

```
$ cat my_malloc.c
#include <unistd.h>
#include <dlfcn.h>
typedef void*(*real_malloc)(size_t)
void* malloc(size_t size) {
   write(1, "my malloc\n", 11);
   void* func = dlsym(RTDL_NEXT, "malloc");
   real_malloc real_malloc = func;
   return real_malloc(size);
}
$ clang -shared my_malloc.c -o my_malloc.so
$ cat main.c
int main() {
   printf("%p\n", malloc(10));
}
$ clang -L. -l:my_malloc.so main.c
```

How to call redefined function?

Let's run it!

```
$ cat my_malloc.c
#include <unistd.h>
#include <dlfcn.h>
typedef void*(*real_malloc)(size_t)
void* malloc(size_t size) {
   write(1, "my malloc\n", 11);
   void* func = dlsym(RTDL_NEXT, "malloc");
   real_malloc real_malloc = func;
   return real_malloc(size);
}
$ clang -shared my_malloc.c -o my_malloc.so
$ cat main.c
int main() {
   printf("%p\n", malloc(10));
}
$ clang -L. -l:my_malloc.so main.c
$ ./a.out
my malloc
my malloc
0x5dbd8acb12a0
```

Sanitizer interceptors IRL

Sanitizer interceptors IRL

```
$ readelf -sW /lib/x86_64-linux-gnu/libasan.so.8 | grep malloc
  2439: 00000000000fa57a     5 FUNC    WEAK   DEFAULT   14 malloc
   902: 00000000000fa57a     5 FUNC    GLOBAL DEFAULT   14 __interceptor_trampoline_malloc
   214: 00000000000fba50   575 FUNC    GLOBAL DEFAULT   14 ___interceptor_malloc
  2173: 00000000000fba50   575 FUNC    WEAK   DEFAULT   14 __interceptor_malloc
```

Sanitizer interceptors IRL

```
$ readelf -sW /lib/x86_64-linux-gnu/libasan.so.8 | grep malloc
  2439: 00000000000fa57a     5 FUNC    WEAK   DEFAULT   14 malloc
   902: 00000000000fa57a     5 FUNC    GLOBAL DEFAULT   14 __interceptor_trampoline_malloc
   214: 00000000000fba50   575 FUNC    GLOBAL DEFAULT   14 ___interceptor_malloc
  2173: 00000000000fba50   575 FUNC    WEAK   DEFAULT   14 __interceptor_malloc
```

It is so complex because it should allow multiple tools to work together.

Sanitizer interceptors IRL

```
$ readelf -sW /lib/x86_64-linux-gnu/libasan.so.8 | grep malloc
  2439: 00000000000fa57a     5 FUNC    WEAK   DEFAULT   14 malloc
   902: 00000000000fa57a     5 FUNC    GLOBAL DEFAULT   14 __interceptor_trampoline_malloc
   214: 00000000000fba50   575 FUNC    GLOBAL DEFAULT   14 ___interceptor_malloc
  2173: 00000000000fba50   575 FUNC    WEAK   DEFAULT   14 __interceptor_malloc
```

First function: trampoline. A short one. Just calls `__interceptor_malloc`
Two symbols one function. WEAK and STRONG

Sanitizer interceptors IRL

```
$ readelf -sW /lib/x86_64-linux-gnu/libasan.so.8 | grep malloc
  2439: 00000000000fa57a     5 FUNC    WEAK   DEFAULT   14 malloc
   902: 00000000000fa57a     5 FUNC    GLOBAL DEFAULT   14 __interceptor_trampoline_malloc
   214: 00000000000fba50   575 FUNC    GLOBAL DEFAULT   14 ___interceptor_malloc
  2173: 00000000000fba50   575 FUNC    WEAK   DEFAULT   14 __interceptor_malloc
```

First function: trampoline. A short one. Just calls `__interceptor_malloc`
Two symbols one function. WEAK and STRONG

Second function: real ASAN malloc implementation.
Again: two symbols (WEAK and STRONG) one function.

Sanitizer interceptors IRL

```
$ readelf -sW /lib/x86_64-linux-gnu/libasan.so.8 | grep malloc
  2439: 00000000000fa57a     5 FUNC    WEAK   DEFAULT   14 malloc
   902: 00000000000fa57a     5 FUNC    GLOBAL DEFAULT   14 __interceptor_trampoline_malloc
   214: 00000000000fba50   575 FUNC    GLOBAL DEFAULT   14 ___interceptor_malloc
  2173: 00000000000fba50   575 FUNC    WEAK   DEFAULT   14 __interceptor_malloc
```

How does it work usually?

Sanitizer interceptors IRL

```
$ readelf -sW /lib/x86_64-linux-gnu/libasan.so.8 | grep malloc
  2439: 00000000000fa57a     5 FUNC    WEAK   DEFAULT   14 malloc
   902: 00000000000fa57a     5 FUNC    GLOBAL DEFAULT   14 __interceptor_trampoline_malloc
   214: 00000000000fba50   575 FUNC    GLOBAL DEFAULT   14 ___interceptor_malloc
  2173: 00000000000fba50   575 FUNC    WEAK   DEFAULT   14 __interceptor_malloc
```

How does it work usually?

Code calls **malloc**. Linker finds **malloc** symbol in ASAN runtime. **malloc is trampoline**

Sanitizer interceptors IRL

```
$ readelf -sW /lib/x86_64-linux-gnu/libasan.so.8 | grep malloc
  2439: 00000000000fa57a     5 FUNC    WEAK   DEFAULT   14 malloc
   902: 00000000000fa57a     5 FUNC    GLOBAL DEFAULT   14 __interceptor_trampoline_malloc
   214: 00000000000fba50   575 FUNC    GLOBAL DEFAULT   14 ___interceptor_malloc
  2173: 00000000000fba50   575 FUNC    WEAK   DEFAULT   14 __interceptor_malloc
```

How does it work usually?

Code calls **malloc**. Linker finds **malloc** symbol in ASAN runtime. **malloc is trampoline**

Trampoline calls **__interceptor_malloc** symbol. **__interceptor_malloc** is actual malloc implementation in ASAN

Sanitizer interceptors IRL

```
$ readelf -sW /lib/x86_64-linux-gnu/libasan.so.8 | grep malloc
  2439: 00000000000fa57a     5 FUNC    WEAK   DEFAULT   14 malloc
   902: 00000000000fa57a     5 FUNC    GLOBAL DEFAULT   14 __interceptor_trampoline_malloc
   214: 00000000000fba50   575 FUNC    GLOBAL DEFAULT   14 ___interceptor_malloc
  2173: 00000000000fba50   575 FUNC    WEAK   DEFAULT   14 __interceptor_malloc
```

Extension points: all WEAK symbols

Sanitizer interceptors IRL

```
$ readelf -sW /lib/x86_64-linux-gnu/libasan.so.8 | grep malloc
  2439: 00000000000fa57a     5 FUNC    WEAK   DEFAULT   14 malloc
   902: 00000000000fa57a     5 FUNC    GLOBAL DEFAULT   14 __interceptor_trampoline_malloc
   214: 00000000000fba50   575 FUNC    GLOBAL DEFAULT   14 ___interceptor_malloc
  2173: 00000000000fba50   575 FUNC    WEAK   DEFAULT   14 __interceptor_malloc
```

Extension points: all WEAK symbols. Other tool can redefine **malloc** or **__interceptor_malloc**

Sanitizer interceptors IRL

```
$ readelf -sW /lib/x86_64-linux-gnu/libasan.so.8 | grep malloc
  2439: 00000000000fa57a     5 FUNC    WEAK   DEFAULT   14 malloc
   902: 00000000000fa57a     5 FUNC    GLOBAL DEFAULT   14 __interceptor_trampoline_malloc
   214: 00000000000fba50   575 FUNC    GLOBAL DEFAULT   14 ___interceptor_malloc
  2173: 00000000000fba50   575 FUNC    WEAK   DEFAULT   14 __interceptor_malloc
```

Extension points: all WEAK symbols. Other tool can redefine **malloc** or **__interceptor_malloc**

In both cases redefinition should call a STRONG symbol from the same pair

Sanitizer interceptors IRL

```
$ readelf -sW /lib/x86_64-linux-gnu/libasan.so.8 | grep malloc
  2439: 00000000000fa57a     5 FUNC    WEAK   DEFAULT   14 malloc
   902: 00000000000fa57a     5 FUNC    GLOBAL DEFAULT   14 __interceptor_trampoline_malloc
   214: 00000000000fba50   575 FUNC    GLOBAL DEFAULT   14 ___interceptor_malloc
  2173: 00000000000fba50   575 FUNC    WEAK   DEFAULT   14 __interceptor_malloc
```

Extension points: all WEAK symbols. Other tool can redefine **malloc** or **__interceptor_malloc**

In both cases redefinition should call a STRONG symbol from the same pair

malloc -> __interceptor_trampoline_malloc
__interceptor_malloc -> ___interceptor_malloc

Sanitizer interceptors IRL

```
$ readelf -sW /lib/x86_64-linux-gnu/libasan.so.8 | grep malloc
  2439: 0000000000fa57a      5 FUNC    WEAK   DEFAULT   14 malloc
   902: 0000000000fa57a      5 FUNC    GLOBAL DEFAULT   14 __interceptor_trampoline_malloc
   214: 0000000000fba50    575 FUNC    GLOBAL DEFAULT   14 ___interceptor_malloc
  2173: 0000000000fba50    575 FUNC    WEAK   DEFAULT   14 __interceptor_malloc
```

If both are redefined we will have a chain of 3 interceptors (including sanitizer one):

malloc -> __interceptor_trampoline_malloc -> **__interceptor_malloc** -> **___interceptor_malloc**

Red - interceptors with custom logic

Memory Manager

Memory Manager. Why?

Control - sanitizers are sensible to (virtual)memory layout

Memory Manager. Why?

Control - sanitizers are sensible to (virtual)memory layout

Metainformation protection

Memory Manager. Why?

Control - sanitizers are sensible to (virtual)memory layout

Metainformation protection

Performance

Memory Manager. Why?

Control - sanitizers are sensible to (virtual)memory layout

Metainformation protection

Performance

Common framework for memory management for all sanitizers, scudo, etc…

Memory Manager. Why?

Control - sanitizers are sensible to (virtual)memory layout

Metainformation protection

Performance

Common framework for memory management for all sanitizers, scudo, etc…

Leak Sanitizer should “scan” all allocated memory blocks. So it should know where to find it.

Memory Manager. Requirements.

1. Able to add some (sanitizer defined) metainformation for each allocated memory block

Memory Manager. Requirements.

1. Able to add some (sanitizer defined) metainformation for each allocated memory block
2. Protect internal memory manager data structures and (optionally) metainformation from buffer overflow in user code. Do not store it next to userdata.

Memory Manager. Requirements.

1. Able to add some (sanitizer defined) metainformation for each allocated memory block
2. Protect internal memory manager data structures and (optionally) metainformation from buffer overflow in user code. Do not store it next to userdata.
3. It should have a reasonable performance. Multithreaded too.

Memory Manager. Implementation.

Top-down architecture:

1. Main allocator is CombinedAllocator

Memory Manager. Implementation.

Top-down architecture:

1. Main allocator is CombinedAllocator
2. CombinedAllocator is a fasade for two other allocators:

Memory Manager. Implementation.

Top-down architecture:

1. Main allocator is CombinedAllocator
2. CombinedAllocator is a fasade for two other allocators:
   a. PrimaryAllocator - very fast! But not always able to allocate.

Memory Manager. Implementation.

Top-down architecture:

1. Main allocator is CombinedAllocator
2. CombinedAllocator is a fasade for two other allocators:
   a. PrimaryAllocator - very fast! But not always able to allocate.
   b. SecondaryAllocator - slow. But universal. Can allocate everything.

Memory Manager. Implementation.

Top-down architecture:

1. Main allocator is CombinedAllocator
2. CombinedAllocator is a fasade for two other allocators:
   a. PrimaryAllocator - very fast! But not always able to allocate.
   b. SecondaryAllocator - slow. But universal. Can allocate everything.
3. CombinedAllocator firstly tries PrimaryAllocator. In case of failure - SecondaryAllocator

Memory Manager. Implementation.

Top-down architecture:

1. Main allocator is CombinedAllocator
2. CombinedAllocator is a fasade for two other allocators:
   a. PrimaryAllocator - very fast! But not always able to allocate.
   b. SecondaryAllocator - slow. But universal. Can allocate everything.
3. CombinedAllocator firstly tries PrimaryAllocator. In case of failure - SecondaryAllocator

PrimaryAllocator and SecondaryAllocator are aliases for instances of template Allocator classes.

Memory Manager. PrimaryAllocator.

PrimaryAllocator:

Memory Manager. PrimaryAllocator.

PrimaryAllocator:

SizeClassAllocator64 - is a PrimaryAllocator (`using PrmaryAllocator=SizeClassAllocator64<...>`). Gets memory from OS. Does not like multithreading.

Memory Manager. PrimaryAllocator.

PrimaryAllocator:

SizeClassAllocator64 - is a PrimaryAllocator (`using PrmaryAllocator=SizeClassAllocator64<...>`). Gets memory from OS. Does not like multithreading.

SizeClassMap – does not contain data, no state. Just for memory classes count (defined in compile time). Able to determine what memory class is suitable for this memory chunk size.

Memory Manager. PrimaryAllocator.

PrimaryAllocator:

SizeClassAllocator64 - is a PrimaryAllocator (`using PrmaryAllocator=SizeClassAllocator64<...>`). Gets memory from OS. Does not like multithreading.

SizeClassMap – does not contain data, no state. Just for memory classes count (defined in compile time). Able to determine what memory class is suitable for this memory chunk size.

SizeClassAllocator64LocalCache – The best friend of threads. Borrows memory from SizeClassAllocator64 for its thread needs.

Memory Manager. Primary Allocator.

PrimaryAllocator:

SizeClassAllocator64 - is a PrimaryAllocator (`using PrmaryAllocator=SizeClassAllocator64<...>`). Gets memory from OS. Does not like multithreading.

SizeClassMap – does not contain data, no state. Just for memory classes count (defined in compile time). Able to determine what memory class is suitable for this memory chunk size.

SizeClassAllocator64LocalCache – The best friend of threads. Borrows memory from SizeClassAllocator64 for its thread needs.

Memory Manager. Primary Allocator.

SizeClassAllocator64:

Memory Manager. Primary Allocator.

SizeClassAllocator64:

For each memory size class there is a **region.** Each region is divided into **chunks** with a given **size** for this memory size class.

Memory Manager. Primary Allocator.

SizeClassAllocator64:

For each memory size class there is a **region.** Each region is divided into **chunks** with a given **size** for this memory size class.

For each **chunk** there is a **MetaChunk** (with metainformation). MetaChunk is stored separately but in the same **region**.

Memory Manager. Primary Allocator.

SizeClassAllocator64:

For each memory size class there is a **region.** Each region is divided into **chunks** with a given **size** for this memory size class.

For each **chunk** there is a **MetaChunk** (with metainformation). MetaChunk is stored separately but in the same **region**.

Also there is a **FreeArray** – array of free chunks. One FreeArray per region.

Memory Manager. Primary Allocator.

SizeClassAllocator64:

For each memory size class there is a **region.** Each region is divided into **chunks** with a given **size** for this memory size class.

For each **chunk** there is a **MetaChunk** (with metainformation). MetaChunk is stored separately but in the same **region**.

Also there is a **FreeArray** – array of free chunks. One FreeArray per region.

All regions for all classes = **Space**

Memory Manager. Primary Allocator.

SizeClassAllocator64:

For each memory size class there is a **region.** Each region is divided into **chunks** with a given **size** for this memory size class.

For each **chunk** there is a **MetaChunk** (with metainformation). MetaChunk is stored separately but in the same **region**.

Also there is a **FreeArray** – array of free chunks. One FreeArray per region.

All regions for all classes = **Space**

Just after the **space** there is info about all regions.

Memory Manager. Primary Allocator.

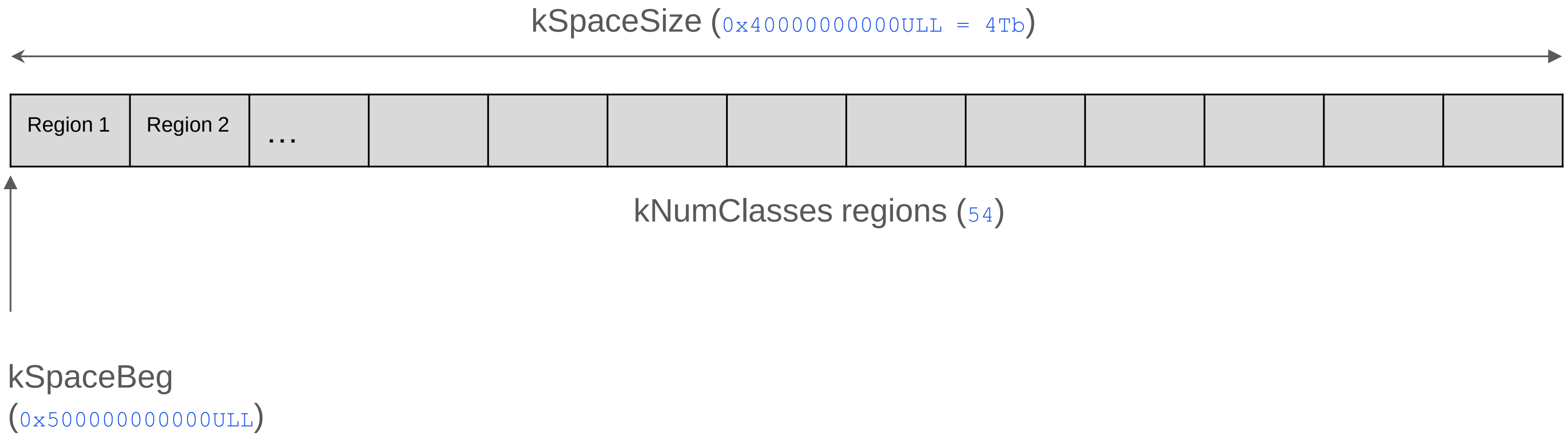
SizeClassAllocator64:

Memory Manager. Primary Allocator.

SizeClassAllocator64:

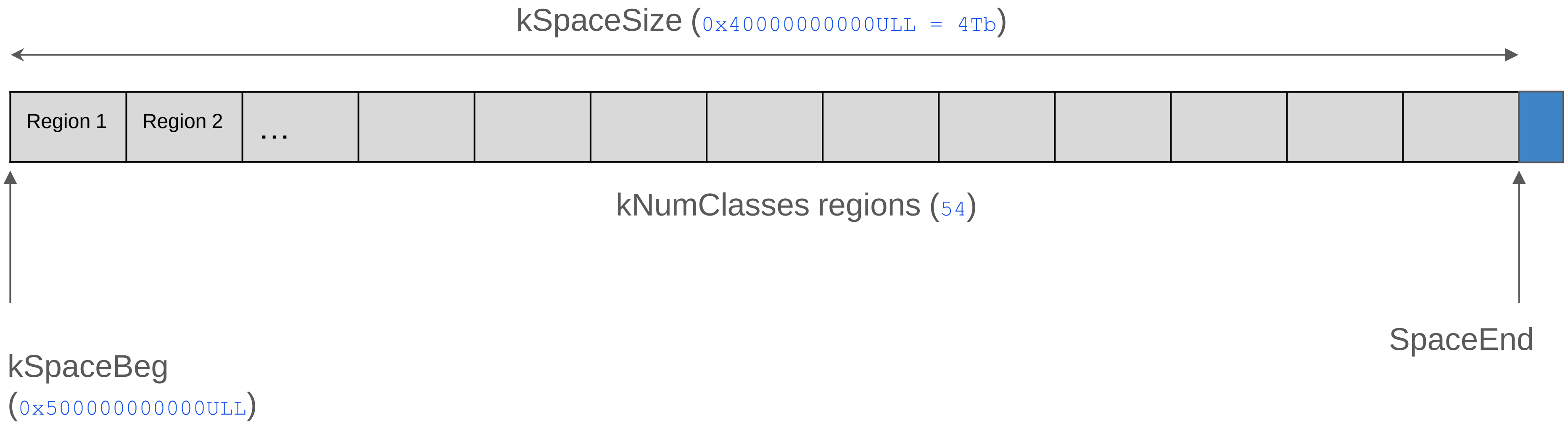
Memory Manager. Primary Allocator.

SizeClassAllocator64:



Memory Manager. Primary Allocator.

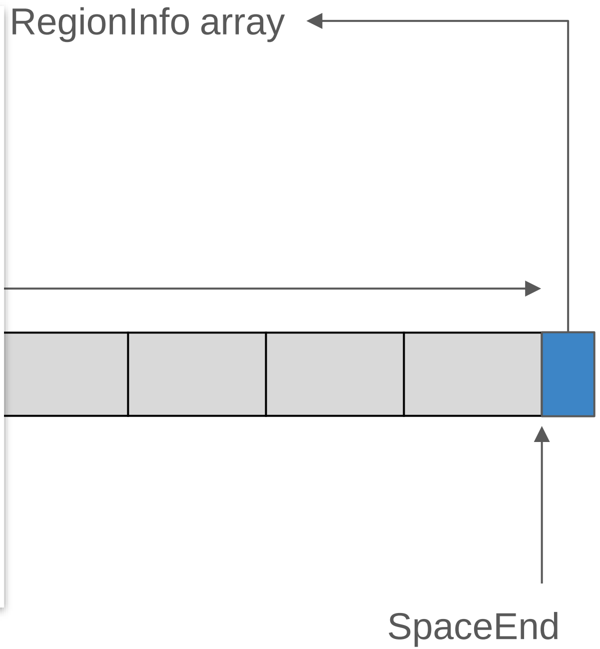
SizeClassAllocator64:



Memory Manager. Primary Allocator.

SizeClassAllocator64:

```
struct ALIGNED(SANITIZER_CACHE_LINE_SIZE) RegionInfo {
  Mutex mutex;
  uptr num_freed_chunks;  // Number of elements in the freearray.
  uptr mapped_free_array;  // Bytes mapped for freearray.
  uptr allocated_user;  // Bytes allocated for user memory.
  uptr allocated_meta;  // Bytes allocated for metadata.
  uptr mapped_user;  // Bytes mapped for user memory.
  uptr mapped_meta;  // Bytes mapped for metadata.
  u32 rand_state;  // Seed for random shuffle, used if kRandomShuffleChunks.
  bool exhausted;  // Whether region is out of space for new chunks.
  Stats stats;
  ReleaseToOsInfo rtoi;
};
```

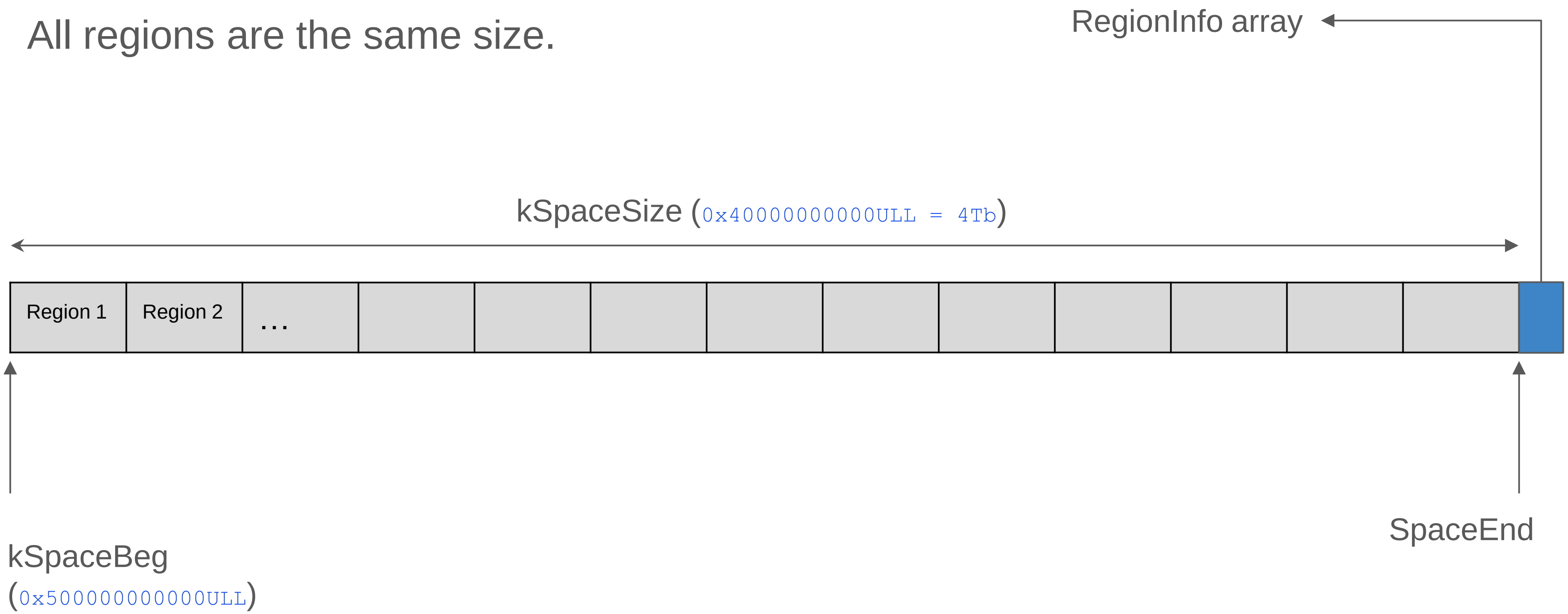


kSpaceBeg
(0x500000000000ULL)

Memory Manager. Primary Allocator.

SizeClassAllocator64:

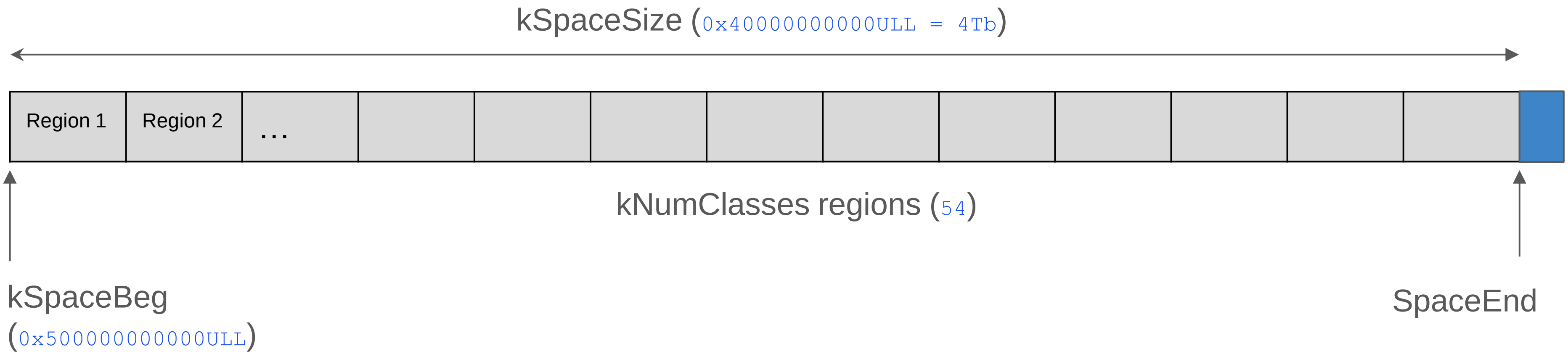
All regions are the same size.



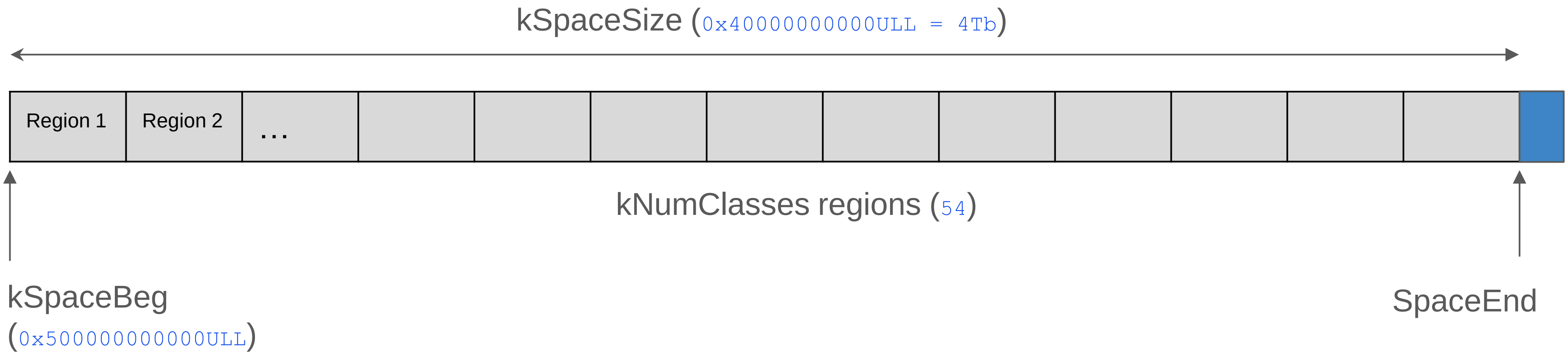
Memory Manager. Primary Allocator.

SizeClassAllocator64:

Initially all memory is mprotected. It will be allocated by request.

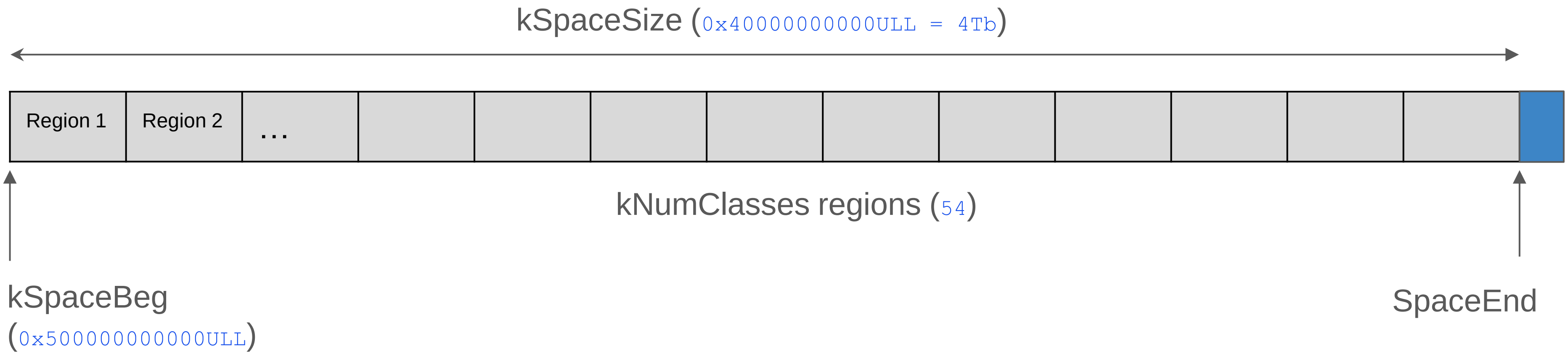

kSpaceSize (0x40000000000ULL = 4Tb)
Region 1
Region 2
...
kNumClasses regions (54)
kSpaceBeg
(0x500000000000ULL)
SpaceEnd

Region:


kSpaceSize (0x40000000000ULL = 4Tb)
Region 1
Region 2
...
kNumClasses regions (54)
kSpaceBeg
(0x500000000000ULL)
SpaceEnd

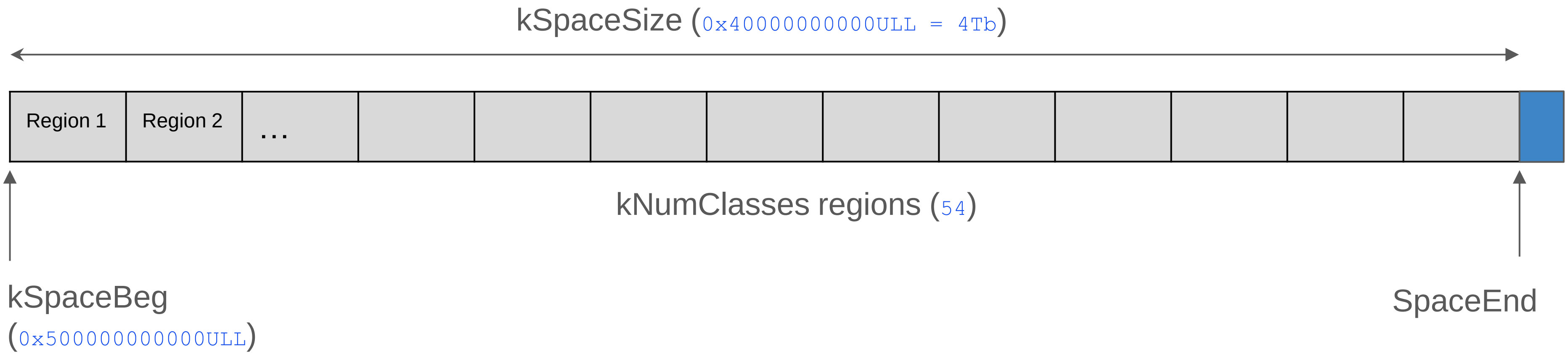
Region:


Chunk 1
Chunk 2
Chunk 3
...
Chunk N


kSpaceSize (0x40000000000ULL = 4Tb)
Region 1
Region 2
...
kNumClasses regions (54)
kSpaceBeg
(0x500000000000ULL)
SpaceEnd

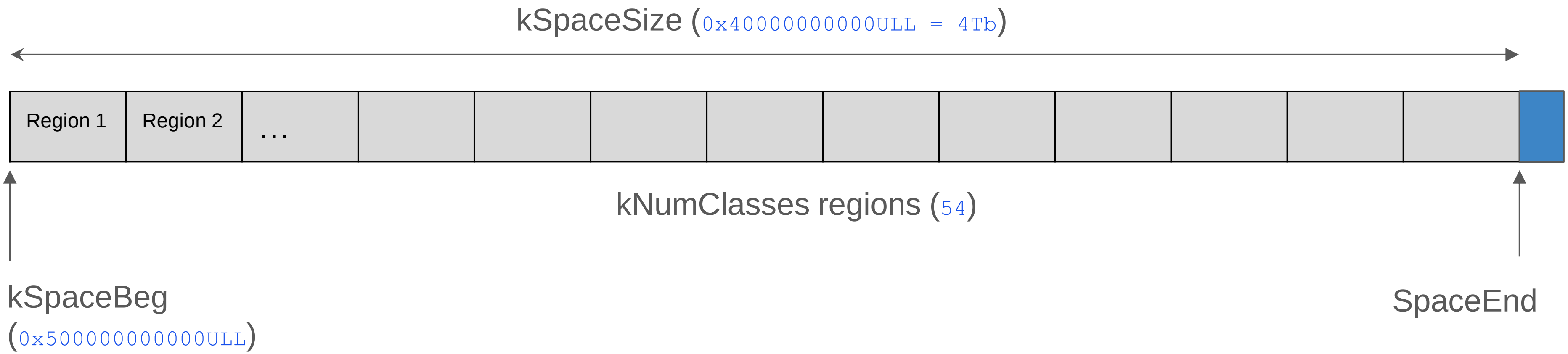
Region:


Chunk 1
Chunk 2
Chunk 3
...
Chunk N
Free Array


kSpaceSize (0x40000000000ULL = 4Tb)
Region 1
Region 2
...
kNumClasses regions (54)
kSpaceBeg
(0x500000000000ULL)
SpaceEnd

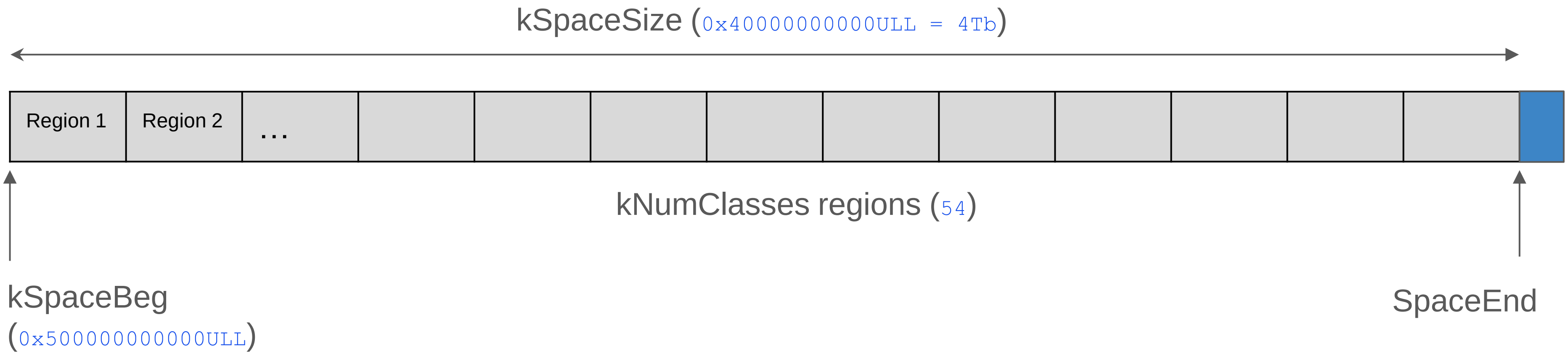
Region:


Chunk 1
Chunk 2
Chunk 3
...
Chunk N
Meta Chunk N
...
Meta Chunk 3
Meta Chunk 2
Meta Chunk 1
Free Array


kSpaceSize (0x40000000000ULL = 4Tb)
Region 1
Region 2
...
kNumClasses regions (54)
kSpaceBeg
(0x500000000000ULL)
SpaceEnd

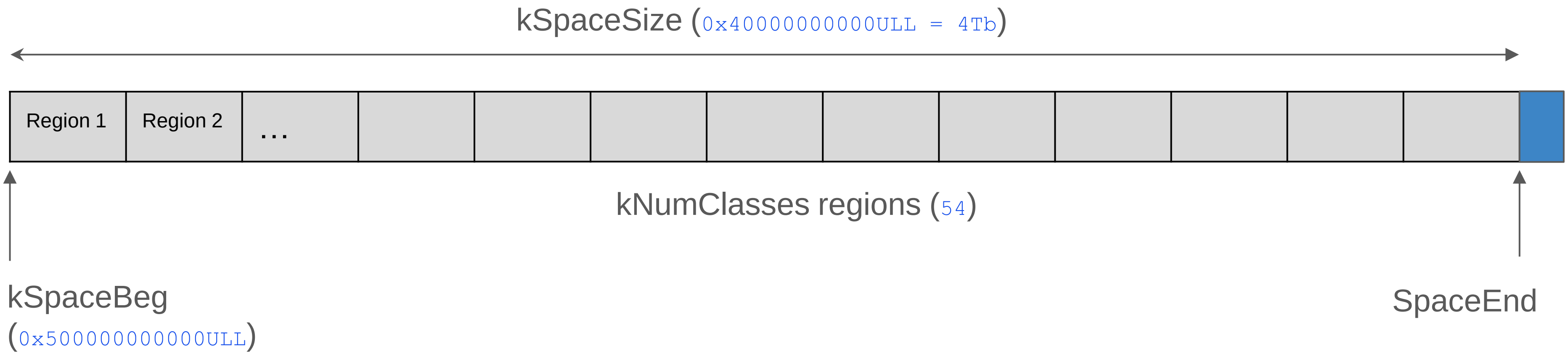
Region:


Chunk 1
Chunk 2
Chunk 3
...
Chunk N
Meta Chunk N
...
Meta Chunk 3
Meta Chunk 2
Meta Chunk 1
Free Array


kSpaceSize (0x40000000000ULL = 4Tb)
Region 1
Region 2
...
kNumClasses regions (54)
kSpaceBeg
(0x500000000000ULL)
SpaceEnd

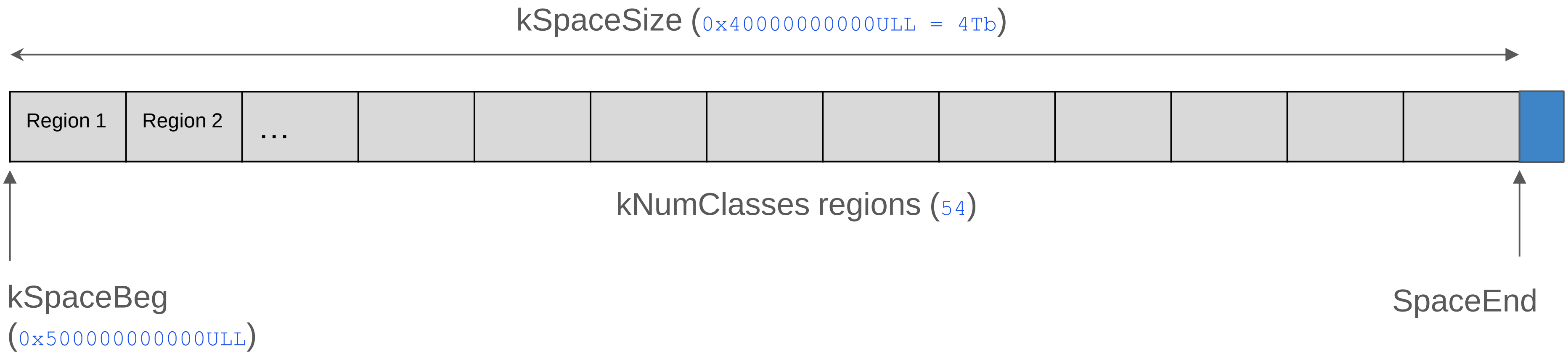
Region:


Chunk 1
Chunk 2
Chunk 3
...
Chunk N
Meta Chunk N
...
Meta Chunk 3
Meta Chunk 2
Meta Chunk 1
Free Array


kSpaceSize (0x40000000000ULL = 4Tb)
Region 1
Region 2
...
kNumClasses regions (54)
kSpaceBeg
(0x500000000000ULL)
SpaceEnd

Region:


Chunk 1
Chunk 2
Chunk 3
...
Chunk N
Meta Chunk N
...
Meta Chunk 3
Meta Chunk 2
Meta Chunk 1
Free Array



Region:

FreeArray is an array free-d chunks (stored as 4-byte offsets)

Memory Manager. Primary Allocator.

Let’s do some experiments

```
$
```

Memory Manager. Primary Allocator.

Let’s do some experiments

```
$ cat main.c
int main() {
   malloc(1);
   int i=0;
   scanf("%d", &i);
}

$ clang -fsanitize=leak main.c
```

Memory Manager. Primary Allocator.

Let’s do some experiments

```
$ cat main.c
int main() {
   malloc(1);
   int i=0;
   scanf("%d", &i);
}

$ clang -fsanitize=leak main.c
$ LSAN_OPTIONS=decorate_proc_maps=1 ./a.out &
```

Memory Manager. Primary Allocator.

Let’s do some experiments

```
$ cat main.c
int main() {
   malloc(1);
   int i=0;
   scanf("%d", &i);
}

$ clang -fsanitize=leak main.c
$ LSAN_OPTIONS=decorate_proc_maps=1 ./a.out &
$ cat /proc/$(pidof a.out)/maps
```

Memory Manager. Primary Allocator.

Let’s do some experiments

```
$ cat /proc/$(pidof a.out)/maps
```

Memory Manager. Primary Allocator.


```
$ cat /proc/$(pidof a.out)/maps
500000000000-501000000000 ---p  /dev/shm/160736 [SizeClassAllocator]
501000000000-501000040000 rw-p  /dev/shm/160736 [SizeClassAllocator: region data]
501000040000-501dfffc0000 ---p  /dev/shm/160736 [SizeClassAllocator]
501dfffc0000-501e00000000 rw-p  /dev/shm/160736 [SizeClassAllocator: region metadata]
501e00000000-501e00040000 rw-p  /dev/shm/160736 [SizeClassAllocator: freearray]
501e00040000-518000000000 ---p  /dev/shm/160736 [SizeClassAllocator]
518000000000-518000040000 rw-p  /dev/shm/160736 [SizeClassAllocator: region data]
518000040000-518dffff0000 ---p  /dev/shm/160736 [SizeClassAllocator]
518dffff0000-518e00000000 rw-p  /dev/shm/160736 [SizeClassAllocator: region metadata]
518e00000000-518e00040000 rw-p  /dev/shm/160736 [SizeClassAllocator: freearray]
518e00040000-540000000000 ---p  /dev/shm/160736 [SizeClassAllocator]
540000000000-540000002000 rw-p  /dev/shm/160736 [SizeClassAllocator: region info]
```

Memory Manager. Primary Allocator.

Region 1	Region 2	…											

```
$ cat /proc/$(pidof a.out)/maps
500000000000-501000000000 ---p  /dev/shm/160736 [SizeClassAllocator]
501000000000-501000040000 rw-p  /dev/shm/160736 [SizeClassAllocator: region data]
501000040000-501dfffc0000 ---p  /dev/shm/160736 [SizeClassAllocator]
501dfffc0000-501e00000000 rw-p  /dev/shm/160736 [SizeClassAllocator: region metadata]
501e00000000-501e00040000 rw-p  /dev/shm/160736 [SizeClassAllocator: freearray]
501e00040000-518000000000 ---p  /dev/shm/160736 [SizeClassAllocator]
518000000000-518000040000 rw-p  /dev/shm/160736 [SizeClassAllocator: region data]
518000040000-518dffff0000 ---p  /dev/shm/160736 [SizeClassAllocator]
518dffff0000-518e00000000 rw-p  /dev/shm/160736 [SizeClassAllocator: region metadata]
518e00000000-518e00040000 rw-p  /dev/shm/160736 [SizeClassAllocator: freearray]
518e00040000-540000000000 ---p  /dev/shm/160736 [SizeClassAllocator]
540000000000-540000002000 rw-p  /dev/shm/160736 [SizeClassAllocator: region info]
```

Memory Manager. Primary Allocator.


```
$ cat /proc/$(pidof a.out)/maps
500000000000-501000000000 ---p  /dev/shm/160736 [SizeClassAllocator]
501000000000-501000040000 rw-p  /dev/shm/160736 [SizeClassAllocator: region data]
501000040000-501dfffc0000 ---p  /dev/shm/160736 [SizeClassAllocator]
501dfffc0000-501e00000000 rw-p  /dev/shm/160736 [SizeClassAllocator: region metadata]
501e00000000-501e00040000 rw-p  /dev/shm/160736 [SizeClassAllocator: freearray]
501e00040000-518000000000 ---p  /dev/shm/160736 [SizeClassAllocator]
518000000000-518000040000 rw-p  /dev/shm/160736 [SizeClassAllocator: region data]
518000040000-518dffff0000 ---p  /dev/shm/160736 [SizeClassAllocator]
518dffff0000-518e00000000 rw-p  /dev/shm/160736 [SizeClassAllocator: region metadata]
518e00000000-518e00040000 rw-p  /dev/shm/160736 [SizeClassAllocator: freearray]
518e00040000-540000000000 ---p  /dev/shm/160736 [SizeClassAllocator]
540000000000-540000002000 rw-p  /dev/shm/160736 [SizeClassAllocator: region info]
```

Memory Manager. Primary Allocator.


```
$ cat /proc/$(pidof a.out)/maps
500000000000-501000000000 ---p  /dev/shm/160736 [SizeClassAllocator]
501000000000-501000040000 rw-p  /dev/shm/160736 [SizeClassAllocator: region data]
501000040000-501dfffc0000 ---p  /dev/shm/160736 [SizeClassAllocator]
501dfffc0000-501e00000000 rw-p  /dev/shm/160736 [SizeClassAllocator: region metadata]
501e00000000-501e00040000 rw-p  /dev/shm/160736 [SizeClassAllocator: freearray]
501e00040000-518000000000 ---p  /dev/shm/160736 [SizeClassAllocator]
518000000000-518000040000 rw-p  /dev/shm/160736 [SizeClassAllocator: region data]
518000040000-518dffff0000 ---p  /dev/shm/160736 [SizeClassAllocator]
518dffff0000-518e00000000 rw-p  /dev/shm/160736 [SizeClassAllocator: region metadata]
518e00000000-518e00040000 rw-p  /dev/shm/160736 [SizeClassAllocator: freearray]
518e00040000-540000000000 ---p  /dev/shm/160736 [SizeClassAllocator]
540000000000-540000002000 rw-p  /dev/shm/160736 [SizeClassAllocator: region info]
```

Memory Manager. Primary Allocator.


```
$ cat /proc/$(pidof a.out)/maps
500000000000-501000000000 ---p  /dev/shm/160736 [SizeClassAllocator]
501000000000-501000040000 rw-p  /dev/shm/160736 [SizeClassAllocator: region data]
501000040000-501dfffc0000 ---p  /dev/shm/160736 [SizeClassAllocator]
501dfffc0000-501e00000000 rw-p  /dev/shm/160736 [SizeClassAllocator: region metadata]
501e00000000-501e00040000 rw-p  /dev/shm/160736 [SizeClassAllocator: freearray]
501e00040000-518000000000 ---p  /dev/shm/160736 [SizeClassAllocator]
518000000000-518000040000 rw-p  /dev/shm/160736 [SizeClassAllocator: region data]
518000040000-518dffff0000 ---p  /dev/shm/160736 [SizeClassAllocator]
518dffff0000-518e00000000 rw-p  /dev/shm/160736 [SizeClassAllocator: region metadata]
518e00000000-518e00040000 rw-p  /dev/shm/160736 [SizeClassAllocator: freearray]
518e00040000-540000000000 ---p  /dev/shm/160736 [SizeClassAllocator]
540000000000-540000002000 rw-p  /dev/shm/160736 [SizeClassAllocator: region info]
```

Memory Manager. Primary Allocator.

0x500000000000ULL = kSpaceBeg

```
500000000000-501000000000 ---p  /dev/shm/160736 [SizeClassAllocator]
501000000000-501000040000 rw-p  /dev/shm/160736 [SizeClassAllocator: region data]
501000040000-501dfffc0000 ---p  /dev/shm/160736 [SizeClassAllocator]
501dfffc0000-501e00000000 rw-p  /dev/shm/160736 [SizeClassAllocator: region metadata]
501e00000000-501e00040000 rw-p  /dev/shm/160736 [SizeClassAllocator: freearray]
501e00040000-518000000000 ---p  /dev/shm/160736 [SizeClassAllocator]
518000000000-518000040000 rw-p  /dev/shm/160736 [SizeClassAllocator: region data]
518000040000-518dffff0000 ---p  /dev/shm/160736 [SizeClassAllocator]
518dffff0000-518e00000000 rw-p  /dev/shm/160736 [SizeClassAllocator: region metadata]
518e00000000-518e00040000 rw-p  /dev/shm/160736 [SizeClassAllocator: freearray]
518e00040000-540000000000 ---p  /dev/shm/160736 [SizeClassAllocator]
540000000000-540000002000 rw-p  /dev/shm/160736 [SizeClassAllocator: region info]
```

kSpaceSize (0x40000000000ULL = 4Tb)

Memory Manager. Primary Allocator.

Region 1	Region 2	…											

```
500000000000-501000000000 ---p  /dev/shm/160736 [SizeClassAllocator]
501000000000-501000040000 rw-p  /dev/shm/160736 [SizeClassAllocator: region data]

518000000000-518000040000 rw-p  /dev/shm/160736 [SizeClassAllocator: region data]
```

We have 3 regions

Chunk 1	Chunk 2	Chunk 3	…	Chunk N			Meta Chunk N	…	Meta Chunk 3	Meta Chunk 2	Meta Chunk 1	Free Array

Memory Manager. Primary Allocator.

500000000000
501000000000

518000000000

We have 3 regions

Let’s check what size classes correspond to these regions.

SizeClassMap

500000000000
501000000000
518000000000

https://godbolt.org/z/5M8cs8s7v

```
inline uptr MostSignificantSetBitIndex(uptr x) {
  unsigned long up;
  up = 64 - 1 - __builtin_clzl(x);
  return up;
}

template <class T>
constexpr T Max(T a, T b) {
  return a > b ? a : b;
}

template <class T>
constexpr T Min(T a, T b) {
  return a < b ? a : b;
}

template <uptr kNumBits, uptr kMinSizeLog, uptr kMidSizeLog, uptr kMaxSizeLog,
          uptr kMaxNumCachedHintT, uptr kMaxBytesCachedLog>
class SizeClassMap {
  static const uptr kMinSize = 1 << kMinSizeLog;
  static const uptr kMidSize = 1 << kMidSizeLog;
  static const uptr kMidClass = kMidSize / kMinSize;
  static const uptr S = kNumBits - 1;
  static const uptr M = (1 << S) - 1;

 public:
  // kMaxNumCachedHintT is a power of two. It serves as a hint
  // for the size of TransferBatch, the actual size could be a bit smaller.
  static const uptr kMaxNumCachedHint = kMaxNumCachedHintT;

  static const uptr kMaxSize = 1UL << kMaxSizeLog;
  static const uptr kNumClasses =
      kMidClass + ((kMaxSizeLog - kMidSizeLog) << S) + 1 + 1;
  static const uptr kLargestClassID = kNumClasses - 2;
  static const uptr kBatchClassID = kNumClasses - 1;
  static const uptr kNumClassesRounded =
      kNumClasses <= 32  ? 32 :
      kNumClasses <= 64  ? 64 :
      kNumClasses <= 128 ? 128 : 256;
```

SizeClassMap

```
500000000000
501000000000
518000000000
```

https://godbolt.org/z/5M8cs8s7v

```
typedef SizeClassMap<3, 4, 8, 17, 128, 16> DefaultSizeClassMap;
DefaultSizeClassMap::Print();
for (int i=0; i<DefaultSizeClassMap::kNumClasses; ++i)
    printf("%d - %p\n", i, GetRegionBeginBySizeClass(i));
```

SizeClassMap

500000000000
501000000000
518000000000

https://godbolt.org/z/5M8cs8s7v

```
typedef SizeClassMap<3, 4, 8, 17, 128, 16> DefaultSizeClassMap;
DefaultSizeClassMap::Print();
for (int i=0; i<DefaultSizeClassMap::kNumClasses; ++i)
    printf("%d - %p\n", i, GetRegionBeginBySizeClass(i));
```

```
0 - 0x500000000000
1 - 0x501000000000
…
24 - 0x518000000000
…
53 - 0x535000000000
```

SizeClassMap

500000000000
501000000000
518000000000

https://godbolt.org/z/5M8cs8s7v

```
typedef SizeClassMap<3, 4, 8, 17, 128, 16> DefaultSizeClassMap;
DefaultSizeClassMap::Print();
for (int i=0; i<DefaultSizeClassMap::kNumClasses; ++i)
    printf("%d - %p\n", i, GetRegionBeginBySizeClass(i));
```

```
0 - 0x500000000000
1 - 0x501000000000
…
24 - 0x518000000000
…
```

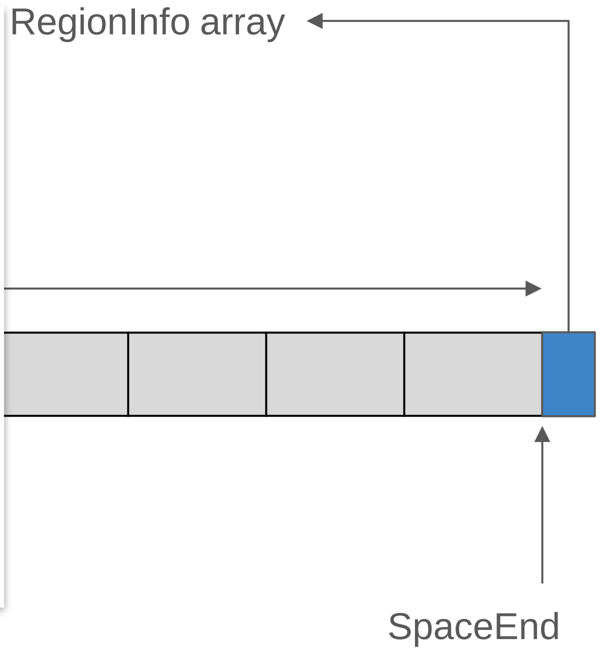
```
c00 => s: 0 diff: +0 00% l 0 cached: 0 0; id 0
c01 => s: 16 diff: +16 00% l 4 cached: 128 2048; id 1
…
c24 => s: 1024 diff: +128 14% l 10 cached: 64 65536; id 24
…
```

```
int main() {
    malloc(1);
    int i=0;
    scanf("%d", &i);
}
```

Memory Manager. Primary Allocator.

SizeClassAllocator64:

```
struct ALIGNED(SANITIZER_CACHE_LINE_SIZE) RegionInfo {
  Mutex mutex;
  uptr num_freed_chunks;  // Number of elements in the freearray.
  uptr mapped_free_array;  // Bytes mapped for freearray.
  uptr allocated_user;  // Bytes allocated for user memory.
  uptr allocated_meta;  // Bytes allocated for metadata.
  uptr mapped_user;  // Bytes mapped for user memory.
  uptr mapped_meta;  // Bytes mapped for metadata.
  u32 rand_state;  // Seed for random shuffle, used if kRandomShuffleChunks.
  bool exhausted;  // Whether region is out of space for new chunks.
  Stats stats;
  ReleaseToOsInfo rtoi;
};
```

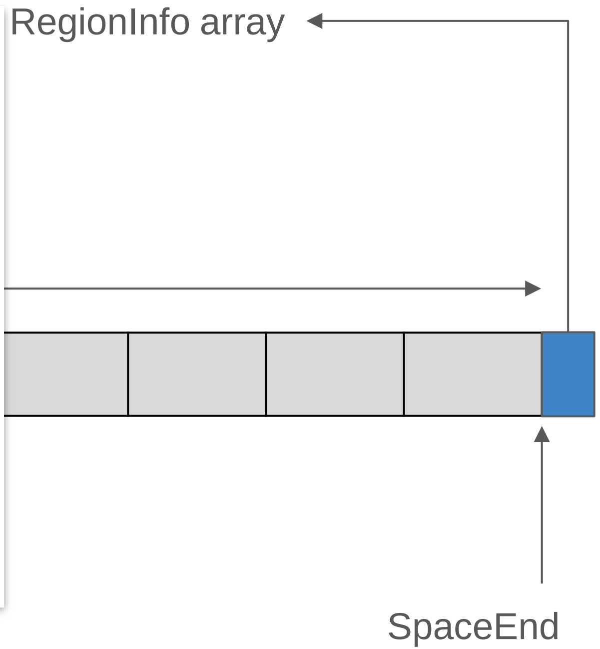


kSpaceBeg
(0x500000000000ULL)

Memory Manager. Primary Allocator.

SizeClassAllocator64:

```
struct ALIGNED(SANITIZER_CACHE_LINE_SIZE) RegionInfo {
  Mutex mutex;
  uptr num_freed_chunks;  // Number of elements in the freearray.
  uptr mapped_free_array;  // Bytes mapped for freearray.
  uptr allocated_user;  // Bytes allocated for user memory.
  uptr allocated_meta;  // Bytes allocated for metadata.
  uptr mapped_user;  // Bytes mapped for user memory.
  uptr mapped_meta;  // Bytes mapped for metadata.
  u32 rand_state;  // Seed for random shuffle, used if kRandomShuffleChunks.
  bool exhausted;  // Whether region is out of space for new chunks.
  Stats stats;
  ReleaseToOsInfo rtoi;
};
```



kSpaceBeg
(0x500000000000ULL)

Thread safe but not effective in multithreaded app

Memory Manager. Primary Allocator.

SizeClassAllocator64LocalCache:

Memory Manager. Primary Allocator.

SizeClassAllocator64LocalCache:

Objects of this class are always in **thread-local storage (TLS)**

Memory Manager. Primary Allocator.

SizeClassAllocator64LocalCache:

Objects of this class are always in **thread-local storage (TLS)**

One per thread

```
static THREADLOCAL AllocatorCache allocator_cache;
AllocatorCache *GetAllocatorCache() { return &allocator_cache; }
```

Memory Manager. Primary Allocator.

SizeClassAllocator64LocalCache:

Objects of this class are always in **thread-local storage (TLS)**

One per thread

```
static THREADLOCAL AllocatorCache allocator_cache;
AllocatorCache *GetAllocatorCache() { return &allocator_cache; }
```

Each SizeClassAllocator64LocalCache object contains array of free chunks:

```
struct PerClass {
 u32 count;
 u32 max_count;
 uptr class_size;
 CompactPtrT chunks[2 * SizeClassMap::kMaxNumCachedHint];
};
PerClass per_class_[kNumClasses];
```

Memory Manager. Primary Allocator.

SizeClassAllocator64LocalCache:

```
struct PerClass {
 u32 count;
 u32 max_count;
 uptr class_size;
 CompactPtrT chunks[2 * SizeClassMap::kMaxNumCachedHint];
};
PerClass per_class_[kNumClasses];
```

It borrows free chunks from SizeClassAllocator64 in a lazy way – if the **count** for necessary size class is zero, it will refill free chunks reserve for this size class only.

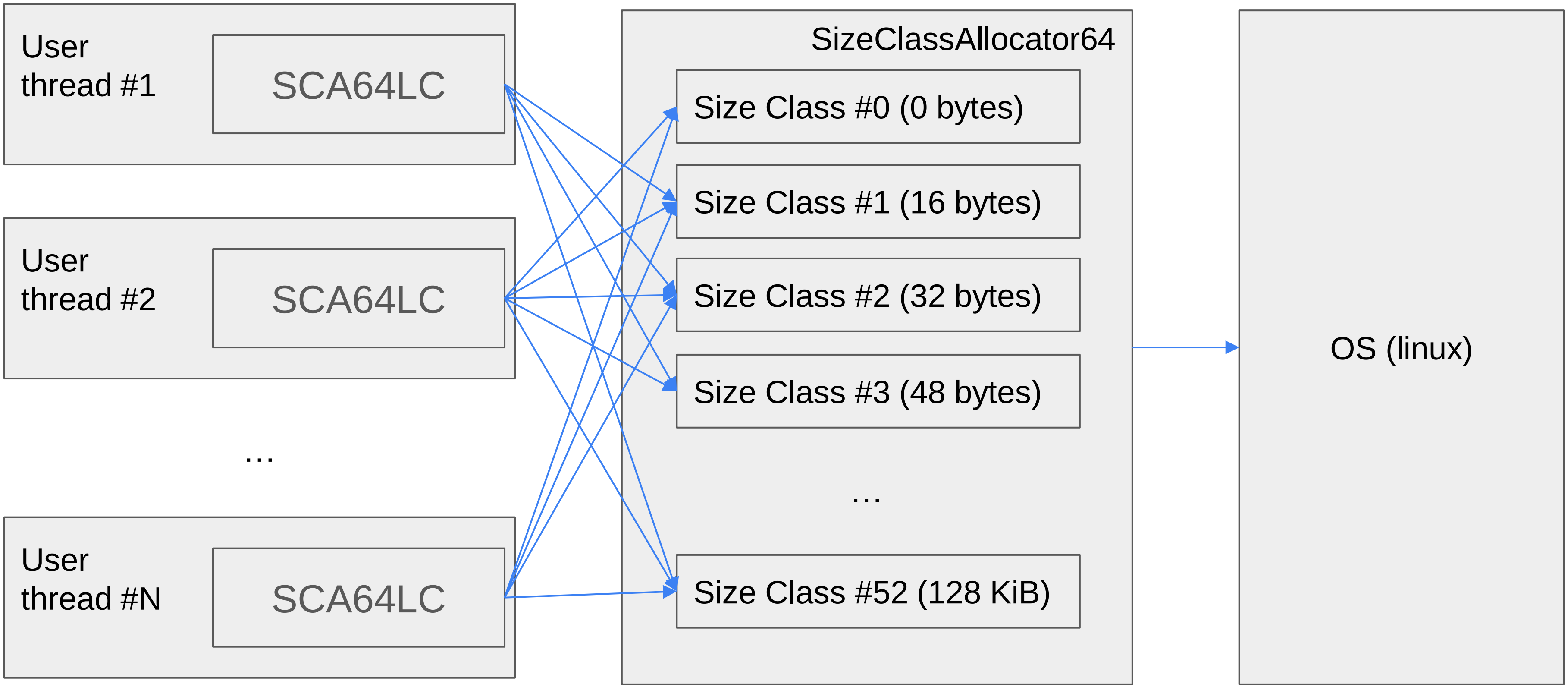
Memory Manager. Primary Allocator.

SizeClassAllocator64LocalCache (SCALC):

```
struct PerClass {
 u32 count;
 u32 max_count;
 uptr class_size;
 CompactPtrT chunks[2 * SizeClassMap::kMaxNumCachedHint];
};
PerClass per_class_[kNumClasses];
```

Personal quick access coin purse
for each thread

Memory Manager. PrimaryAllocator



SCALC == SizeClassAllocator64LocalCache

Memory Manager. PrimaryAllocator

SCALC == SizeClassAllocator64LocalCache

Memory Manager. Secondary Allocator

Memory Manager. Secondary Allocator

Which is LargeMmapAllocator

Memory Manager. Secondary Allocator

Which is LargeMmapAllocator

```
// This class can (de)allocate only large chunks of memory using mmap/unmap.
// The main purpose of this allocator is to cover large and rare allocation
// sizes not covered by more efficient allocators (e.g. SizeClassAllocator64).
```

Memory Manager. Secondary Allocator

Which is LargeMmapAllocator

```
// This class can (de)allocate only large chunks of memory using mmap/unmap.
// The main purpose of this allocator is to cover large and rare allocation
// sizes not covered by more efficient allocators (e.g. SizeClassAllocator64).
```

So, it just uses mmap/unmap for each memmory allocation

Memory Manager. Secondary Allocator

Which is LargeMmapAllocator

```
// This class can (de)allocate only large chunks of memory using mmap/unmap.
// The main purpose of this allocator is to cover large and rare allocation
// sizes not covered by more efficient allocators (e.g. SizeClassAllocator64).
```

So, it just uses mmap/unmap for each memmory allocation

For each allocation it allocates a memory chunk (via mmap) and stores pointer to it in internal array:

Memory Manager. Secondary Allocator

Which is LargeMmapAllocator

```
// This class can (de)allocate only large chunks of memory using mmap/unmap.
// The main purpose of this allocator is to cover large and rare allocation
// sizes not covered by more efficient allocators (e.g. SizeClassAllocator64).
```

So, it just uses mmap/unmap for each memmory allocation

For each allocation it allocates a memory chunk (via mmap) and stores pointer to it in internal array:

Uses first page (4096 bytes) for storing a header:

```
struct Header {
   uptr map_beg;
   uptr map_size;
   uptr size;
   uptr chunk_idx;
 };
```

Memory Manager. Secondary Allocator

Which is LargeMmapAllocator

```
// This class can (de)allocate only large chunks of memory using mmap/unmap.
// The main purpose of this allocator is to cover large and rare allocation
// sizes not covered by more efficient allocators (e.g. SizeClassAllocator64).
```

So, it just uses mmap/unmap for each memmory allocation

For each allocation it allocates a memory chunk (via mmap) and stores pointer to it in internal array:

Uses first page (4096 bytes) for storing a header:
… and metadata

```
struct Header {
   uptr map_beg;
   uptr map_size;
   uptr size;
   uptr chunk_idx;
  };
```

Memory Manager. Secondary Allocator

Which is LargeMmapAllocator

```
// This class can (de)allocate only large chunks of memory using mmap/unmap.
// The main purpose of this allocator is to cover large and rare allocation
// sizes not covered by more efficient allocators (e.g. SizeClassAllocator64).
```

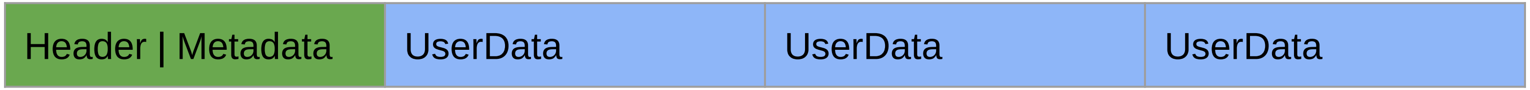
So, it just uses mmap/unmap for each memmory allocation

For each allocation it allocates a memory chunk (via mmap) and stores pointer to it in internal array:

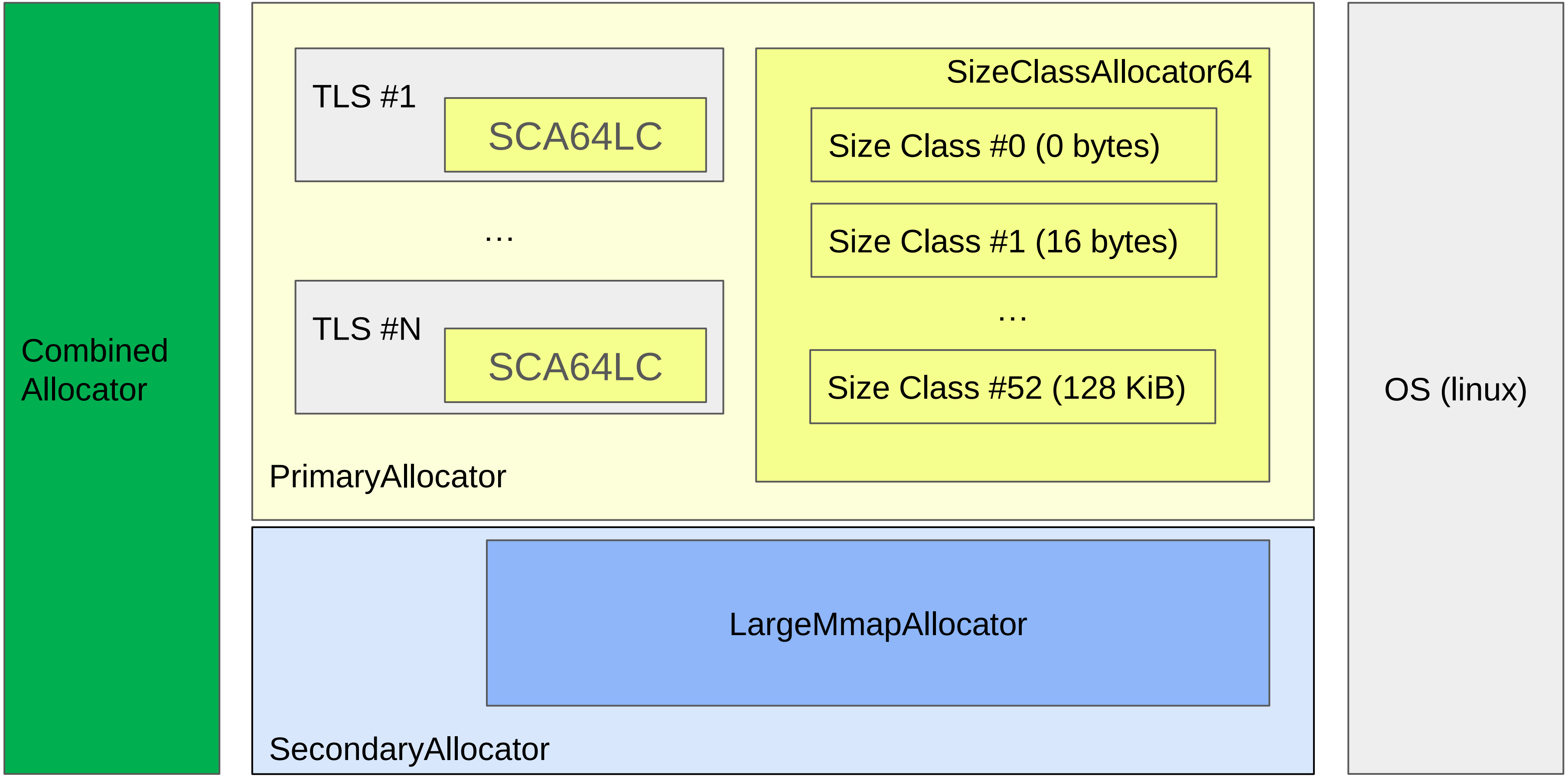
Uses first page (4096 bytes) for storing a header:
… and metadata

The rest space is for user data

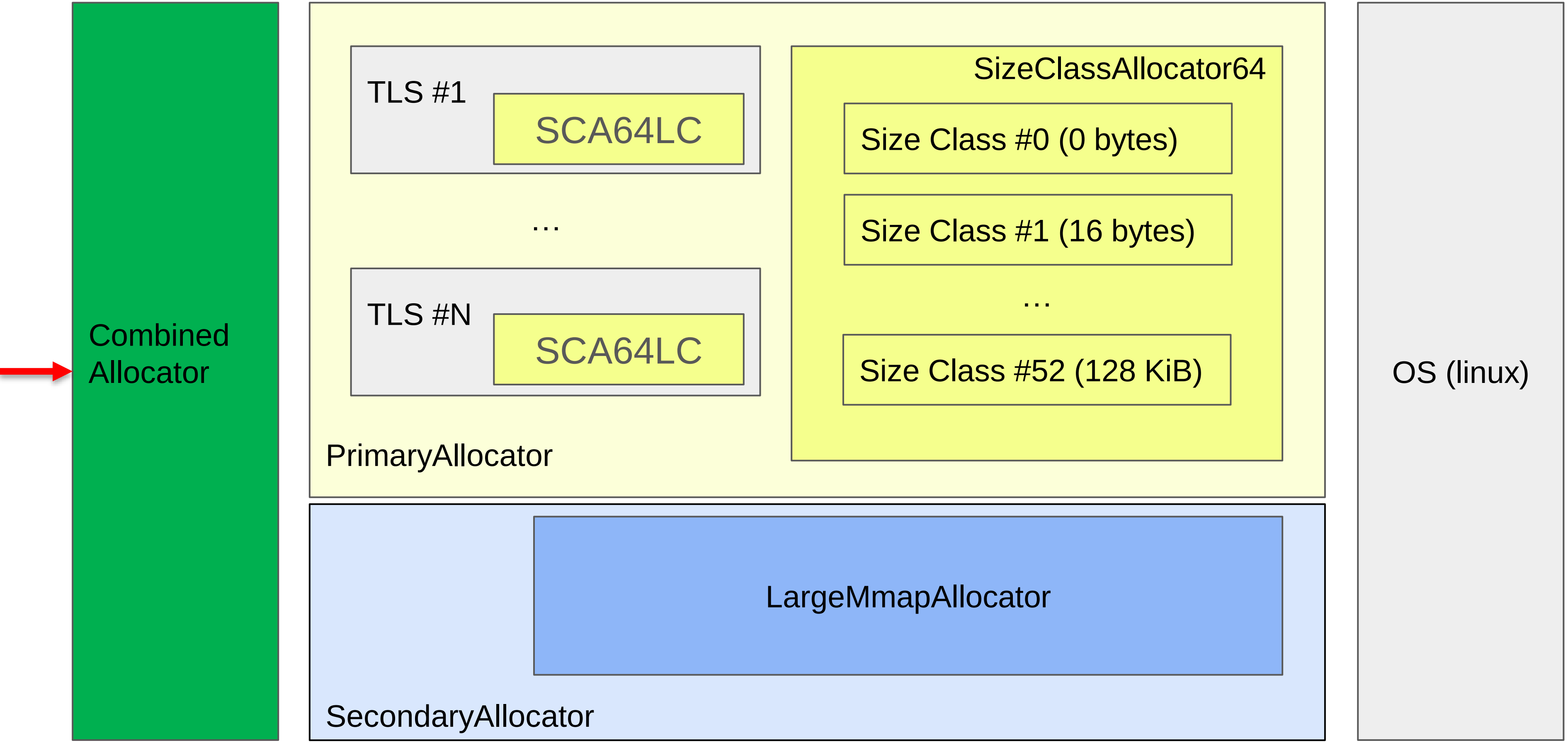
```
struct Header {
   uptr map_beg;
   uptr map_size;
   uptr size;
   uptr chunk_idx;
 };
```



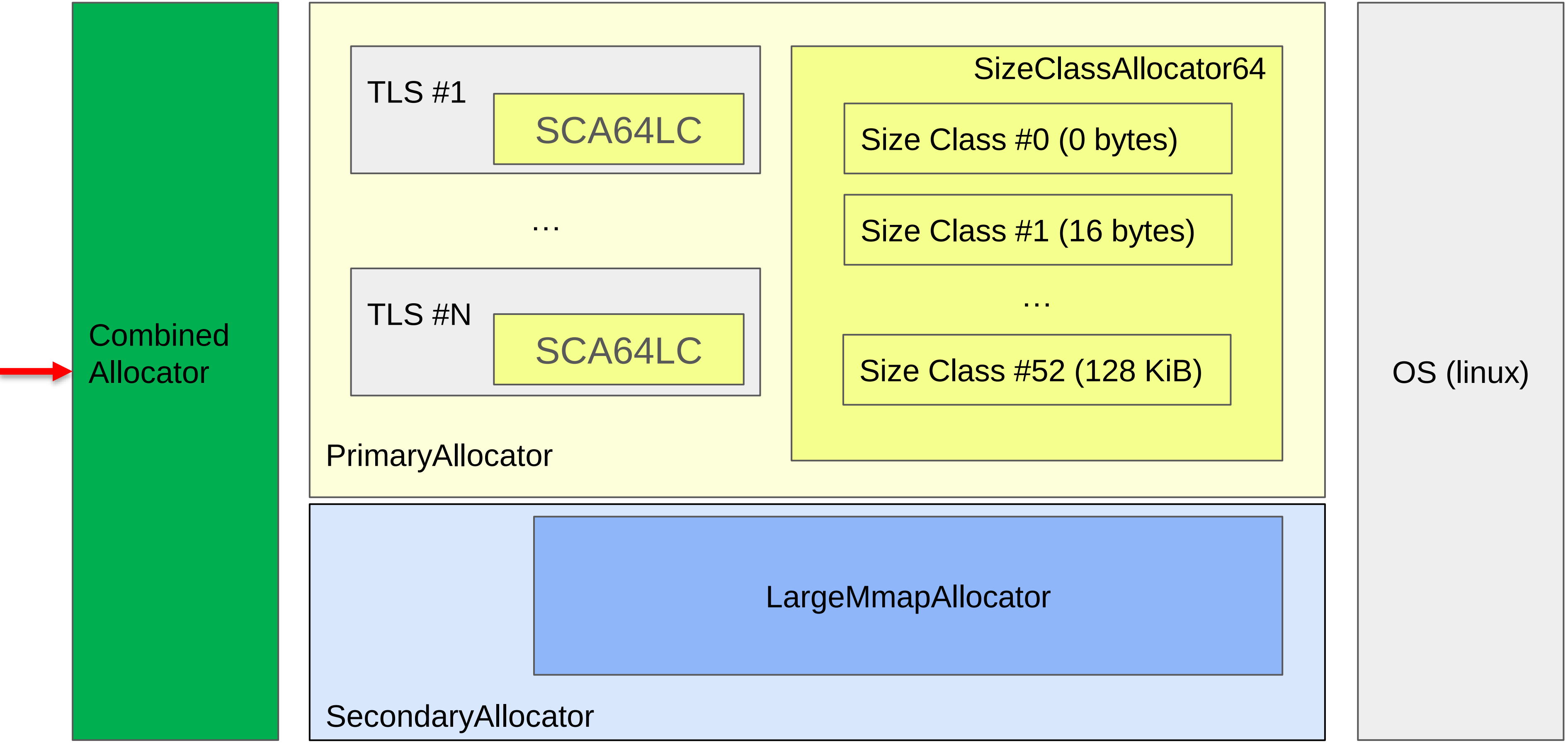
Memory Manager. Full picture



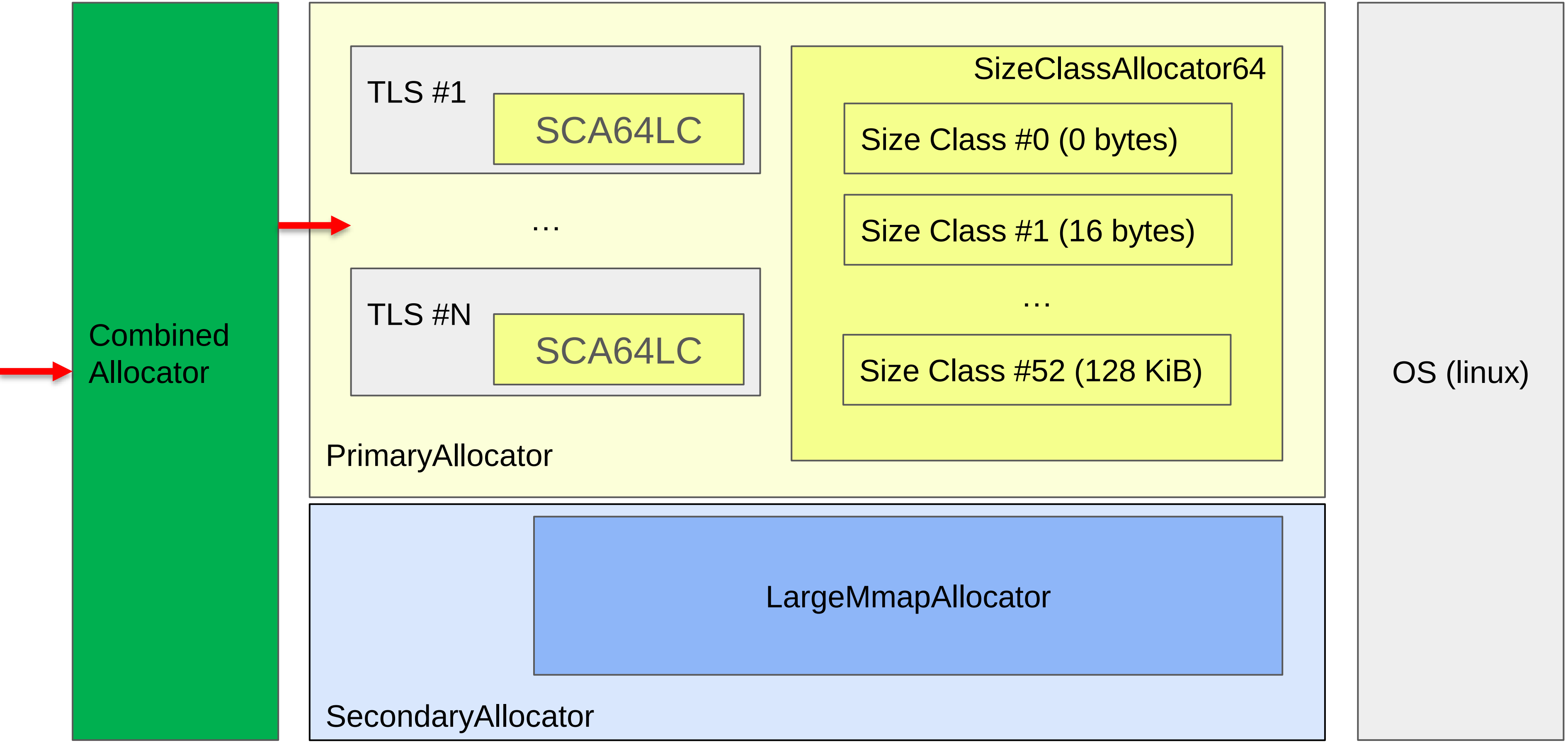
User calls CombinedAllocator to allocate memory



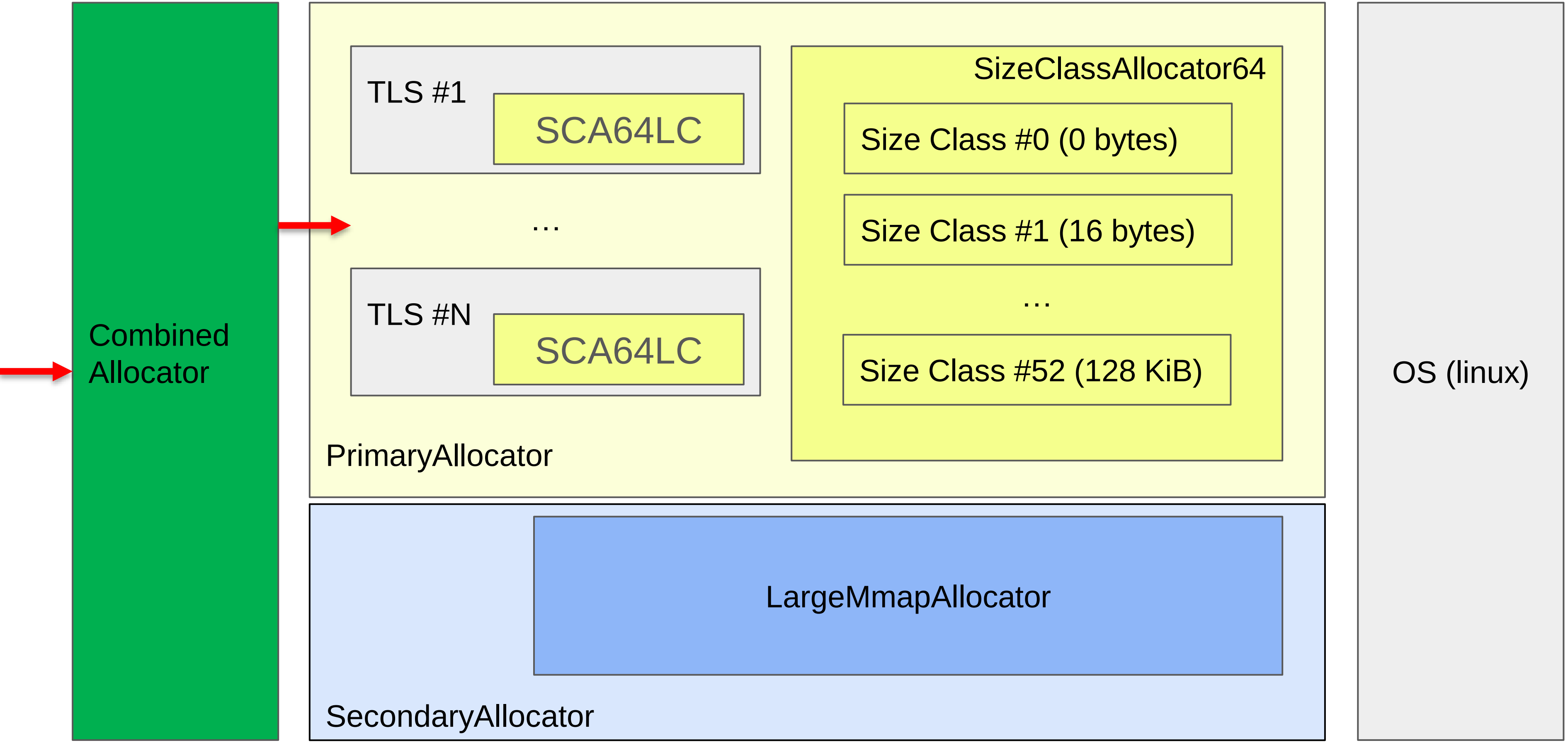
CombinedAllocator calls PrimaryAllocator firstly to allocate memory



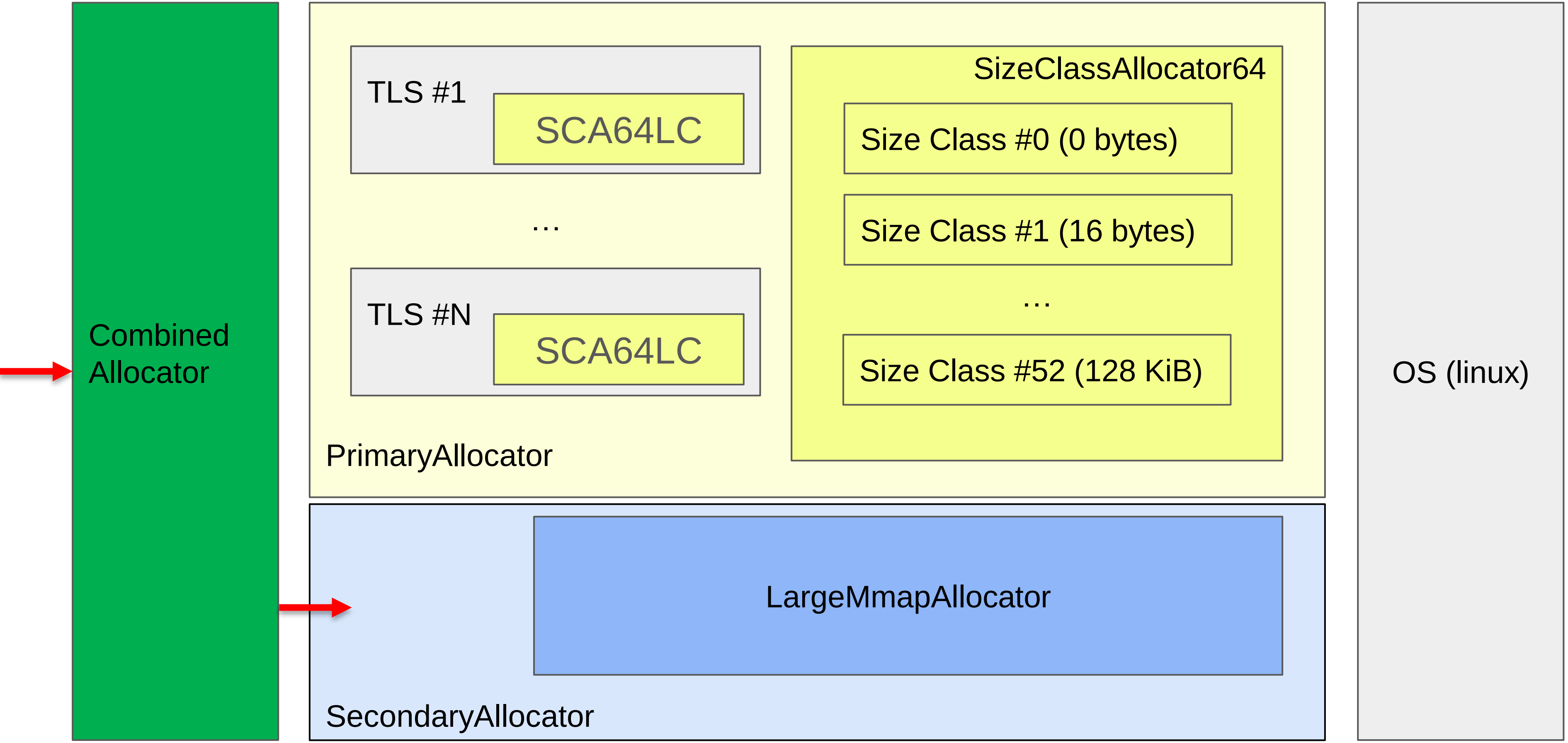
CombinedAllocator calls PrimaryAllocator firstly to allocate memory



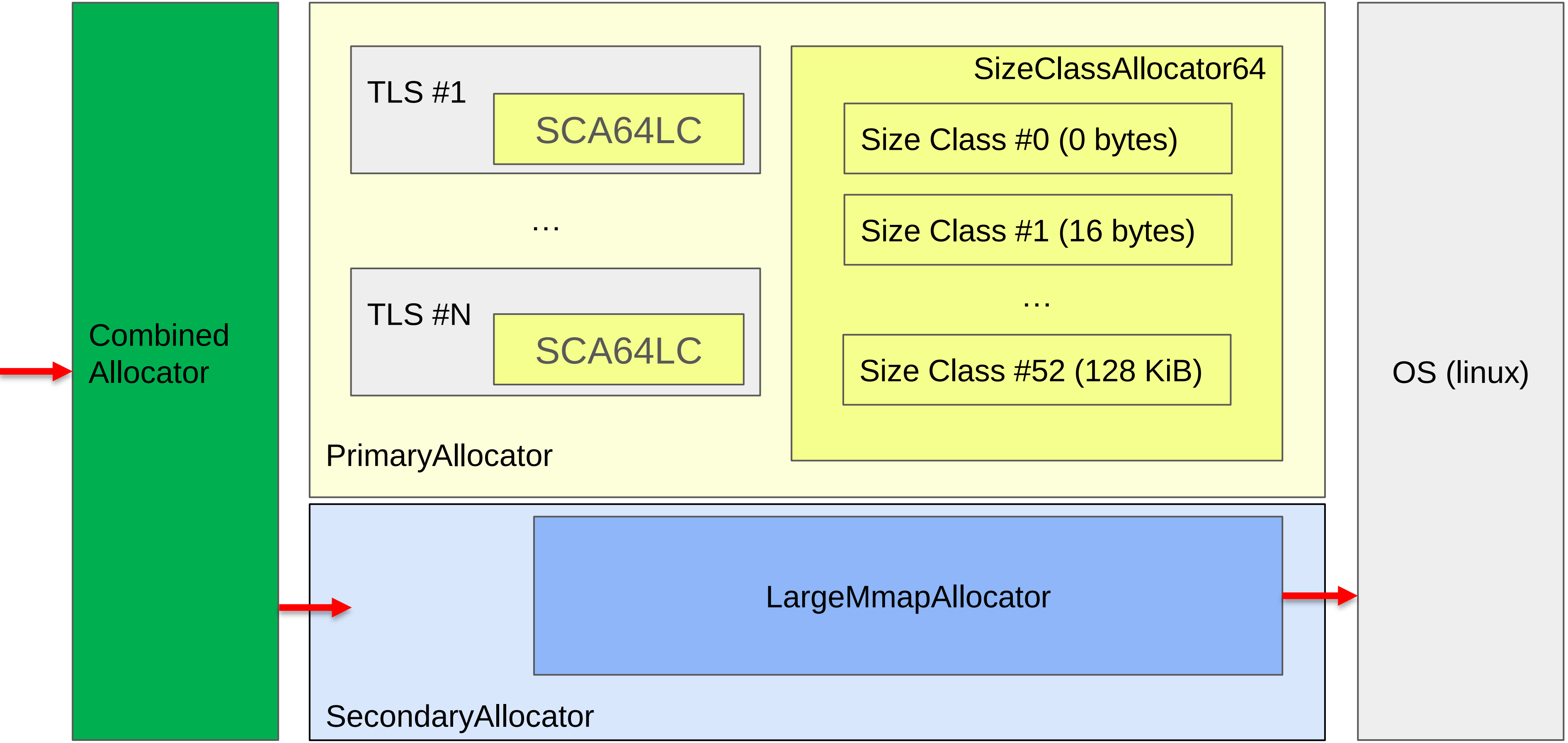
PrimaryAllocator checks if it can allocate this memory (hint: it can’t if size > 128 KiB)



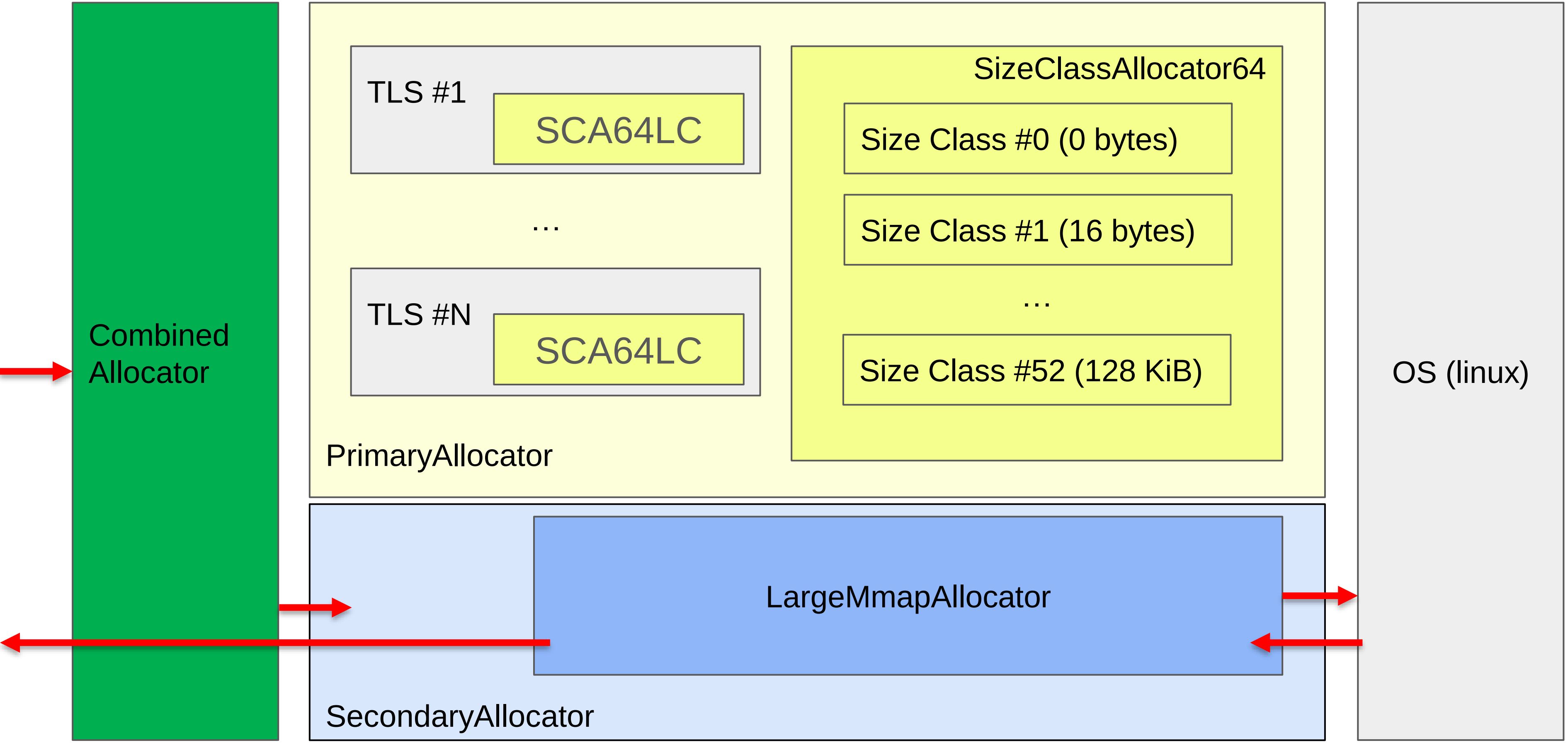
If PrimaryAllocator failed, CombinedAllocator will call SecondaryAllocator to allocate this alloc request



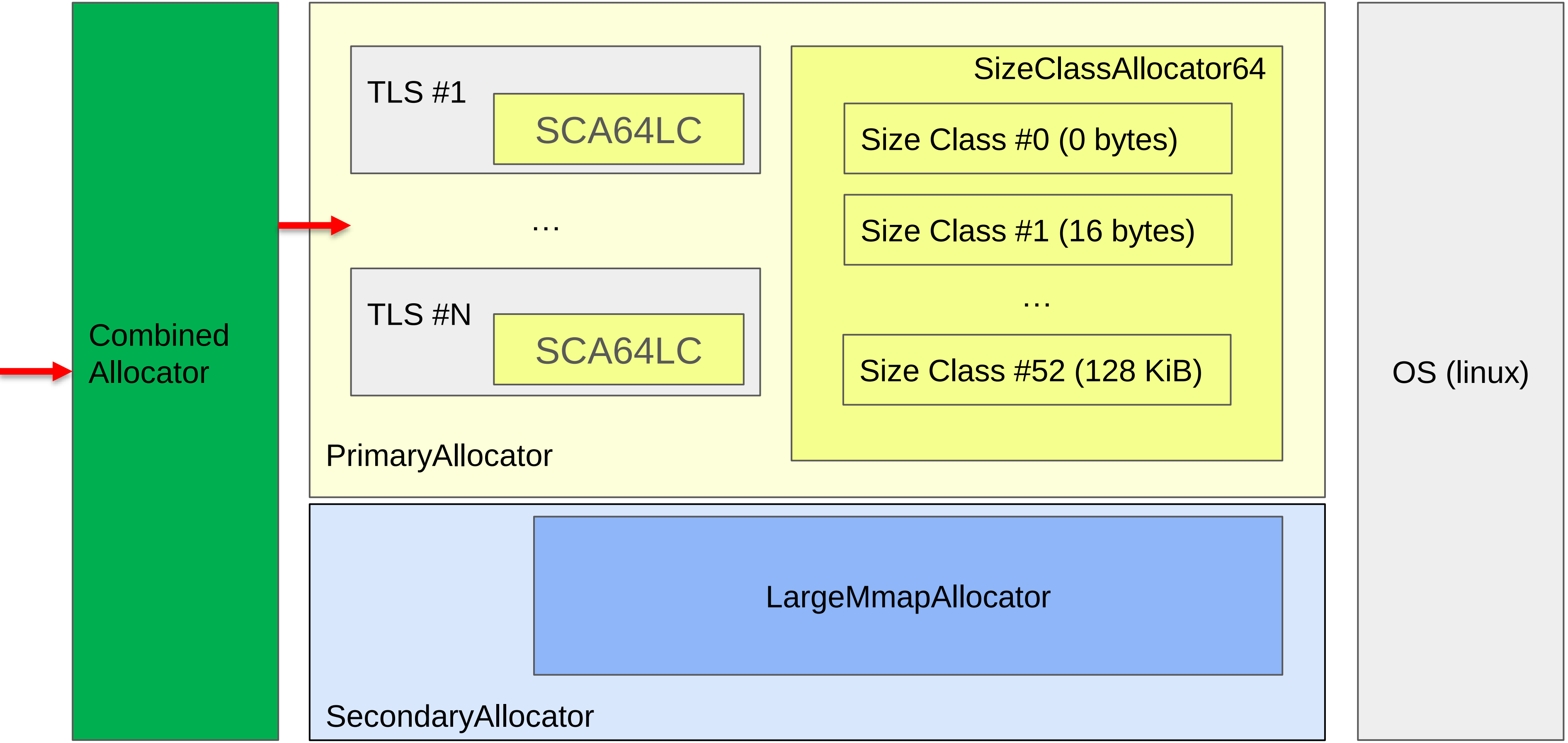
SecondaryAllocator (LargeMmapAllocator) will call mmap to allocate this memory



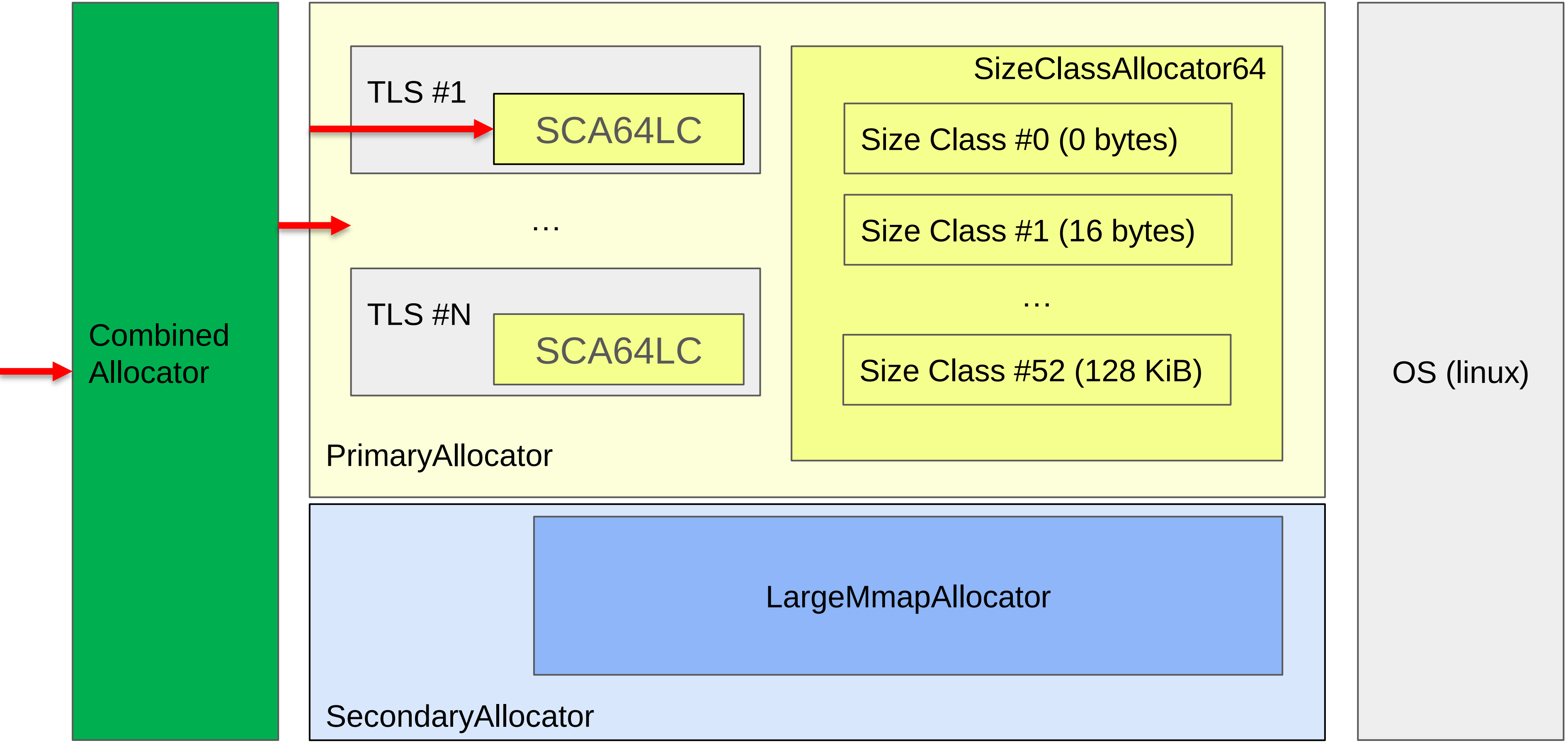
Pointer to the beginning of userdata memory will be returned



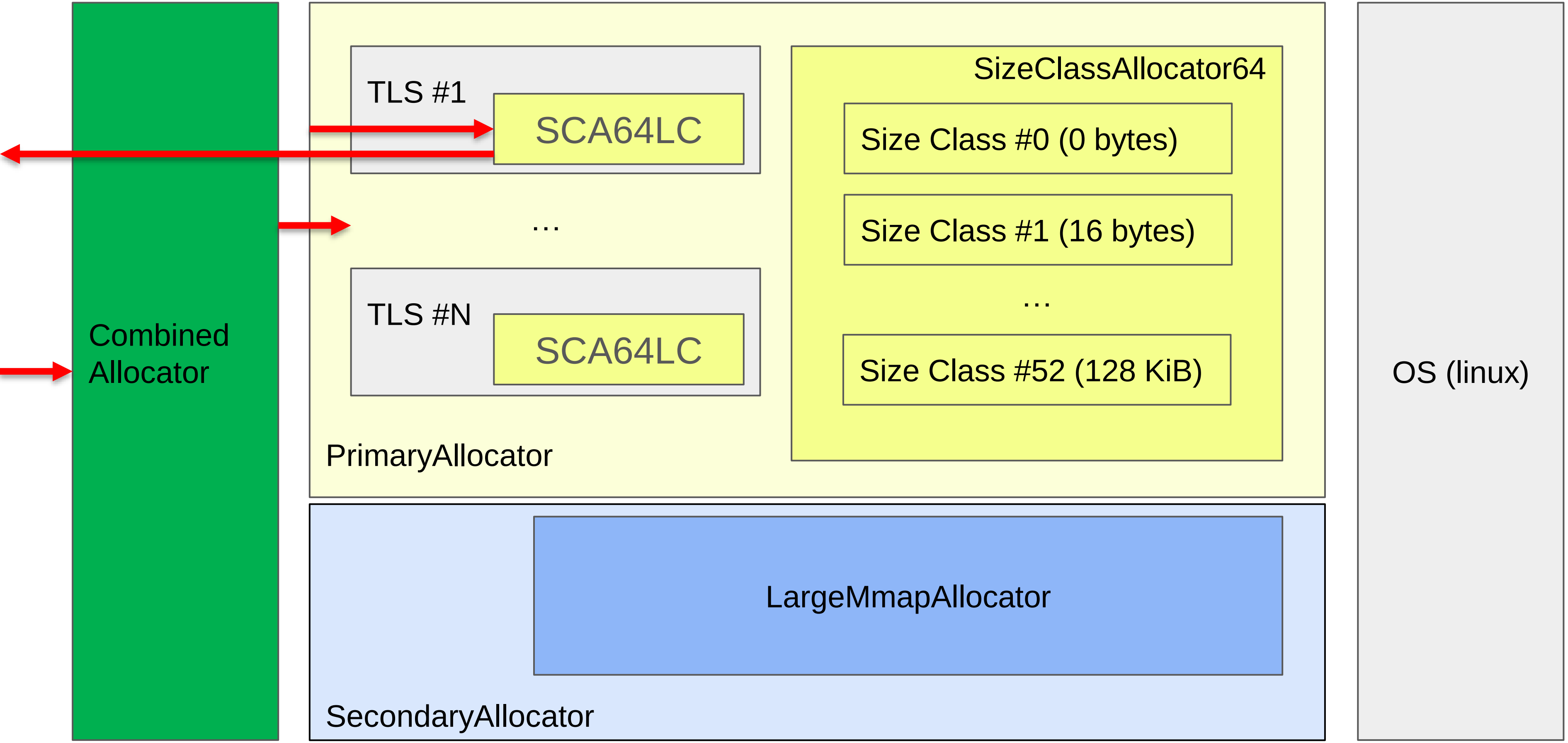
If memory size is suitable for PrimaryAllocator it will proceed the request



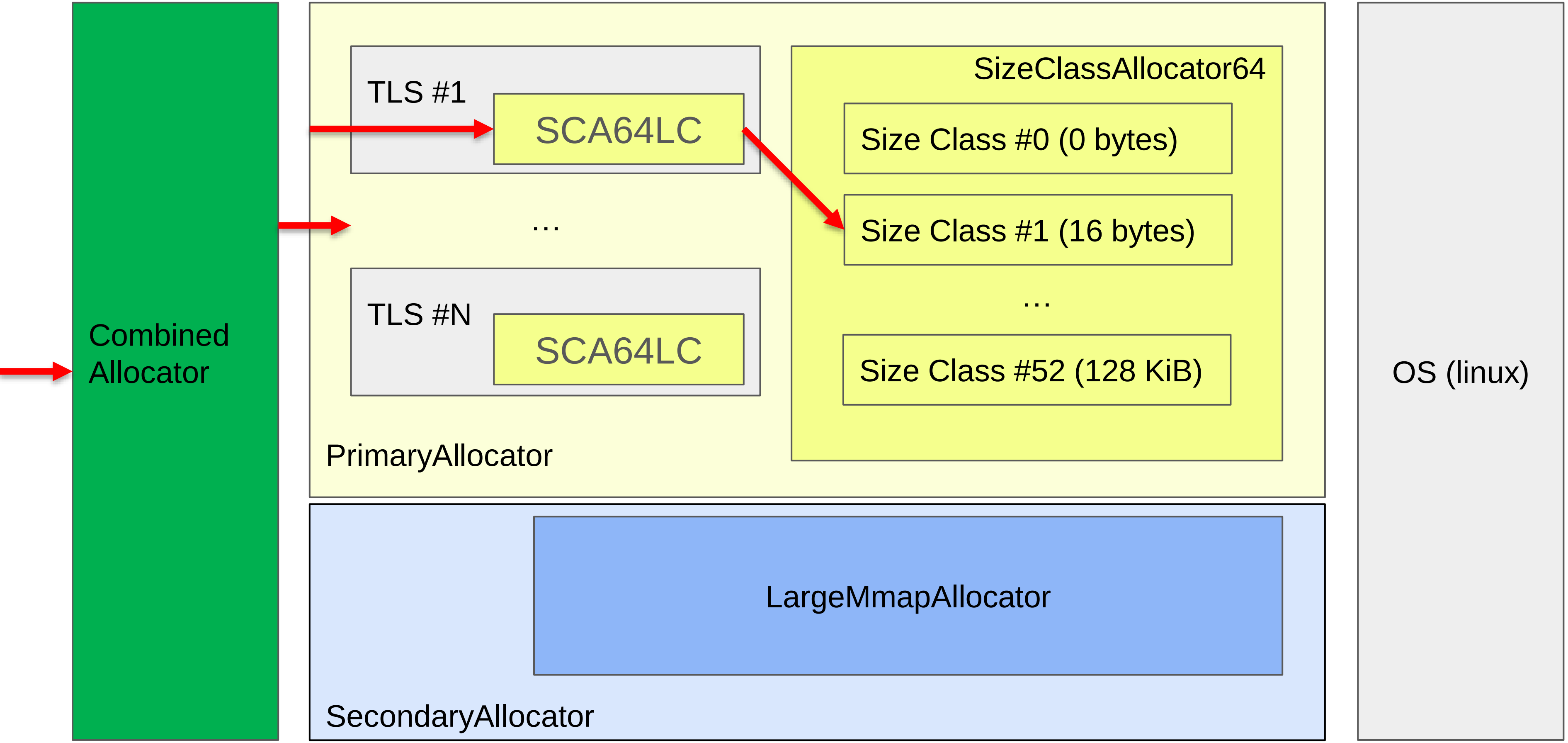
SizeClassAllocator64LocalCache will be used for allocation



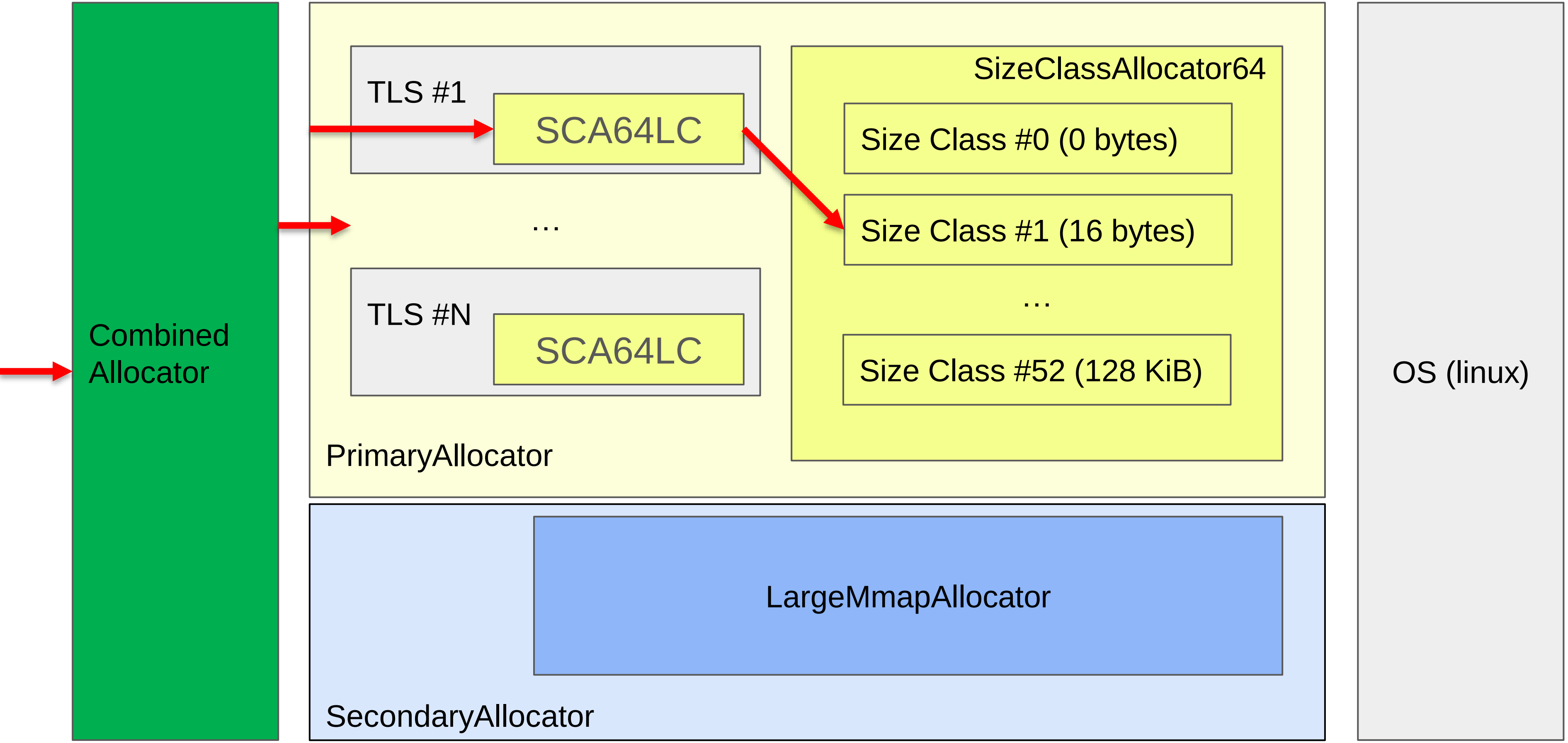
If LocalCache has free chunk for a given size it will be returned to user



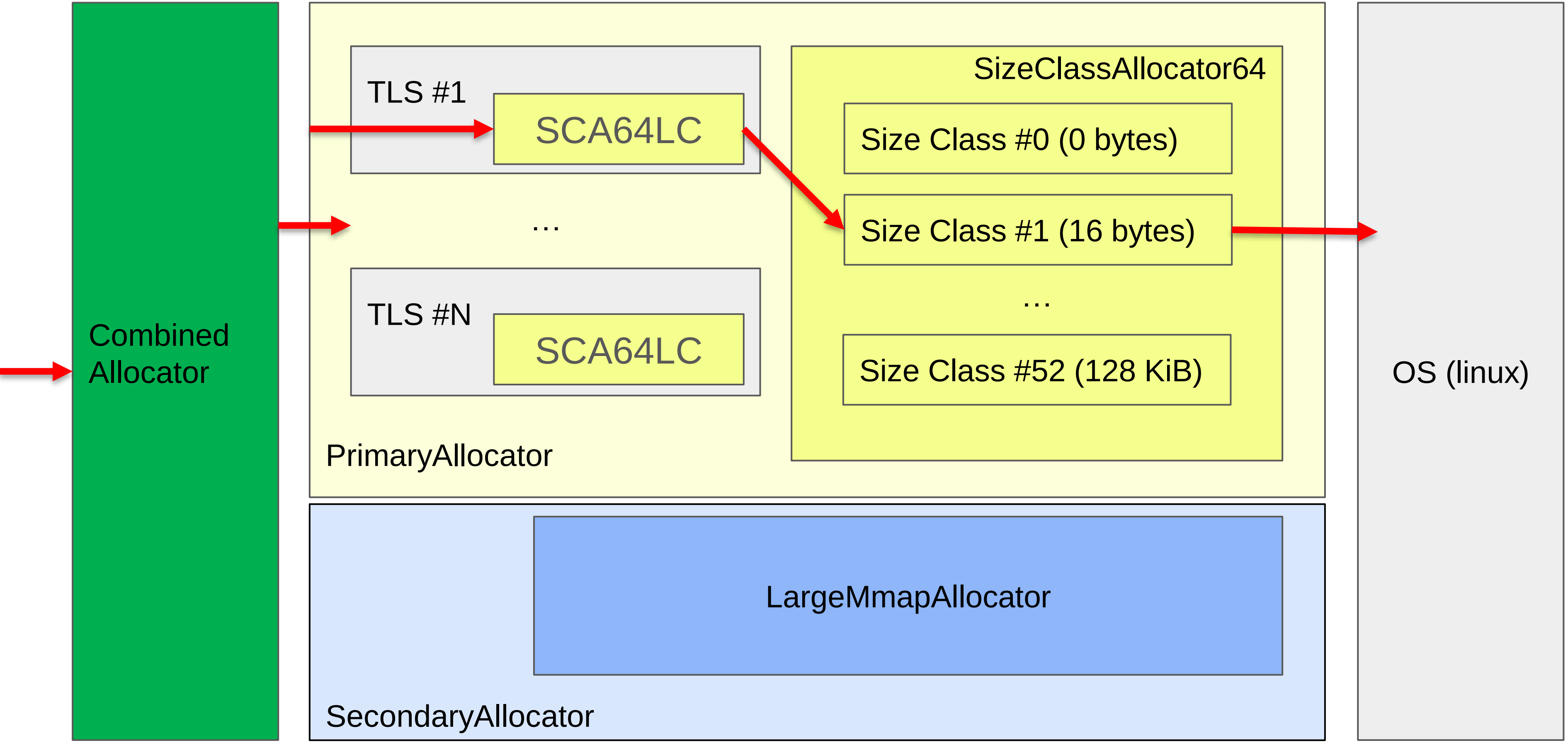
If not – LocalCache will refill its free chunk list for this size. Will borrow it from SizeClassAllocator64



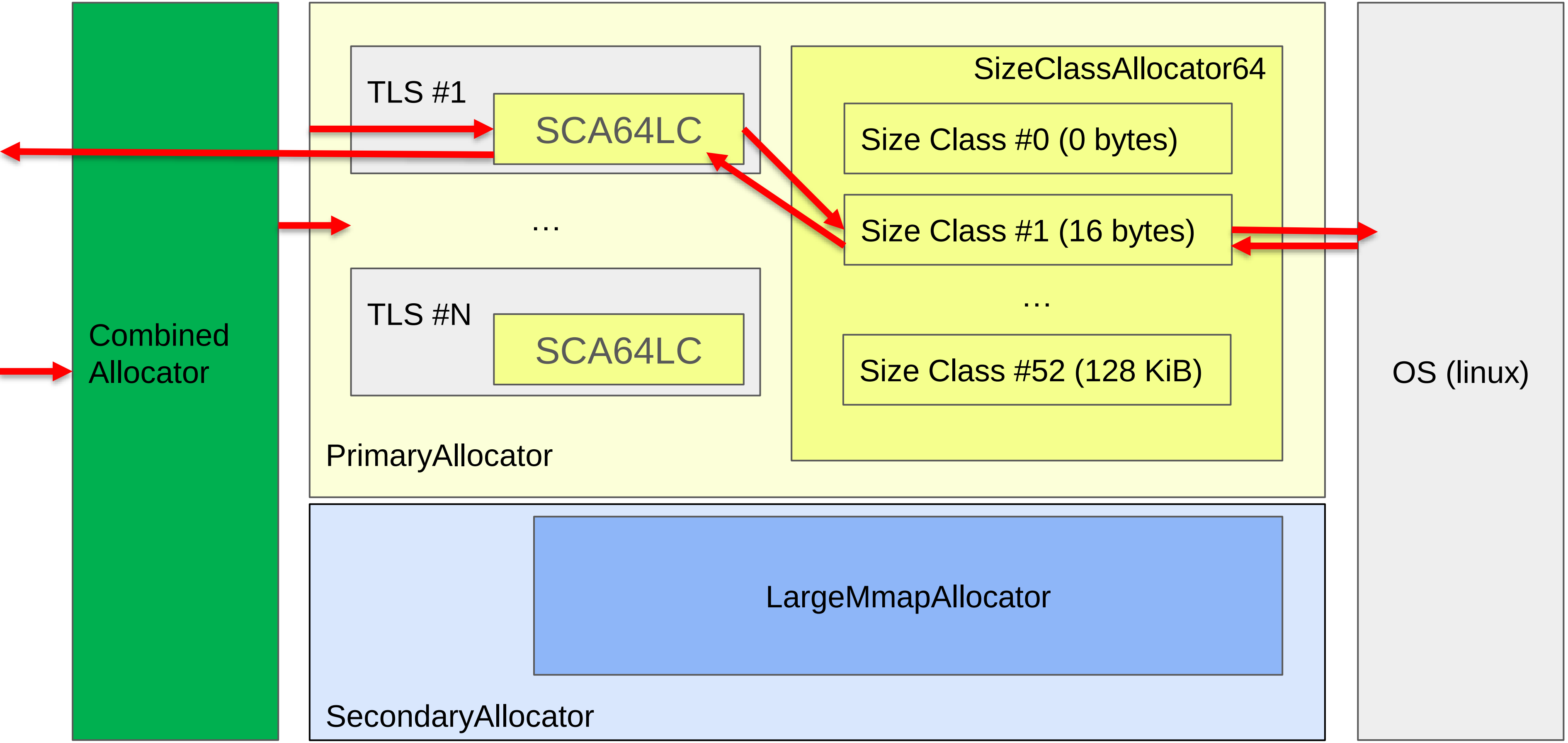
If SizeClassAllocator64 has free chunks for this size (from previous deallocations) – it will return them.



Otherwise, it will allocate new chunks in corresponding region via mmap



After refilling LocalCache will have free chunks of required size. Pointer to one will be returned to user



Leak Sanitizer algorithm

Leak Sanitizer algorithm

It works as a conservative mark and sweep GC

Leak Sanitizer algorithm

It works as a conservative mark and sweep GC

With one exception

Leak Sanitizer algorithm

It works as a conservative mark and sweep GC

With one exception

It does not collect any garbage

Leak Sanitizer algorithm

It works as a conservative mark and sweep GC

With one exception

It does not collect any garbage

It tells you that you need to collect it

Leak Sanitizer algorithm

It works as a conservative mark and sweep GC

With one exception

It does not collect any garbage

It tells you that you need to collect it

But anyway, it should find
the garbage first

Leak Sanitizer algorithm

General algorithm:

Leak Sanitizer algorithm

General algorithm:

1. Stop the world

Leak Sanitizer algorithm

General algorithm:

1. Stop the world
2. Find all roots

Leak Sanitizer algorithm

General algorithm:

1. Stop the world
2. Find all roots
3. Mark all reachable chunks

Leak Sanitizer algorithm

General algorithm:

1. Stop the world
2. Find all roots
3. Mark all reachable chunks
4. All not marked chunks are leaks

Leak Sanitizer algorithm

General algorithm:

1. Stop the world
2. Find all roots
3. Mark all reachable chunks
4. All not marked chunks are leaks
5. Mark indirect leaks

Leak Sanitizer algorithm

General algorithm:

1. Stop the world
2. Find all roots
3. Mark all reachable chunks
4. All not marked chunks are leaks
5. Mark indirect leaks
6. Report all leaks

Leak Sanitizer: stop the world

General algorithm:

1. **Stop the world**
2. Find all roots
3. Mark all reachable chunks
4. All not marked chunks are leaks
5. Mark indirect leaks
6. Report all leaks

Leak Sanitizer: stop the world

LSAN runtime creates a special worker thread

Leak Sanitizer: stop the world

LSAN runtime creates a special worker thread

All other threads will be stopped via `internal_ptrace(PTRACE_ATTACH, tid, nullptr, nullptr)` call.

Leak Sanitizer: stop the world

LSAN runtime creates a special worker thread

All other threads will be stopped via `internal_ptrace(PTRACE_ATTACH, tid, nullptr, nullptr)` call.

That’s why LSAN can’t work under GDB:

Leak Sanitizer: stop the world

LSAN runtime creates a special worker thread

All other threads will be stopped via `internal_ptrace(PTRACE_ATTACH, tid, nullptr, nullptr)` call.

That's why LSAN can't work under GDB:

only one thread could be attached to a target thread

Leak Sanitizer: stop the world

LSAN runtime creates a special worker thread

All other threads will be stopped via `internal_ptrace(PTRACE_ATTACH, tid, nullptr, nullptr)` call.

That's why LSAN can't work under GDB:

only one thread could be attached to a target thread

```
$ man ptrace
ERRORS
…
 EPERM  The specified process cannot be traced.  This could be because the tracer has  in-
        sufficient  privileges  (the  required capability is CAP_SYS_PTRACE); unprivileged
        processes cannot trace processes that they cannot send signals to or those running
        set-user-ID/set-group-ID   programs,  for  obvious  reasons.   Alternatively,  the
        process may already be being traced, or (on kernels before 2.6.26) be init(1) (PID
        1).
```

Leak Sanitizer: find all roots

General algorithm:

1. Stop the world
2. **Find all roots**
3. Mark all reachable chunks
4. All not marked chunks are leaks
5. Mark indirect leaks
6. Report all leaks

Leak Sanitizer: find all roots

There are 4 possible places where roots could be located:

Leak Sanitizer: find all roots

There are 4 possible places where roots could be located:

1. Any memory chunk which was explicitly marked as ignored via `__lsan_ignore_object(const void *p)` call by user

Leak Sanitizer: find all roots

There are 4 possible places where roots could be located:

1. Any memory chunk which was explicitly marked as ignored via `__lsan_ignore_object(const void *p)` call by user
2. Any writable global region

Leak Sanitizer: find all roots

There are 4 possible places where roots could be located:

1. Any memory chunk which was explicitly marked as ignored via `__lsan_ignore_object(const void *p)` call by user
2. Any writable global region
3. Any thread related memory region: TLS, registers, stack

Leak Sanitizer: find all roots

There are 4 possible places where roots could be located:

1. Any memory chunk which was explicitly marked as ignored via `__lsan_ignore_object(const void *p)` call by user
2. Any writable global region
3. Any thread related memory region: TLS, registers, stack
4. Any mmaped memory which was explicitly marked as roots source by user via `__lsan_register_root_region(const void *p, size_t size);` call

Leak Sanitizer: find all roots

There are 4 possible places where roots could be located:

1. Any memory chunk which was explicitly marked as ignored via `__lsan_ignore_object(const void *p)` call by user
2. Any writable global region
3. **Any thread related memory region: TLS, registers, stack**
4. Any mmaped memory which was explicitly marked as roots source by user via `__lsan_register_root_region(const void *p, size_t size);` call

Thread related memory could be excluded from roots source via LSAN_OPTIONS:

```
$ LSAN_OPTIONS=use_registers=0:use_stacks=0:use_tls=0 ./a.out
…
$ export LSAN_OPTIONS=use_registers=0:use_stacks=0:use_tls=0
$ ./a.out
…
```

Leak Sanitizer: find all roots

We have a memory chunk. What next?

Leak Sanitizer: find all roots

We have a memory chunk. What next?
Scan chunk to detect pointers.

Leak Sanitizer: find all roots

We have a memory chunk. What next?
Scan chunk to detect pointers.
We don’t have some metainformation. Impossible to precisely detect pointers.

Leak Sanitizer: find all roots

We have a memory chunk. What next?
Scan chunk to detect pointers.
We don’t have some metainformation. Impossible to precisely detect pointers.
If something looks like a pointer and points to live memory – it is a pointer.

Leak Sanitizer: find all roots

We have a memory chunk. What next?
Scan chunk to detect pointers.
We don’t have some metainformation. Impossible to precisely detect pointers.
If something looks like a pointer and points to live memory – it is a pointer.
That’s why it’s called “conservative GC”.

Leak Sanitizer: find all roots

Looks like a pointer means:

Leak Sanitizer: find all roots

Looks like a pointer means:

1. It is aligned as a pointer (but see `LSAN_OPTION=use_unaligned=1`)

Leak Sanitizer: find all roots

Looks like a pointer means:

1. It is aligned as a pointer (but see `LSAN_OPTION=use_unaligned=1`)
2. Value looks like a pointer value

Leak Sanitizer: find all roots

Looks like a pointer means:

1. It is aligned as a pointer (but see `LSAN_OPTION=use_unaligned=1`)
2. Value looks like a pointer value

It means:

1. Value is not too small (because we're using mmaped heap):

```
const uptr kMidAddress = 4*4096;
if (p < kMidAddress)
    return false;
```

Leak Sanitizer: find all roots

Looks like a pointer means:

1. It is aligned as a pointer (but see `LSAN_OPTION=use_unaligned=1`)
2. Value looks like a pointer value

It means:

1. Value is not too small (because we're using mmaped heap):

```
const uptr kMidAddress = 4*4096;
if (p < kMidAddress)
    return false;
```

2. Value matches the pattern for pointers for a given architecture (x86_64 in our case):

Leak Sanitizer: find all roots

It means:

1. Value is not too small (because we’re using mmaped heap):

```
const uptr kMidAddress = 4*4096;
if (p < kMidAddress)
    return false;
```

2. Value matches the pattern for pointers for a given architecture (x86_64 in our case):

Leak Sanitizer: find all roots

It means:

1. Value is not too small (because we're using mmaped heap):

```
const uptr kMidAddress = 4*4096;
if (p < kMidAddress)
    return false;
```

2. Value matches the pattern for pointers for a given architecture (x86_64 in our case):

```
// TODO: support LAM48 and 5 level page tables.
// LAM_U57 mask format
//  * top byte: 0x81 because the format is: [0] [6-bit tag] [0]
//  * top-1 byte: 0xff because it should be 0
//  * top-2 byte: 0x80 because Linux uses 128 TB VMA ending at 0x7fffffffffff
constexpr uptr kLAM_U57Mask = 0x81ff80;
constexpr uptr kPointerMask = kLAM_U57Mask << 40;
return ((p & kPointerMask) == 0);
```

```
kPointerMask = 10000001 11111111 10000000 00000000 00000000 00000000 00000000 00000000
```

Leak Sanitizer: find all roots

If value looks like a pointer than let’s check if it points to a live memory

Leak Sanitizer: find all roots

If value looks like a pointer than let’s check if it points to a live memory
(we can do it because we’ve replaced memory manager by own implementation)

Leak Sanitizer: find all roots

If value looks like a pointer than let's check if it points to a live memory
(we can do it because we've replaced memory manager by own implementation)

```
uptr chunk = reinterpret_cast<uptr>(allocator.GetBlockBeginFastLocked(p));
```

Leak Sanitizer: find all roots

If value looks like a pointer than let's check if it points to a live memory
(we can do it because we've replaced memory manager by own implementation)

```
uptr chunk = reinterpret_cast<uptr>(allocator.GetBlockBeginFastLocked(p));
if (!chunk) return 0;
```

Leak Sanitizer: find all roots

If value looks like a pointer than let’s check if it points to a live memory
(we can do it because we’ve replaced memory manager by own implementation)

```
uptr chunk = reinterpret_cast<uptr>(allocator.GetBlockBeginFastLocked(p));
if (!chunk) return 0;
ChunkMetadata *m = Metadata(reinterpret_cast<void *>(chunk));
```

Leak Sanitizer: find all roots

If value looks like a pointer than let’s check if it points to a live memory
(we can do it because we’ve replaced memory manager by own implementation)

```
uptr chunk = reinterpret_cast<uptr>(allocator.GetBlockBeginFastLocked(p));
if (!chunk) return 0;
ChunkMetadata *m = Metadata(reinterpret_cast<void *>(chunk));
if (!m->allocated)
          return 0;
```

Leak Sanitizer: find all roots

If value looks like a pointer than let’s check if it points to a live memory
(we can do it because we’ve replaced memory manager by own implementation)

```
uptr chunk = reinterpret_cast<uptr>(allocator.GetBlockBeginFastLocked(p));
if (!chunk) return 0;
ChunkMetadata *m = Metadata(reinterpret_cast<void *>(chunk));
if (!m->allocated)
          return 0;
if (p < chunk + m->requested_size)
          return chunk;
```

Leak Sanitizer: mark all reachable chunks

General algorithm:

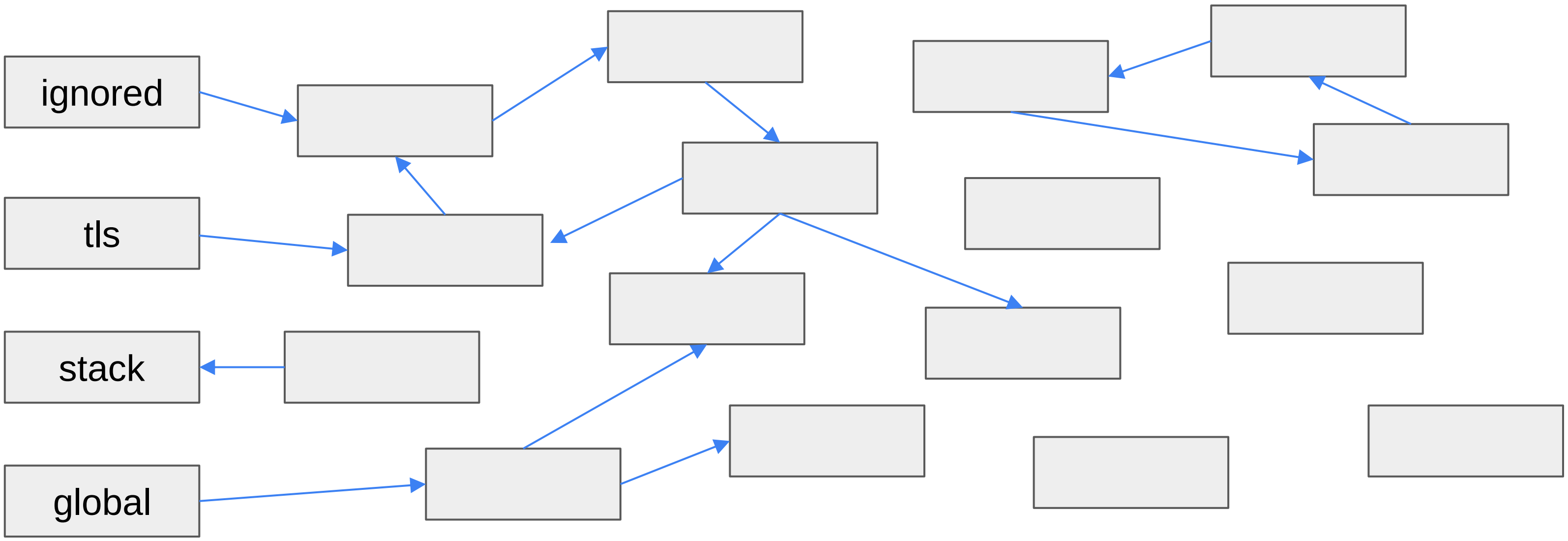
1. Stop the world
2. Find all roots
3. **Mark all reachable chunks**
4. All not marked chunks are leaks
5. Mark indirect leaks
6. Report all leaks

Leak Sanitizer: mark all reachable chunks

Will use Depth-First Search (aka DFS) algorithm:

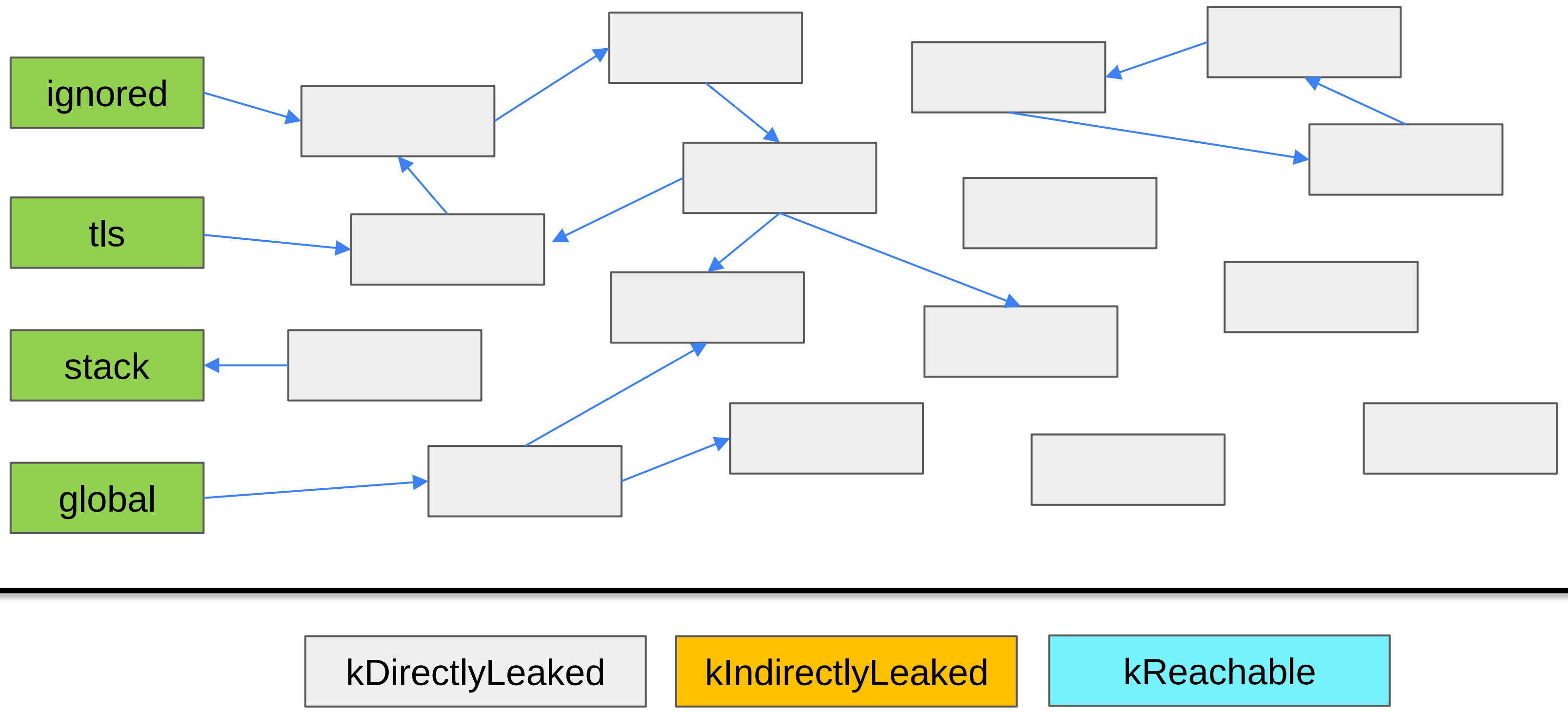
Leak Sanitizer: mark all reachable chunks

Will use Depth-First Search (aka DFS) algorithm:



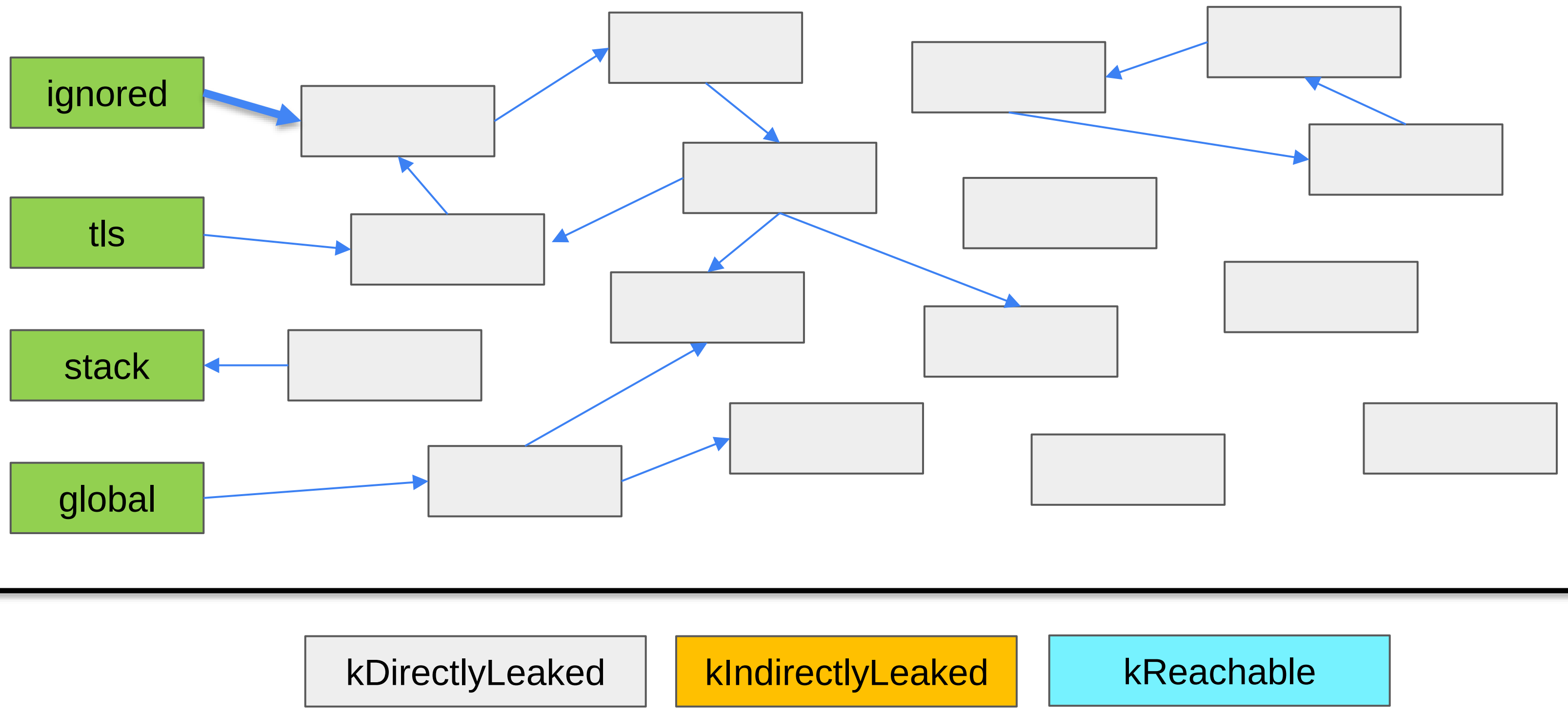
Leak Sanitizer: mark all reachable chunks

Will use Depth-First Search (aka DFS) algorithm:



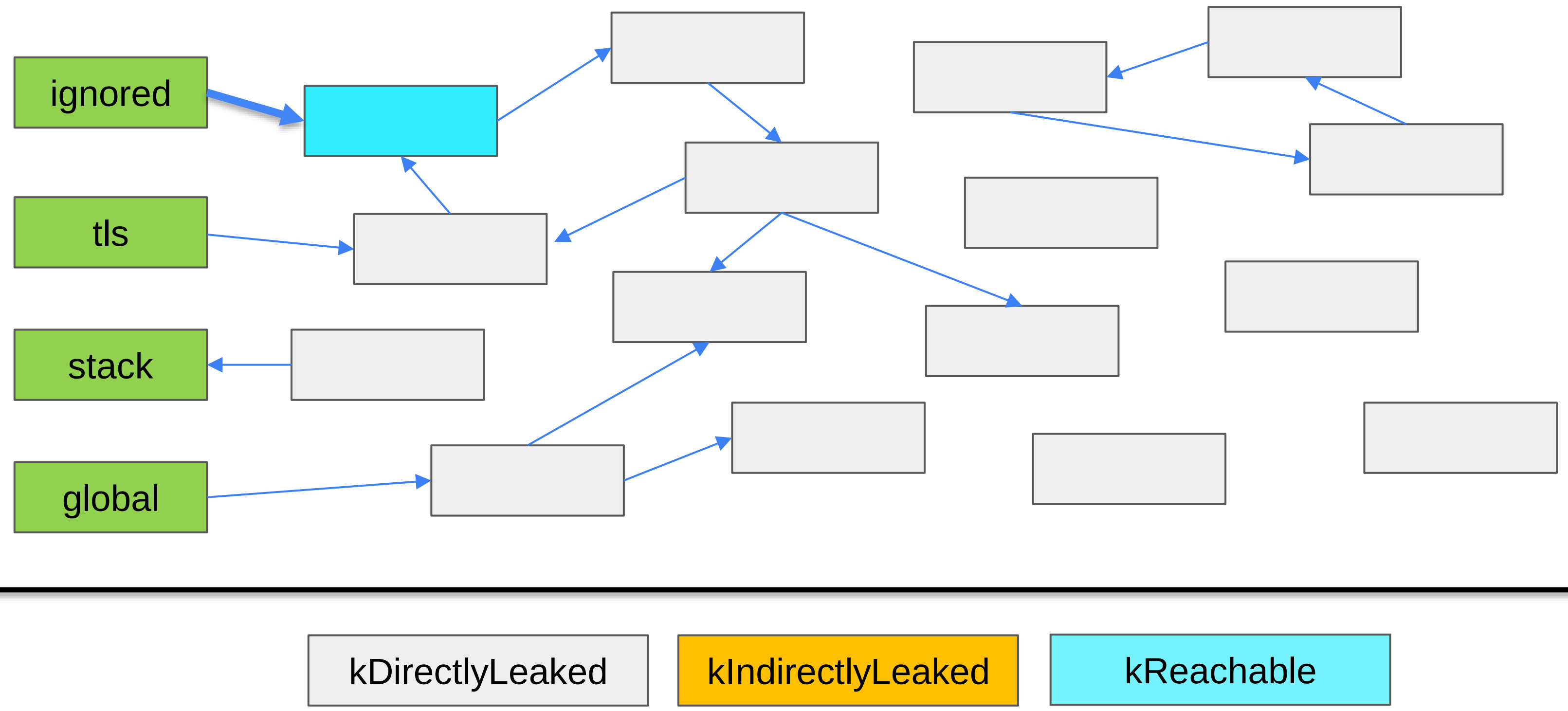
Leak Sanitizer: mark all reachable chunks

Will use Depth-First Search (aka DFS) algorithm:



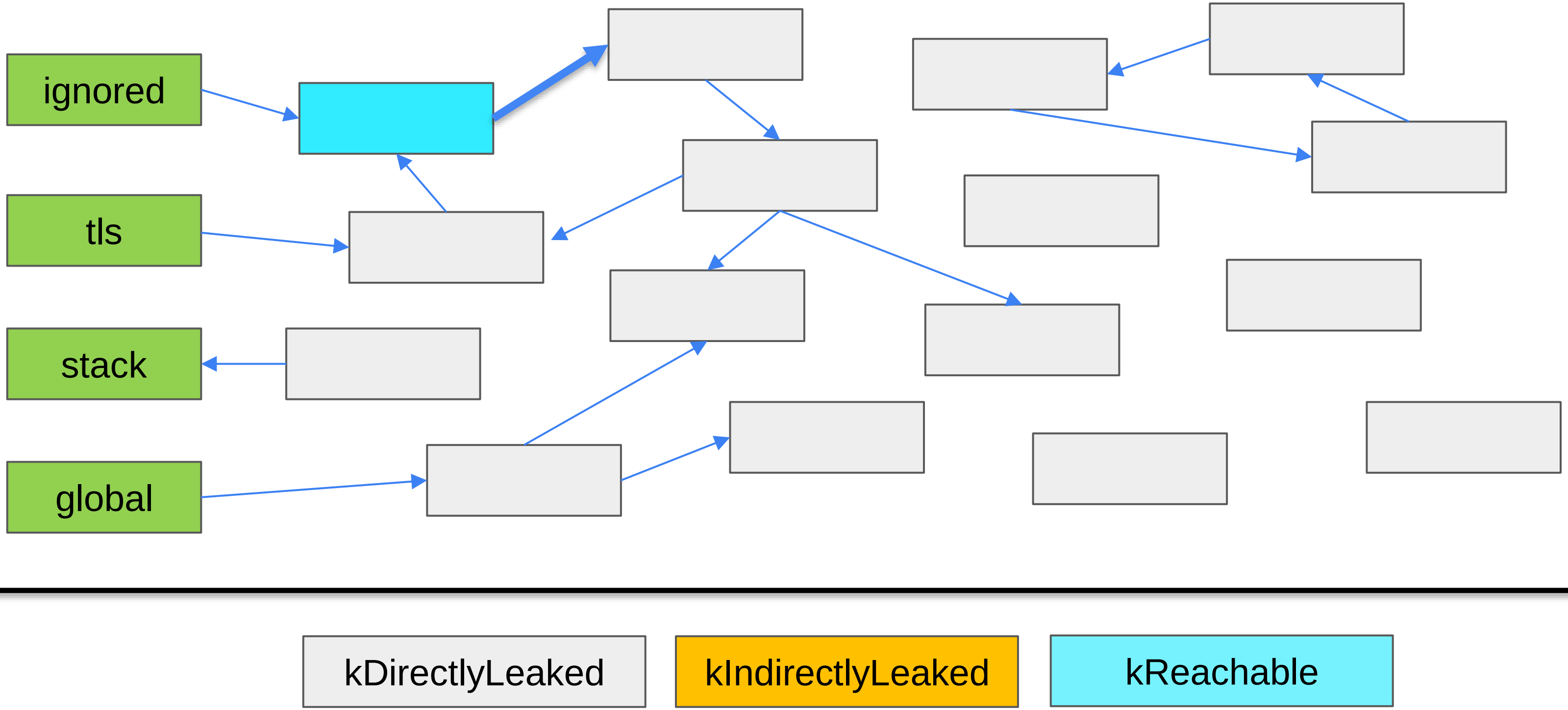
Leak Sanitizer: mark all reachable chunks

Will use Depth-First Search (aka DFS) algorithm:



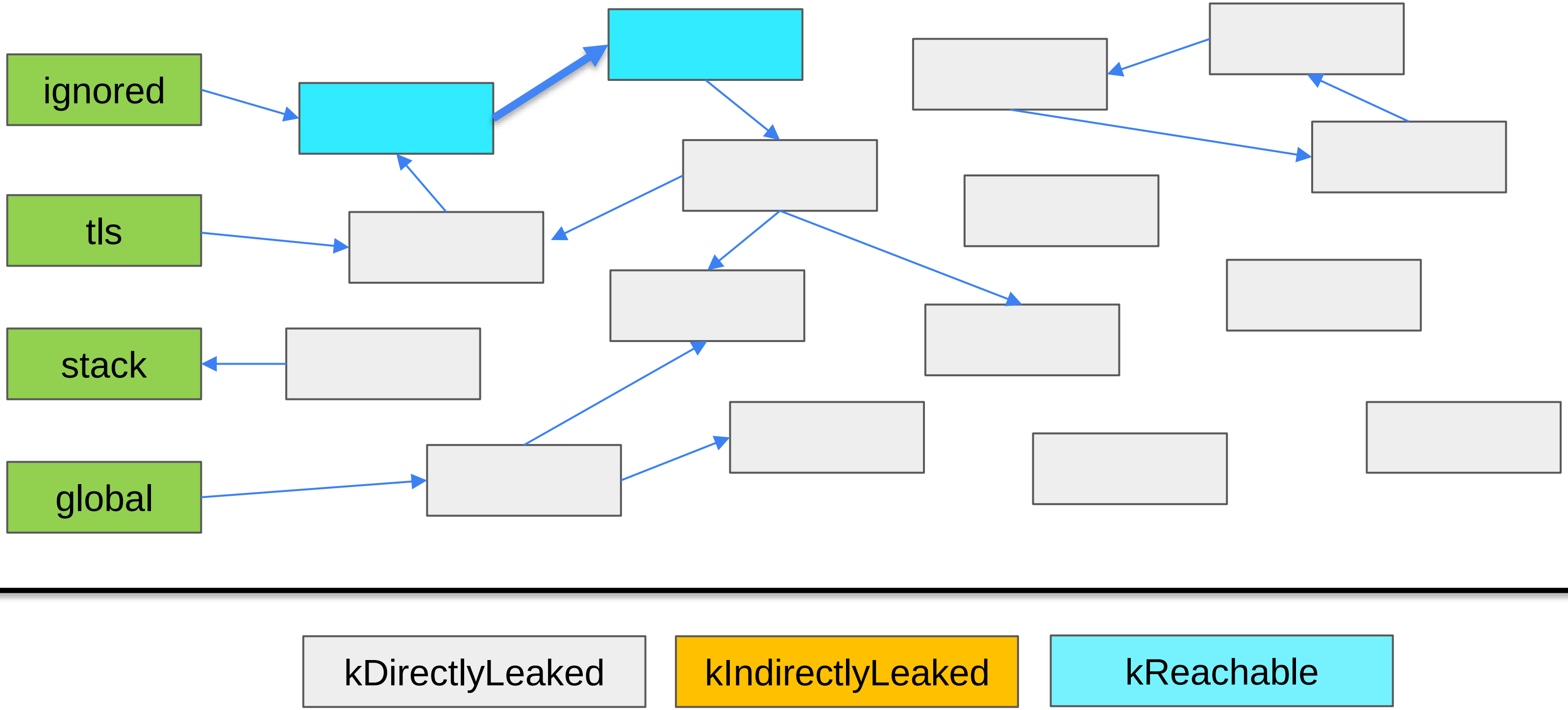
Leak Sanitizer: mark all reachable chunks

Will use Depth-First Search (aka DFS) algorithm:



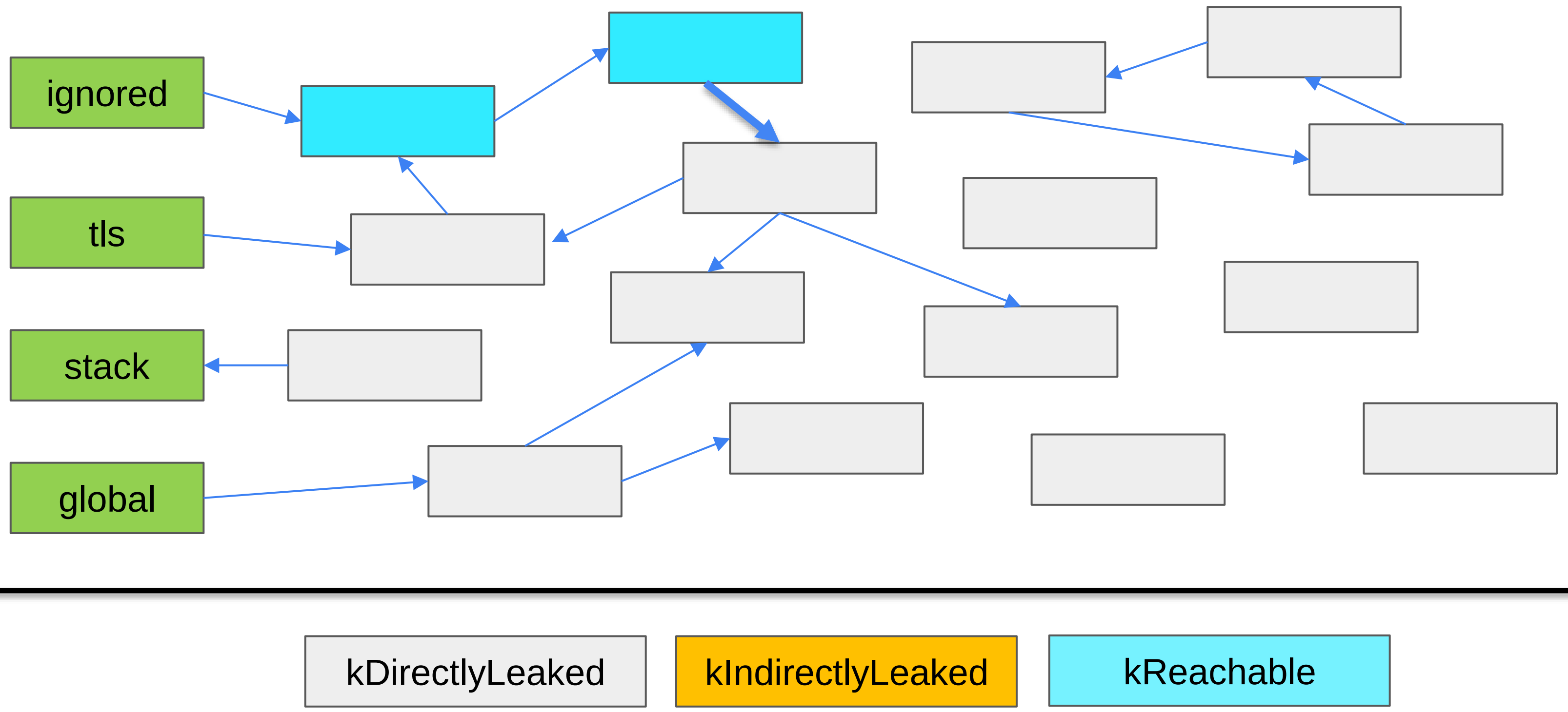
Leak Sanitizer: mark all reachable chunks

Will use Depth-First Search (aka DFS) algorithm:



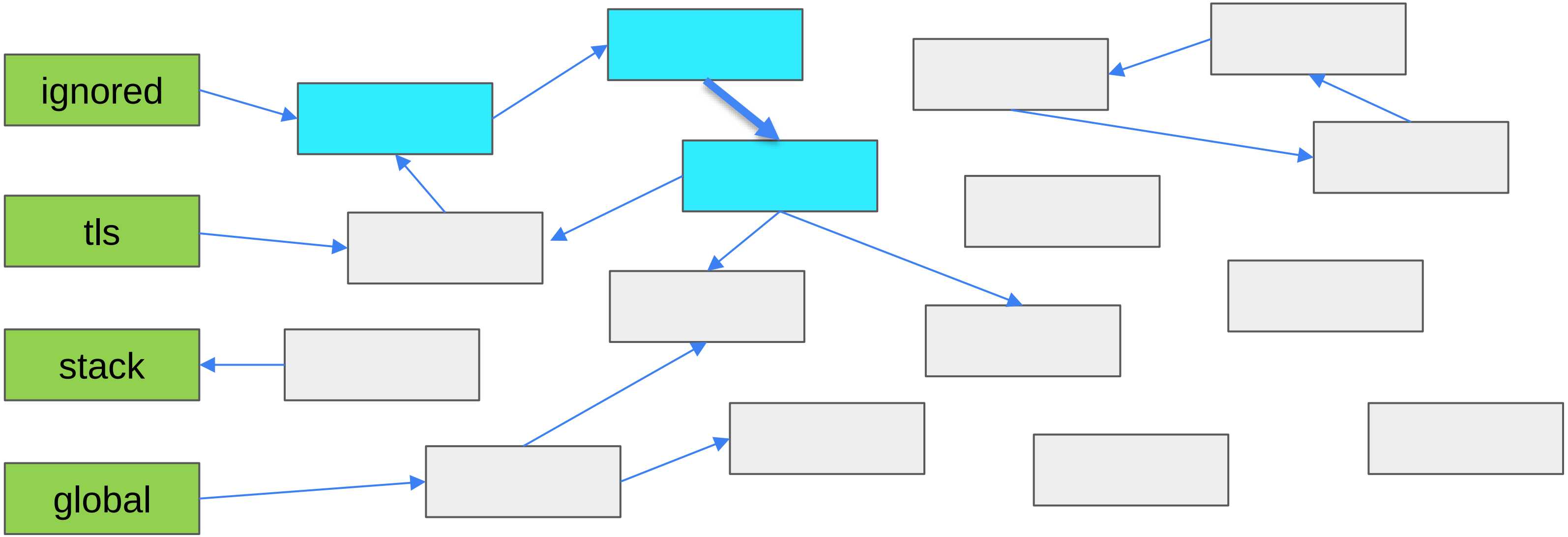
Leak Sanitizer: mark all reachable chunks

Will use Depth-First Search (aka DFS) algorithm:



Leak Sanitizer: mark all reachable chunks

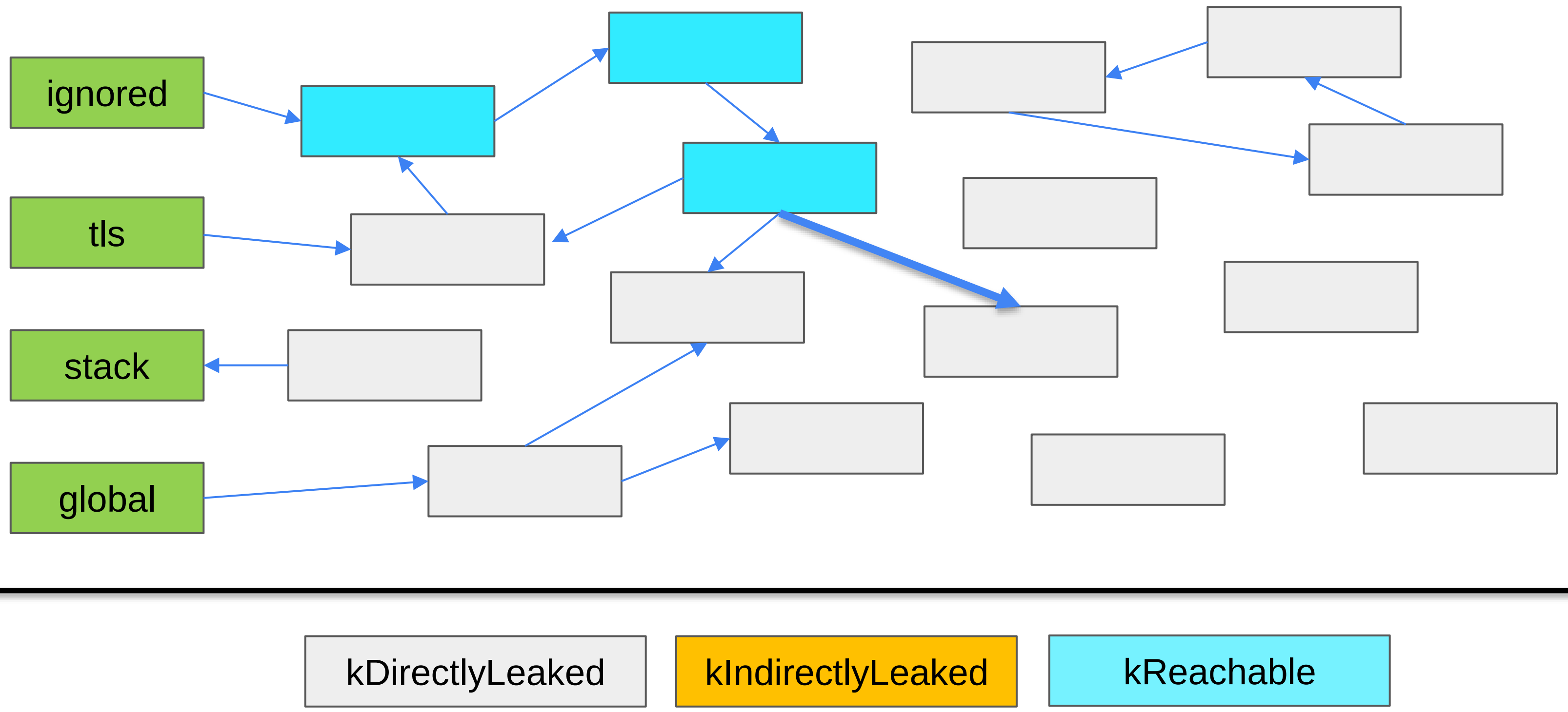
Will use Depth-First Search (aka DFS) algorithm:



kDirectlyLeaked | kIndirectlyLeaked | kReachable

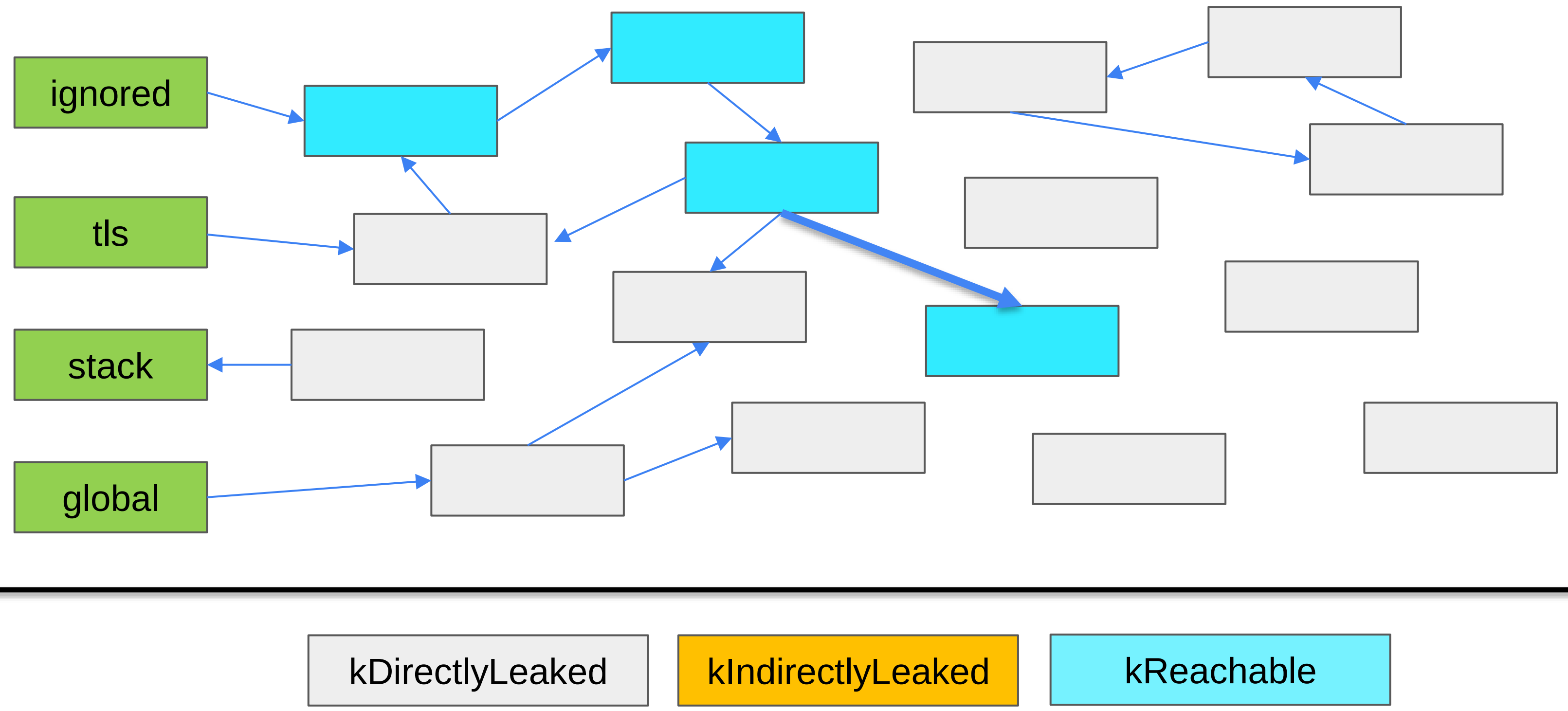
Leak Sanitizer: mark all reachable chunks

Will use Depth-First Search (aka DFS) algorithm:



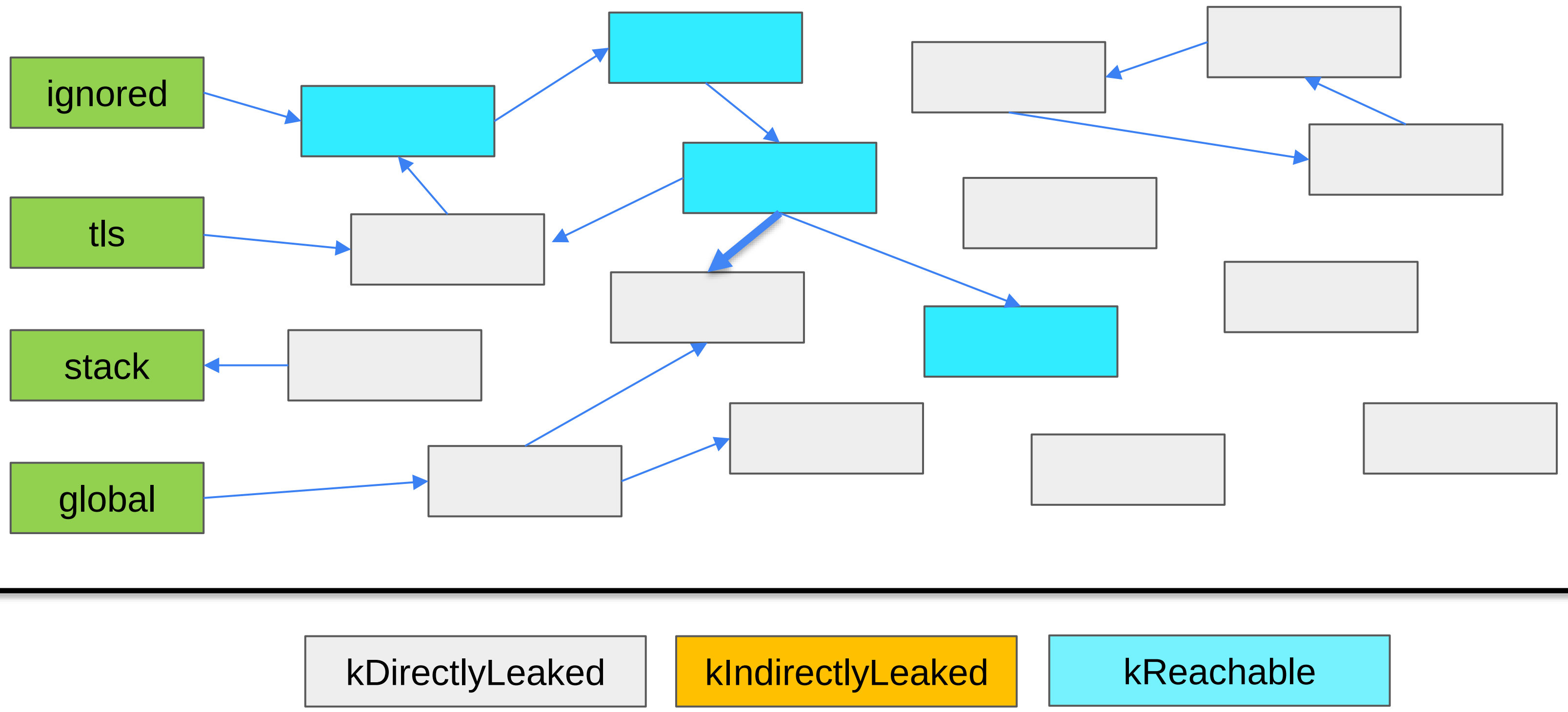
Leak Sanitizer: mark all reachable chunks

Will use Depth-First Search (aka DFS) algorithm:



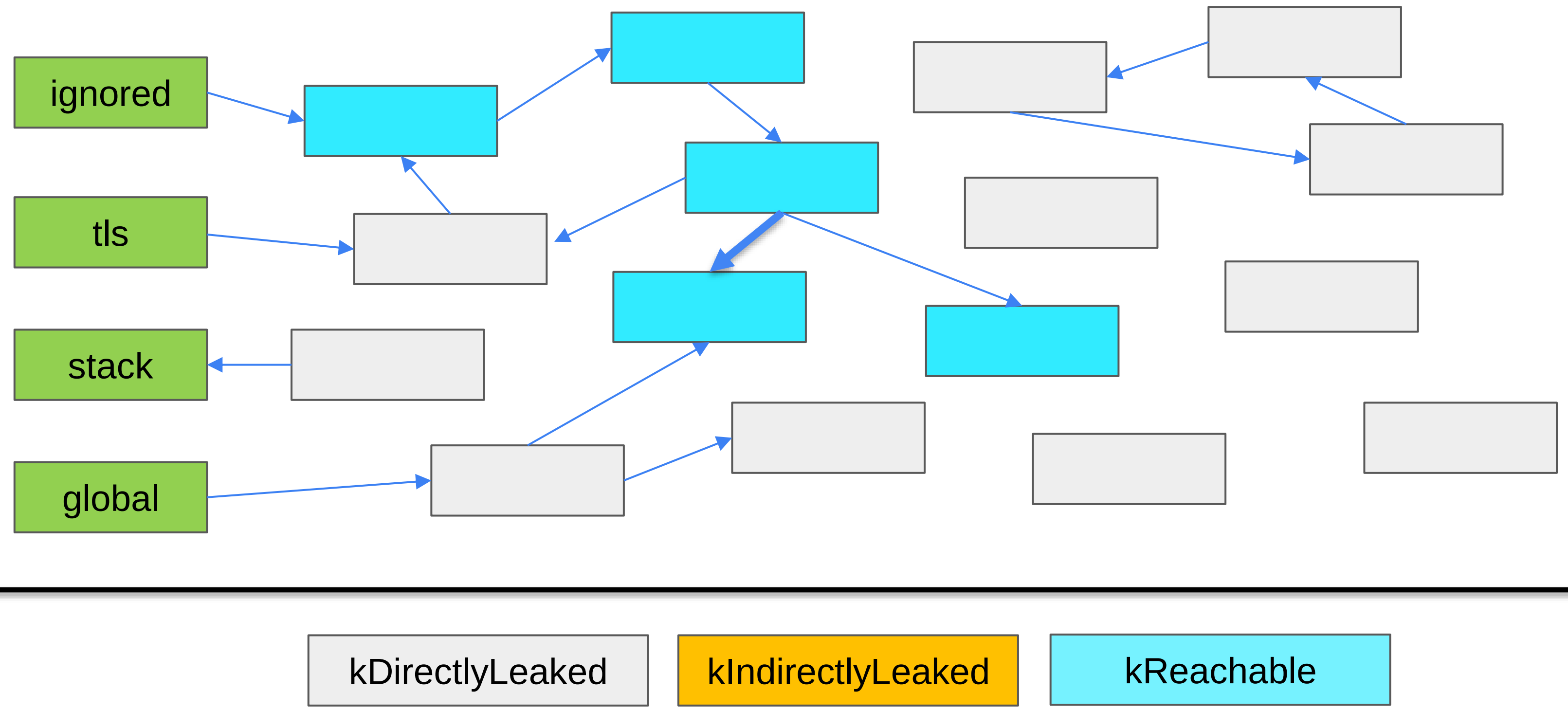
Leak Sanitizer: mark all reachable chunks

Will use Depth-First Search (aka DFS) algorithm:



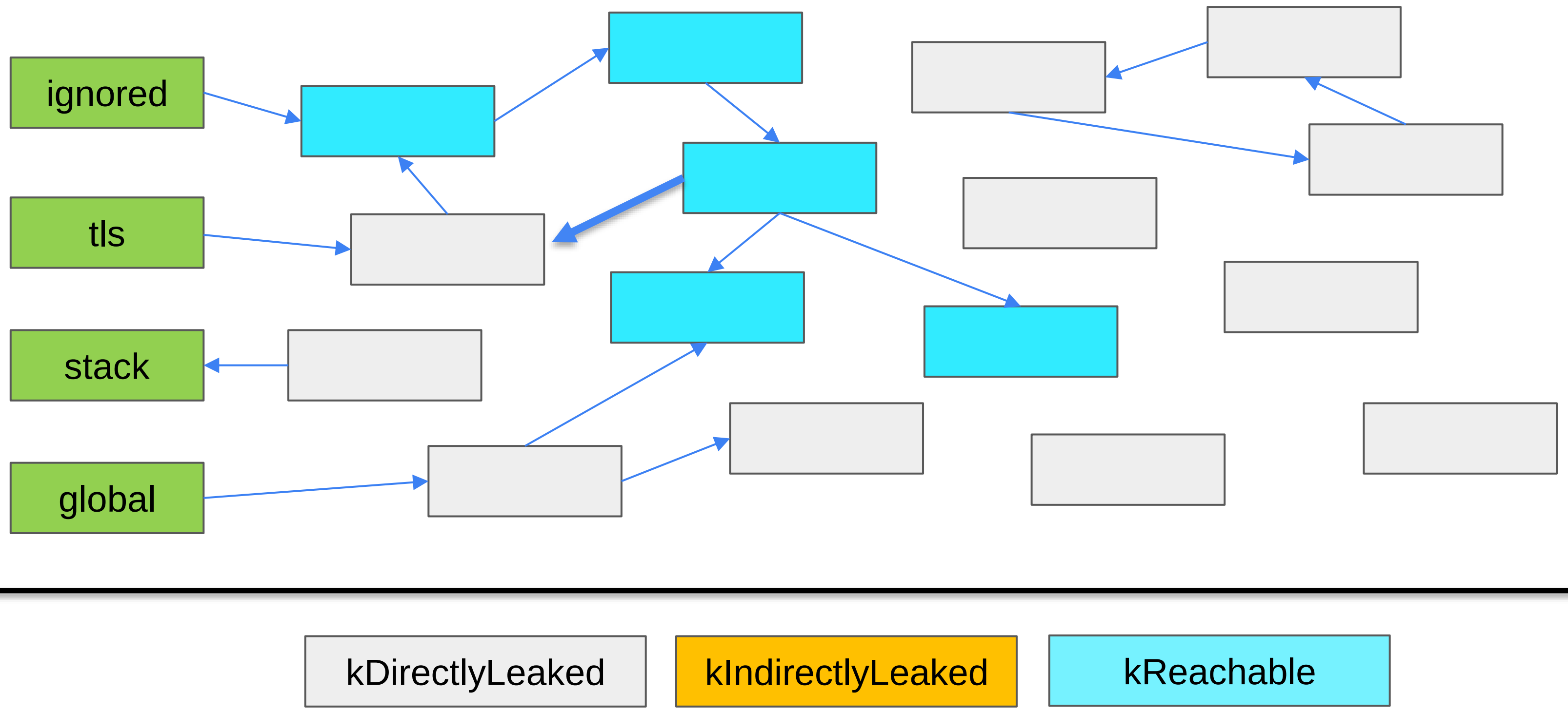
Leak Sanitizer: mark all reachable chunks

Will use Depth-First Search (aka DFS) algorithm:



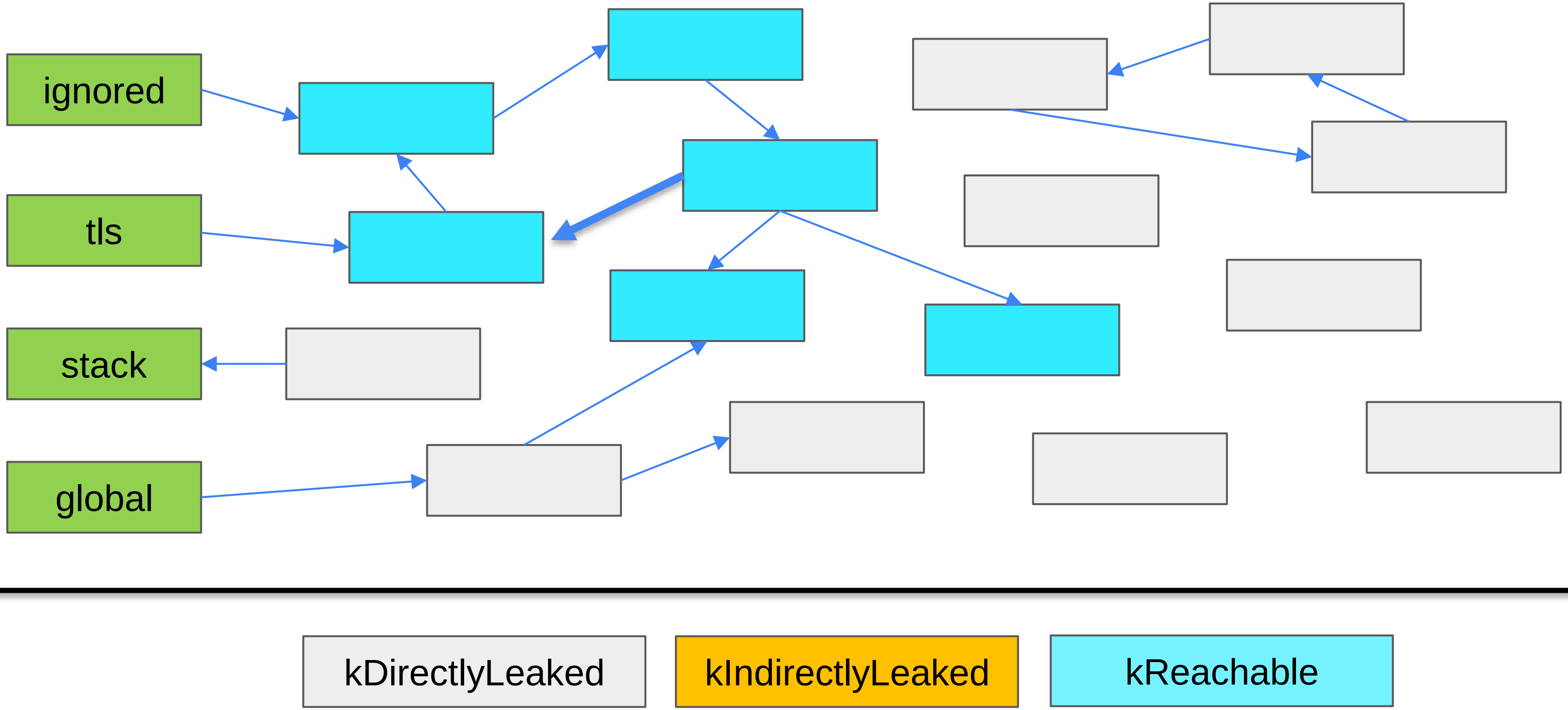
Leak Sanitizer: mark all reachable chunks

Will use Depth-First Search (aka DFS) algorithm:



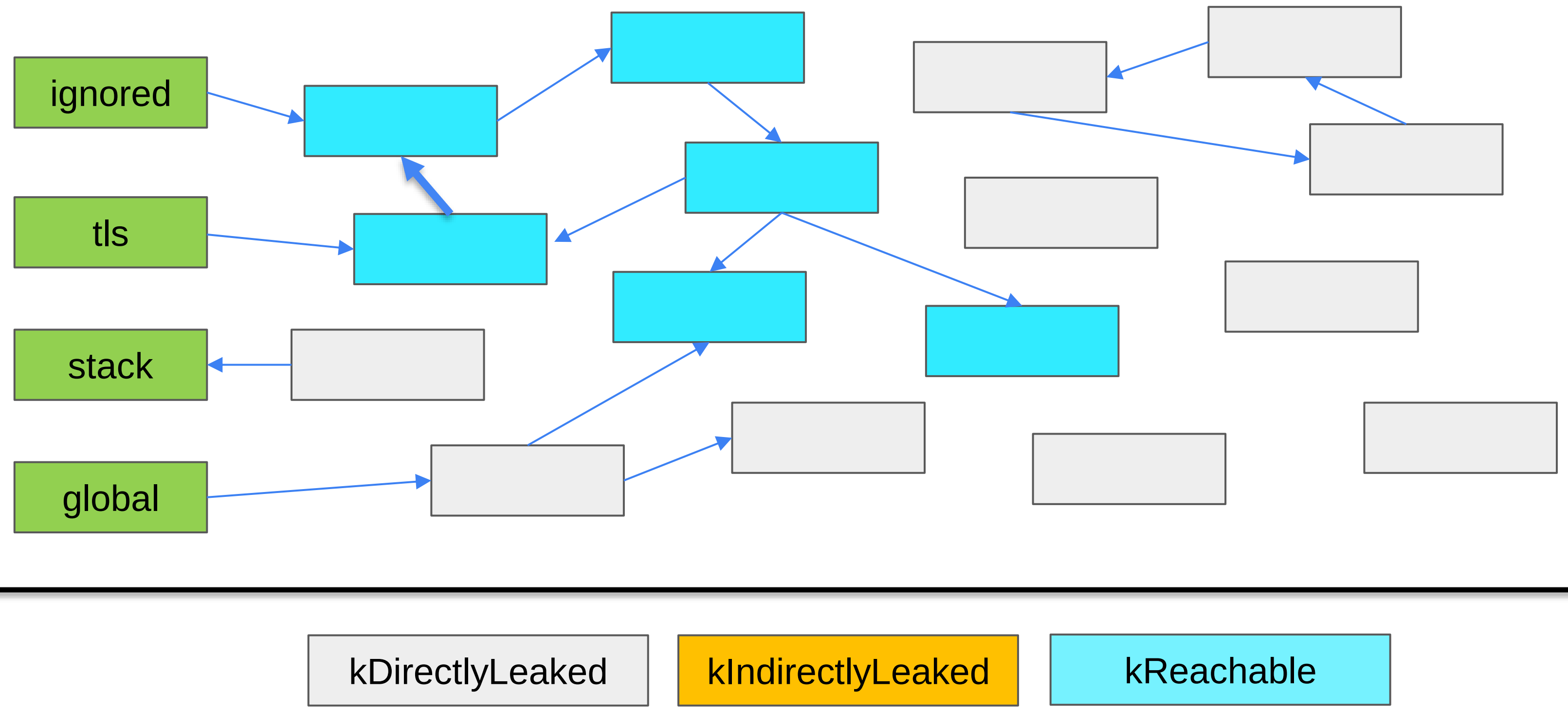
Leak Sanitizer: mark all reachable chunks

Will use Depth-First Search (aka DFS) algorithm:



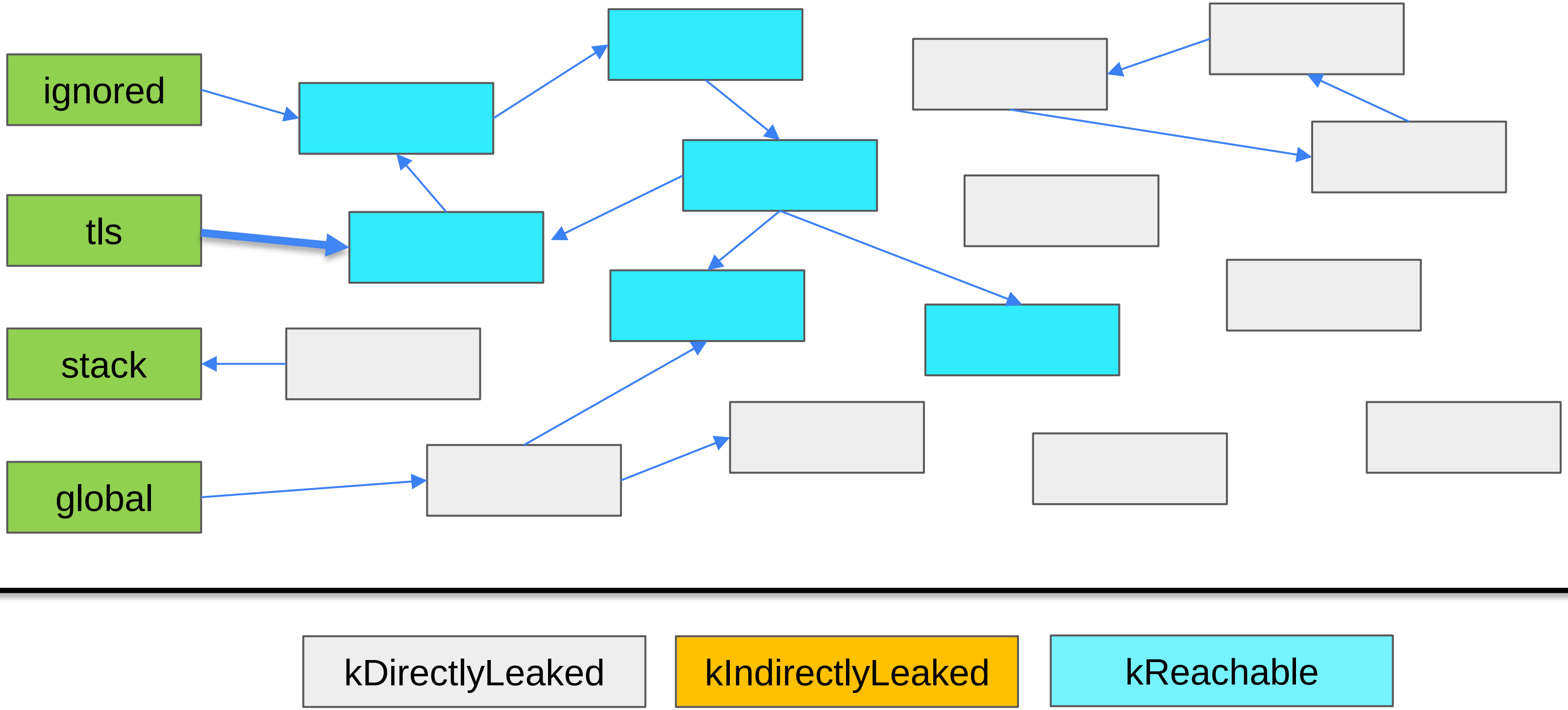
Leak Sanitizer: mark all reachable chunks

Will use Depth-First Search (aka DFS) algorithm:



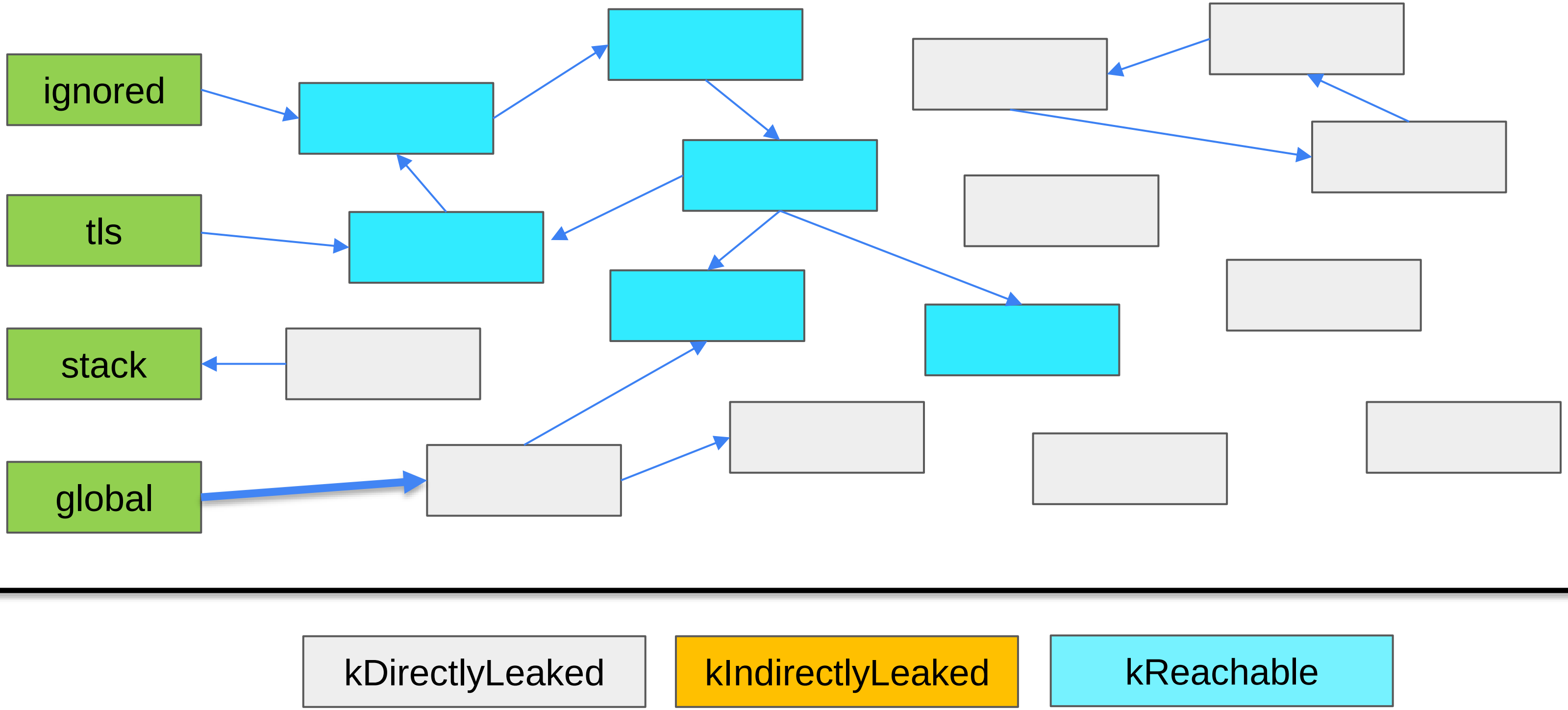
Leak Sanitizer: mark all reachable chunks

Will use Depth-First Search (aka DFS) algorithm:



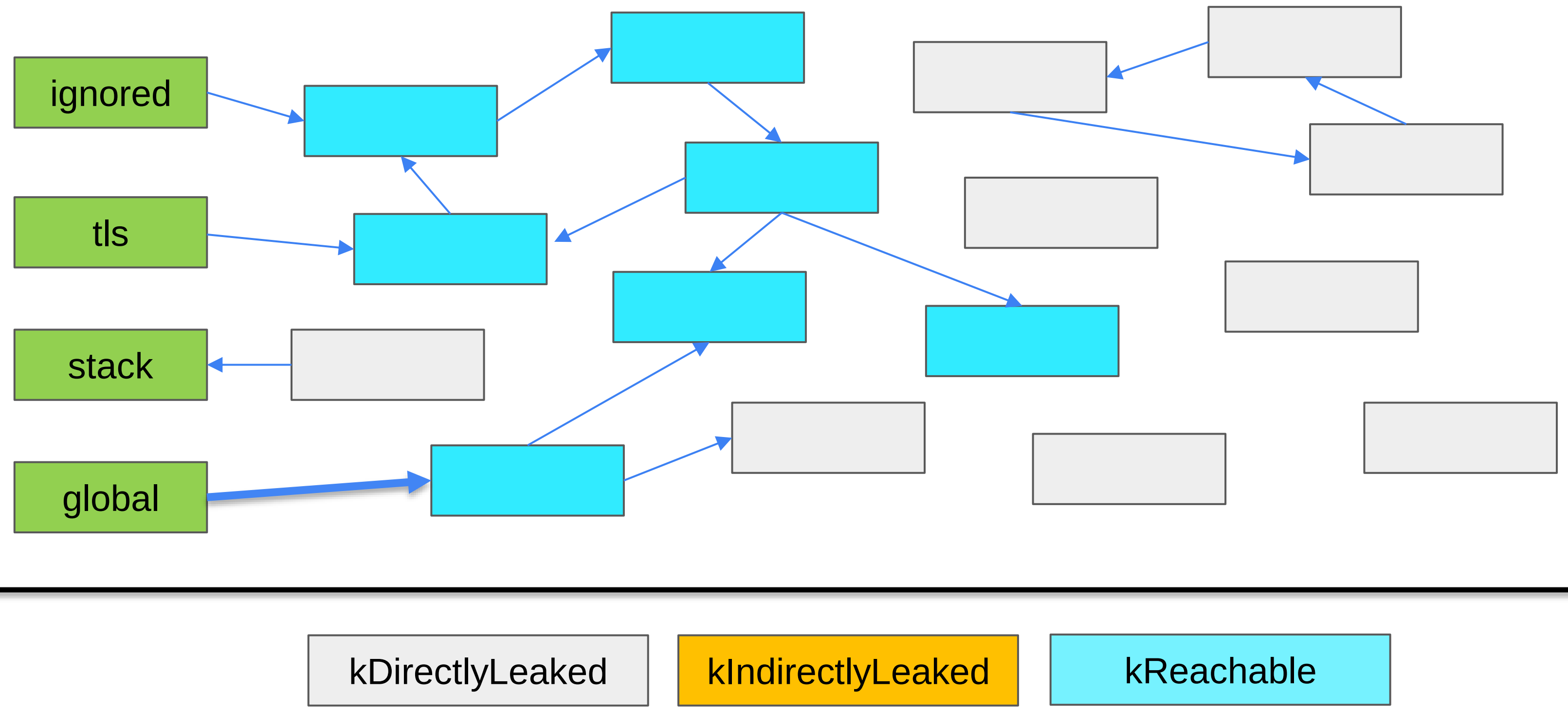
Leak Sanitizer: mark all reachable chunks

Will use Depth-First Search (aka DFS) algorithm:



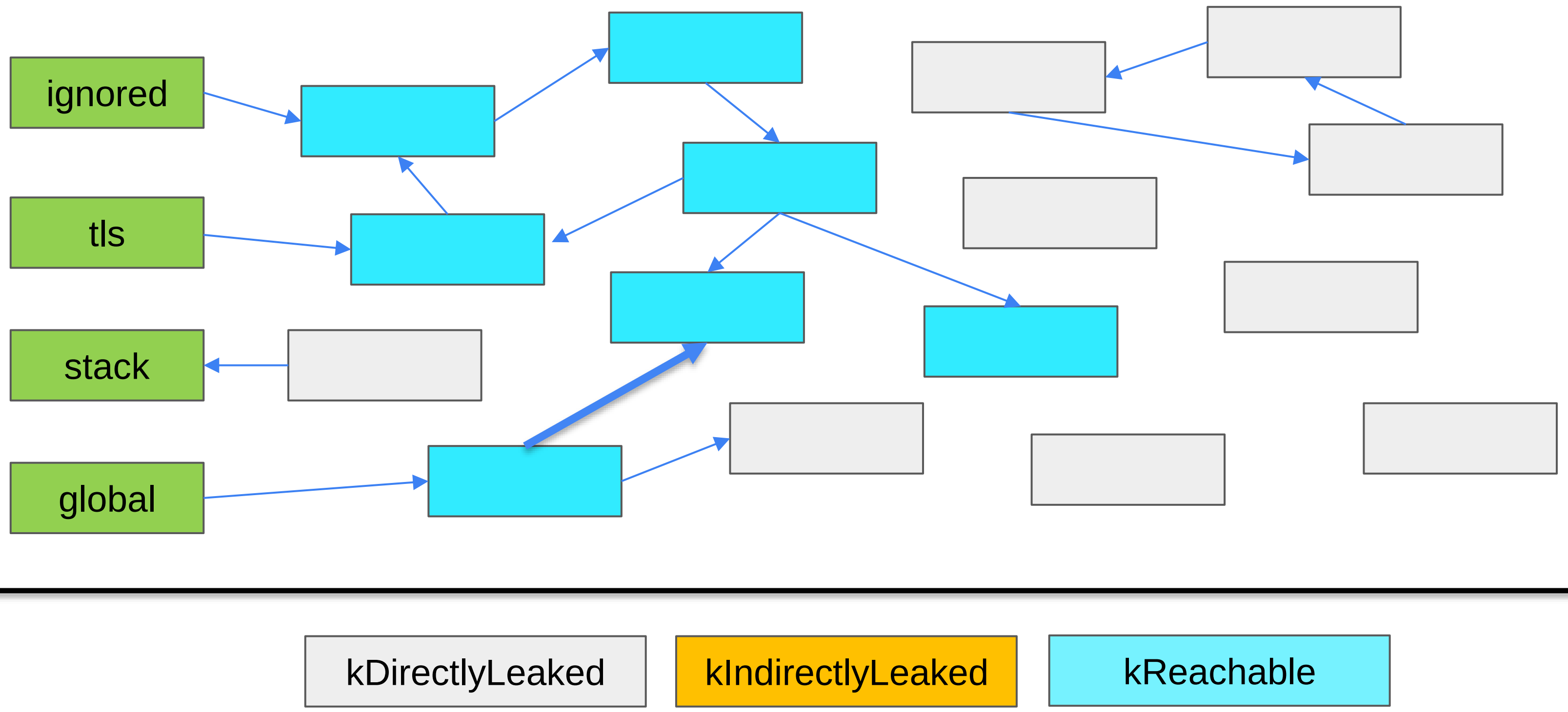
Leak Sanitizer: mark all reachable chunks

Will use Depth-First Search (aka DFS) algorithm:



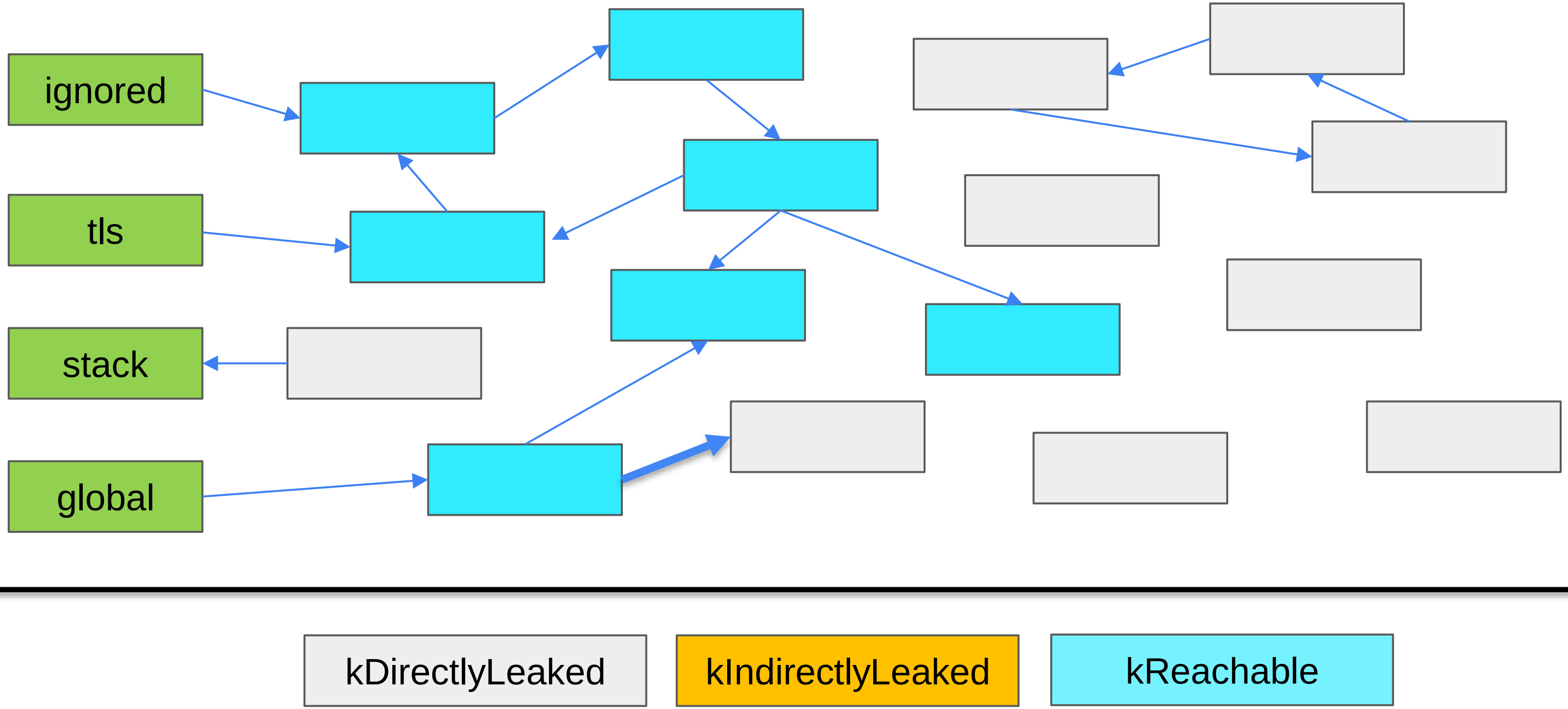
Leak Sanitizer: mark all reachable chunks

Will use Depth-First Search (aka DFS) algorithm:



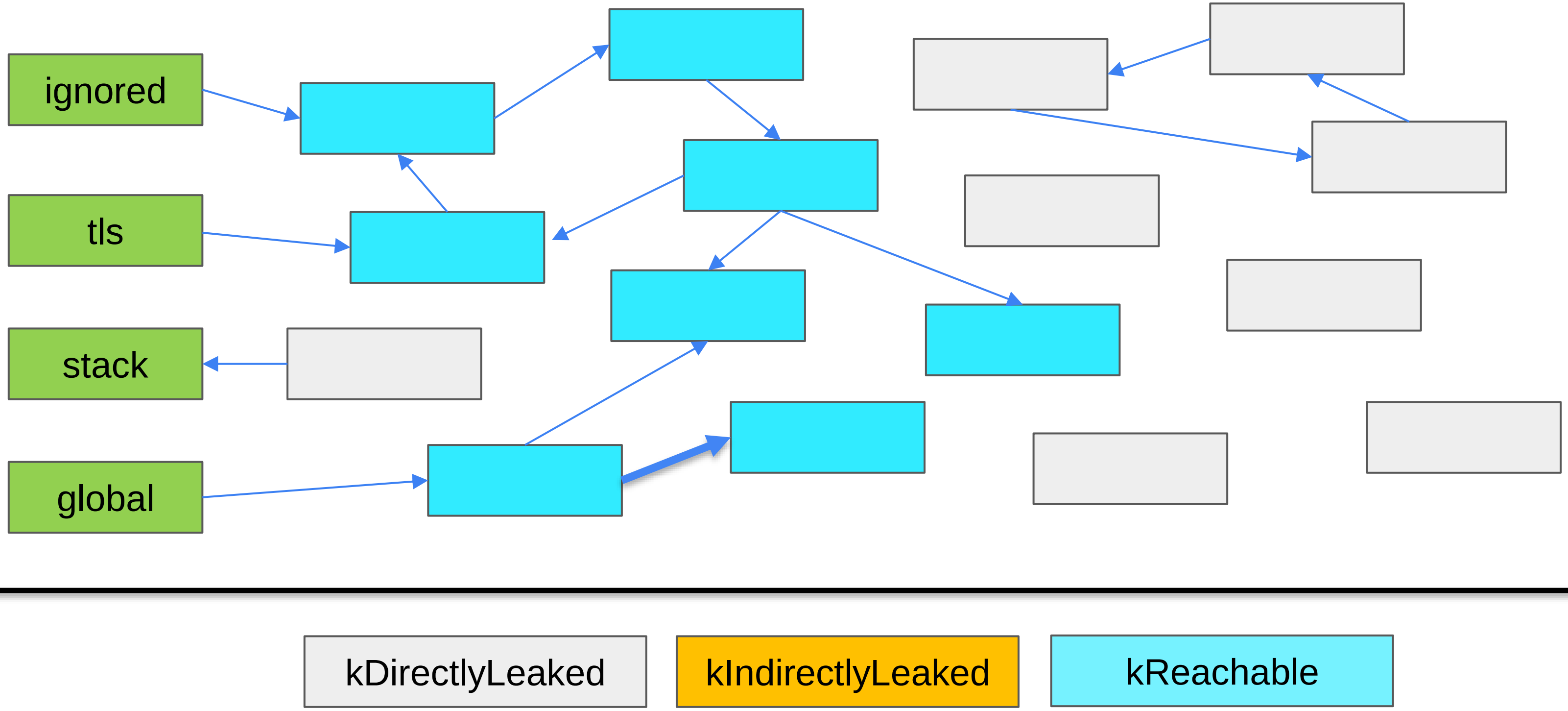
Leak Sanitizer: mark all reachable chunks

Will use Depth-First Search (aka DFS) algorithm:



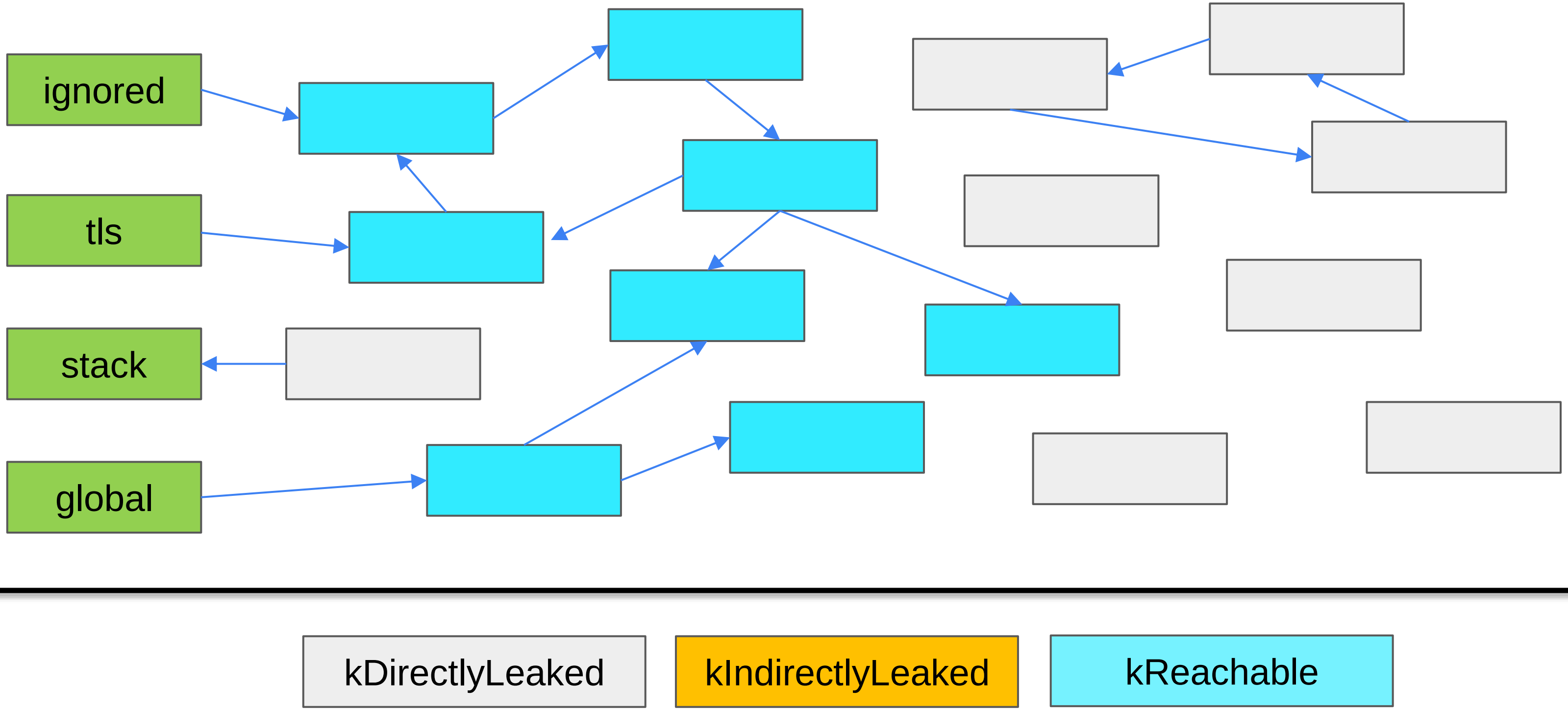
Leak Sanitizer: mark all reachable chunks

Will use Depth-First Search (aka DFS) algorithm:



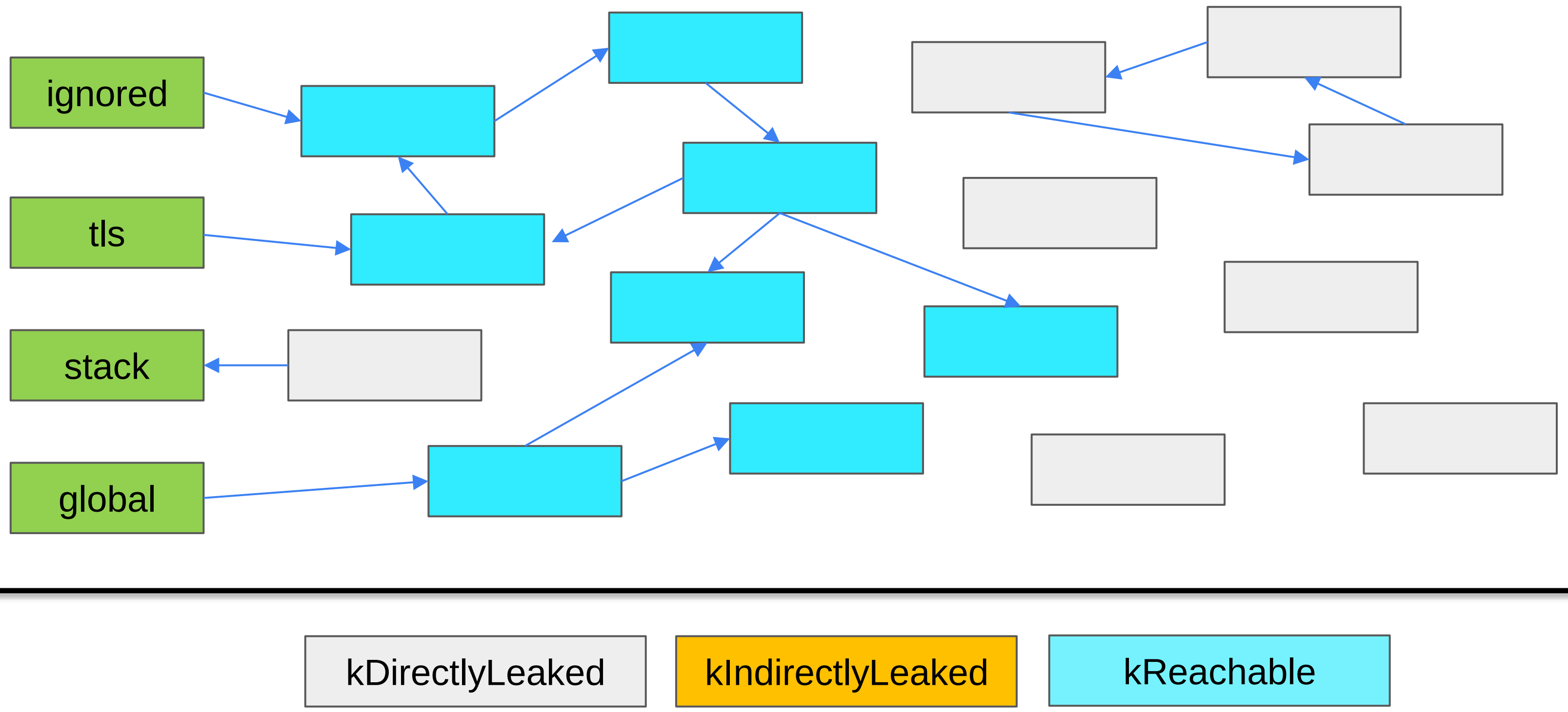
Leak Sanitizer: mark all reachable chunks

Will use Depth-First Search (aka DFS) algorithm:



Leak Sanitizer: mark all reachable chunks

So, we’ve marked all reachable chunks as kReachable (in metainfo tag)



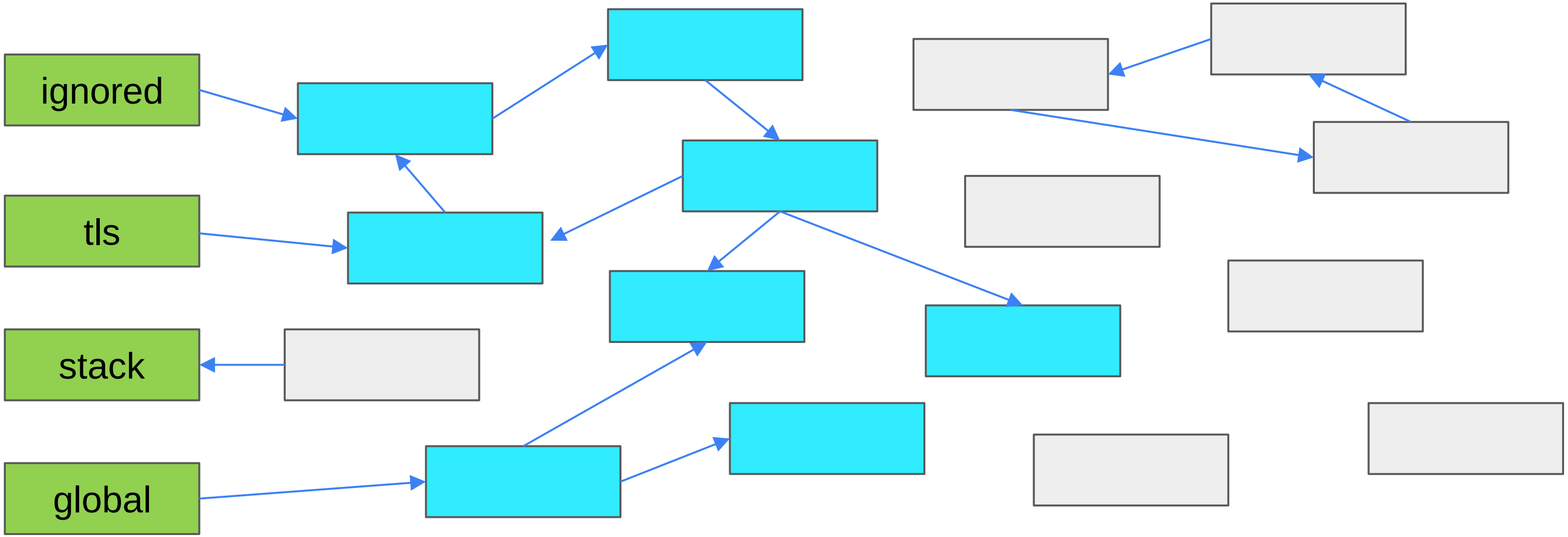
Leak Sanitizer: all not marked chunks are leaks

General algorithm:

1. Stop the world
2. Find all roots
3. Mark all reachable chunks
4. **All not marked chunks are leaks**
5. Mark indirect leaks
6. Report all leaks

Leak Sanitizer: all not marked chunks are leaks

kDirectlyLeaked metainfo tag is a default tag



Leak Sanitizer: mark indirect leaks

General algorithm:

1. Stop the world
2. Find all roots
3. Mark all reachable chunks
4. All not marked chunks are leaks
5. **Mark indirect leaks**
6. Report all leaks

Leak Sanitizer: mark indirect leaks

Indirect leak is a chunk referenced by another leaked chunk

kDirectlyLeaked

kIndirectlyLeaked

kReachable

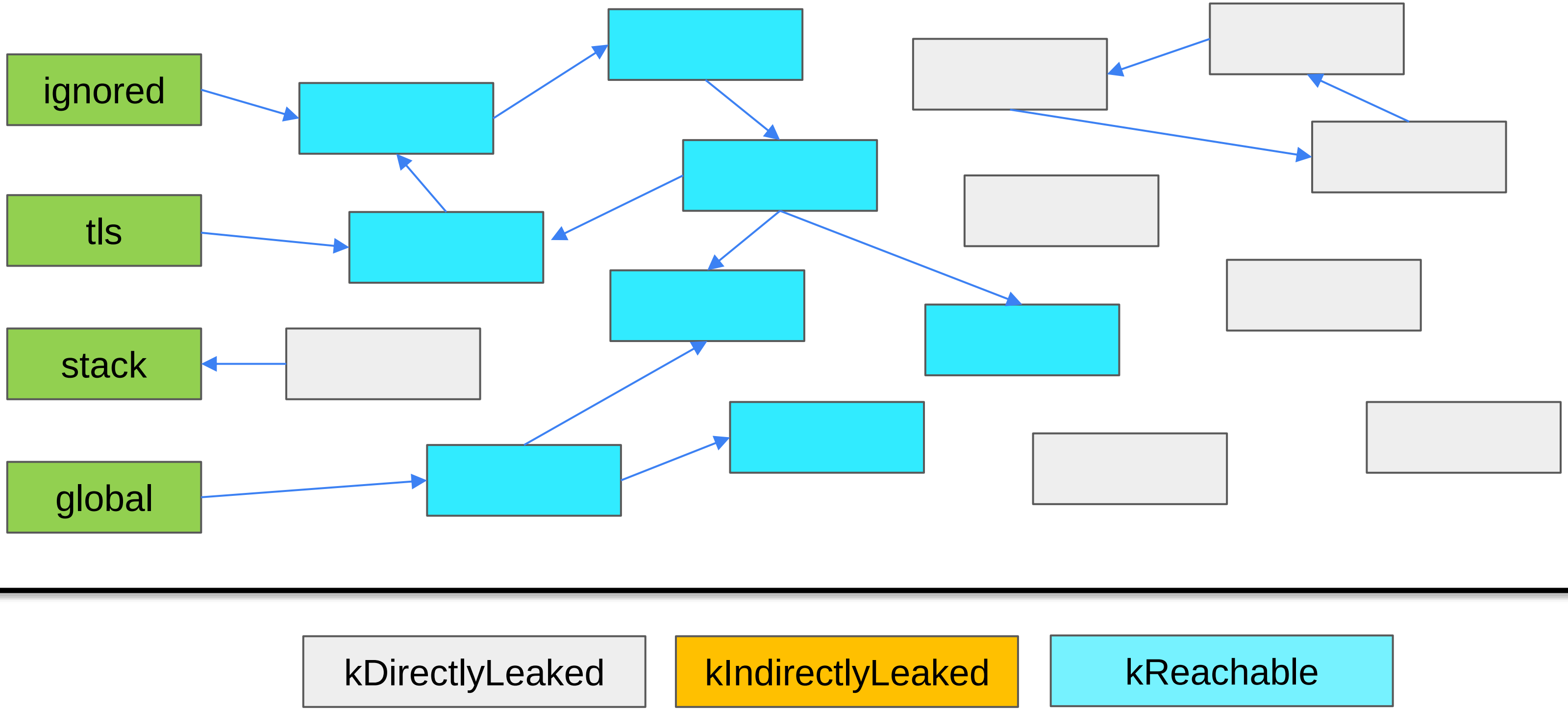
Leak Sanitizer: mark indirect leaks

Indirect leak is a chunk referenced by another leaked chunk. Example:

kDirectlyLeaked | kIndirectlyLeaked | kReachable

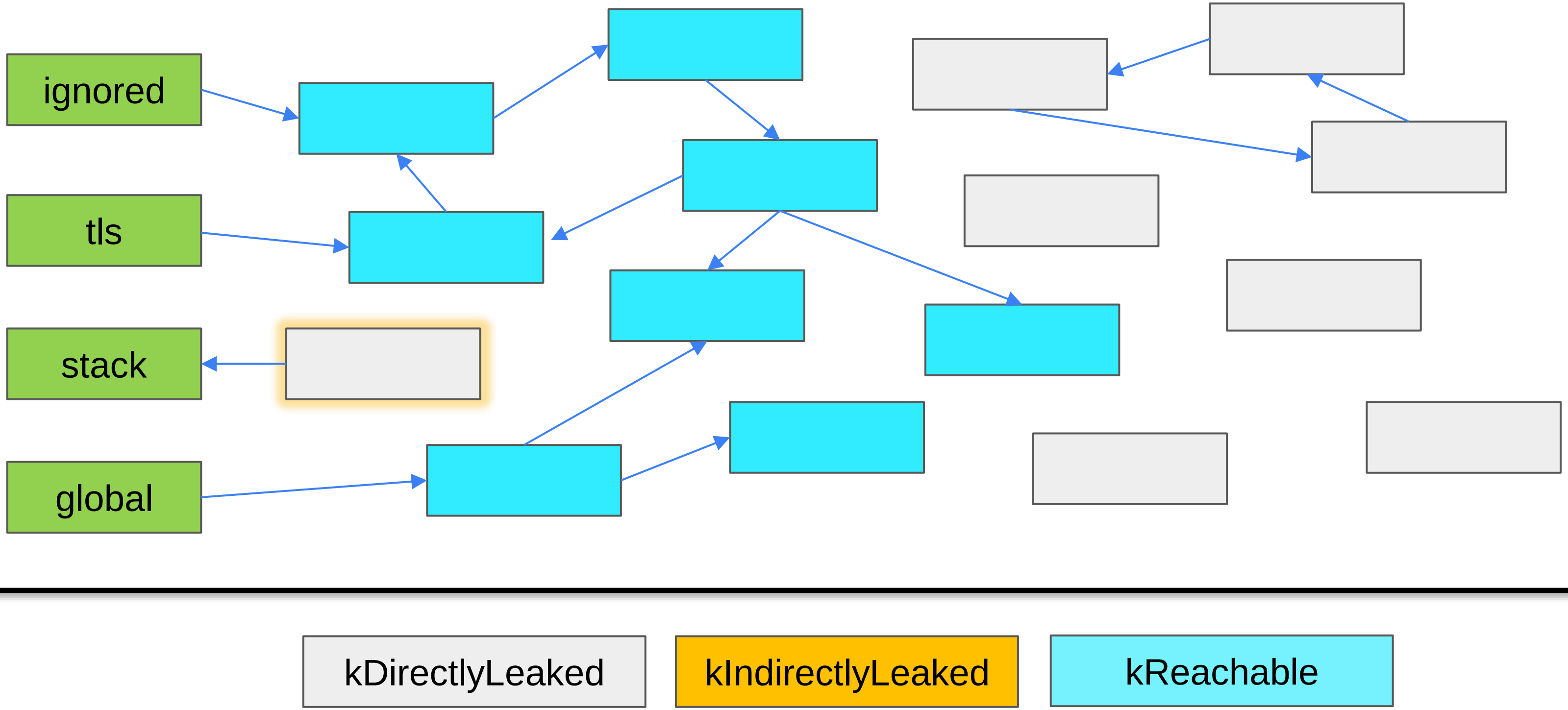
Leak Sanitizer: mark indirect leaks

Will check **each** leaked chunk for pointers to other leaked chunks



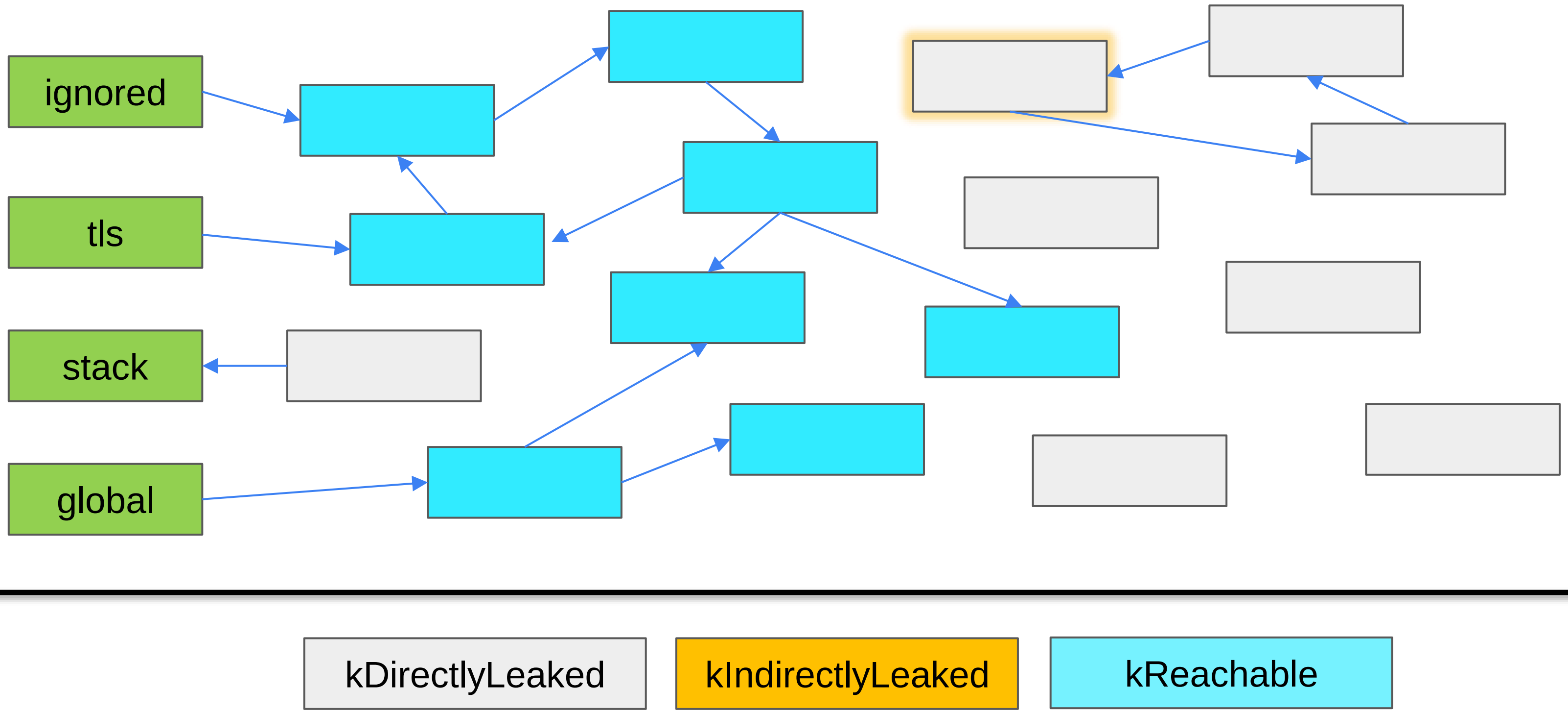
Leak Sanitizer: mark indirect leaks

Will check each leaked chunk for pointers to other leaked chunks



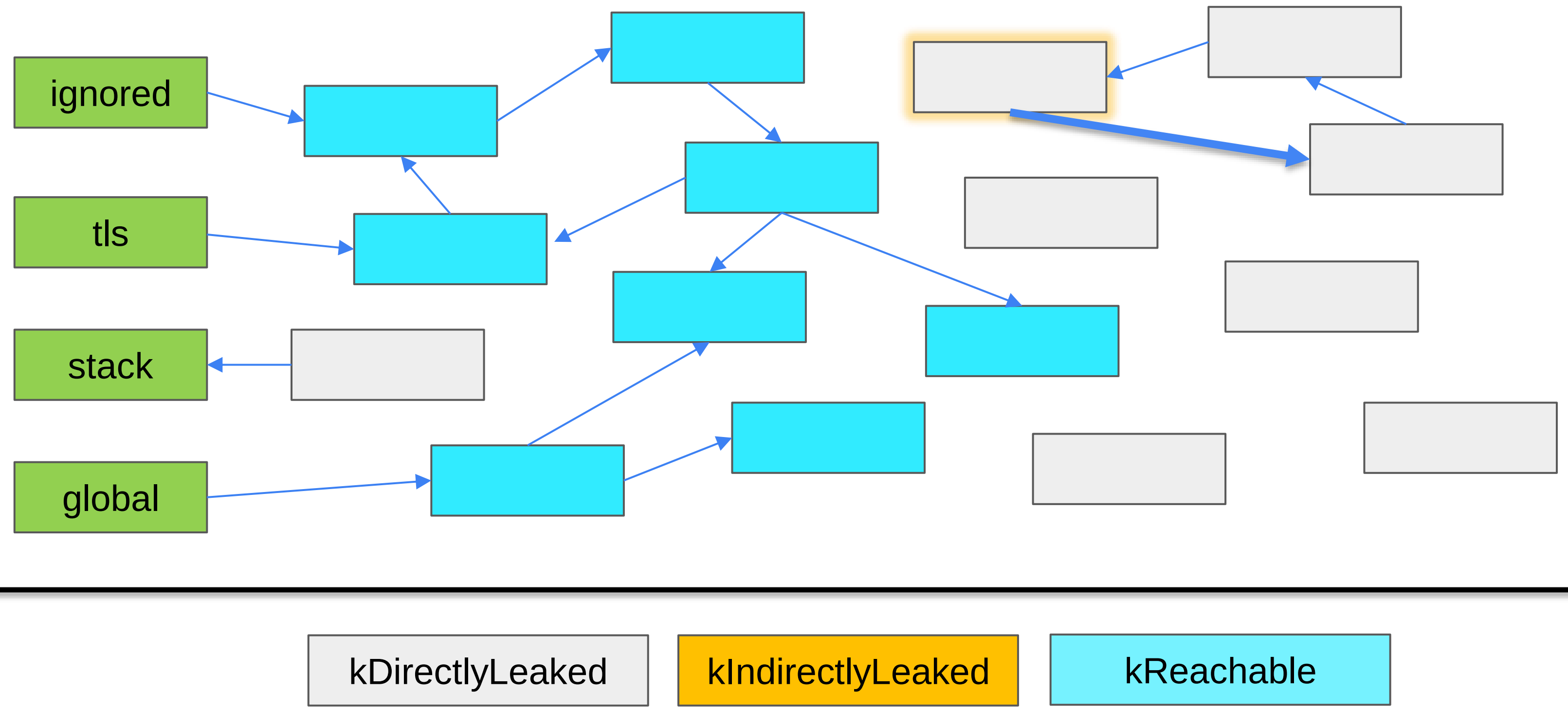
Leak Sanitizer: mark indirect leaks

Will check each leaked chunk for pointers to other leaked chunks



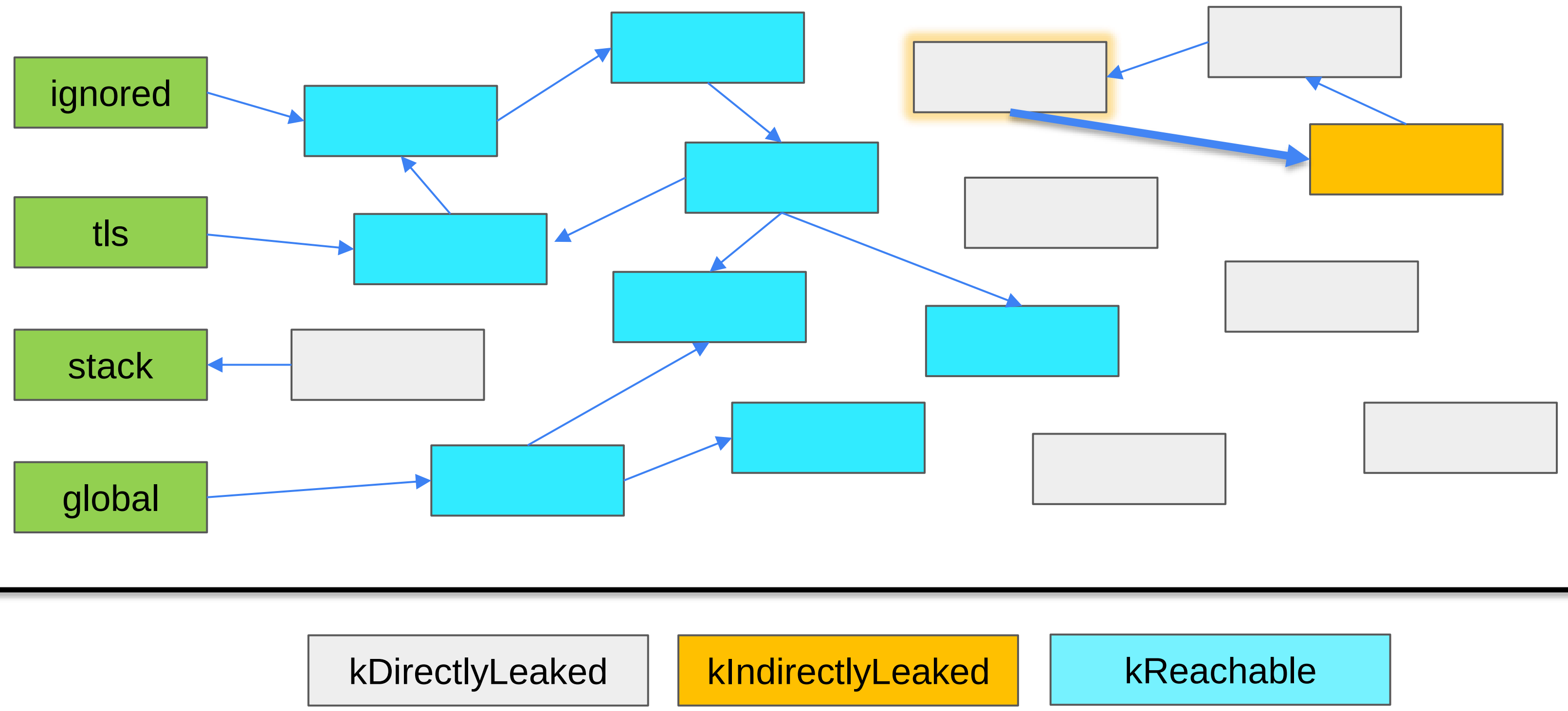
Leak Sanitizer: mark indirect leaks

Will check each leaked chunk for pointers to other leaked chunks



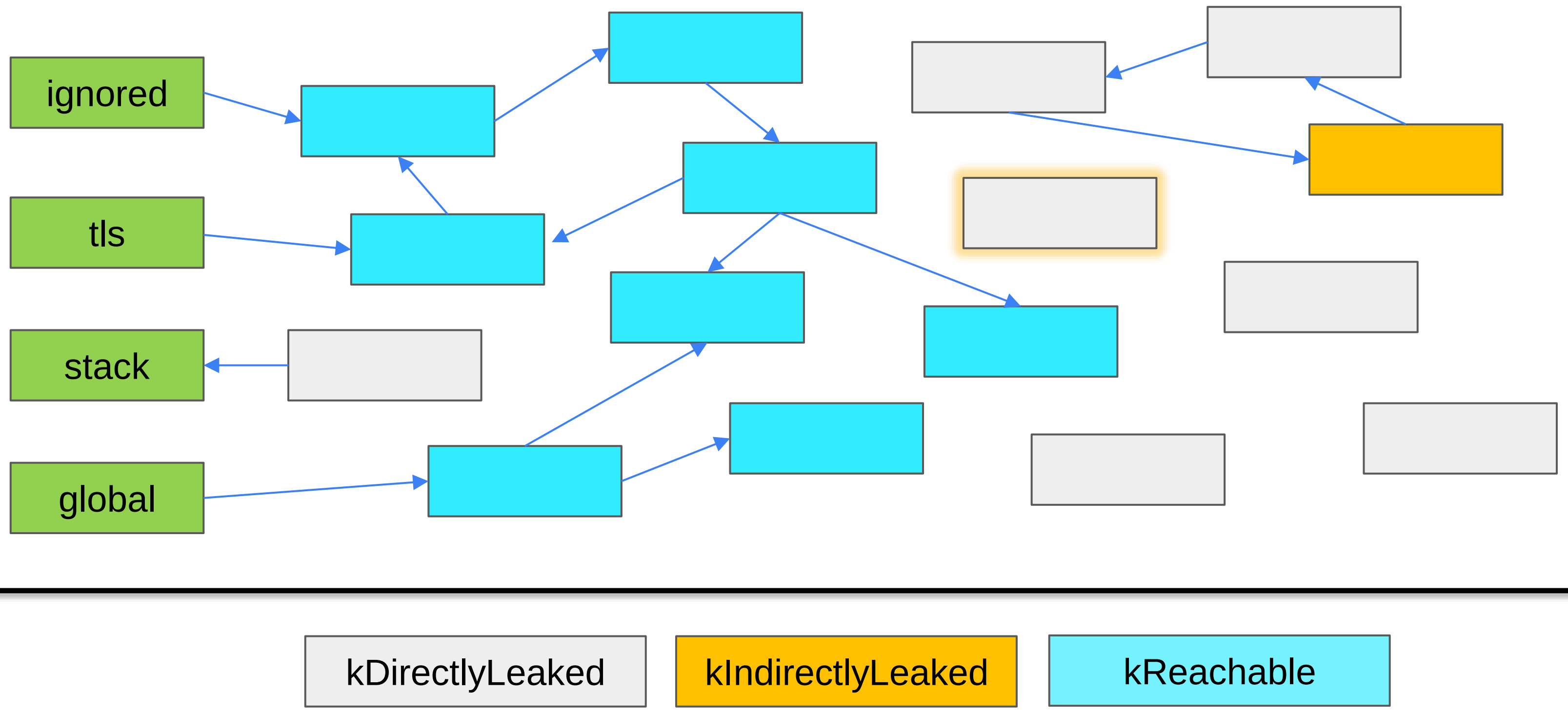
Leak Sanitizer: mark indirect leaks

Will check each leaked chunk for pointers to other leaked chunks



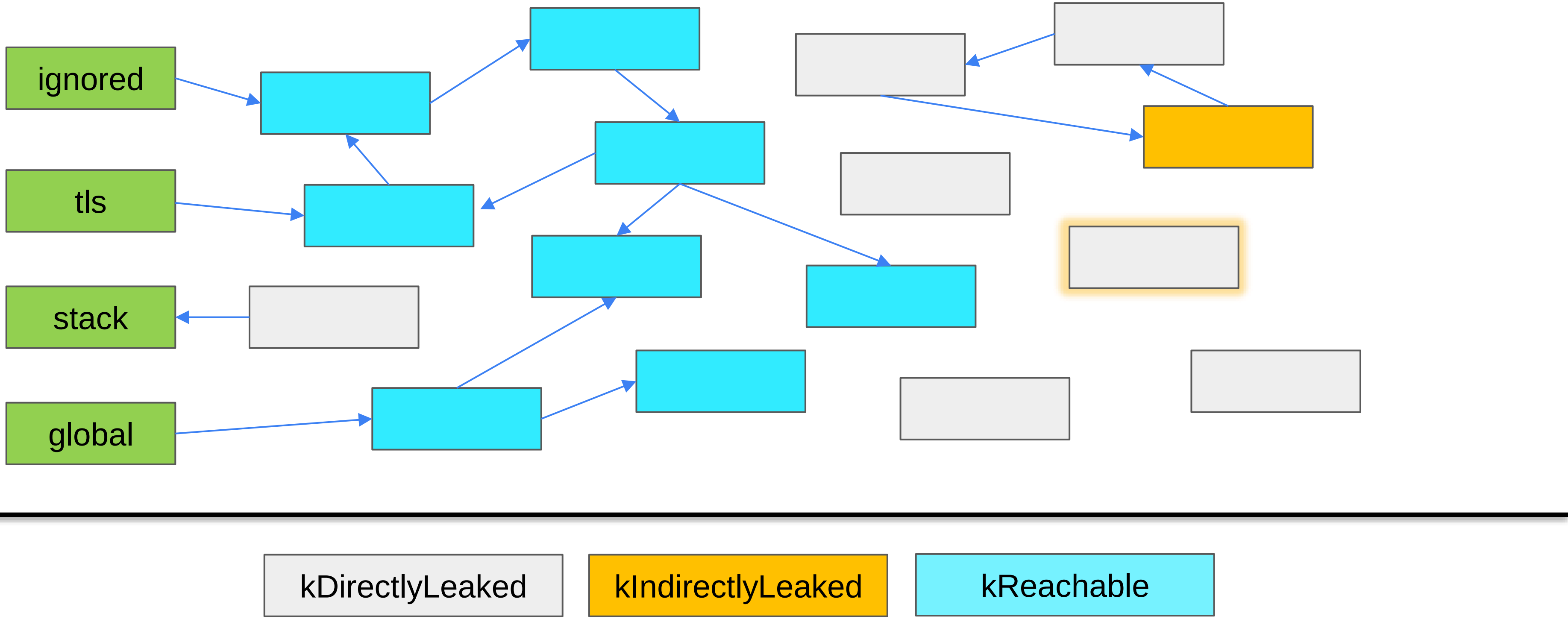
Leak Sanitizer: mark indirect leaks

Will check each leaked chunk for pointers to other leaked chunks



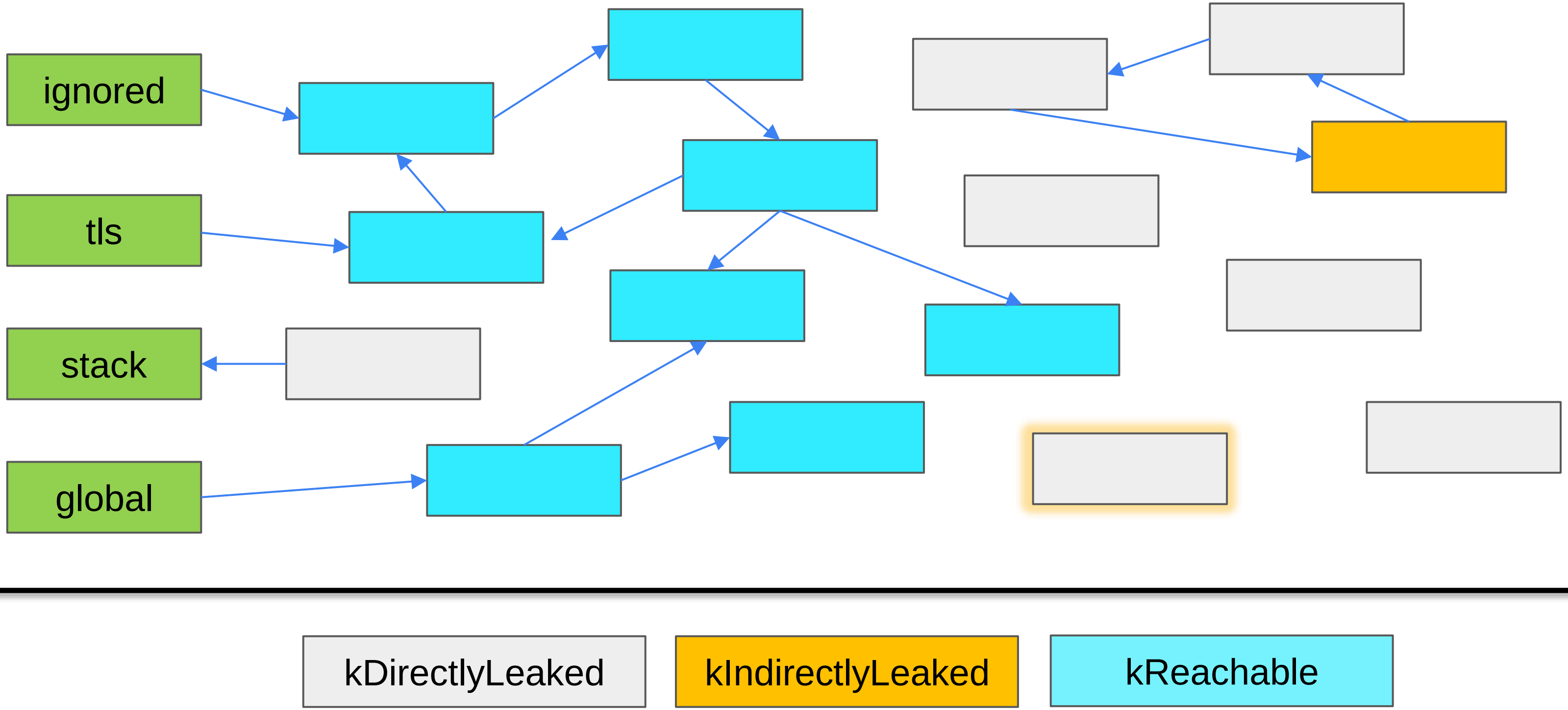
Leak Sanitizer: mark indirect leaks

Will check each leaked chunk for pointers to other leaked chunks



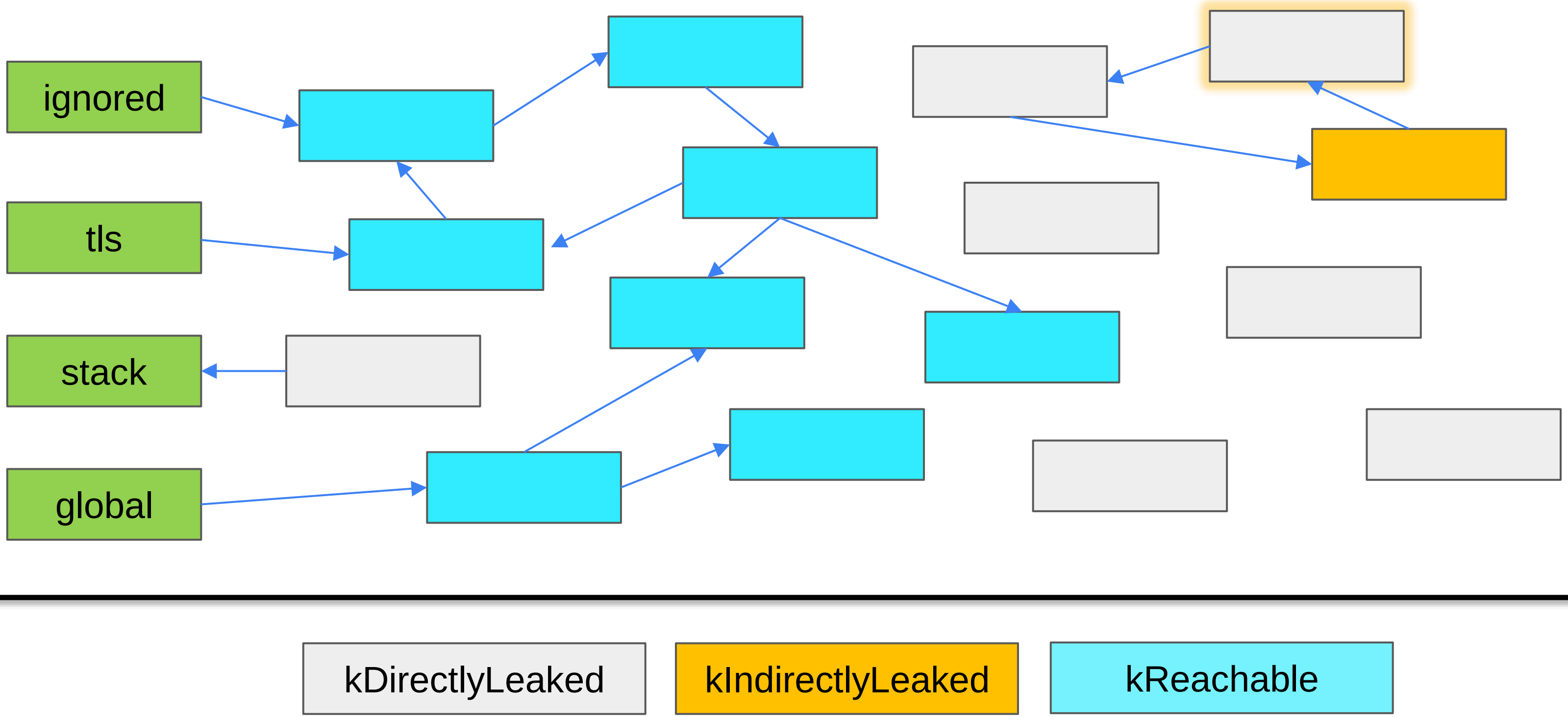
Leak Sanitizer: mark indirect leaks

Will check each leaked chunk for pointers to other leaked chunks



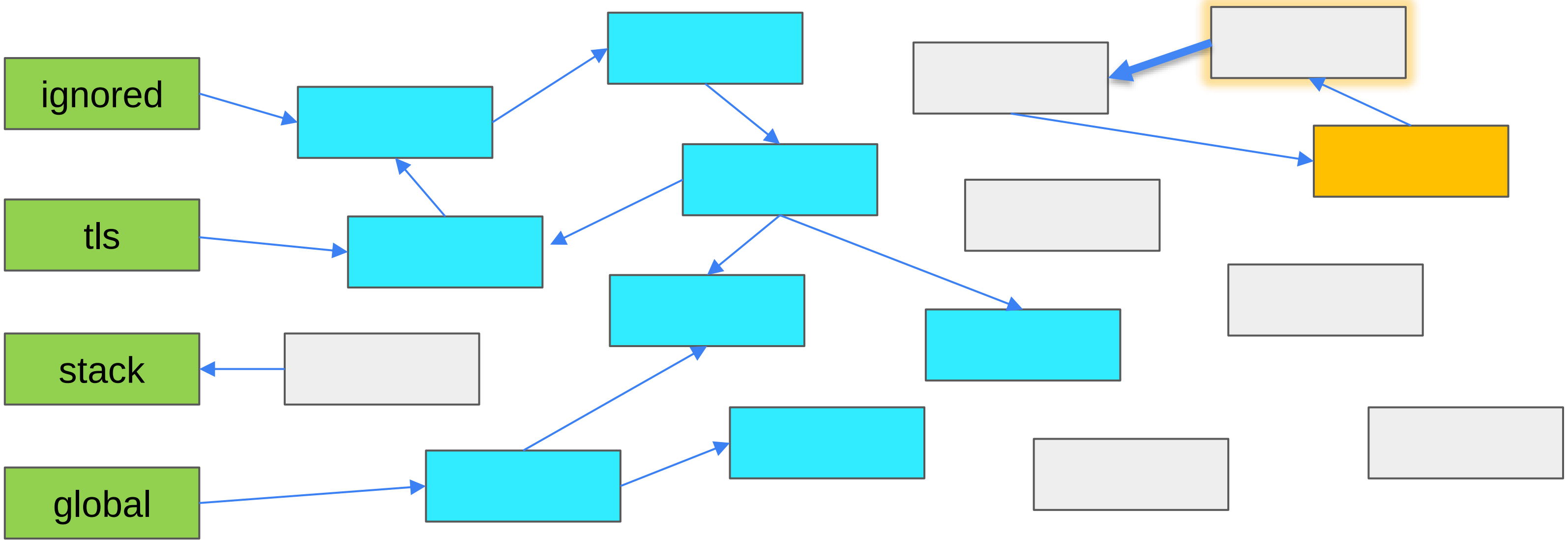
Leak Sanitizer: mark indirect leaks

Will check each leaked chunk for pointers to other leaked chunks



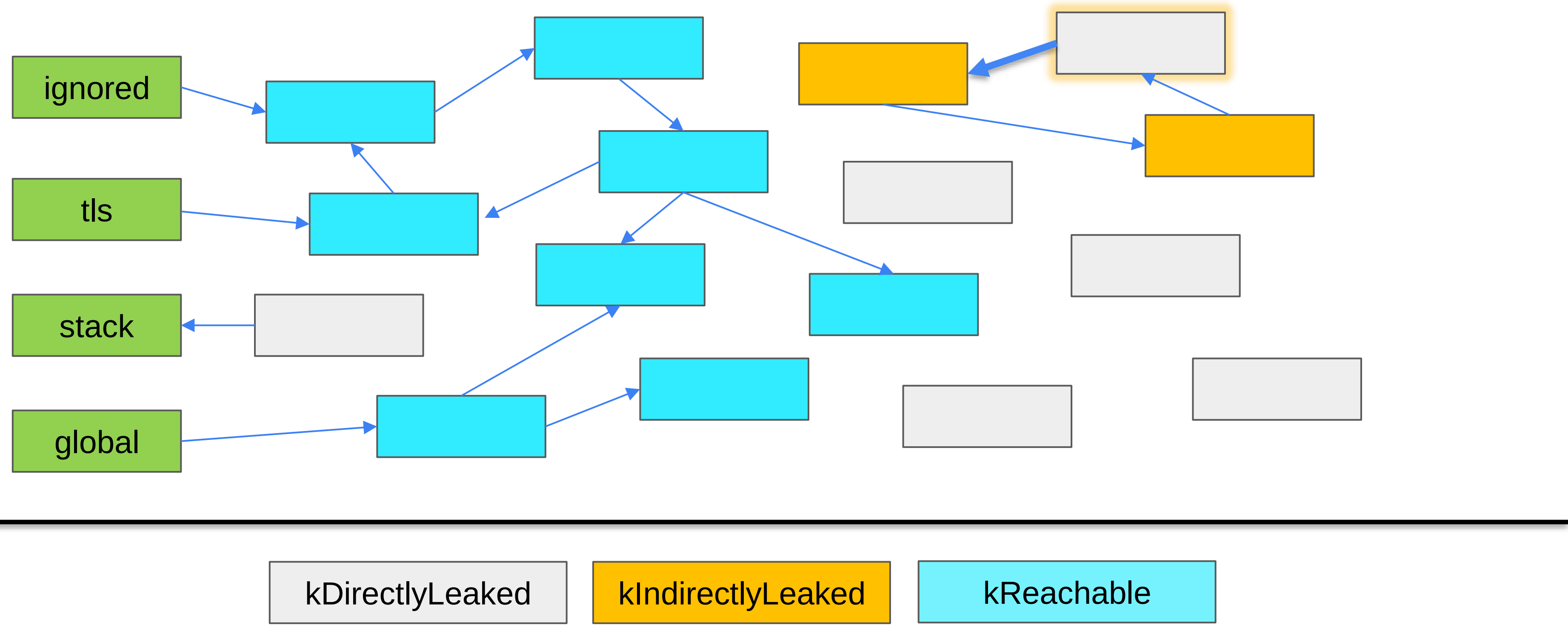
Leak Sanitizer: mark indirect leaks

Will check each leaked chunk for pointers to other leaked chunks



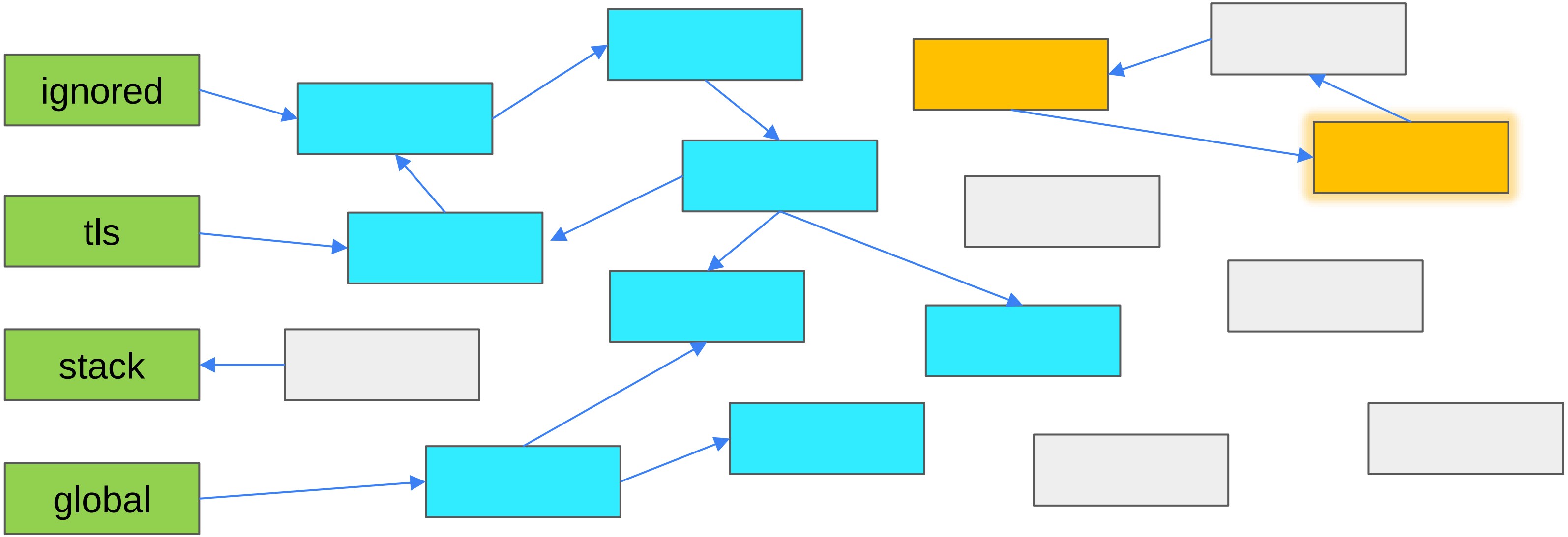
Leak Sanitizer: mark indirect leaks

Will check each leaked chunk for pointers to other leaked chunks



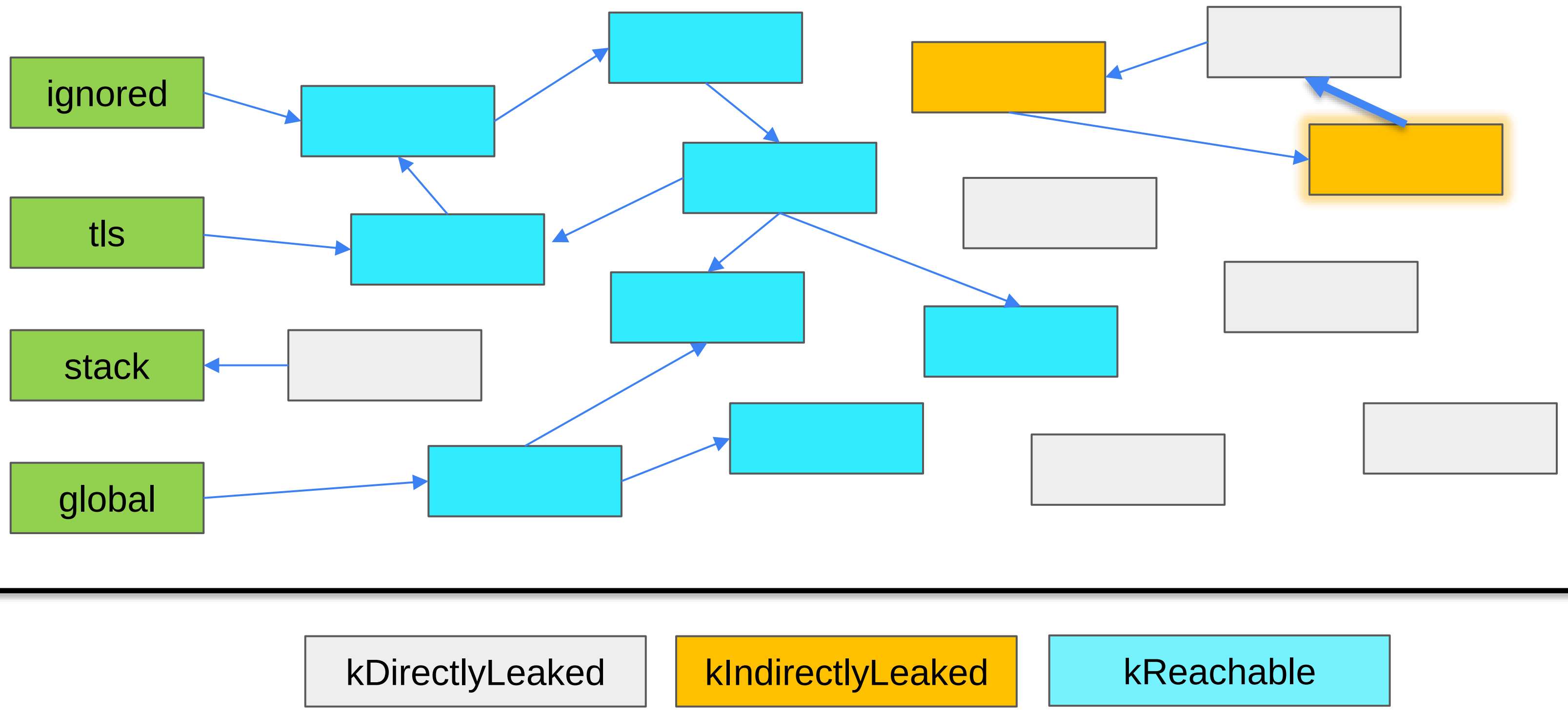
Leak Sanitizer: mark indirect leaks

Will check each leaked chunk for pointers to other leaked chunks



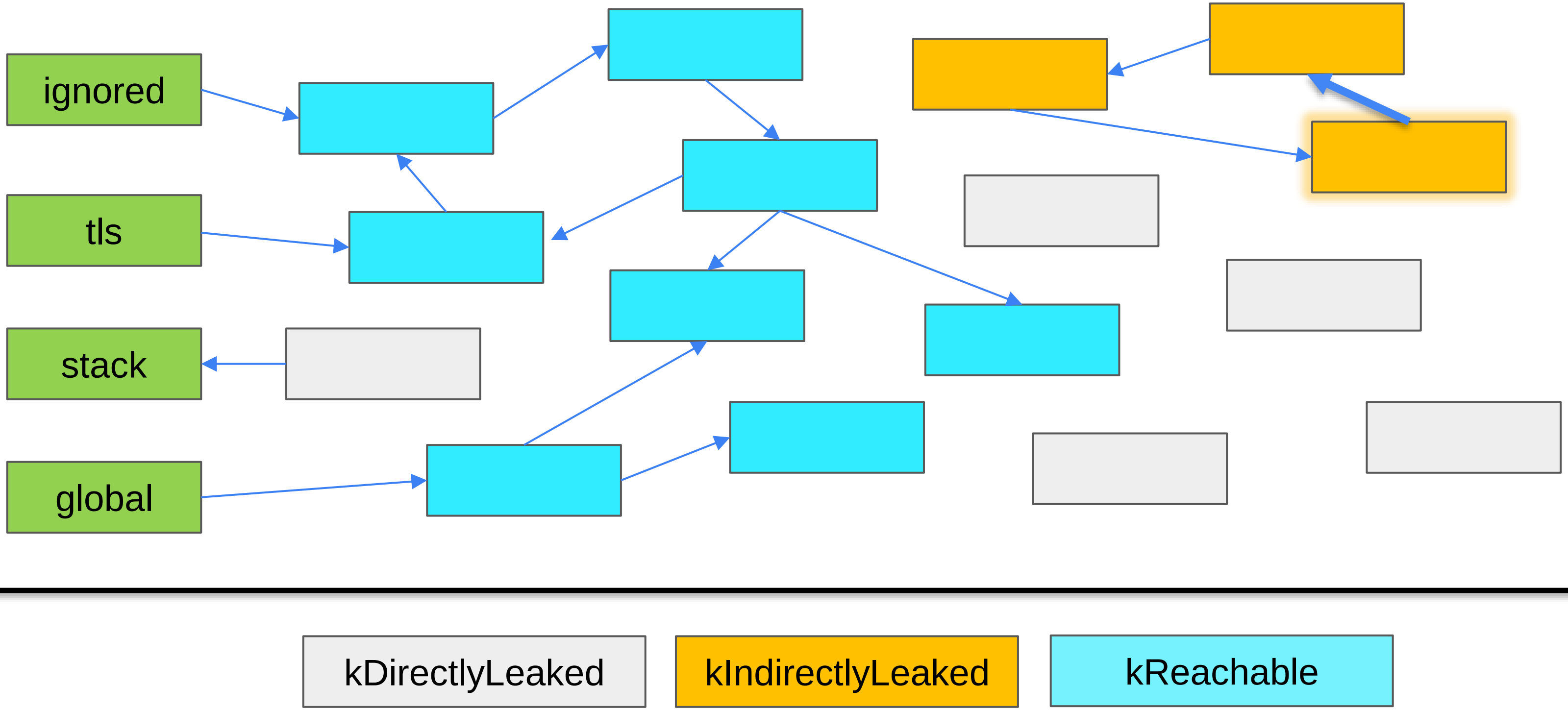
Leak Sanitizer: mark indirect leaks

Will check each leaked chunk for pointers to other leaked chunks



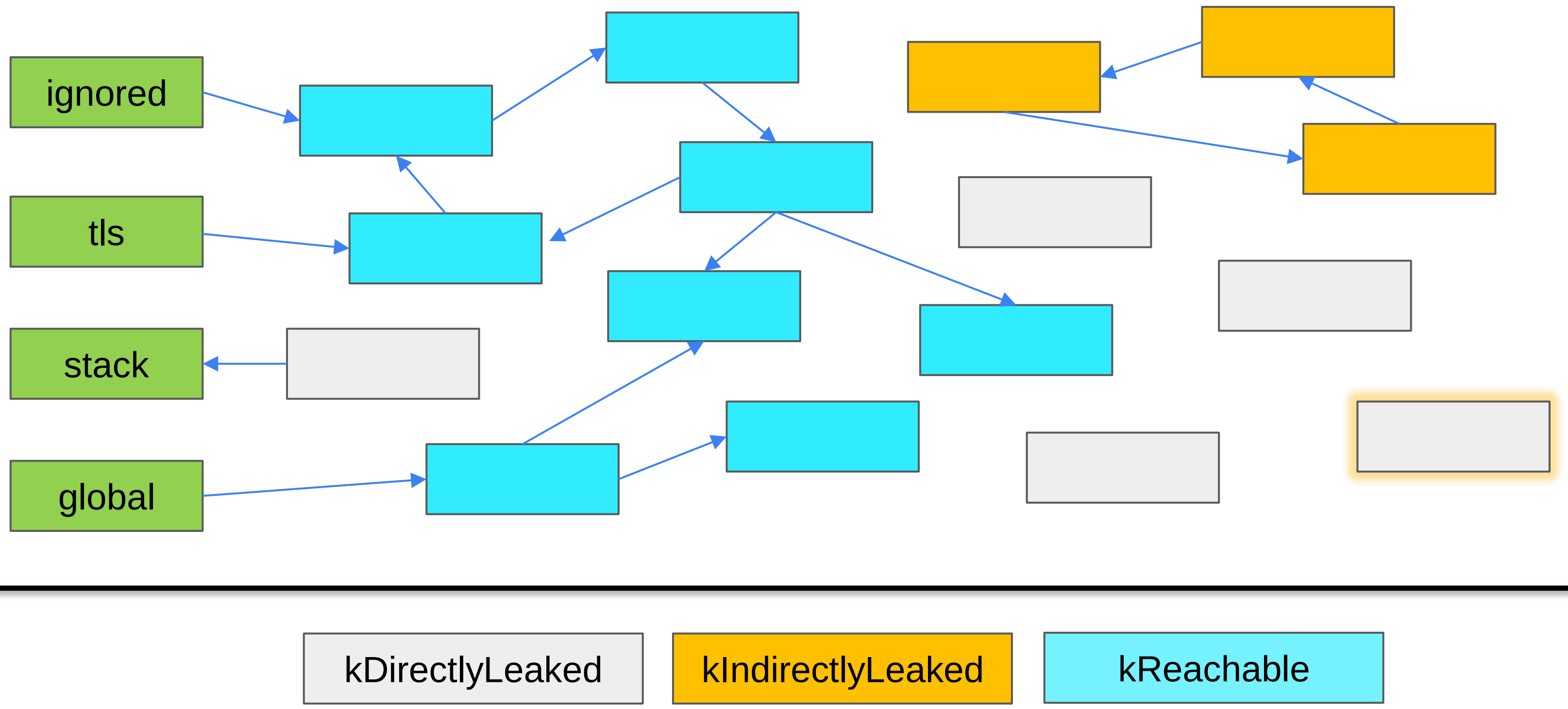
Leak Sanitizer: mark indirect leaks

Will check each leaked chunk for pointers to other leaked chunks



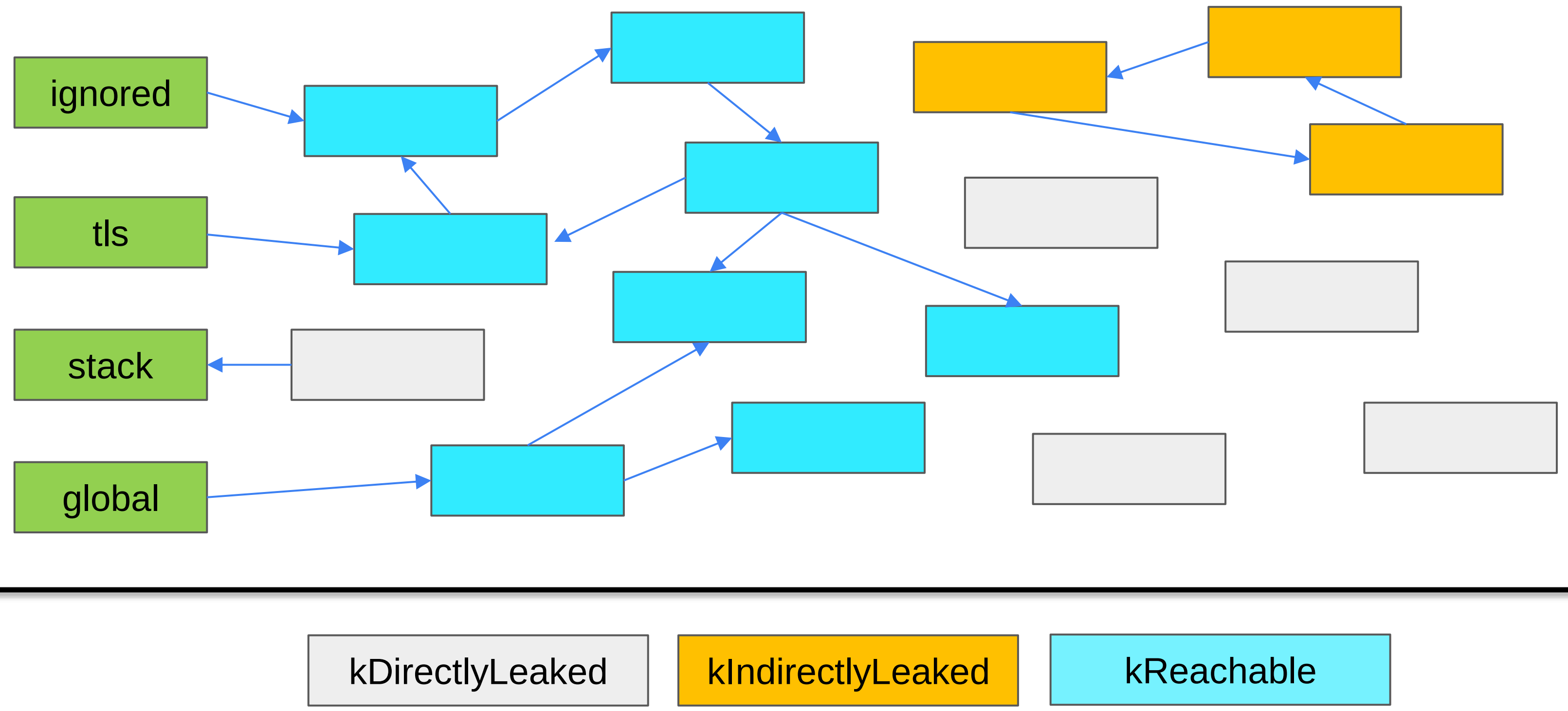
Leak Sanitizer: mark indirect leaks

Will check each leaked chunk for pointers to other leaked chunks



Leak Sanitizer: mark indirect leaks

Will check each leaked chunk for pointers to other leaked chunks



Leak Sanitizer: report all leaks

General algorithm:

1. Stop the world
2. Find all roots
3. Mark all reachable chunks
4. All not marked chunks are leaks
5. Mark indirect leaks
6. **Report all leaks**

Leak Sanitizer: mark indirect leaks

For each chunk do:

```
chunk = GetUserBegin(chunk);
LsanMetadata m(chunk);
if (!m.allocated())
  return;
if (m.tag() == kDirectlyLeaked || m.tag() == kIndirectlyLeaked)
  leaks->push_back({chunk, m.stack_trace_id(), m.requested_size(), m.tag()})
```

Check if the chunk leaked.
Add to report.

Let’s visualize leaked objects graph

Let’s visualize leaked objects graph

Let’s check what options are available for LSAN:

Let’s visualize leaked objects graph

Let’s check what options are available for LSAN:
Just run any application with LSAN and LSAN_OPTIONS=**help=1**

```
$ LSAN_OPTIONS=help=1 LD_PRELOAD="/lib/x86_64-linux-gnu/liblsan.so.0" ./a.out
```

Let’s visualize leaked objects graph

Let’s check what options are available for LSAN:
Just run any application with LSAN and LSAN_OPTIONS=**help=1**
There are a lot of them:

```
$ LSAN_OPTIONS=help=1 LD_PRELOAD="/lib/x86_64-linux-gnu/liblsan.so.0" ./a.out
Available flags for LeakSanitizer:
report_objects - Print addresses of leaked objects after main leak report. (Current Value: false)
resolution - Aggregate two objects into one leak if this many stack frames match. If zero, the
             entire stack trace must match. (Current Value: 0)
max_leaks   - The number of leaks reported. (Current Value: 0)
use_globals - Root set: include global variables (.data and .bss) (Current Value: true) use_stacks
- Root set: include thread stacks (Current Value: true)
use_registers - Root set: include thread registers (Current Value: true)
use_tls - Root set: include TLS and thread-specific storage (Current Value: true)
use_root_regions - Root set: include regions added via __lsan_register_root_region().
use_ld_allocations - Root set: mark as reachable all allocations made from dynamic linker. This was
the old way to handle dynamic TLS, and will be removed soon. Do not use this flag. (Current Value:
true)
use_unaligned - Consider unaligned pointers valid. (Current Value: false)
…
```

Let’s visualize leaked objects graph

Let’s check what options are available for LSAN:
Just run any application with LSAN and LSAN_OPTIONS=**help=1**
But we need only one:

```
$ LSAN_OPTIONS=help=1 LD_PRELOAD="/lib/x86_64-linux-gnu/liblsan.so.0" ./a.out
log_pointers - Debug logging (Current Value: false)
```

Let’s visualize leaked objects graph

A simple application with circular reference

```
$ LSAN_OPTIONS=help=1 LD_PRELOAD="/lib/x86_64-linux-gnu/liblsan.so.0" ./a.out
log_pointers - Debug logging (Current Value: false)
$ cat main.cpp
struct N {N* next;};
int main() {
    auto a = new N{};
    auto b = new N{};
    a->next = b;
    b->next = a;
    return 0;
}
```

Let’s visualize leaked objects graph

A simple application with circular reference

```
$ LSAN_OPTIONS=help=1 LD_PRELOAD="/lib/x86_64-linux-gnu/liblsan.so.0" ./a.out
log_pointers - Debug logging (Current Value: false)
$ cat main.cpp
struct N {N* next;};
int main() {
    auto a = new N{};
    auto b = new N{};
    a->next = b;
    b->next = a;
    return 0;
}
$ g++ -g main.cpp
```

Let’s visualize leaked objects graph

A simple application with circular reference

```
$ LSAN_OPTIONS=help=1 LD_PRELOAD="/lib/x86_64-linux-gnu/liblsan.so.0" ./a.out
log_pointers - Debug logging (Current Value: false)
$ cat main.cpp
struct N {N* next;};
int main() {
    auto a = new N{};
    auto b = new N{};
    a->next = b;
    b->next = a;
    return 0;
}
$ g++ -g main.cpp
```

Let’s visualize leaked objects graph

A simple application with circular reference
Run with LSAN

```
$ g++ -g main.cpp
$ LD_PRELOAD=/lib/x86_64-linux-gnu/liblsan.so.0 ./a.out
==15853==ERROR: LeakSanitizer: detected memory leaks
Indirect leak of 8 byte(s) in 1 object(s) allocated from:
    #0 0x7fe4befd6a92 in operator new(unsigned long) libsanitizer/lsan/lsan_interceptors.cpp:248
    #1 0x55cae748f173 in main /home/valexey/Projects/circ/main.cpp:5
    #2 0x7fe4beb88d8f in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58

Indirect leak of 8 byte(s) in 1 object(s) allocated from:
    #0 0x7fe4befd6a92 in operator new(unsigned long) libsanitizer/lsan/lsan_interceptors.cpp:248
    #1 0x55cae748f15e in main /home/valexey/Projects/circ/main.cpp:4
    #2 0x7fe4beb88d8f in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58

SUMMARY: LeakSanitizer: 16 byte(s) leaked in 2 allocation(s).
```

Let’s visualize leaked objects graph

A simple application with circular reference
Run with LSAN and pointer logging:

```
$ g++ -g main.cpp
$ LSAN_OPTIONS=log_pointers=1  LD_PRELOAD=/lib/x86_64-linux-gnu/liblsan.so.0 ./a.out
```

```
$ g++ -g main.cpp
$ LSAN_OPTIONS=log_pointers=1  LD_PRELOAD=/lib/x86_64-linux-gnu/liblsan.so.0 ./a.out
==15854==Scanning GLOBAL range 0x55cae7491da8-0x55cae7492018.
==15854==Scanning GLOBAL range 0x7fe4bf027a88-0x7fe4bf02be40.
==15854==Scanning GLOBAL range 0x7fe4bf02c118-0x7fe4bf57d168.
==15854==Scanning GLOBAL range 0x7fe4befa3880-0x7fe4befb38c0.
==15854==0x7fe4befb0328: found 0x631000000000 pointing into chunk 0x631000000000-0x631000011c00 of
size 72704.
==15854==Scanning GLOBAL range 0x7fe4bed758f0-0x7fe4bed87e50.
==15854==Scanning GLOBAL range 0x7fe4bf5b8620-0x7fe4bf5bb2d8.
==15854==Scanning GLOBAL range 0x7fe4beb5ddc8-0x7fe4beb5e2e8.
==15854==Scanning GLOBAL range 0x7fe4beb3dd80-0x7fe4beb3e108.
==15854==Scanning REGISTERS range 0x7fe4be90e000-0x7fe4be90e418.
==15854==Scanning STACK range 0x7ffe68a6cc98-0x7ffe68a70000.
==15854==Scanning TLS range 0x7fe4bea46000-0x7fe4bea46760.
==15854==Scanning TLS range 0x7fe4bea542e0-0x7fe4bea54cc0.
==15854==Scanning HEAP range 0x631000000000-0x631000011c00.
==15854==Processing platform-specific allocations.
==15854==Scanning leaked chunks.
==15854==Scanning HEAP range 0x601000000000-0x601000000008.
==15854==0x601000000000: found 0x601000000010 pointing into chunk 0x601000000010-0x601000000018 of
size 8.
==15854==Scanning HEAP range 0x601000000010-0x601000000018.
==15854==0x601000000010: found 0x601000000000 pointing into chunk 0x601000000000-0x601000000008 of
size 8.
```

```
$ g++ -g main.cpp
$ LSAN_OPTIONS=log_pointers=1  LD_PRELOAD=/lib/x86_64-linux-gnu/liblsan.so.0 ./a.out
==15854==Scanning GLOBAL range 0x55cae7491da8-0x55cae7492018.
==15854==Scanning GLOBAL range 0x7fe4bf027a88-0x7fe4bf02be40.
==15854==Scanning GLOBAL range 0x7fe4bf02c118-0x7fe4bf57d168.
==15854==Scanning GLOBAL range 0x7fe4befa3880-0x7fe4befb38c0.
==15854==0x7fe4befb0328: found 0x631000000000 pointing into chunk 0x631000000000-0x631000011c00 of
size 72704.
==15854==Scanning GLOBAL range 0x7fe4bed758f0-0x7fe4bed87e50.
==15854==Scanning GLOBAL range 0x7fe4bf5b8620-0x7fe4bf5bb2d8.
==15854==Scanning GLOBAL range 0x7fe4beb5ddc8-0x7fe4beb5e2e8.
==15854==Scanning GLOBAL range 0x7fe4beb3dd80-0x7fe4beb3e108.
==15854==Scanning REGISTERS range 0x7fe4be90e000-0x7fe4be90e418.
==15854==Scanning STACK range 0x7ffe68a6cc98-0x7ffe68a70000.
==15854==Scanning TLS range 0x7fe4bea46000-0x7fe4bea46760.
==15854==Scanning TLS range 0x7fe4bea542e0-0x7fe4bea54cc0.
==15854==Scanning HEAP range 0x631000000000-0x631000011c00.
==15854==Processing platform-specific allocations.
==15854==Scanning leaked chunks.
==15854==Scanning HEAP range 0x601000000000-0x601000000008.
==15854==0x601000000000: found 0x601000000010 pointing into chunk 0x601000000010-0x601000000018 of
size 8.
==15854==Scanning HEAP range 0x601000000010-0x601000000018.
==15854==0x601000000010: found 0x601000000000 pointing into chunk 0x601000000000-0x601000000008 of
size 8.
```

```
$ g++ -g main.cpp
$ LSAN_OPTIONS=log_pointers=1  LD_PRELOAD=/lib/x86_64-linux-gnu/liblsan.so.0 ./a.out
==15854==Scanning leaked chunks.
==15854==Scanning HEAP range 0x601000000000-0x601000000008.
==15854==0x601000000000: found 0x601000000010 pointing into chunk 0x601000000010-0x601000000018 of
size 8.
==15854==Scanning HEAP range 0x601000000010-0x601000000018.
==15854==0x601000000010: found 0x601000000000 pointing into chunk 0x601000000000-0x601000000008 of
size 8.
```

```
$ g++ -g main.cpp
$ LSAN_OPTIONS=log_pointers=1  LD_PRELOAD=/lib/x86_64-linux-gnu/liblsan.so.0 ./a.out
==15854==Scanning leaked chunks.
==15854==Scanning HEAP range 0x601000000000-0x601000000008.
==15854==0x601000000000: found 0x601000000010 pointing into chunk 0x601000000010-0x601000000018 of
size 8.
==15854==Scanning HEAP range 0x601000000010-0x601000000018.
==15854==0x601000000010: found 0x601000000000 pointing into chunk 0x601000000000-0x601000000008 of
size 8.
```

```
$ g++ -g main.cpp
$ LSAN_OPTIONS=log_pointers=1  LD_PRELOAD=/lib/x86_64-linux-gnu/liblsan.so.0 ./a.out
==15854==Scanning leaked chunks.
==15854==Scanning HEAP range 0x601000000000-0x601000000008.
==15854==0x601000000000: found 0x601000000010 pointing into chunk 0x601000000010-0x601000000018 of size 8.
==15854==Scanning HEAP range 0x601000000010-0x601000000018.
==15854==0x601000000010: found 0x601000000000 pointing into chunk 0x601000000000-0x601000000008 of size 8.
```

```
$ g++ -g main.cpp
$ LSAN_OPTIONS=log_pointers=1  LD_PRELOAD=/lib/x86_64-linux-gnu/liblsan.so.0 ./a.out
==15854==Scanning leaked chunks.
==15854==Scanning HEAP range 0x601000000000-0x601000000008.
==15854==0x601000000000: found 0x601000000010 pointing into chunk 0x601000000010-0x601000000018 of
size 8.
==15854==Scanning HEAP range 0x601000000010-0x601000000018.
==15854==0x601000000010: found 0x601000000000 pointing into chunk 0x601000000000-0x601000000008 of
size 8.
```

```
$ g++ -g main.cpp
$ LSAN_OPTIONS=log_pointers=1  LD_PRELOAD=/lib/x86_64-linux-gnu/liblsan.so.0 ./a.out
==15854==Scanning leaked chunks.
==15854==Scanning HEAP range 0x601000000000-0x601000000008.
==15854==0x601000000000: found 0x601000000010 pointing into chunk 0x601000000010-0x601000000018 of
size 8.
==15854==Scanning HEAP range 0x601000000010-0x601000000018.
==15854==0x601000000010: found 0x601000000000 pointing into chunk 0x601000000000-0x601000000008 of
size 8.
```

```
$ g++ -g main.cpp
$ LSAN_OPTIONS=log_pointers=1  LD_PRELOAD=/lib/x86_64-linux-gnu/liblsan.so.0 ./a.out
==15854==Scanning leaked chunks.
==15854==Scanning HEAP range 0x601000000000-0x601000000008.
==15854==0x601000000000: found 0x601000000010 pointing into chunk 0x601000000010-0x601000000018 of
size 8.
==15854==Scanning HEAP range 0x601000000010-0x601000000018.
==15854==0x601000000010: found 0x601000000000 pointing into chunk 0x601000000000-0x601000000008 of
size 8.
```

Let’s visualize leaked objects graph

Is it possible to visualize live object graph by the same way?

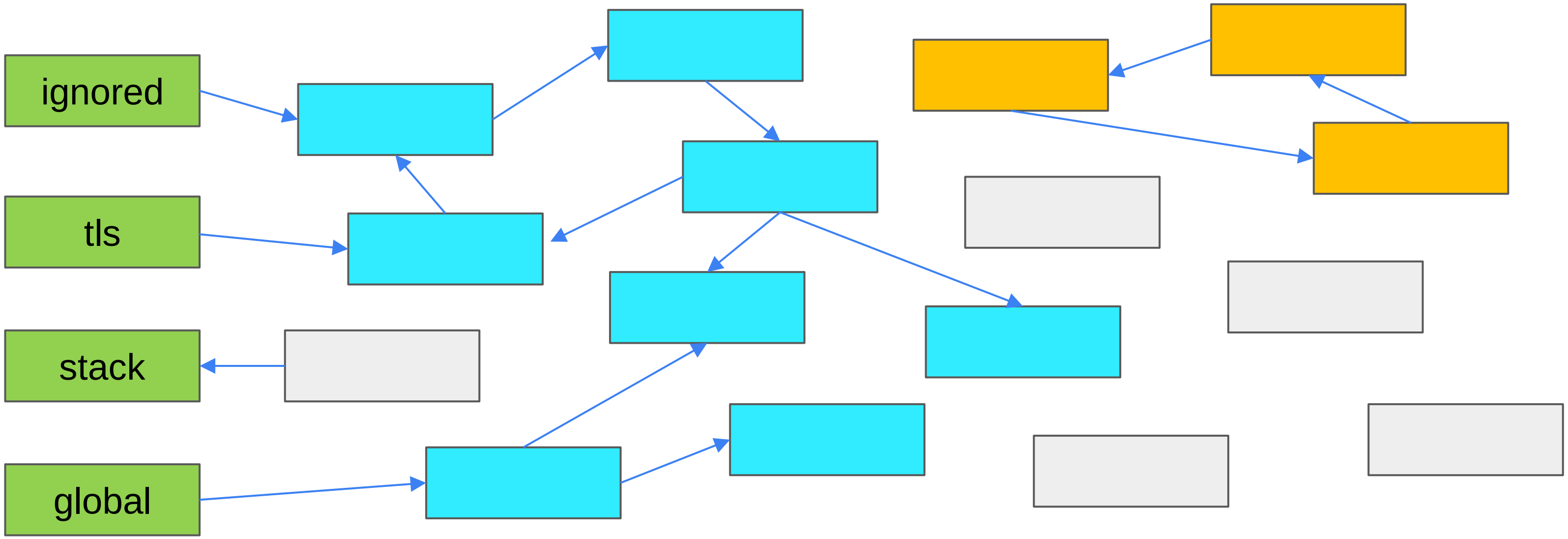
Let’s visualize leaked objects graph

Is it possible to visualize live object graph by the same way?

No

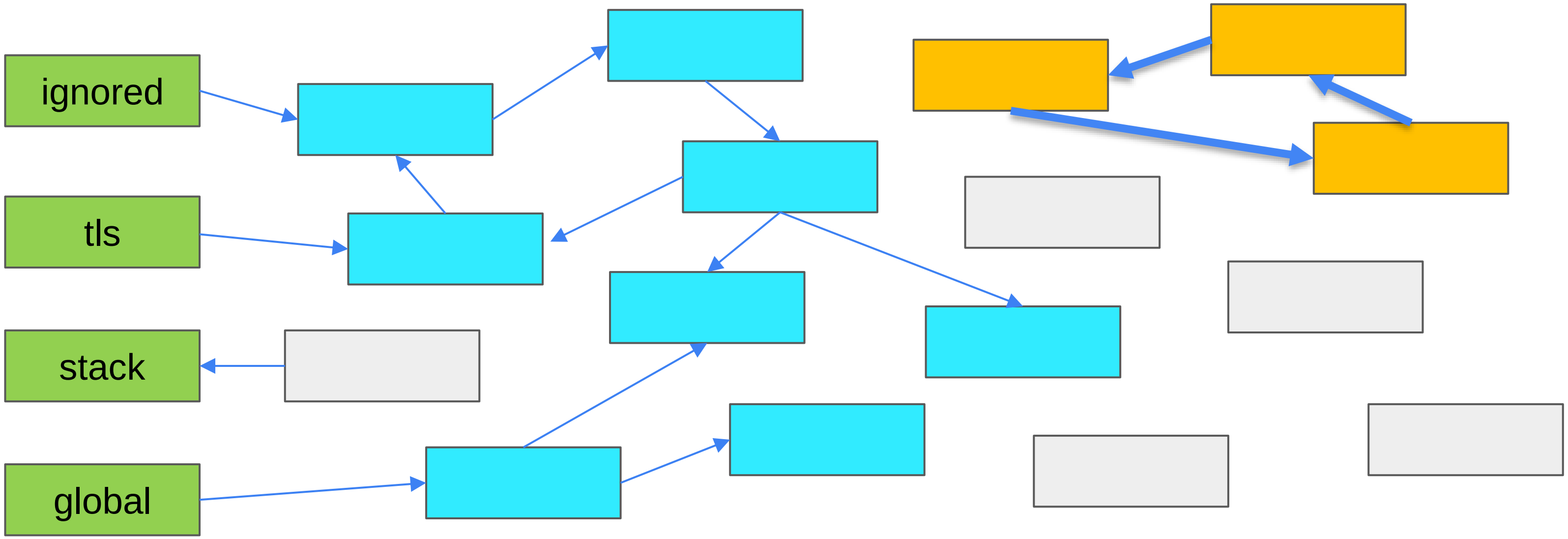
Let’s visualize leaked objects graph

We see all edges for leaked objects because LSAN scans ALL leaked objects to find and mark all indirect leaks. Without flood fill.



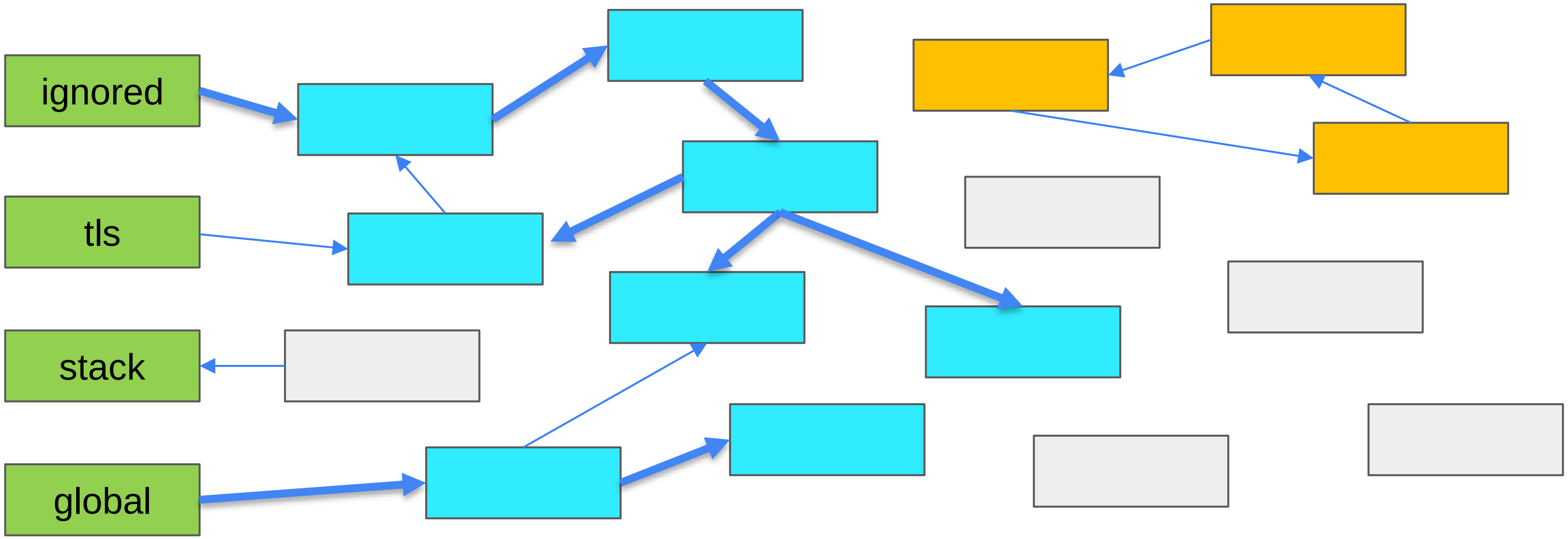
Let’s visualize leaked objects graph

We see all edges for leaked objects because LSAN scans ALL leaked objects to find and mark all indirect leaks. And logs all edges. Without flood fill.



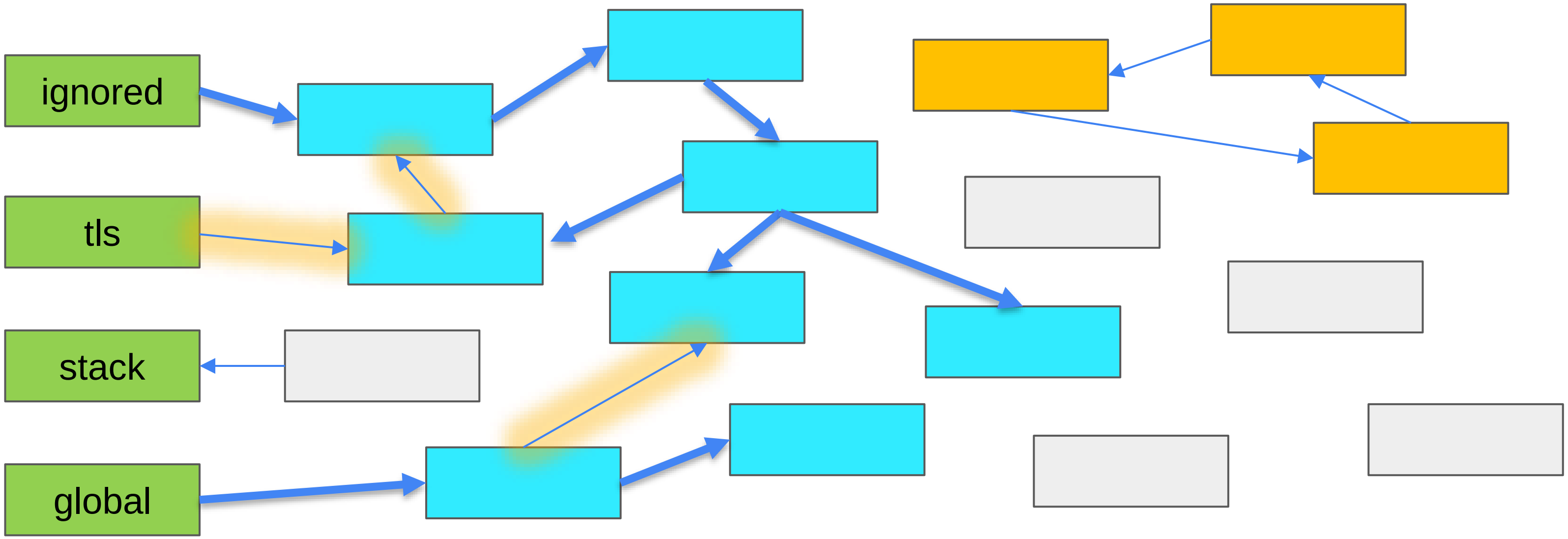
Let's visualize leaked objects graph

For live objects we'll see only part of real edges.



Let’s visualize leaked objects graph

For live objects we’ll see only part of real edges.



Bonus part: LSAN/ASAN monitoring

Bonus part: LSAN/ASAN monitoring

It would be great to know (approximately) WHEN leaks were occurred

Bonus part: LSAN/ASAN monitoring

It would be great to know (approximately) WHEN leaks were occurred
For this purposes we can create a simple lib

```
$ cat lsan_monitor.cpp
#include <thread>
#include <chrono>
extern "C" {
    void __lsan_do_recoverable_leak_check();
}

namespace {
void run();
std::thread lsan_monitor_thread(run);
void run() {
    lsan_monitor_thread.detach();
    for(;;) {
            std::this_thread::sleep_for(std::chrono::milliseconds(5000));
            __lsan_do_recoverable_leak_check();
    }
}
} // namespace
```

Bonus part: LSAN/ASAN monitoring

Compile and run it with our sample app:

```
$ clang -fPIC -shared lsan_monitor.cpp -o lsan_monitor.so
$ cat main.cpp
#include <thread>
#include <chrono>

int main() {
    int* mem;
    while (1) {
        mem = new int;
        std::this_thread::sleep_for(std::chrono::milliseconds(1000));
    }
}
```

Bonus part: LSAN/ASAN monitoring

Compile and run it with our sample app:

```
$  LD_PRELOAD="./lsan_monitor.so /lib/x86_64-linux-gnu/liblsan.so.0" ./a.out
```

Bonus part: LSAN/ASAN monitoring

5 sec later…

```
$  LD_PRELOAD="./lsan_monitor.so /lib/x86_64-linux-gnu/liblsan.so.0" ./a.out
==================================================================24893==ERROR: LeakSanitizer:
detected memory leaks

Direct leak of 16 byte(s) in 4 object(s) allocated from:#0 0x7fcd5c232a92 in operator new(unsigned
long) libsanitizer/lsan/lsan_interceptors.cpp:248
#1 0x5627f00d21cd in main (/home/valexey/Projects/lsan_reader/a.out+0x11cd)
#2 0x7fcd5bde4d8f in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58

SUMMARY: LeakSanitizer: 16 byte(s) leaked in 4 allocation(s).
```

Bonus part: LSAN/ASAN monitoring

10 sec later…

```
$  LD_PRELOAD="./lsan_monitor.so /lib/x86_64-linux-gnu/liblsan.so.0" ./a.out
==================================================================24893==ERROR: LeakSanitizer:
detected memory leaks

Direct leak of 16 byte(s) in 4 object(s) allocated from:#0 0x7fcd5c232a92 in operator new(unsigned
long) libsanitizer/lsan/lsan_interceptors.cpp:248
#1 0x5627f00d21cd in main (/home/valexey/Projects/lsan_reader/a.out+0x11cd)
#2 0x7fcd5bde4d8f in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58

SUMMARY: LeakSanitizer: 16 byte(s) leaked in 4 allocation(s).
=================================================================
==24893==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 36 byte(s) in 9 object(s) allocated from:
    #0 0x7fcd5c232a92 in operator new(unsigned long) libsanitizer/lsan/lsan_interceptors.cpp:248
    #1 0x5627f00d21cd in main (/home/valexey/Projects/lsan_reader/a.out+0x11cd)
    #2 0x7fcd5bde4d8f in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58
SUMMARY: LeakSanitizer: 36 byte(s) leaked in 9 allocation(s).
```

Bonus part: LSAN/ASAN monitoring

15 sec later…
..and so on

```
$  LD_PRELOAD="./lsan_monitor.so /lib/x86_64-linux-gnu/liblsan.so.0" ./a.out
=================================================================
==24893==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 36 byte(s) in 9 object(s) allocated from:
    #0 0x7fcd5c232a92 in operator new(unsigned long) libsanitizer/lsan/lsan_interceptors.cpp:248
    #1 0x5627f00d21cd in main (/home/valexey/Projects/lsan_reader/a.out+0x11cd)
    #2 0x7fcd5bde4d8f in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58
SUMMARY: LeakSanitizer: 36 byte(s) leaked in 9 allocation(s).
=================================================================
==24893==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 56 byte(s) in 14 object(s) allocated from:
    #0 0x7fcd5c232a92 in operator new(unsigned long) libsanitizer/lsan/lsan_interceptors.cpp:248
    #1 0x5627f00d21cd in main (/home/valexey/Projects/lsan_reader/a.out+0x11cd)
    #2 0x7fcd5bde4d8f in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58

SUMMARY: LeakSanitizer: 56 byte(s) leaked in 14 allocation(s).
```

Bonus part: LSAN/ASAN monitoring

How does it work: our library just calls `__lsan_do_recoverable_leak_check` each 5 seconds...

```
$ cat lsan_monitor.cpp
#include <thread>
#include <chrono>
extern "C" {
    void __lsan_do_recoverable_leak_check();
}

namespace {
void run();
std::thread lsan_monitor_thread(run);
void run() {
    lsan_monitor_thread.detach();
    for(;;) {
            std::this_thread::sleep_for(std::chrono::milliseconds(5000));
            __lsan_do_recoverable_leak_check();
    }
}
} // namespace
```

Bonus part: LSAN/ASAN monitoring

Good candidates for such (monitoring) usage from LSAN or ASAN API:

```
__sanitizer_print_memory_profile
__sanitizer_get_current_allocated_bytes
__sanitizer_get_heap_size
__sanitizer_get_free_bytes
__asan_print_accumulated_stats
```

Other API functions are here:

```
#include <sanitizer/lsan_interface.h>
#include <sanitizer/allocator_interface.h>
#include <sanitizer/asan_interface.h>
```

Thanks!

References:

- [LLVM compiler-rt github]
- [How Linux Elf Symbols Work and How They Are Used in C++]
- [Mask Ray: All about LSAN]

Contacts:

- linkedin: https://www.linkedin.com/in/valexey/
- e-mail: alexey.veselovsky@gmail.com
- telegram: @l_vlxy_l

Illustrator: elenaarm1805@gmail.com