C++ONLINE

JF Bastien

KEYNOTE:

The Bytes Before the Types

2024

The Bytes Before the Types

Unveiling Uninitialized Uses

JF Bastien
Distinguished Engineer, Woven by Toyota
Chair, WG21 C++ evolution working group

jfb@woven.toyota
jfbastien.com
@jfbastien

ジェイエフ　バスティエン
Distinguished エンジニア、ウーブン・バイ・トヨタ
議長、WG21 C++ 進化作業グループ

```
void get_address(unsigned char *out) {
  unsigned char addr[MAX_ADDR_LEN];

  //  ... do something

  memcpy(out, addr, sizeof(addr));
}
```

what's in here?

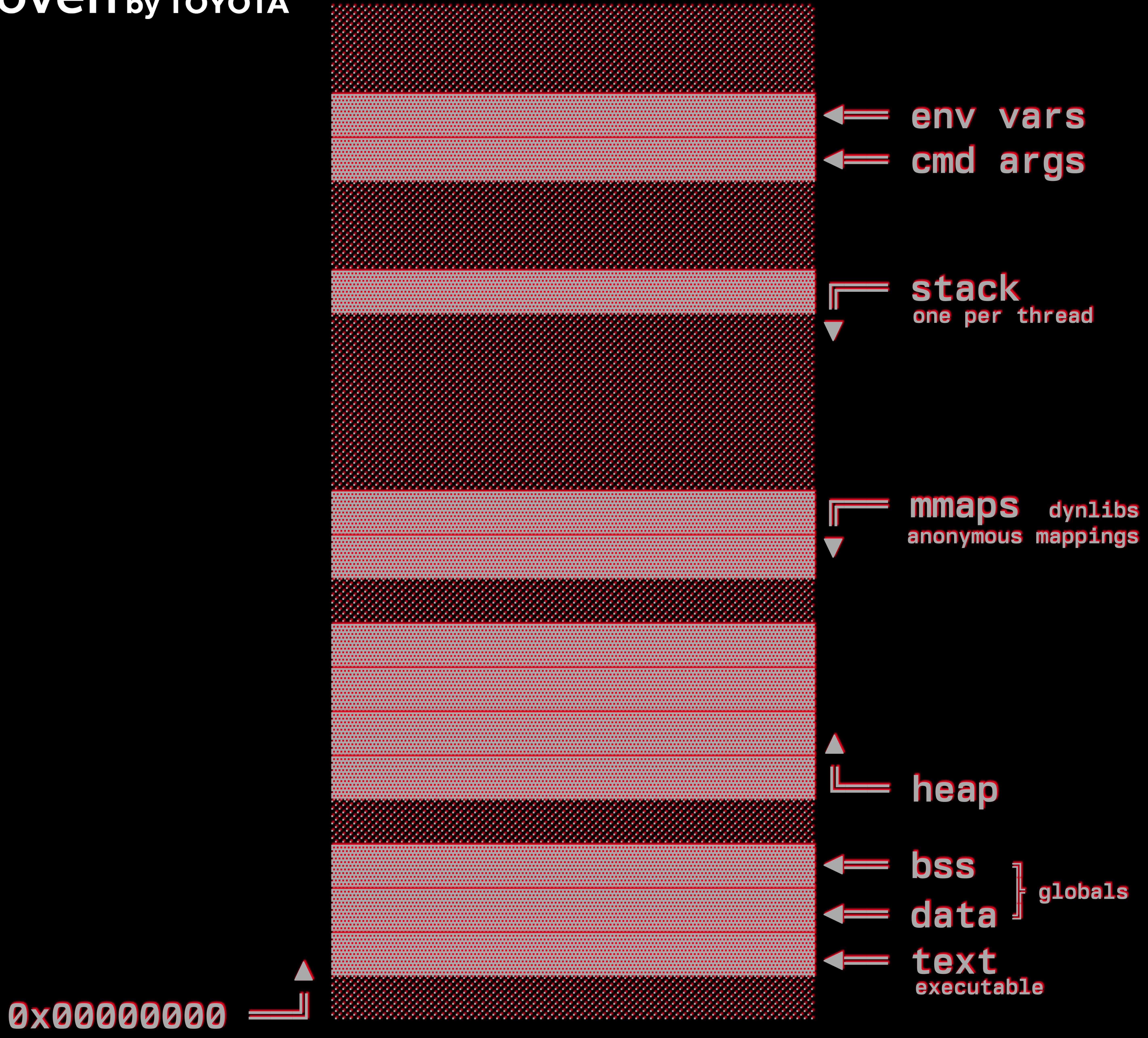
what will come out?

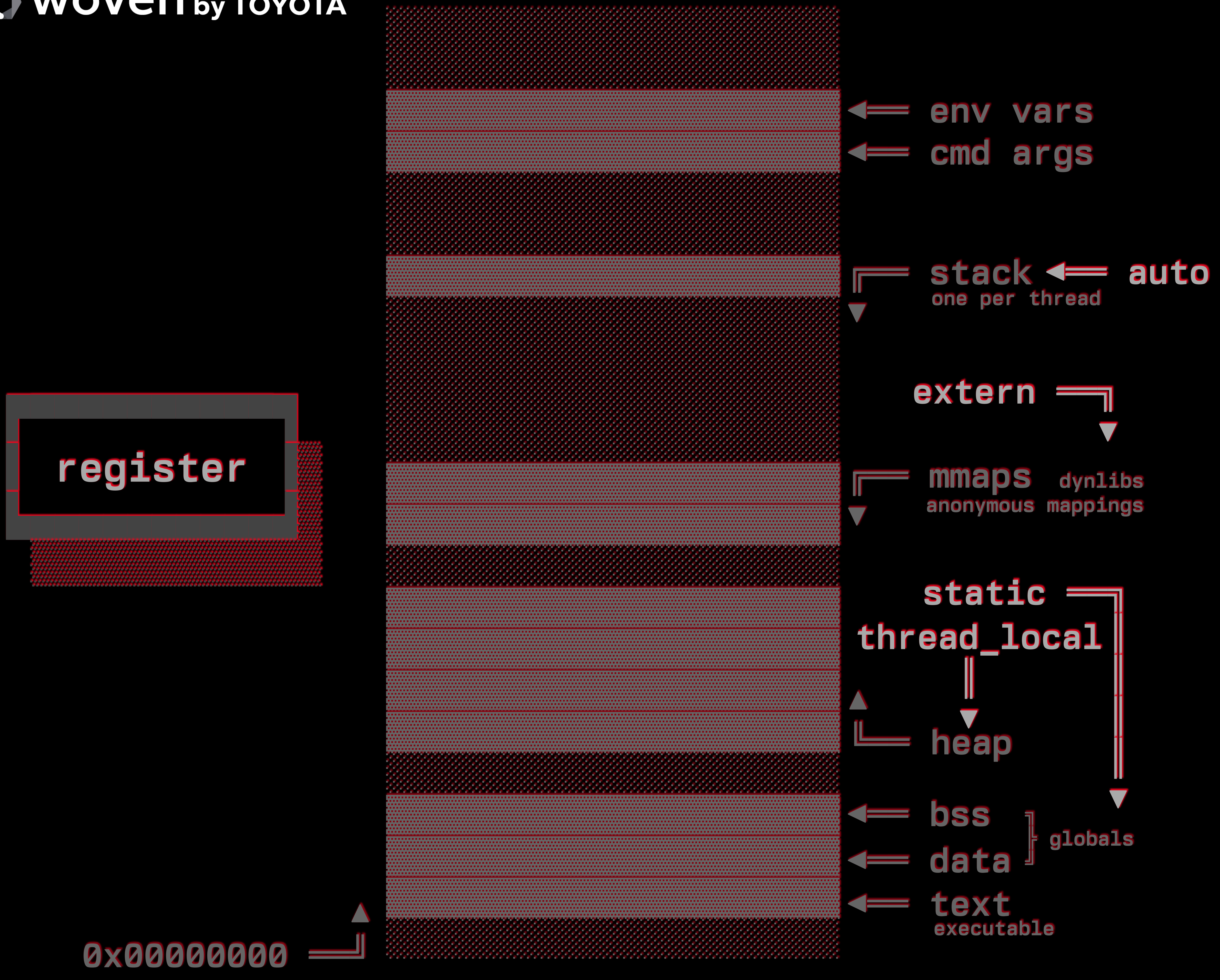
what's in there?

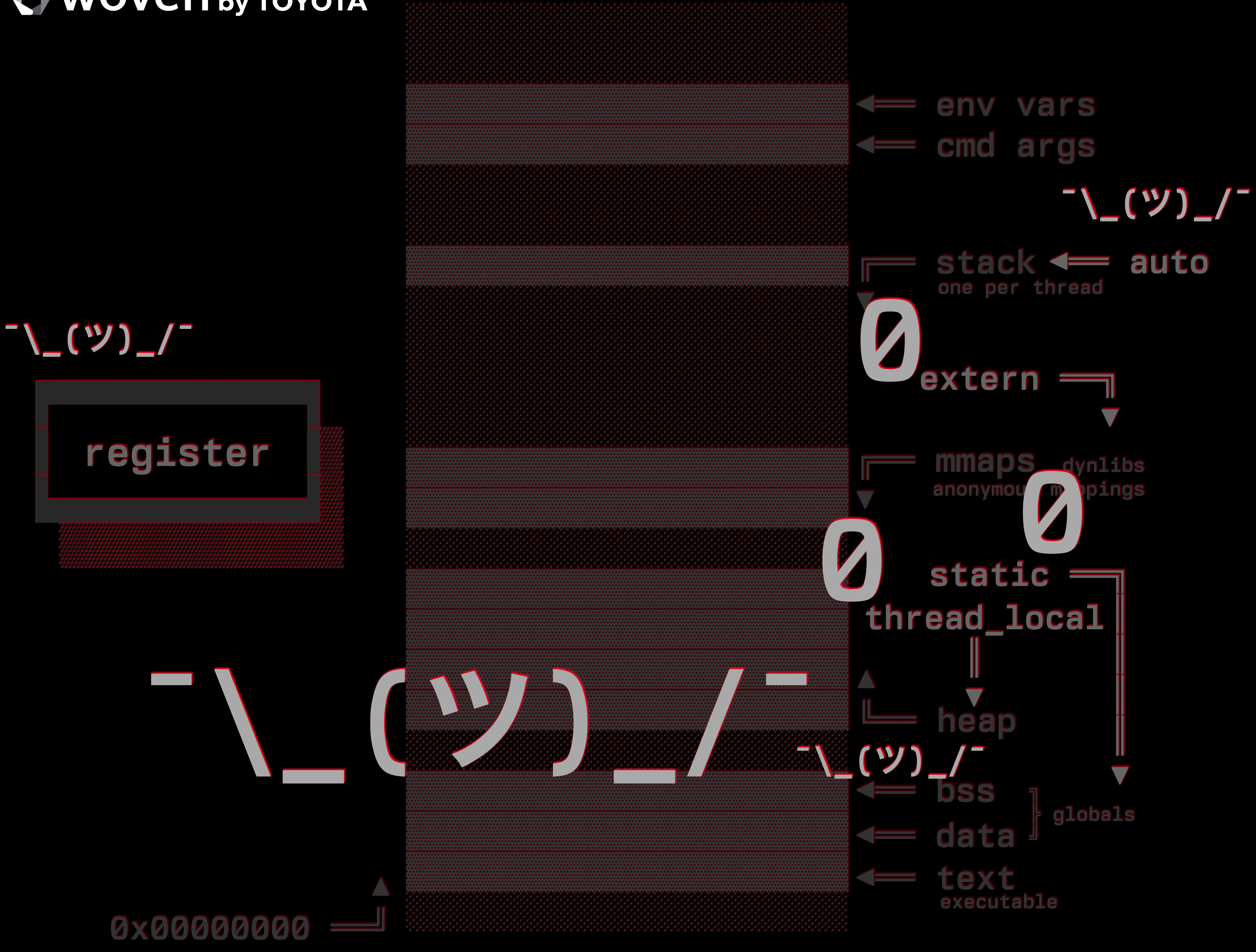
```
new unsigned char[128];
```

WARNING: herein lies undefined behavior

†but implementations provide concrete behavior


env vars
cmd args
stack
one per thread
mmaps
dynlibs
anonymous mappings
heap
bss
data
globals
text
executable
0x00000000


env vars
cmd args
stack
auto
one per thread
extern
mmaps
dynlibs
anonymous mappings
register
static
thread_local
heap
bss
data
globals
text
executable
0x00000000



env vars

cmd args

¯_(ツ)_/¯

stack ⟸ auto

one per thread

0

extern

mmaps dynlibs

anonymous mappings

0

¯_(ツ)_/¯

register

0

static

thread_local

¯_(ツ)_/¯

heap

¯_(ツ)_/¯

bss

data

globals

text

executable

0x00000000

woven by TOYOTA

why not just initialize everything?

2.7–4.5%

runtime cost

“Why Nothing Matters: The Impact of Zeroing” is there even a problem?

typical outcome: read stale value

best case: unexpected result

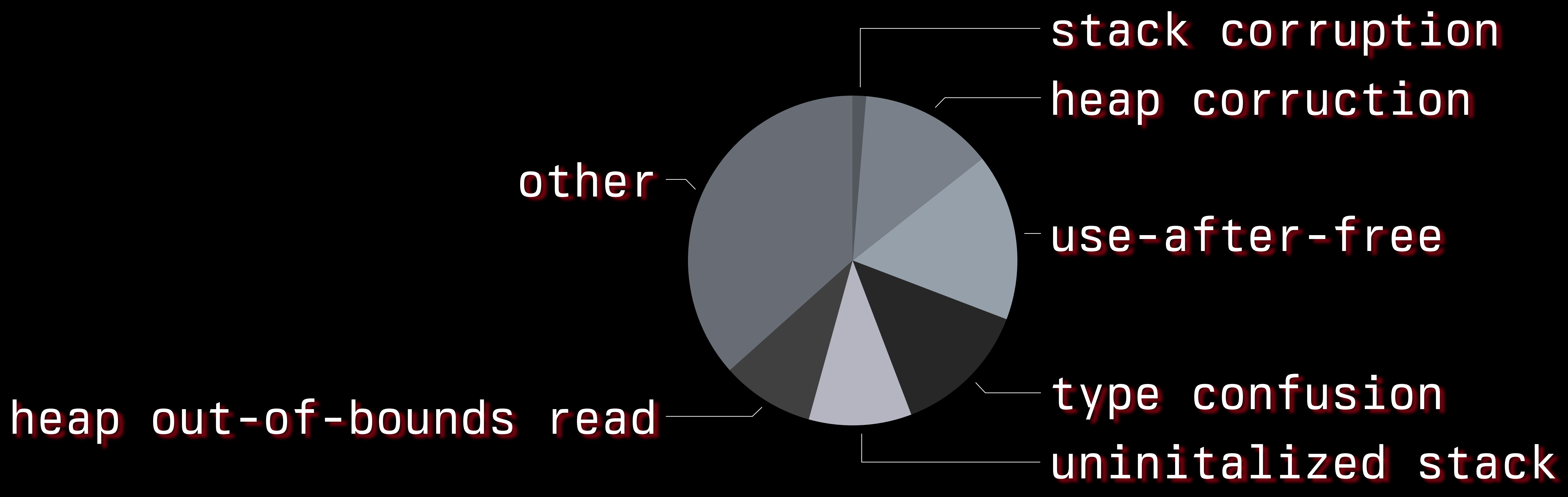
:-(

worst case: exploit

- leak secret
- use attacker-controlled value

8-||

root cause of exploits



this brings us to →
↘
↓

what is in

the bytes
before the types
↘
?

unveiling
uninitialized
uses

```c
int main() {
    char garbage[128];
    for (int i = 0; i != sizeof(garbage); ++i)
        printf("%02x", garbage[i]);
}
```

```c
int main() {
    char garbage[128];
    for (int i = 0; i != sizeof(garbage); ++i)
        printf("%02x", garbage[i]);
}
```

```
00000000000000000000000000000000
00000000000000000000000000000000
02000000000000007c00000079000000
ffffff80000000ffffffeb000000ffff
ffff110000000000002e65685f706f6f
6cffffff80ffffffacffffff811effff
ffdb7f0000002001000000000000ffffff
ff110000000000002e65685f706f6f6c
```

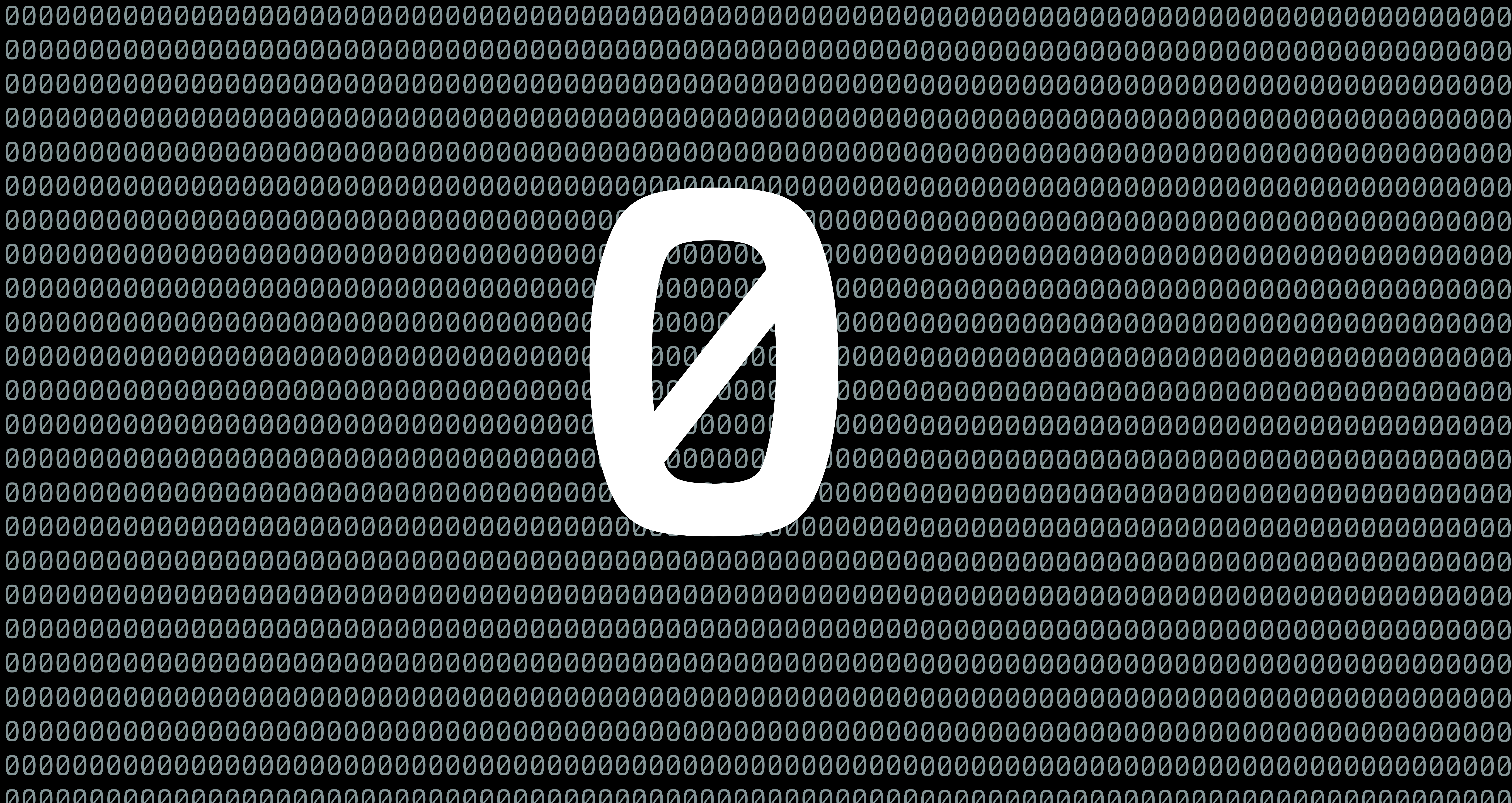
```c
int main() {
    char garbage[128];
    for (int i = 0; i ≠ sizeof(garbage); ++i)
        printf("%02x", garbage[i]);
}
```

```
ffffff80040000000000000001120000ff
ffffb0040000ffffff90000000000000
00120000000000000002000000000000
007c00000007900000ffffff80000000
ffffffeb000000ffffffff1100000000
00002e65685f706f6f6cffffff80ffff
ffac0138207f00000020010000000000
ffffffff110000000000002e65685f70
```

```
int main() {
    char garbage[128];
    for (int i = 0; i != sizeof(garbage); ++i)
        printf("%c", garbage[i]);
}
```

```
|y¿¿¿.eh_pool¿¿a]? ¿.eh_pool¿h]?bRJ]?glibcxx.¿H¿1¿
```

```
    struct Garbage { char bits[128]; };

    int main() {
        bool printed = false;
        for (int tries = 0; tries != 2048; ++tries) {
            auto *garbage = new Garbage;  <== create
            for (int i = 0; i != sizeof(Garbage); ++i) {
                if (garbage->bits[i])
                    printed = true;
                printf("%02x", garbage->bits[i]);
            }
            printf("\n");
leak ==     if (printed)
                break;
        }
    }
```



```cpp
struct Garbage {
    char bits[128];
    template <class Engine, class Distribution>
    void randomize(Engine &engine, Distribution &distribution) {
        for (int i = 0; i != sizeof(bits); ++i)
            bits[i] = distribution(engine);
    }
};
int main() {
    std::random_device device;
    std::mt19937 enigne(device());
    std::uniform_int_distribution<> distribution(0, 255);

    bool printed = false;
    for (int tries = 0; tries != 2048; ++tries) {
        auto *garbage = new Garbage;
        for (int i = 0; i != sizeof(Garbage); ++i) {
            if (garbage->bits[i])
                printed = true;
            printf("%02x", garbage->bits[i]);
        }
        printf("\n");
        if (printed)
            break;
        garbage->randomize(engine, distribution);
        delete garbage;
    }
}
```

lookitme!
proper C++
random

reuse

```
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
35ffffff89fffffff960050000000000
00000000000038ffffffe3096f5bffff
ff8d2a1affffffddffffffaf5e58ffff
ffd2ffffff95ffffffc85bfffffffe4c
ffffff90fffffff628ffffff99ffffff
a03dffffffebffffffbcffffffc9ffff
ffbdffffffafffffffe9fffffff2ffff
ffcfffffffd9ffffffce09ffffff92ff
ffff8d18ffffffa0ffffffb8ffffffbe
```

random!???

```
struct Garbage {
    char bits[128];
    void hackerize() {
        for (int i = 0; i ≠ sizeof(bits); ++i)
            bits[i] = 'A';  ⟸  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
    }
};
int main() {
    bool printed = false;
    for (int tries = 0; tries ≠ 2048; ++tries) {
        auto *garbage = new Garbage;
        for (int i = 0; i ≠ sizeof(Garbage); ++i) {
            if (garbage→bits[i])
                printed = true;
            printf("%02x", garbage→bits[i]);
        }
        printf("\n");
        if (printed)
            break;
        garbage→hackerize();
        delete garbage;
    }
}
```

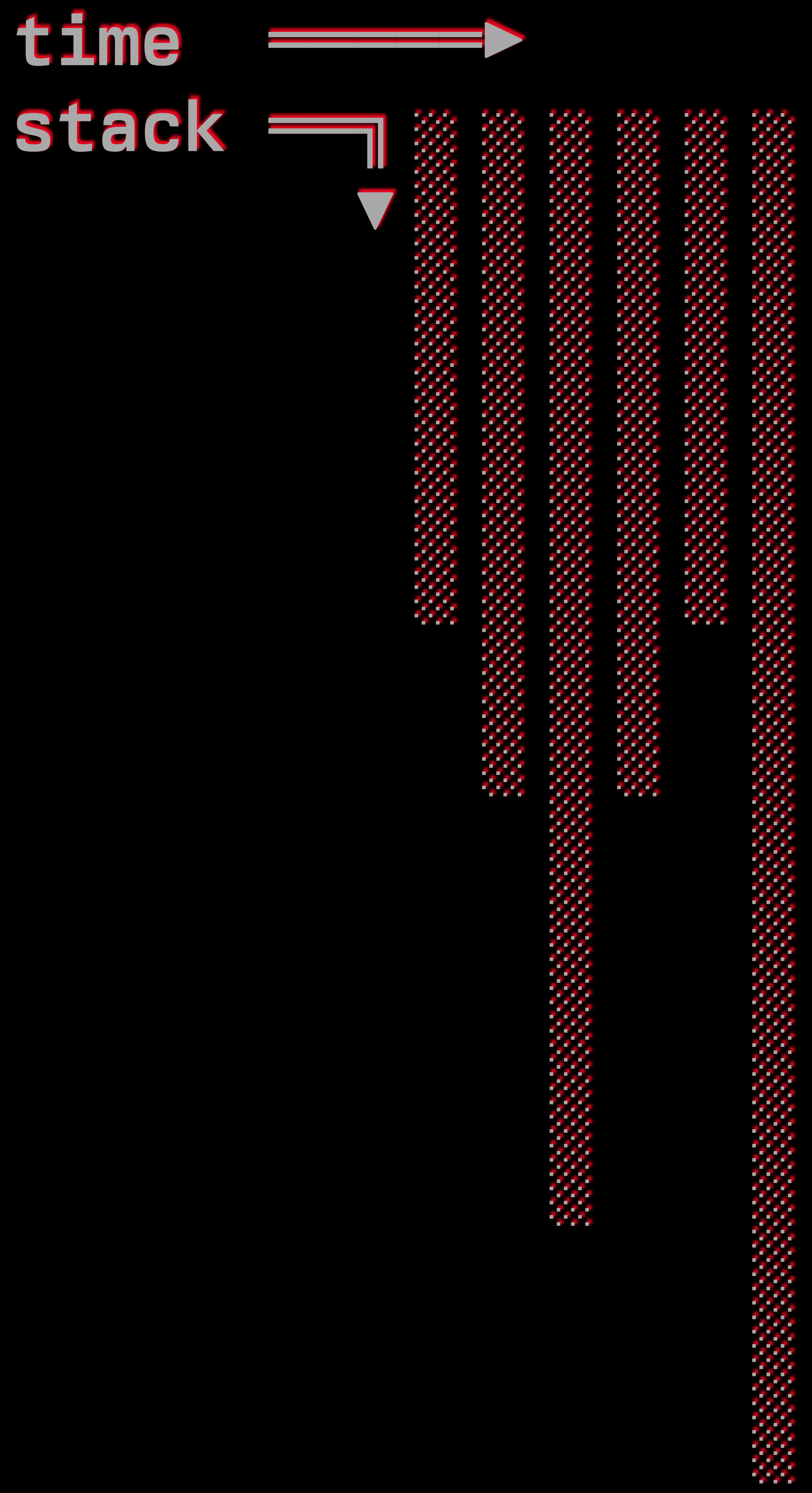
```
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
ffffff82ffffffd86a64050000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
```

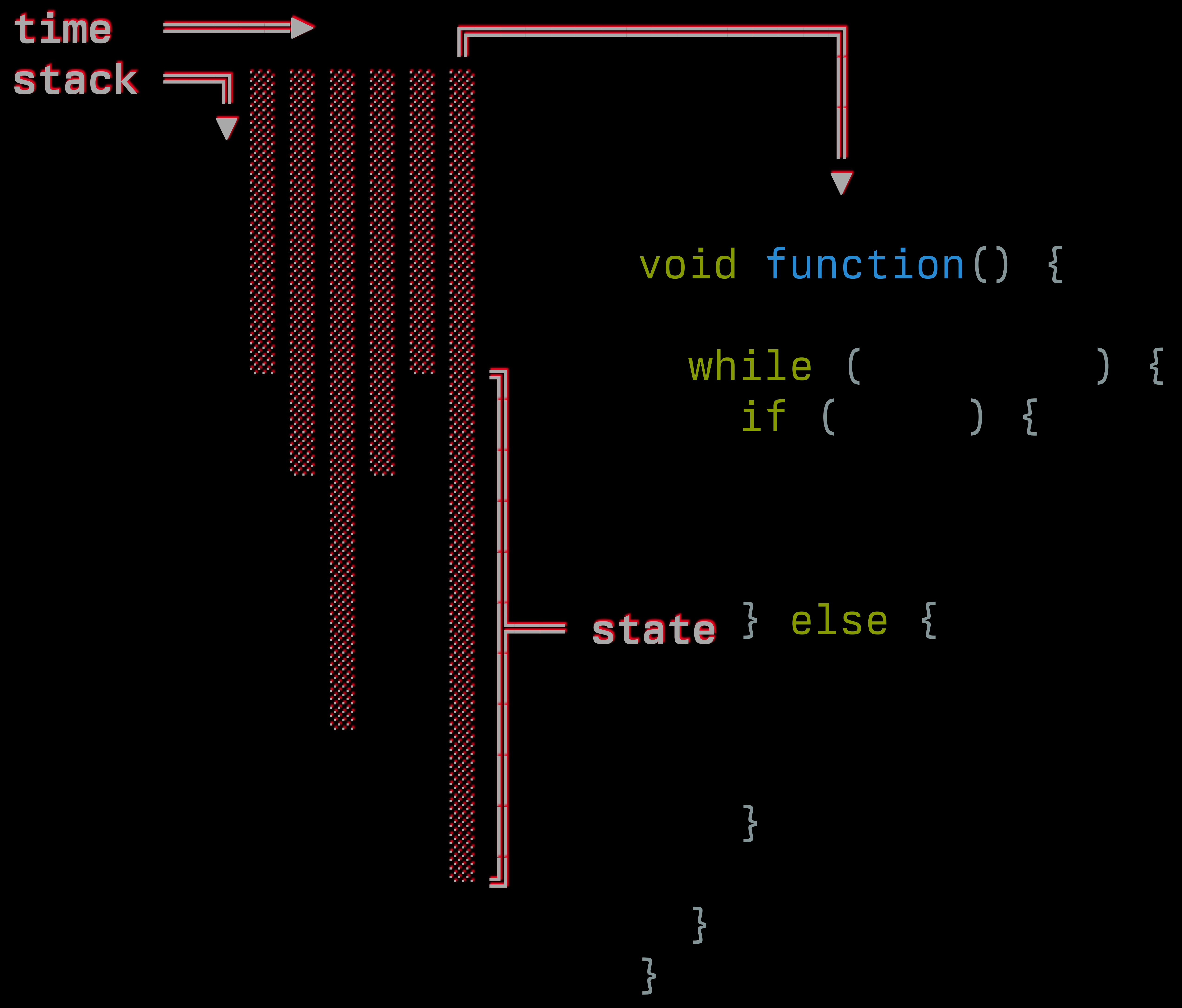
```
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
1dfffff9dffffffae5c0500000000000
00000000000000414141414141414141
41414141414141414141414141414141
41414141414141414141414141414141
41414141414141414141414141414141
41414141414141414141414141414141
41414141414141414141414141414141
41414141414141414141414141414141
```

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

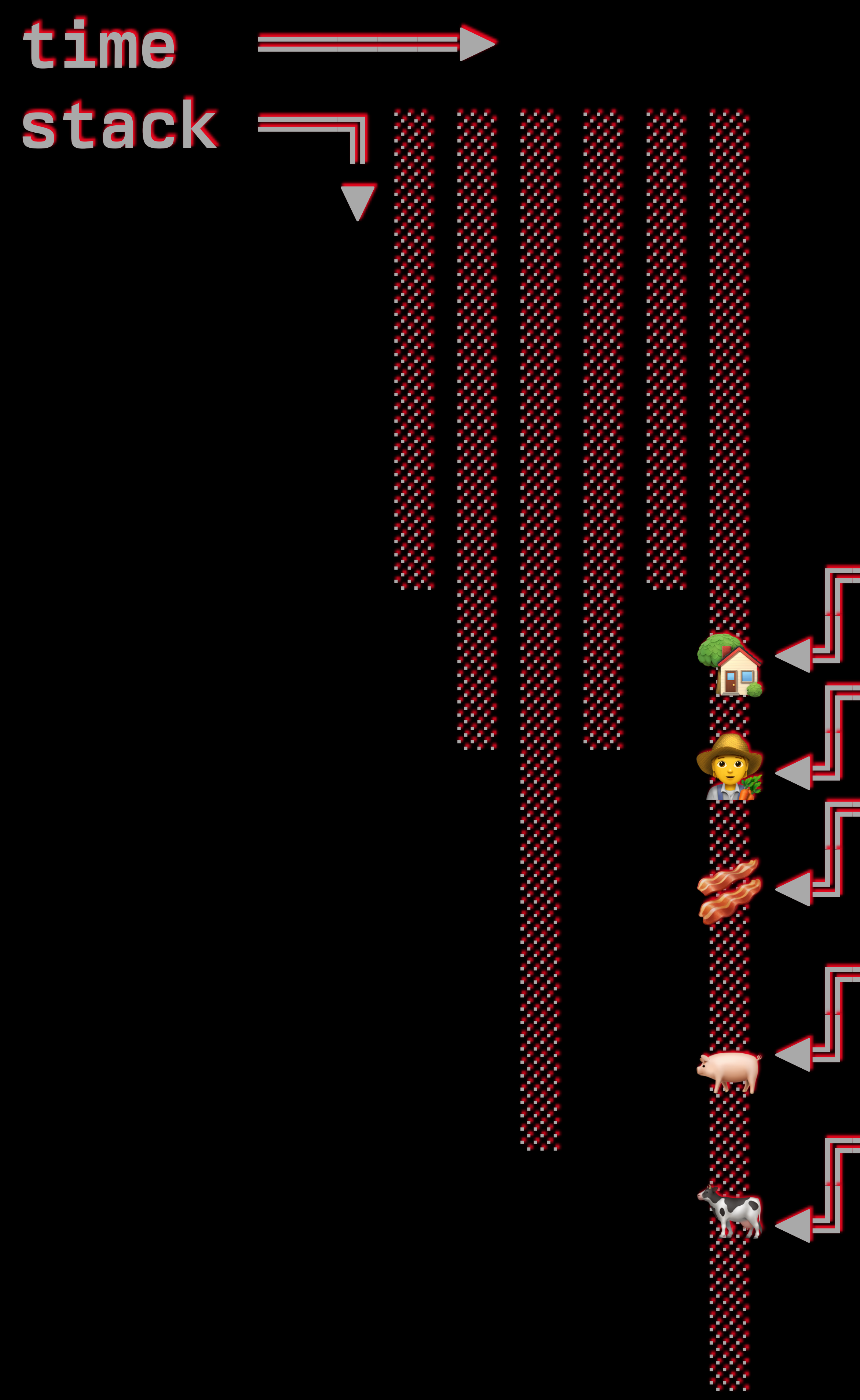
↵

```
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
1dffffff9dffffffae5c050000000000
00000000000041414141414141414141
41414141414141414141414141414141
41414141414141414141414141414141
41414141414141414141414141414141
41414141414141414141414141414141
41414141414141414141414141414141
41414141414141414141414141414141
```

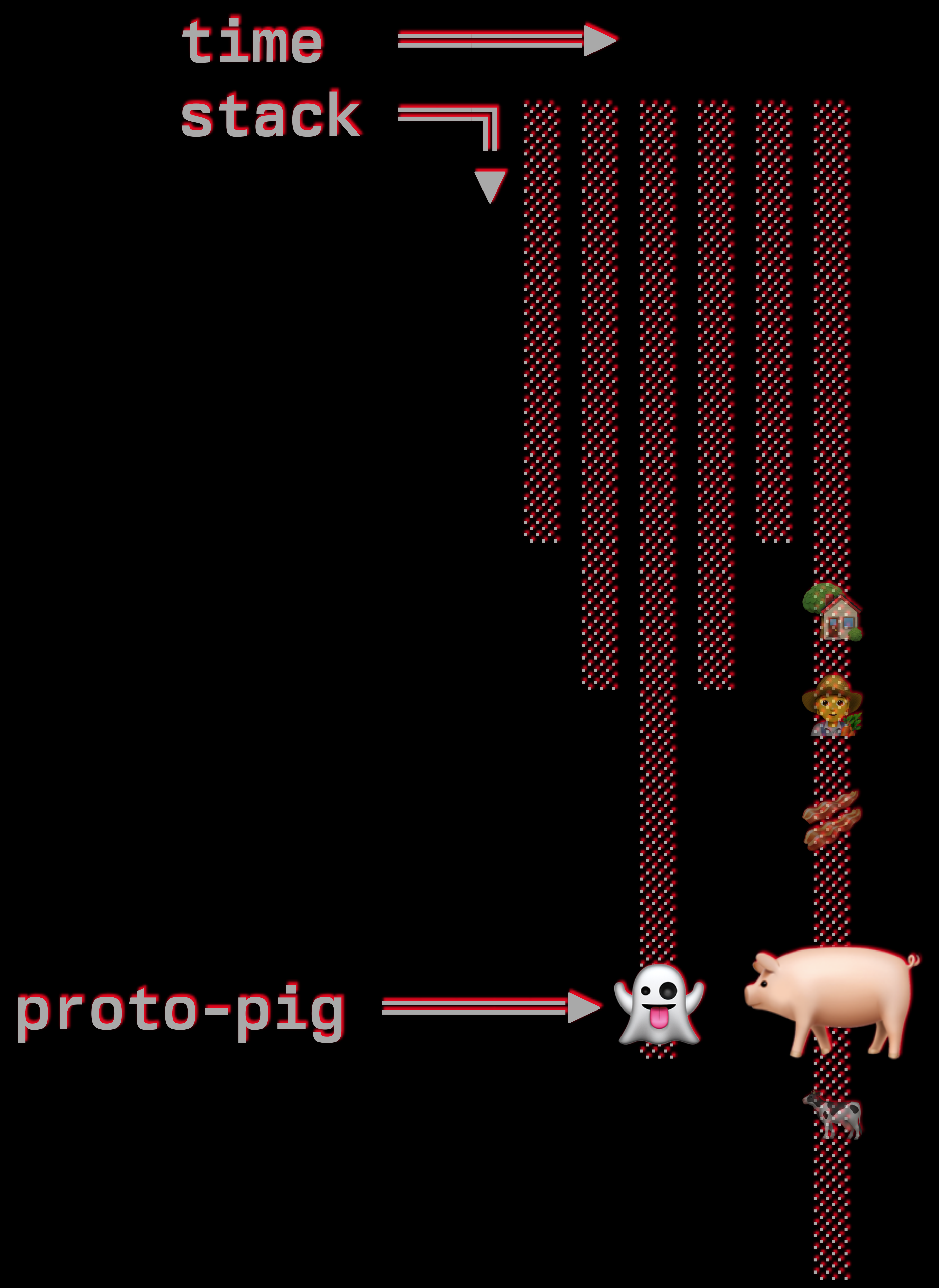

time
stack



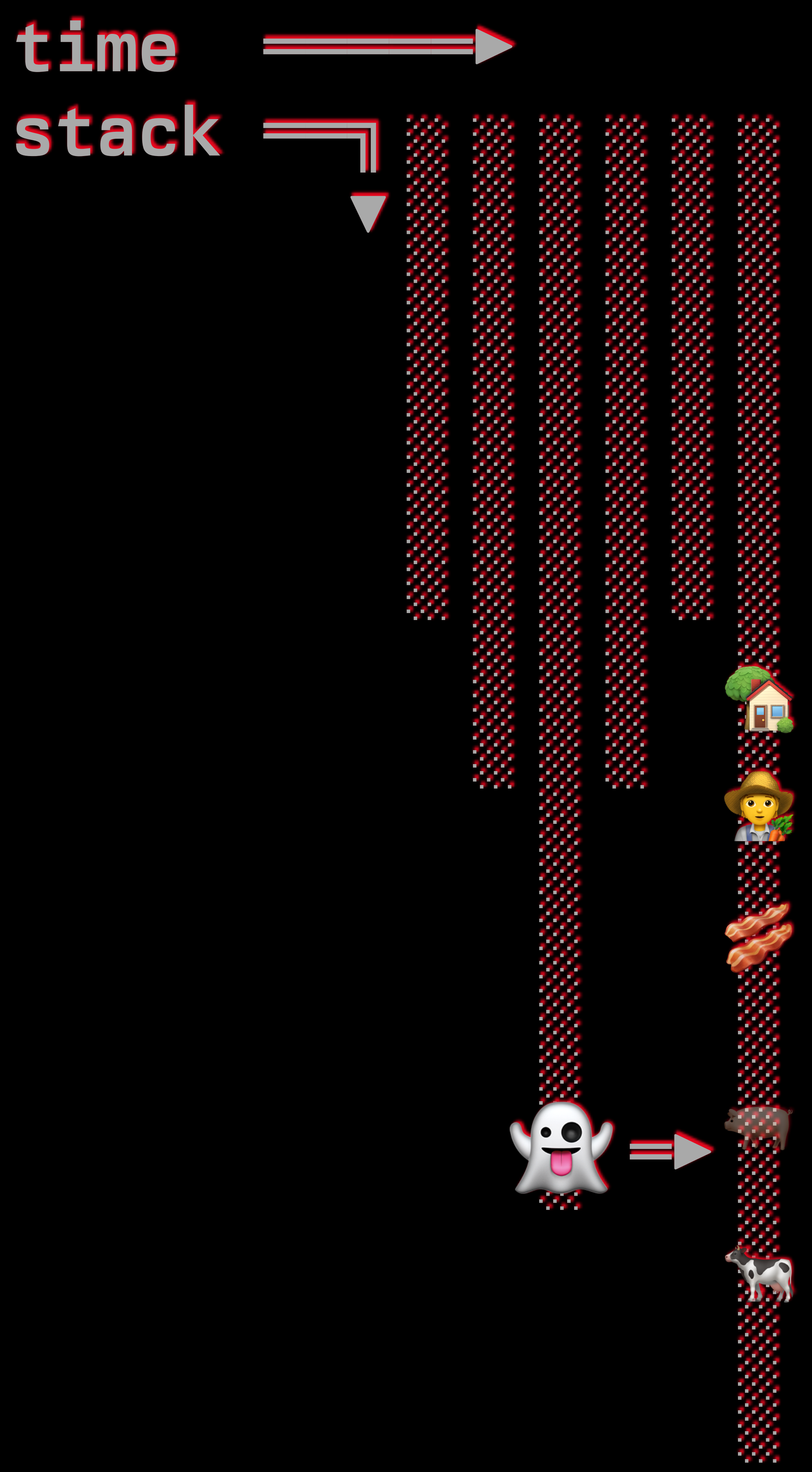
```
void function() {

  while (         ) {
    if (     ) {



    } else {



    }

  }
}
```



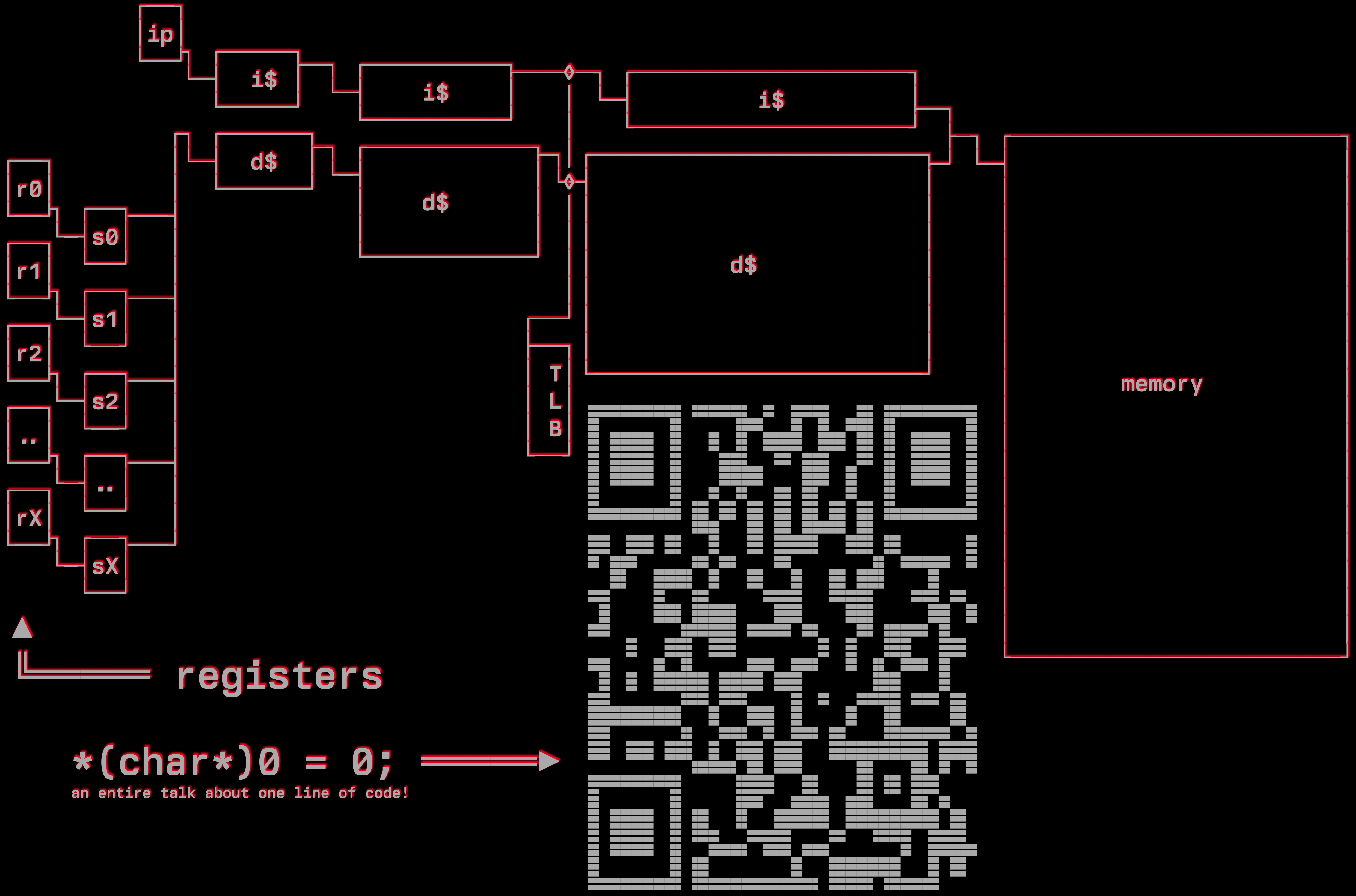
```
class Animal;
class Cow : public Animal { };
class Pig : public Animal { };

void function() {
  Farm farm;
  while (husbandry) {
    if (bacon) {
      Pig pig;
      // ... do piggy things
      farm.insert(pig);
    } else {
      Cow cow;
      // ... do cowy things
      farm.insert(cow);
    }
    // ...
  }
}
```



```
class Animal;
class Cow : public Animal { };
class Pig : public Animal { };

void function() {
  Farm farm;
  while (husbandry) {
    if (bacon) {
      Pig pig;
      //  ... do piggy things
      farm.insert(pig);
    } else {
      Cow cow;
      //  ... do cowy things
      farm.insert(cow);
    }
    //  ...
  }
}
```



```cpp
class Animal;
class Cow : public Animal { };
class Pig : public Animal { };

void function() {
  Farm farm;
  while (husbandry) {
    if (bacon) {
      Pig pig;
      // ... do piggy things
      farm.insert(pig);
    } else {
      Cow cow;
      // ... do cowy things
      farm.insert(cow);
    }
    // ...
  }
}
```



registers

`*(char*)0 = 0;`

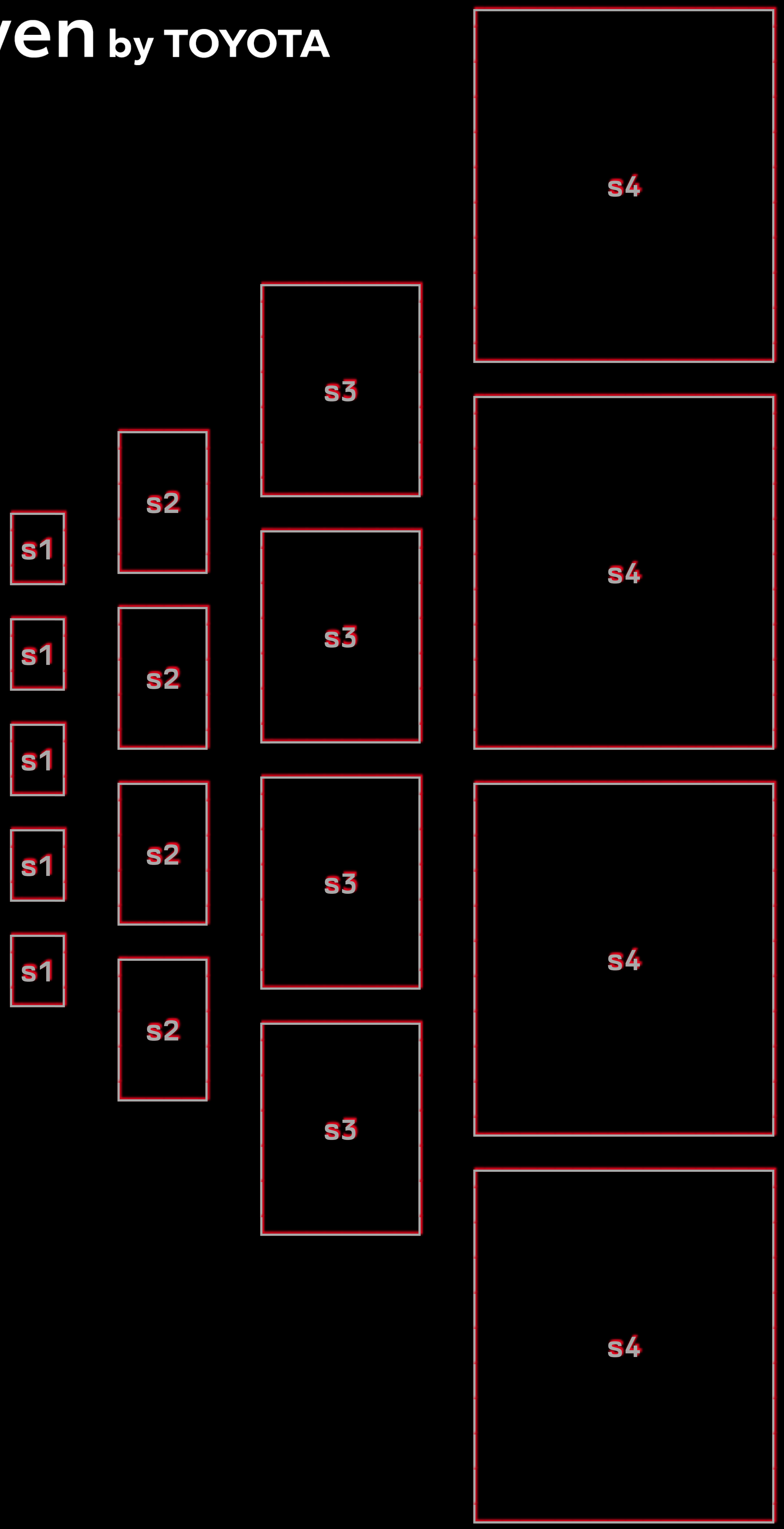
an entire talk about one line of code!

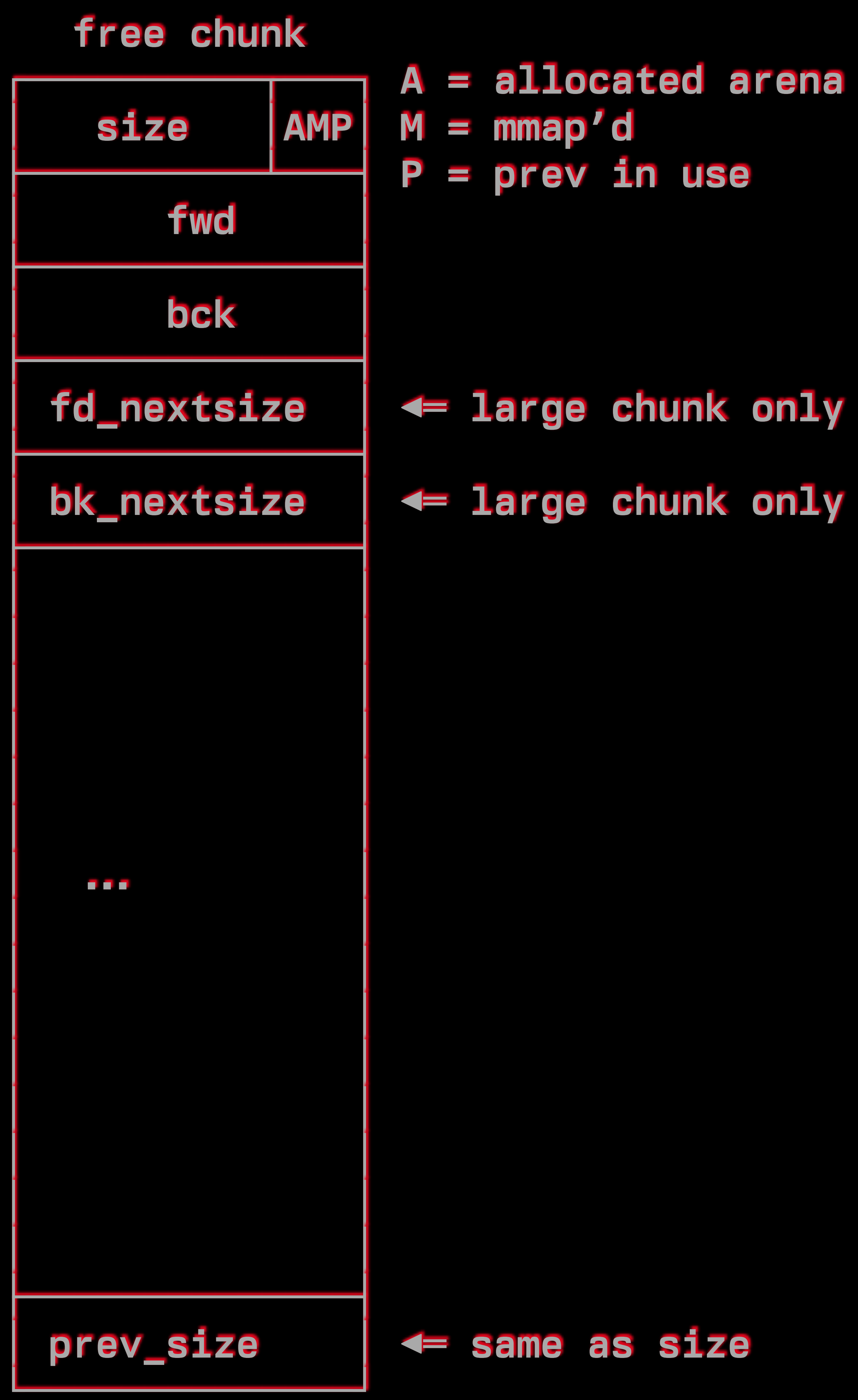
time →

r0	r0	r0	r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2	r2	👻	🐖
..	..	..	..	..	..	..	..	..
rX	rX	rX	rX	rX	rX	rX	rX	rX

```
+-----------+----------+---------------------------------------------+
| Register  | Special  | Role in the procedure call standard         |
+===========+==========+=============================================+
| SP        |          | Stack Pointer                               |
+-----------+----------+---------------------------------------------+
| r30       | LR       | Link Register                               |
+-----------+----------+---------------------------------------------+
| r29       | FP       | Frame Pointer                               |
+-----------+----------+---------------------------------------------+
| r19…r28   |          | Callee-saved registers                      |
+-----------+----------+---------------------------------------------+
| r18       |          | The Platform Register                       |
+-----------+----------+---------------------------------------------+
| r17       | IP1      | 2nd intra-procedure-call temp register      |
+-----------+----------+---------------------------------------------+
| r16       | IP0      | 1st intra-procedure-call scratch register   |
+-----------+----------+---------------------------------------------+
| r9…r15    |          | Temporary registers                         |
+-----------+----------+---------------------------------------------+
| r8        |          | Indirect result location register           |
+-----------+----------+---------------------------------------------+
| r0…r7     |          | Parameter/result registers                  |
+-----------+----------+---------------------------------------------+
int func(int);
```

HEAP


s4
s3
s2
s1
s4
s2
s1
s3
s1
s2
s3
s4
s1
s2
s3
s1
s2
s4



free chunk

size	AMP
fwd	
bck	
fd_nextsize	
bk_nextsize	
...	
prev_size	

A = allocated arena
M = mmap'd
P = prev in use

fd_nextsize ⇐ large chunk only

bk_nextsize ⇐ large chunk only

prev_size ⇐ same as size

Bins for sizes < 512 bytes contain chunks of all the same size, spaced 8 bytes apart. Larger bins are approximately logarithmically spaced:

```
64 bins of size       8
32 bins of size      64
16 bins of size     512
 8 bins of size    4096
 4 bins of size   32768
 2 bins of size  262144
 1 bin  of size what's left
```

```c
void __libc_free (void *mem) {
  mstate ar_ptr;
  mchunkptr p;                          /* chunk corresponding to mem */

  void (*hook) (void *, const void *) = atomic_forced_read (__free_hook);
  if (__builtin_expect (hook != NULL, 0)) {
      (*hook)(mem, RETURN_ADDRESS (0));
      return;
  }

  if (mem == 0)                         /* free(0) has no effect */
    return;

  p = mem2chunk (mem);
  if (chunk_is_mmapped (p)) {          /* release mmapped memory. */
      /* See if the dynamic brk/mmap threshold needs adjusting.
         Dumped fake mmapped chunks do not affect the threshold.  */
      if (!mp_.no_dyn_threshold
          && chunksize_nomask (p) > mp_.mmap_threshold
          && chunksize_nomask (p) <= DEFAULT_MMAP_THRESHOLD_MAX
     && !DUMPED_MAIN_ARENA_CHUNK (p)) {
          mp_.mmap_threshold = chunksize (p);
          mp_.trim_threshold = 2 * mp_.mmap_threshold;
          LIBC_PROBE (memory_mallopt_free_dyn_thresholds, 2,
                      mp_.mmap_threshold, mp_.trim_threshold);
      }
      munmap_chunk (p);
      return;
  }

  MAYBE_INIT_TCACHE ();
  ar_ptr = arena_for_chunk (p);
  _int_free (ar_ptr, p, 0);
}
```

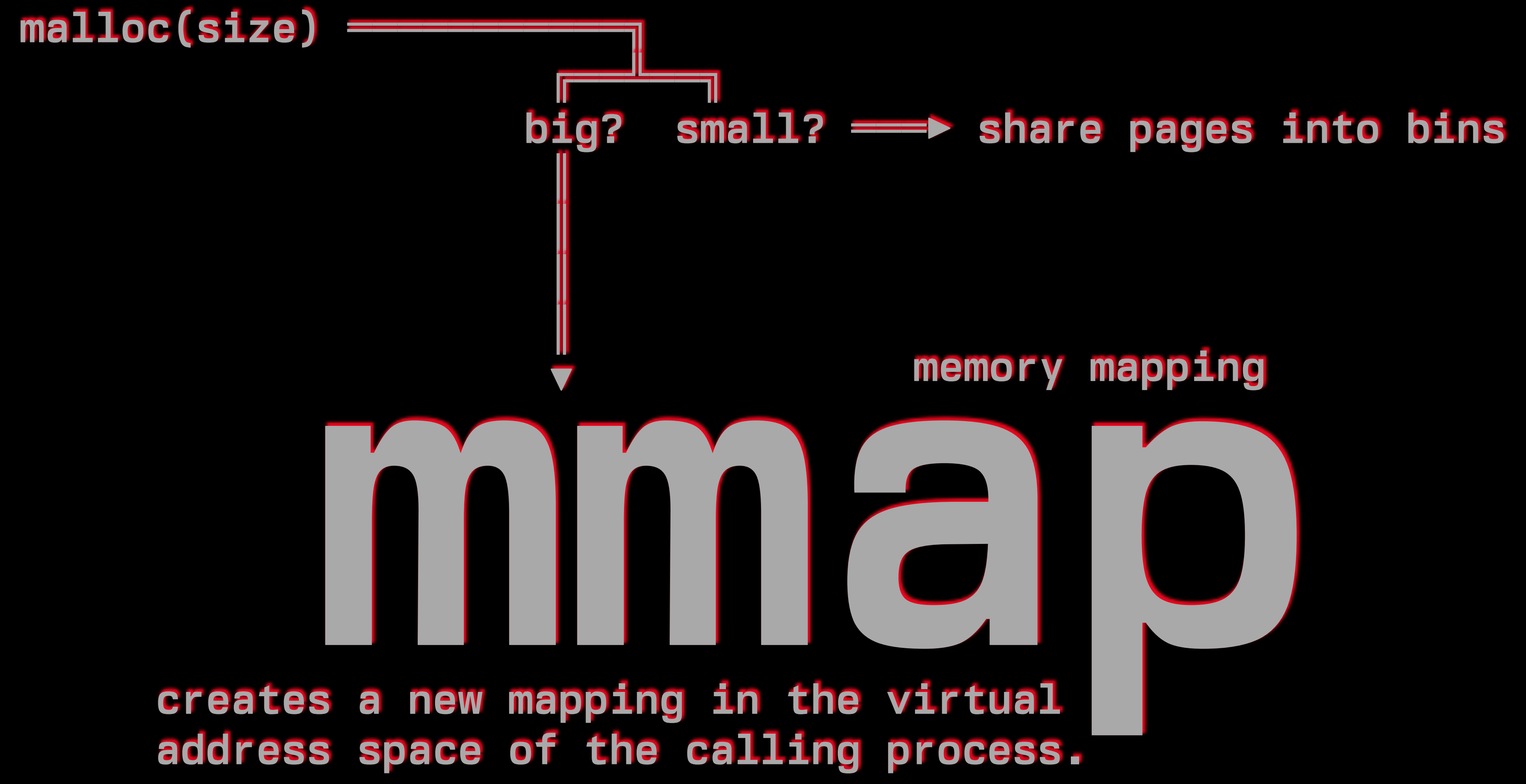
```
void __libc_free (void *mem) {
  mstate ar_ptr;
  mchunkptr p;                          /* chunk corresponding to mem */

  void (*hook) (void *, const void *) = atomic_forced_read (__free_hook);
  if (__builtin_expect (hook != NULL, 0)) {
      (*hook)(mem, RETURN_ADDRESS (0));
      return;
  }

  if (mem == 0)                         /* free(0) has no effect */
    return;

  p = mem2chunk (mem);
  if (chunk_is_mmapped (p)) {          /* release mmapped memory. */
      /* See if the dynamic brk/mmap threshold needs adjusting.
         Dumped fake mmapped chunks do not affect the threshold.  */
      if (!mp_.no_dyn_threshold
          && chunksize_nomask (p) > mp_.mmap_threshold
          && chunksize_nomask (p) <= DEFAULT_MMAP_THRESHOLD_MAX
     && !DUMPED_MAIN_ARENA_CHUNK (p)) {
          mp_.mmap_threshold = chunksize (p);
          mp_.trim_threshold = 2 * mp_.mmap_threshold;
          LIBC_PROBE (memory_mallopt_free_dyn_thresholds, 2,
                      mp_.mmap_threshold, mp_.trim_threshold);
      }
      munmap_chunk (p);
      return;
  }

  MAYBE_INIT_TCACHE ();
  ar_ptr = arena_for_chunk (p);
  _int_free (ar_ptr, p, 0);
}
```

```
void __libc_free (void *mem) {
  mstate ar_ptr;
  mchunkptr p;                          /* chunk corresponding to mem */

  void (*hook) (void *, const void *) = atomic_forced_read (__free_hook);
  if (__builtin_expect (hook != NULL, 0)) {
      (*hook)(mem, RETURN_ADDRESS (0));
      return;
  }

  if (mem == 0)                         /* free(0) has no effect */
    return;

  p = mem2chunk (mem);
  if (chunk_is_mmapped (p)) {          /* release mmapped memory. */
      /* See if the dynamic brk/mmap threshold needs adjusting.
         Dumped fake mmapped chunks do not affect the threshold.  */
      if (!mp_.no_dyn_threshold
          && chunksize_nomask (p) > mp_.mmap_threshold
          && chunksize_nomask (p) <= DEFAULT_MMAP_THRESHOLD_MAX
     && !DUMPED_MAIN_ARENA_CHUNK (p)) {
          mp_.mmap_threshold = chunksize (p);
          mp_.trim_threshold = 2 * mp_.mmap_threshold;
          LIBC_PROBE (memory_mallopt_free_dyn_thresholds, 2,
                      mp_.mmap_threshold, mp_.trim_threshold);
      }
      munmap_chunk (p);
      return;
  }

  MAYBE_INIT_TCACHE ();
  ar_ptr = arena_for_chunk (p);
  _int_free (ar_ptr, p, 0);
}
```

memory mapping

mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm **mmap**

creates a new mapping in the virtual
address space of the calling process.



malloc(size)

big?

small?

share pages into bins

memory mapping

mmap

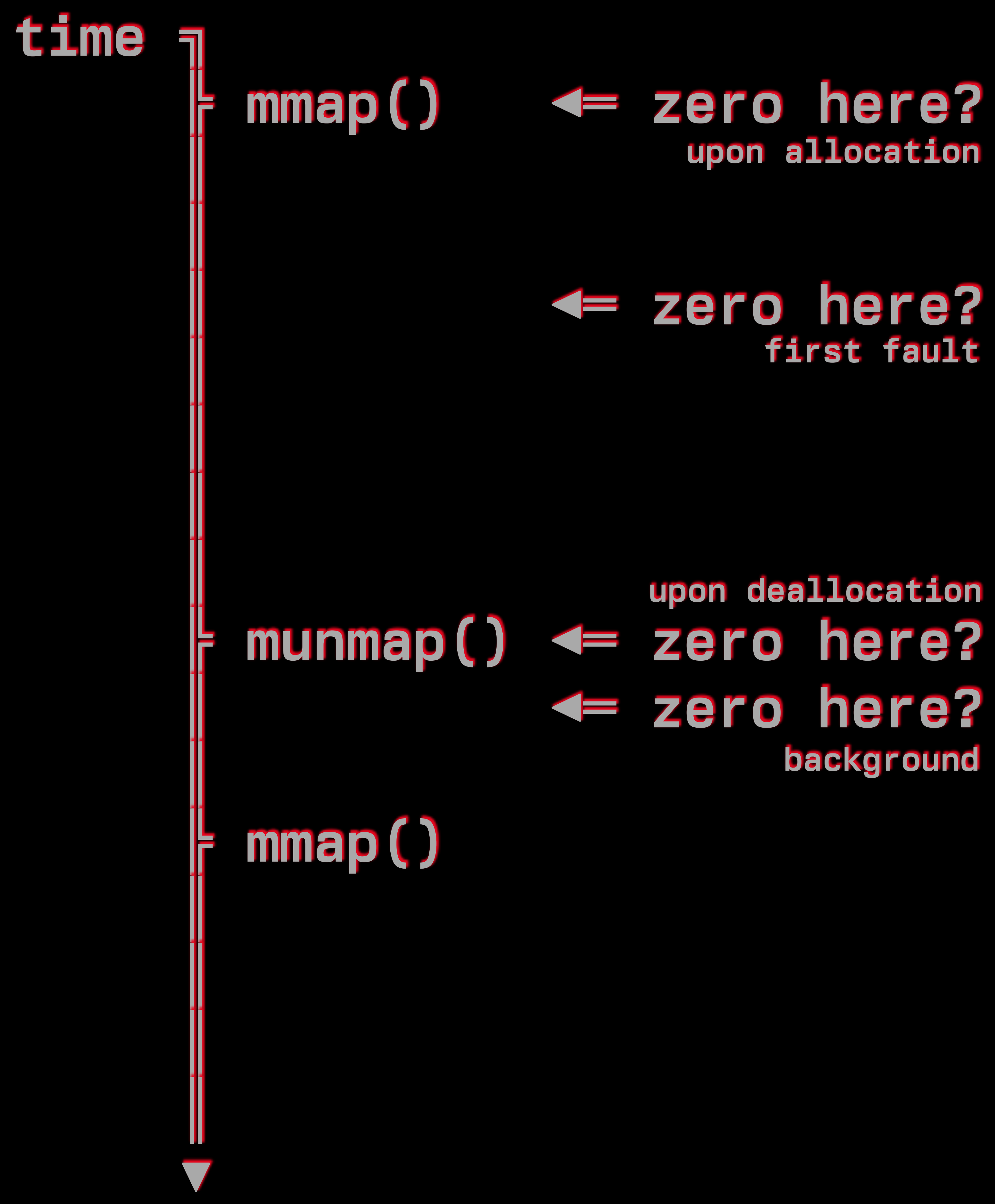
creates a new mapping in the virtual
address space of the calling process.

mmap

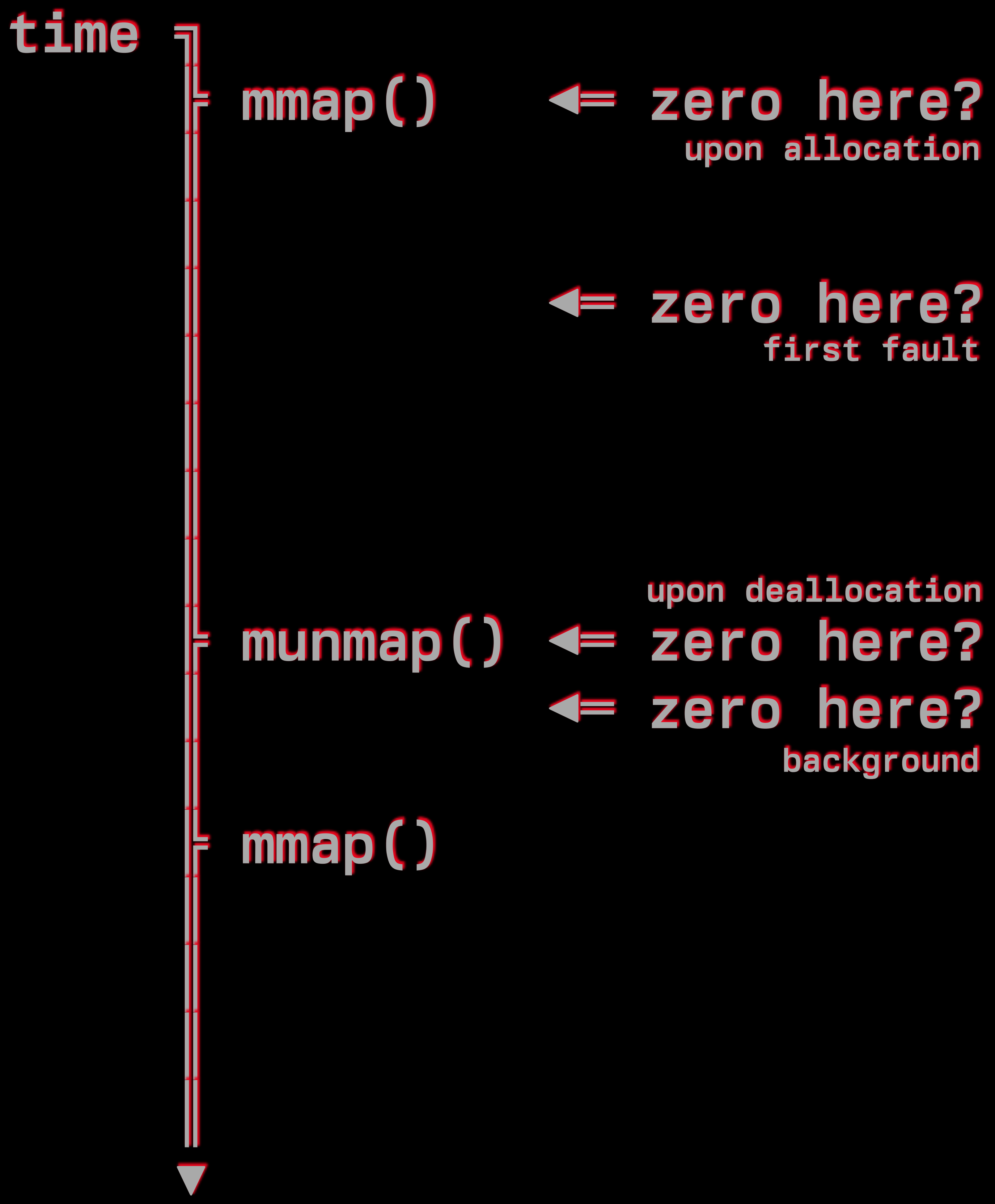
i zero memory

```
void *mmap(
    void addr[.length],
    size_t length,
    int prot,
    int flags,
    int fd,
    off_t offset);
```

```c
int page_size = sysconf(_SC_PAGESIZE);
char *memory = (char*)mmap(
    NULL,
    page_size * 4,
    PROT_READ | PROT_WRITE,
    MAP_PRIVATE | MAP_ANONYMOUS,
    -1,
    0);
```



time
mmap()
⇐ zero here?
upon allocation
⇐ zero here?
first fault
upon deallocation
munmap()
⇐ zero here?
⇐ zero here?
background
mmap()



time

mmap() ⇐ zero here?
upon allocation

⇐ zero here?
first fault

upon deallocation
munmap() ⇐ zero here?

⇐ zero here?
background

mmap()

definitely NOT rickroll

```
*(char*)0 = 0;
```

an entire talk about one line of code!

```c
void *
__mmap (void *addr, size_t len, int prot,
        int flags, int fd, off_t offset)
{
  MMAP_CHECK_PAGE_UNIT ();

  if (offset & MMAP_OFF_LOW_MASK)
    return (void *)
        INLINE_SYSCALL_ERROR_RETURN_VALUE (EINVAL);

  return (void *) INLINE_SYSCALL_CALL (
          mmap2, addr, len, prot, flags, fd,
          offset / (uint32_t) MMAP2_PAGE_UNIT);
}
```

```
void *
__mmap (void *addr, size_t len, int prot,
        int flags, int fd, off_t offset)
{
  MMAP_CHECK_PAGE_UNIT ();

  if (offset & MMAP_OFF_LOW_MASK)
    return (void *)
        INLINE_SYSCALL_ERROR_RETURN_VALUE (EINVAL);

  return (void *) INLINE_SYSCALL_CALL (
          mmap2, addr, len, prot, flags, fd,
          offset / (uint32_t) MMAP2_PAGE_UNIT);
}
```

```
SYSCALL_DEFINE6(mmap, unsigned long,
    addr, unsigned long, len,
    unsigned long, prot, unsigned long, flags,
    unsigned long, fd, unsigned long, off)
{
  if (offset_in_page(off) ≠ 0)
    return -EINVAL;

  return ksys_mmap_pgoff(addr, len, prot,
      flags, fd, off >> PAGE_SHIFT);
}
```

```c
unsigned long ksys_mmap_pgoff(unsigned long addr, unsigned long len,
                 unsigned long prot, unsigned long flags,
                 unsigned long fd, unsigned long pgoff)
{
    struct file *file = NULL;
    unsigned long retval;

    if (!(flags & MAP_ANONYMOUS)) {
        audit_mmap_fd(fd, flags);
        file = fget(fd);
        if (!file)
            return -EBADF;
        if (is_file_hugepages(file)) {
            len = ALIGN(len, huge_page_size(hstate_file(file)));
        } else if (unlikely(flags & MAP_HUGETLB)) {
            retval = -EINVAL;
            goto out_fput;
        }
    } else if (flags & MAP_HUGETLB) {
        struct hstate *hs;

        hs = hstate_sizelog((flags >> MAP_HUGE_SHIFT) & MAP_HUGE_MASK);
        if (!hs)
            return -EINVAL;

        len = ALIGN(len, huge_page_size(hs));
        /*
         * VM_NORESERVE is used because the reservations will be
         * taken when vm_ops->mmap() is called
         */
        file = hugetlb_file_setup(HUGETLB_ANON_FILE, len,
               VM_NORESERVE,
               HUGETLB_ANONHUGE_INODE,
               (flags >> MAP_HUGE_SHIFT) & MAP_HUGE_MASK);
        if (IS_ERR(file))
            return PTR_ERR(file);
    }

    retval = vm_mmap_pgoff(file, addr, len, prot, flags, pgoff);
out_fput:
    if (file)
        fput(file);
    return retval;
}
```

```
unsigned long ksys_mmap_pgoff(unsigned long addr, unsigned long len,
                 unsigned long prot, unsigned long flags,
                 unsigned long fd, unsigned long pgoff)
{
    struct file *file = NULL;
    unsigned long retval;

    if (!(flags & MAP_ANONYMOUS)) {
        audit_mmap_fd(fd, flags);
        file = fget(fd);
        if (!file)
            return -EBADF;
        if (is_file_hugepages(file)) {
            len = ALIGN(len, huge_page_size(hstate_file(file)));
        } else if (unlikely(flags & MAP_HUGETLB)) {
            retval = -EINVAL;
            goto out_fput;
        }
    } else if (flags & MAP_HUGETLB) {
        struct hstate *hs;

        hs = hstate_sizelog((flags >> MAP_HUGE_SHIFT) & MAP_HUGE_MASK);
        if (!hs)
            return -EINVAL;

        len = ALIGN(len, huge_page_size(hs));
        /*
         * VM_NORESERVE is used because the reservations will be
         * taken when vm_ops->mmap() is called
         */
        file = hugetlb_file_setup(HUGETLB_ANON_FILE, len,
                VM_NORESERVE,
                HUGETLB_ANONHUGE_INODE,
                (flags >> MAP_HUGE_SHIFT) & MAP_HUGE_MASK);
        if (IS_ERR(file))
            return PTR_ERR(file);
    }

    retval = vm_mmap_pgoff(file, addr, len, prot, flags, pgoff);
out_fput:
    if (file)
        fput(file);
    return retval;
}
```

```c
unsigned long vm_mmap_pgoff(struct file *file,
  unsigned long addr,
  unsigned long len, unsigned long prot,
  unsigned long flag, unsigned long pgoff)
{
  unsigned long ret;
  struct mm_struct *mm = current->mm;
  unsigned long populate;
  LIST_HEAD(uf);

  ret = security_mmap_file(file, prot, flag);
  if (!ret) {
    if (mmap_write_lock_killable(mm))
      return -EINTR;
    ret = do_mmap(file, addr, len, prot,
            flag, 0, pgoff, &populate, &uf);
    mmap_write_unlock(mm);
    userfaultfd_unmap_complete(mm, &uf);
    if (populate)
      mm_populate(ret, populate);
  }
  return ret;
}
```

```
unsigned long vm_mmap_pgoff(struct file *file,
  unsigned long addr,
  unsigned long len, unsigned long prot,
  unsigned long flag, unsigned long pgoff)
{
  unsigned long ret;
  struct mm_struct *mm = current->mm;
  unsigned long populate;
  LIST_HEAD(uf);

  ret = security_mmap_file(file, prot, flag);
  if (!ret) {
    if (mmap_write_lock_killable(mm))
      return -EINTR;
    ret = do_mmap(file, addr, len, prot,
            flag, 0, pgoff, &populate, &uf);
    mmap_write_unlock(mm);
    userfaultfd_unmap_complete(mm, &uf);
    if (populate)
      mm_populate(ret, populate);
  }
  return ret;
}
```

```
unsigned long do_mmap(struct file *file, unsigned long addr,
                unsigned long len, unsigned long prot,
                unsigned long flags, vm_flags_t vm_flags,
                unsigned long pgoff, unsigned long *populate,
                struct list_head *uf) {
      struct mm_struct *mm = current->mm;
      int pkey = 0;

      *populate = 0;

      if (!len)
            return -EINVAL;

      /*
       * Does the application expect PROT_READ to imply PROT_EXEC?
       *
       * (the exception is when the underlying filesystem is noexec
       *  mounted, in which case we don't add PROT_EXEC.)
       */
      if ((prot & PROT_READ) && (current->personality & READ_IMPLIES_EXEC))
            if (!(file && path_noexec(&file->f_path)))
                  prot |= PROT_EXEC;

      /* force arch specific MAP_FIXED handling in get_unmapped_area */
      if (flags & MAP_FIXED_NOREPLACE)
            flags |= MAP_FIXED;

      if (!(flags & MAP_FIXED))
            addr = round_hint_to_min(addr);

      /* Careful about overflows.. */
      len = PAGE_ALIGN(len);
      if (!len)
            return -ENOMEM;

      /* offset overflow? */
      if ((pgoff + (len >> PAGE_SHIFT)) < pgoff)
            return -EOVERFLOW;

      /* Too many mappings? */
      if (mm->map_count > sysctl_max_map_count)
            return -ENOMEM;

      /* Obtain the address to map to. we verify (or select) it and ensure
       * that it represents a valid section of the address space.
       */
      addr = get_unmapped_area(file, addr, len, pgoff, flags);
      if (IS_ERR_VALUE(addr))
            return addr;

      if (flags & MAP_FIXED_NOREPLACE) {
            if (find_vma_intersection(mm, addr, addr + len))
                  return -EEXIST;
      }

      if (prot == PROT_EXEC) {
            pkey = execute_only_pkey(mm);
            if (pkey < 0)
                  pkey = 0;
      }
```

```
      /* Do simple checking here so the lower-level routines won't have
       * to. we assume access permissions have been handled by the open
       * of the memory object, so we don't do any here.
       */
      vm_flags |= calc_vm_prot_bits(prot, pkey) | calc_vm_flag_bits(flags) |
                  mm->def_flags | VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC;

      if (flags & MAP_LOCKED)
            if (!can_do_mlock())
                  return -EPERM;

      if (!mlock_future_ok(mm, vm_flags, len))
            return -EAGAIN;

      if (file) {
            struct inode *inode = file_inode(file);
            unsigned long flags_mask;

            if (!file_mmap_ok(file, inode, pgoff, len))
                  return -EOVERFLOW;

            flags_mask = LEGACY_MAP_MASK | file->f_op->mmap_supported_flags;

            switch (flags & MAP_TYPE) {
            case MAP_SHARED:
                  /*
                   * Force use of MAP_SHARED_VALIDATE with non-legacy
                   * flags. E.g. MAP_SYNC is dangerous to use with
                   * MAP_SHARED as you don't know which consistency model
                   * you will get. We silently ignore unsupported flags
                   * with MAP_SHARED to preserve backward compatibility.
                   */
                  flags &= LEGACY_MAP_MASK;
                  fallthrough;
            case MAP_SHARED_VALIDATE:
                  if (flags & ~flags_mask)
                        return -EOPNOTSUPP;
                  if (prot & PROT_WRITE) {
                        if (!(file->f_mode & FMODE_WRITE))
                              return -EACCES;
                        if (IS_SWAPFILE(file->f_mapping->host))
                              return -ETXTBSY;
                  }

                  /*
                   * Make sure we don't allow writing to an append-only
                   * file..
                   */
                  if (IS_APPEND(inode) && (file->f_mode & FMODE_WRITE))
                        return -EACCES;

                  vm_flags |= VM_SHARED | VM_MAYSHARE;
                  if (!(file->f_mode & FMODE_WRITE))
                        vm_flags &= ~(VM_MAYWRITE | VM_SHARED);
                  fallthrough;
            case MAP_PRIVATE:
                  if (!(file->f_mode & FMODE_READ))
                        return -EACCES;
```

```
                  if (path_noexec(&file->f_path)) {
                        if (vm_flags & VM_EXEC)
                              return -EPERM;
                        vm_flags &= ~VM_MAYEXEC;
                  }

                  if (!file->f_op->mmap)
                        return -ENODEV;
                  if (vm_flags & (VM_GROWSDOWN|VM_GROWSUP))
                        return -EINVAL;
                  break;

            default:
                  return -EINVAL;
            }
      } else {
            switch (flags & MAP_TYPE) {
            case MAP_SHARED:
                  if (vm_flags & (VM_GROWSDOWN|VM_GROWSUP))
                        return -EINVAL;
                  /*
                   * Ignore pgoff.
                   */
                  pgoff = 0;
                  vm_flags |= VM_SHARED | VM_MAYSHARE;
                  break;
            case MAP_PRIVATE:
                  /*
                   * Set pgoff according to addr for anon_vma.
                   */
                  pgoff = addr >> PAGE_SHIFT;
                  break;
            default:
                  return -EINVAL;
            }
      }

      /*
       * Set 'VM_NORESERVE' if we should not account for the
       * memory use of this mapping.
       */
      if (flags & MAP_NORESERVE) {
            /* We honor MAP_NORESERVE if allowed to overcommit */
            if (sysctl_overcommit_memory != OVERCOMMIT_NEVER)
                  vm_flags |= VM_NORESERVE;

            /* hugetlb applies strict overcommit unless MAP_NORESERVE */
            if (file && is_file_hugepages(file))
                  vm_flags |= VM_NORESERVE;
      }

      addr = mmap_region(file, addr, len, vm_flags, pgoff, uf);
      if (!IS_ERR_VALUE(addr) &&
          ((vm_flags & VM_LOCKED) ||
           (flags & (MAP_POPULATE | MAP_NONBLOCK)) == MAP_POPULATE))
            *populate = len;

      return addr;
}
```

```
addr = mmap_region(file, addr, len, vm_flags, pgoff, uf);
```

```
unsigned long do_mmap(struct file *file, unsigned long addr,
            unsigned long len, unsigned long prot,
            unsigned long flags, vm_flags_t vm_flags,
            unsigned long pgoff, unsigned long *populate,
            struct list_head *uf) {
    struct mm_struct *mm = current->mm;
    int pkey = 0;

    *populate = 0;

    if (!len)
        return -EINVAL;

    /*
     * Does the application expect PROT_READ to imply PROT_EXEC?
     *
     * (the exception is when the underlying filesystem is noexec
     *  mounted, in which case we don't add PROT_EXEC.)
     */
    if ((prot & PROT_READ) && (current->personality & READ_IMPLIES_EXEC))
        if (!(file && path_noexec(&file->f_path)))
            prot |= PROT_EXEC;

    /* force arch specific MAP_FIXED handling in get_unmapped_area */
    if (flags & MAP_FIXED_NOREPLACE)
        flags |= MAP_FIXED;

    if (!(flags & MAP_FIXED))
        addr = round_hint_to_min(addr);

    /* Careful about overflows.. */
    len = PAGE_ALIGN(len);
    if (!len)
        return -ENOMEM;

    /* offset overflow? */
    if ((pgoff + (len >> PAGE_SHIFT)) < pgoff)
        return -EOVERFLOW;

    /* Too many mappings? */
    if (mm->map_count > sysctl_max_map_count)
        return -ENOMEM;

    /* Obtain the address to map to. we verify (or select) it and ensure
     * that it represents a valid section of the address space.
     */
    addr = get_unmapped_area(file, addr, len, pgoff, flags);
    if (IS_ERR_VALUE(addr))
        return addr;

    if (flags & MAP_FIXED_NOREPLACE) {
        if (find_vma_intersection(mm, addr, addr + len))
            return -EEXIST;
    }

    if (prot == PROT_EXEC) {
        pkey = execute_only_pkey(mm);
        if (pkey < 0)
            pkey = 0;
    }

    /* Do simple checking here so the lower-level routines won't have
     * to. we assume access permissions have been handled by the open
     * of the memory object, so we don't do any here.
     */
    vm_flags |= calc_vm_prot_bits(prot, pkey) | calc_vm_flag_bits(flags) |
            mm->def_flags | VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC;

    if (flags & MAP_LOCKED)
        if (!can_do_mlock())
            return -EPERM;

    if (!mlock_future_ok(mm, vm_flags, len))
        return -EAGAIN;

    if (file) {
        struct inode *inode = file_inode(file);
        unsigned long flags_mask;

        if (!file_mmap_ok(file, inode, pgoff, len))
            return -EOVERFLOW;

        flags_mask = LEGACY_MAP_MASK | file->f_op->mmap_supported_flags;

        switch (flags & MAP_TYPE) {
        case MAP_SHARED:
            /*
             * Force use of MAP_SHARED_VALIDATE with non-legacy
             * flags. E.g. MAP_SYNC is dangerous to use with
             * MAP_SHARED as you don't know which consistency model
             * you will get. We silently ignore unsupported flags
             * with MAP_SHARED to preserve backward compatibility.
             */
            flags &= LEGACY_MAP_MASK;
            fallthrough;
        case MAP_SHARED_VALIDATE:
            if (flags & ~flags_mask)
                return -EOPNOTSUPP;
            if (prot & PROT_WRITE) {
                if (!(file->f_mode & FMODE_WRITE))
                    return -EACCES;
                if (IS_SWAPFILE(file->f_mapping->host))
                    return -ETXTBSY;
            }

            /*
             * Make sure we don't allow writing to an append-only
             * file..
             */
            if (IS_APPEND(inode) && (file->f_mode & FMODE_WRITE))
                return -EACCES;

            vm_flags |= VM_SHARED | VM_MAYSHARE;
            if (!(file->f_mode & FMODE_WRITE))
                vm_flags &= ~(VM_MAYWRITE | VM_SHARED);
            fallthrough;
        case MAP_PRIVATE:
            if (!(file->f_mode & FMODE_READ))
                return -EACCES;
            if (path_noexec(&file->f_path)) {
                if (vm_flags & VM_EXEC)
                    return -EPERM;
                vm_flags &= ~VM_MAYEXEC;
            }

            if (!file->f_op->mmap)
                return -ENODEV;
            if (vm_flags & (VM_GROWSDOWN|VM_GROWSUP))
                return -EINVAL;
            break;

        default:
            return -EINVAL;
        }
    } else {
        switch (flags & MAP_TYPE) {
        case MAP_SHARED:
            if (vm_flags & (VM_GROWSDOWN|VM_GROWSUP))
                return -EINVAL;
            /*
             * Ignore pgoff.
             */
            pgoff = 0;
            vm_flags |= VM_SHARED | VM_MAYSHARE;
            break;
        case MAP_PRIVATE:
            /*
             * Set pgoff according to addr for anon_vma.
             */
            pgoff = addr >> PAGE_SHIFT;
            break;
        default:
            return -EINVAL;
        }
    }

    /*
     * Set 'VM_NORESERVE' if we should not account for the
     * memory use of this mapping.
     */
    if (flags & MAP_NORESERVE) {
        /* We honor MAP_NORESERVE if allowed to overcommit */
        if (sysctl_overcommit_memory != OVERCOMMIT_NEVER)
            vm_flags |= VM_NORESERVE;

        /* hugetlb applies strict overcommit unless MAP_NORESERVE */
        if (file && is_file_hugepages(file))
            vm_flags |= VM_NORESERVE;
    }

    addr = mmap_region(file, addr, len, vm_flags, pgoff, uf);
    if (!IS_ERR_VALUE(addr) &&
        ((vm_flags & VM_LOCKED) ||
         (flags & (MAP_POPULATE | MAP_NONBLOCK)) == MAP_POPULATE))
        *populate = len;

    return addr;
```

```c
unsigned long mmap_region(struct file *file, unsigned long addr,
		unsigned long len, vm_flags_t vm_flags, unsigned long pgoff,
		struct list_head *uf)
{
	struct mm_struct *mm = current->mm;
	struct vm_area_struct *vma = NULL;
	struct vm_area_struct *next, *prev, *merge;
	pgoff_t pglen = len >> PAGE_SHIFT;
	unsigned long charged = 0;
	unsigned long end = addr + len;
	unsigned long merge_start = addr, merge_end = end;
	bool writable_file_mapping = false;
	pgoff_t vm_pgoff;
	int error;
	VMA_ITERATOR(vmi, mm, addr);

	/* Check against address space limit. */
	if (!may_expand_vm(mm, vm_flags, len >> PAGE_SHIFT)) {
		unsigned long nr_pages;

		/*
		 * MAP_FIXED may remove pages of mappings that intersects with
		 * requested mapping. Account for the pages it would unmap.
		 */
		nr_pages = count_vma_pages_range(mm, addr, end);

		if (!may_expand_vm(mm, vm_flags,
					(len >> PAGE_SHIFT) - nr_pages))
			return -ENOMEM;
	}

	/* Unmap any existing mapping in the area */
	if (do_vmi_munmap(&vmi, mm, addr, len, uf, false))
		return -ENOMEM;

	/*
	 * Private writable mapping: check memory availability
	 */
	if (accountable_mapping(file, vm_flags)) {
		charged = len >> PAGE_SHIFT;
		if (security_vm_enough_memory_mm(mm, charged))
			return -ENOMEM;
		vm_flags |= VM_ACCOUNT;
	}

	next = vma_next(&vmi);
	prev = vma_prev(&vmi);
	if (vm_flags & VM_SPECIAL) {
		if (prev)
			vma_iter_next_range(&vmi);
		goto cannot_expand;
	}

	/* Attempt to expand an old mapping */
	/* Check next */
	if (next && next->vm_start == end && !vma_policy(next) &&
	    can_vma_merge_before(next, vm_flags, NULL, file, pgoff+pglen,
				 NULL_VM_UFFD_CTX, NULL)) {
		merge_end = next->vm_end;
		vma = next;
		vm_pgoff = next->vm_pgoff - pglen;
	}

/* Check prev */
	if (prev && prev->vm_end == addr && !vma_policy(prev) &&
	    (vma ? can_vma_merge_after(prev, vm_flags, vma->anon_vma, file,
				       pgoff, vma->vm_userfaultfd_ctx, NULL) :
		   can_vma_merge_after(prev, vm_flags, NULL, file, pgoff,
				       NULL_VM_UFFD_CTX, NULL))) {
		merge_start = prev->vm_start;
		vma = prev;
		vm_pgoff = prev->vm_pgoff;
	} else if (prev) {
		vma_iter_next_range(&vmi);
	}

	/* Actually expand, if possible */
	if (vma &&
	    !vma_expand(&vmi, vma, merge_start, merge_end, vm_pgoff, next)) {
		khugepaged_enter_vma(vma, vm_flags);
		goto expanded;
	}

	if (vma == prev)
		vma_iter_set(&vmi, addr);
cannot_expand:

	/*
	 * Determine the object being mapped and call the appropriate
	 * specific mapper. the address has already been validated, but
	 * not unmapped, but the maps are removed from the list.
	 */
	vma = vm_area_alloc(mm);
	if (!vma) {
		error = -ENOMEM;
		goto unacct_error;
	}

	vma_iter_config(&vmi, addr, end);
	vma->vm_start = addr;
	vma->vm_end = end;
	vm_flags_init(vma, vm_flags);
	vma->vm_page_prot = vm_get_page_prot(vm_flags);
	vma->vm_pgoff = pgoff;

	if (file) {
		vma->vm_file = get_file(file);
		error = call_mmap(file, vma);
		if (error)
			goto unmap_and_free_vma;

		if (vma_is_shared_maywrite(vma)) {
			error = mapping_map_writable(file->f_mapping);
			if (error)
				goto close_and_free_vma;

			writable_file_mapping = true;
		}


		/*
		 * Expansion is handled above, merging is handled below.
		 * Drivers should not alter the address of the VMA.
		 */
		error = -EINVAL;
		if (WARN_ON((addr != vma->vm_start)))
			goto close_and_free_vma;

		vma_iter_config(&vmi, addr, end);
		/*
		 * If vm_flags changed after call_mmap(), we should try merge
		 * vma again as we may succeed this time.
		 */
		if (unlikely(vm_flags != vma->vm_flags && prev)) {
			merge = vma_merge_new_vma(&vmi, prev, vma,
						  vma->vm_start, vma->vm_end,
						  vma->vm_pgoff);



			if (merge) {
				/*
				 * ->mmap() can change vma->vm_file and fput
				 * the original file. So fput the vma->vm_file
				 * here or we would add an extra fput for file
				 * and cause general protection fault
				 * ultimately.
				 */
				fput(vma->vm_file);
				vm_area_free(vma);
				vma = merge;
				/* Update vm_flags to pick up the change. */
				vm_flags = vma->vm_flags;
				goto unmap_writable;
			}
		}

		vm_flags = vma->vm_flags;
	} else if (vm_flags & VM_SHARED) {
		error = shmem_zero_setup(vma);
		if (error)
			goto free_vma;
	} else {
		vma_set_anonymous(vma);
	}

	if (map_deny_write_exec(vma, vma->vm_flags)) {
		error = -EACCES;
		goto close_and_free_vma;
	}

	/* Allow architectures to sanity-check the vm_flags */
	error = -EINVAL;
	if (!arch_validate_flags(vma->vm_flags))
		goto close_and_free_vma;

	error = -ENOMEM;
	if (vma_iter_prealloc(&vmi, vma))
		goto close_and_free_vma;


	/* Lock the VMA since it is modified after insertion into VMA tree */
	vma_start_write(vma);
	vma_iter_store(&vmi, vma);
	mm->map_count++;
	if (vma->vm_file) {
		i_mmap_lock_write(vma->vm_file->f_mapping);
		if (vma_is_shared_maywrite(vma))
			mapping_allow_writable(vma->vm_file->f_mapping);

		flush_dcache_mmap_lock(vma->vm_file->f_mapping);
		vma_interval_tree_insert(vma, &vma->vm_file->f_mapping->i_mmap);
		flush_dcache_mmap_unlock(vma->vm_file->f_mapping);
		i_mmap_unlock_write(vma->vm_file->f_mapping);
	}

	/*
	 * vma_merge() calls khugepaged_enter_vma() either, the below
	 * call covers the non-merge case.
	 */
	khugepaged_enter_vma(vma, vma->vm_flags);

	/* Once vma denies write, undo our temporary denial count */
unmap_writable:
	if (writable_file_mapping)
		mapping_unmap_writable(file->f_mapping);
	file = vma->vm_file;
	ksm_add_vma(vma);
expanded:
	perf_event_mmap(vma);

	vm_stat_account(mm, vm_flags, len >> PAGE_SHIFT);
	if (vm_flags & VM_LOCKED) {
		if ((vm_flags & VM_SPECIAL) || vma_is_dax(vma) ||
					is_vm_hugetlb_page(vma) ||
					vma == get_gate_vma(current->mm))
			vm_flags_clear(vma, VM_LOCKED_MASK);
		else
			mm->locked_vm += (len >> PAGE_SHIFT);
	}

	if (file)
		uprobe_mmap(vma);

	/*
	 * New (or expanded) vma always get soft dirty status.
	 * Otherwise user-space soft-dirty page tracker won't
	 * be able to distinguish situation when vma area unmapped,
	 * then new mapped in-place (which must be aimed as
	 * a completely new data area).
	 */
	vm_flags_set(vma, VM_SOFTDIRTY);

	vma_set_page_prot(vma);

	validate_mm(mm);
	return addr;

close_and_free_vma:
	if (file && vma->vm_ops && vma->vm_ops->close)
		vma->vm_ops->close(vma);

	if (file || vma->vm_file) {
unmap_and_free_vma:
		fput(vma->vm_file);
		vma->vm_file = NULL;

		vma_iter_set(&vmi, vma->vm_end);
		/* Undo any partial mapping done by a device driver. */
		unmap_region(mm, &vmi.mas, vma, prev, next, vma->vm_start,
			     vma->vm_end, vma->vm_end, true);
	}
	if (writable_file_mapping)
		mapping_unmap_writable(file->f_mapping);
free_vma:
	vm_area_free(vma);
unacct_error:
	if (charged)
		vm_unacct_memory(charged);
	validate_mm(mm);
	return error;
}
```

```c
int shmem_zero_setup(struct vm_area_struct *vma)
{
  struct file *file;
  loff_t size = vma->vm_end - vma->vm_start;

  /*
   * Cloning a new file under mmap_lock leads to a lock ordering conflict
   * between XFS directory reading and selinux: since this file is only
   * accessible to the user through its mapping, use S_PRIVATE flag to
   * bypass file security, in the same way as shmem_kernel_file_setup().
   */
  file = shmem_kernel_file_setup("dev/zero", size, vma->vm_flags);
  if (IS_ERR(file))
    return PTR_ERR(file);

  if (vma->vm_file)
    fput(vma->vm_file);
  vma->vm_file = file;
  vma->vm_ops = &shmem_anon_vm_ops;

  return 0;
}
```

0

where u at?

```
vm_fault_t handle_mm_fault(struct vm_area_struct *vma, unsigned long address,
                unsigned int flags, struct pt_regs *regs)
{
    /* If the fault handler drops the mmap_lock, vma may be freed */
    struct mm_struct *mm = vma->vm_mm;
    vm_fault_t ret;

    __set_current_state(TASK_RUNNING);

    ret = sanitize_fault_flags(vma, &flags);
    if (ret)
        goto out;

    if (!arch_vma_access_permitted(vma, flags & FAULT_FLAG_WRITE,
                         flags & FAULT_FLAG_INSTRUCTION,
                         flags & FAULT_FLAG_REMOTE)) {
        ret = VM_FAULT_SIGSEGV;
        goto out;
    }

    /*
     * Enable the memcg OOM handling for faults triggered in user
     * space.  Kernel faults are handled more gracefully.
     */
    if (flags & FAULT_FLAG_USER)
        mem_cgroup_enter_user_fault();

    lru_gen_enter_fault(vma);

    if (unlikely(is_vm_hugetlb_page(vma)))
        ret = hugetlb_fault(vma->vm_mm, vma, address, flags);
    else
        ret = __handle_mm_fault(vma, address, flags);

    lru_gen_exit_fault();

    if (flags & FAULT_FLAG_USER) {
        mem_cgroup_exit_user_fault();
        /*
         * The task may have entered a memcg OOM situation but
         * if the allocation error was handled gracefully (no
         * VM_FAULT_OOM), there is no need to kill anything.
         * Just clean up the OOM state peacefully.
         */
        if (task_in_memcg_oom(current) && !(ret & VM_FAULT_OOM))
            mem_cgroup_oom_synchronize(false);
    }
out:
    mm_account_fault(mm, regs, address, flags, ret);

    return ret;
}
```

woven by TOYOTA

```
vm_fault_t handle_mm_fault(struct vm_area_struct *vma, unsigned long address,
                unsigned int flags, struct pt_regs *regs)
{
    /* If the fault handler drops the mmap_lock, vma may be freed */
    struct mm_struct *mm = vma->vm_mm;
    vm_fault_t ret;

    __set_current_state(TASK_RUNNING);

    ret = sanitize_fault_flags(vma, &flags);
    if (ret)
        goto out;

    if (!arch_vma_access_permitted(vma, flags & FAULT_FLAG_WRITE,
                        flags & FAULT_FLAG_INSTRUCTION,
                        flags & FAULT_FLAG_REMOTE)) {
        ret = VM_FAULT_SIGSEGV;
        goto out;
    }

    /*
     * Enable the memcg OOM handling for faults triggered in user
     * space.  Kernel faults are handled more gracefully.
     */
    if (flags & FAULT_FLAG_USER)
        mem_cgroup_enter_user_fault();

    lru_gen_enter_fault(vma);

    if (unlikely(is_vm_hugetlb_page(vma)))
        ret = hugetlb_fault(vma->vm_mm, vma, address, flags);
    else
        ret = __handle_mm_fault(vma, address, flags);

    lru_gen_exit_fault();

    if (flags & FAULT_FLAG_USER) {
        mem_cgroup_exit_user_fault();
        /*
         * The task may have entered a memcg OOM situation but
         * if the allocation error was handled gracefully (no
         * VM_FAULT_OOM), there is no need to kill anything.
         * Just clean up the OOM state peacefully.
         */
        if (task_in_memcg_oom(current) && !(ret & VM_FAULT_OOM))
            mem_cgroup_oom_synchronize(false);
    }
out:
    mm_account_fault(mm, regs, address, flags, ret);

    return ret;
}
```

ret = __handle_mm_fault(vma, address, flags);

```
static vm_fault_t __handle_mm_fault(struct vm_area_struct *vma,
        unsigned long address, unsigned int flags)
{
    struct vm_fault vmf = {
        .vma = vma,
        .address = address & PAGE_MASK,
        .real_address = address,
        .flags = flags,
        .pgoff = linear_page_index(vma, address),
        .gfp_mask = __get_fault_gfp_mask(vma),
    };
    struct mm_struct *mm = vma->vm_mm;
    unsigned long vm_flags = vma->vm_flags;
    pgd_t *pgd;
    p4d_t *p4d;
    vm_fault_t ret;

    pgd = pgd_offset(mm, address);
    p4d = p4d_alloc(mm, pgd, address);
    if (!p4d)
        return VM_FAULT_OOM;

    vmf.pud = pud_alloc(mm, p4d, address);
    if (!vmf.pud)
        return VM_FAULT_OOM;
retry_pud:
    if (pud_none(*vmf.pud) &&
        thp_vma_allowable_order(vma, vm_flags, false, true, true, PUD_ORDER)) {
        ret = create_huge_pud(&vmf);
        if (!(ret & VM_FAULT_FALLBACK))
            return ret;
    } else {
        pud_t orig_pud = *vmf.pud;

        barrier();
        if (pud_trans_huge(orig_pud) || pud_devmap(orig_pud)) {

            /*
             * TODO once we support anonymous PUDs: NUMA case and
             * FAULT_FLAG_UNSHARE handling.
             */
            if ((flags & FAULT_FLAG_WRITE) && !pud_write(orig_pud)) {
                ret = wp_huge_pud(&vmf, orig_pud);
                if (!(ret & VM_FAULT_FALLBACK))
                    return ret;
            } else {
                huge_pud_set_accessed(&vmf, orig_pud);
                return 0;
            }
        }
    }

    vmf.pmd = pmd_alloc(mm, vmf.pud, address);
    if (!vmf.pmd)
        return VM_FAULT_OOM;

    /* Huge pud page fault raced with pmd_alloc? */
    if (pud_trans_unstable(vmf.pud))
        goto retry_pud;

    if (pmd_none(*vmf.pmd) &&
        thp_vma_allowable_order(vma, vm_flags, false, true, true, PMD_ORDER)) {
        ret = create_huge_pmd(&vmf);
        if (!(ret & VM_FAULT_FALLBACK))
            return ret;
    } else {
        vmf.orig_pmd = pmdp_get_lockless(vmf.pmd);

        if (unlikely(is_swap_pmd(vmf.orig_pmd))) {
            VM_BUG_ON(thp_migration_supported() &&
                       !is_pmd_migration_entry(vmf.orig_pmd));
            if (is_pmd_migration_entry(vmf.orig_pmd))
                pmd_migration_entry_wait(mm, vmf.pmd);
            return 0;
        }
        if (pmd_trans_huge(vmf.orig_pmd) || pmd_devmap(vmf.orig_pmd)) {
            if (pmd_protnone(vmf.orig_pmd) && vma_is_accessible(vma))
                return do_huge_pmd_numa_page(&vmf);

            if ((flags & (FAULT_FLAG_WRITE|FAULT_FLAG_UNSHARE)) &&
                !pmd_write(vmf.orig_pmd)) {
                ret = wp_huge_pmd(&vmf);
                if (!(ret & VM_FAULT_FALLBACK))
                    return ret;
            } else {
                huge_pmd_set_accessed(&vmf);
                return 0;
            }
        }
    }

    return handle_pte_fault(&vmf);
}
```

```
static vm_fault_t __handle_mm_fault(struct vm_area_struct *vma,
        unsigned long address, unsigned int flags)
{
    struct vm_fault vmf = {
        .vma = vma,
        .address = address & PAGE_MASK,
        .real_address = address,
        .flags = flags,
        .pgoff = linear_page_index(vma, address),
        .gfp_mask = __get_fault_gfp_mask(vma),
    };
    struct mm_struct *mm = vma->vm_mm;
    unsigned long vm_flags = vma->vm_flags;
    pgd_t *pgd;
    p4d_t *p4d;
    vm_fault_t ret;

    pgd = pgd_offset(mm, address);
    p4d = p4d_alloc(mm, pgd, address);
    if (!p4d)
        return VM_FAULT_OOM;

    vmf.pud = pud_alloc(mm, p4d, address);
    if (!vmf.pud)
        return VM_FAULT_OOM;
retry_pud:
    if (pud_none(*vmf.pud) &&
        thp_vma_allowable_order(vma, vm_flags, false, true, true, PUD_ORDER)) {
        ret = create_huge_pud(&vmf);
        if (!(ret & VM_FAULT_FALLBACK))
            return ret;
    } else {
        pud_t orig_pud = *vmf.pud;

        barrier();
        if (pud_trans_huge(orig_pud) || pud_devmap(orig_pud)) {

            /*
             * TODO once we support anonymous PUDs: NUMA case and
             * FAULT_FLAG_UNSHARE handling.
             */
            if ((flags & FAULT_FLAG_WRITE) && !pud_write(orig_pud)) {
                ret = wp_huge_pud(&vmf, orig_pud);
                if (!(ret & VM_FAULT_FALLBACK))
                    return ret;
            } else {
                huge_pud_set_accessed(&vmf, orig_pud);
                return 0;
            }
        }
    }

    vmf.pmd = pmd_alloc(mm, vmf.pud, address);
    if (!vmf.pmd)
        return VM_FAULT_OOM;

    /* Huge pud page fault raced with pmd_alloc? */
    if (pud_trans_unstable(vmf.pud))
        goto retry_pud;

    if (pmd_none(*vmf.pmd) &&
        thp_vma_allowable_order(vma, vm_flags, false, true, true, PMD_ORDER)) {
        ret = create_huge_pmd(&vmf);
        if (!(ret & VM_FAULT_FALLBACK))
            return ret;
    } else {
        vmf.orig_pmd = pmdp_get_lockless(vmf.pmd);

        if (unlikely(is_swap_pmd(vmf.orig_pmd))) {
            VM_BUG_ON(thp_migration_supported() &&
                       !is_pmd_migration_entry(vmf.orig_pmd));
            if (is_pmd_migration_entry(vmf.orig_pmd))
                pmd_migration_entry_wait(mm, vmf.pmd);
            return 0;
        }
        if (pmd_trans_huge(vmf.orig_pmd) || pmd_devmap(vmf.orig_pmd)) {
            if (pmd_protnone(vmf.orig_pmd) && vma_is_accessible(vma))
                return do_huge_pmd_numa_page(&vmf);

            if ((flags & (FAULT_FLAG_WRITE|FAULT_FLAG_UNSHARE)) &&
                !pmd_write(vmf.orig_pmd)) {
                ret = wp_huge_pmd(&vmf);
                if (!(ret & VM_FAULT_FALLBACK))
                    return ret;
            } else {
                huge_pmd_set_accessed(&vmf);
                return 0;
            }
        }
    }

    return handle_pte_fault(&vmf);
}
```

```c
static vm_fault_t handle_pte_fault(struct vm_fault *vmf)
{
    pte_t entry;

    if (unlikely(pmd_none(*vmf->pmd))) {
        /*
         * Leave __pte_alloc() until later: because vm_ops->fault may
         * want to allocate huge page, and if we expose page table
         * for an instant, it will be difficult to retract from
         * concurrent faults and from rmap lookups.
         */
        vmf->pte = NULL;
        vmf->flags &= ~FAULT_FLAG_ORIG_PTE_VALID;
    } else {
        /*
         * A regular pmd is established and it can't morph into a huge
         * pmd by anon khugepaged, since that takes mmap_lock in write
         * mode; but shmem or file collapse to THP could still morph
         * it into a huge pmd: just retry later if so.
         */
        vmf->pte = pte_offset_map_nolock(vmf->vma->vm_mm, vmf->pmd,
                         vmf->address, &vmf->ptl);
        if (unlikely(!vmf->pte))
            return 0;
        vmf->orig_pte = ptep_get_lockless(vmf->pte);
        vmf->flags |= FAULT_FLAG_ORIG_PTE_VALID;

        if (pte_none(vmf->orig_pte)) {
            pte_unmap(vmf->pte);
            vmf->pte = NULL;
        }
    }

    if (!vmf->pte)
        return do_pte_missing(vmf);

    if (!pte_present(vmf->orig_pte))
        return do_swap_page(vmf);

    if (pte_protnone(vmf->orig_pte) && vma_is_accessible(vmf->vma))
        return do_numa_page(vmf);

    spin_lock(vmf->ptl);
    entry = vmf->orig_pte;
    if (unlikely(!pte_same(ptep_get(vmf->pte), entry))) {
        update_mmu_tlb(vmf->vma, vmf->address, vmf->pte);
        goto unlock;
    }
    if (vmf->flags & (FAULT_FLAG_WRITE|FAULT_FLAG_UNSHARE)) {
        if (!pte_write(entry))
            return do_wp_page(vmf);
        else if (likely(vmf->flags & FAULT_FLAG_WRITE))
            entry = pte_mkdirty(entry);
    }
    entry = pte_mkyoung(entry);
    if (ptep_set_access_flags(vmf->vma, vmf->address, vmf->pte, entry,
                vmf->flags & FAULT_FLAG_WRITE)) {
        update_mmu_cache_range(vmf, vmf->vma, vmf->address,
                vmf->pte, 1);
    } else {
        /* Skip spurious TLB flush for retried page fault */
        if (vmf->flags & FAULT_FLAG_TRIED)
            goto unlock;
        /*
         * This is needed only for protection faults but the arch code
         * is not yet telling us if this is a protection fault or not.
         * This still avoids useless tlb flushes for .text page faults
         * with threads.
         */
        if (vmf->flags & FAULT_FLAG_WRITE)
            flush_tlb_fix_spurious_fault(vmf->vma, vmf->address,
                             vmf->pte);
    }
unlock:
    pte_unmap_unlock(vmf->pte, vmf->ptl);
    return 0;
}
```

```c
static vm_fault_t handle_pte_fault(struct vm_fault *vmf)
{
    pte_t entry;

    if (unlikely(pmd_none(*vmf->pmd))) {
        /*
         * Leave __pte_alloc() until later: because vm_ops->fault may
         * want to allocate huge page, and if we expose page table
         * for an instant, it will be difficult to retract from
         * concurrent faults and from rmap lookups.
         */
        vmf->pte = NULL;
        vmf->flags &= ~FAULT_FLAG_ORIG_PTE_VALID;
    } else {
        /*
         * A regular pmd is established and it can't morph into a huge
         * pmd by anon khugepaged, since that takes mmap_lock in write
         * mode; but shmem or file collapse to THP could still morph
         * it into a huge pmd: just retry later if so.
         */
        vmf->pte = pte_offset_map_nolock(vmf->vma->vm_mm, vmf->pmd,
                         vmf->address, &vmf->ptl);
        if (unlikely(!vmf->pte))
            return 0;
        vmf->orig_pte = ptep_get_lockless(vmf->pte);
        vmf->flags |= FAULT_FLAG_ORIG_PTE_VALID;

        if (pte_none(vmf->orig_pte)) {
            pte_unmap(vmf->pte);
            vmf->pte = NULL;
        }
    }

    if (!vmf->pte)
        return do_pte_missing(vmf);

    if (!pte_present(vmf->orig_pte))
        return do_swap_page(vmf);

    if (pte_protnone(vmf->orig_pte) && vma_is_accessible(vmf->vma))
        return do_numa_page(vmf);

    spin_lock(vmf->ptl);
    entry = vmf->orig_pte;
    if (unlikely(!pte_same(ptep_get(vmf->pte), entry))) {
        update_mmu_tlb(vmf->vma, vmf->address, vmf->pte);
        goto unlock;
    }
    if (vmf->flags & (FAULT_FLAG_WRITE|FAULT_FLAG_UNSHARE)) {
        if (!pte_write(entry))
            return do_wp_page(vmf);
        else if (likely(vmf->flags & FAULT_FLAG_WRITE))
            entry = pte_mkdirty(entry);
    }
    entry = pte_mkyoung(entry);
    if (ptep_set_access_flags(vmf->vma, vmf->address, vmf->pte, entry,
                vmf->flags & FAULT_FLAG_WRITE)) {
        update_mmu_cache_range(vmf, vmf->vma, vmf->address,
                vmf->pte, 1);
    } else {
        /* Skip spurious TLB flush for retried page fault */
        if (vmf->flags & FAULT_FLAG_TRIED)
            goto unlock;
        /*
         * This is needed only for protection faults but the arch code
         * is not yet telling us if this is a protection fault or not.
         * This still avoids useless tlb flushes for .text page faults
         * with threads.
         */
        if (vmf->flags & FAULT_FLAG_WRITE)
            flush_tlb_fix_spurious_fault(vmf->vma, vmf->address,
                             vmf->pte);
    }
unlock:
    pte_unmap_unlock(vmf->pte, vmf->ptl);
    return 0;
}
```

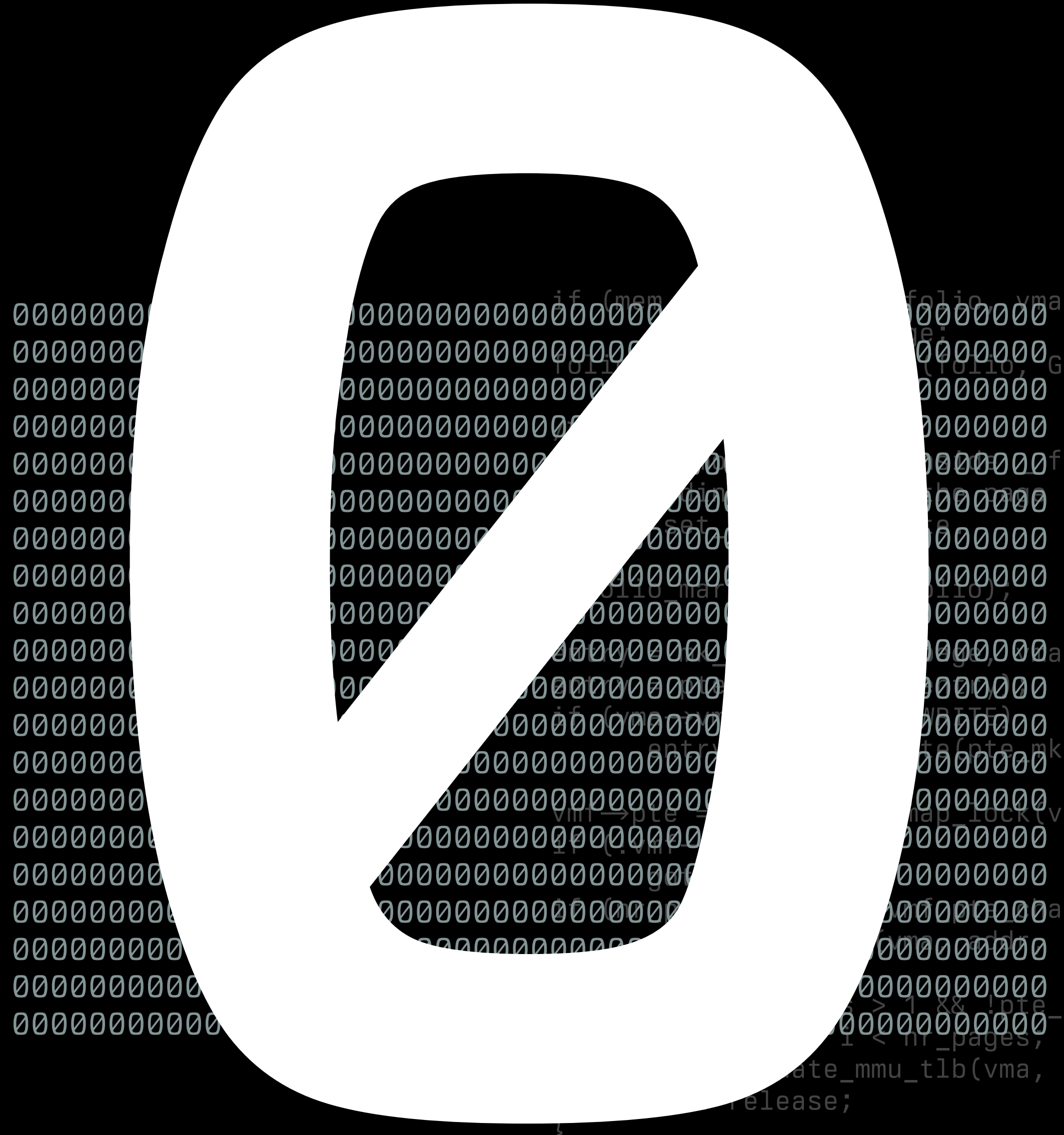
```c
if (!vmf->pte)
    return do_pte_missing(vmf);
```

```
static vm_fault_t do_pte_missing(struct vm_fault *vmf)
{
  if (vma_is_anonymous(vmf->vma))
    return do_anonymous_page(vmf);
  else
    return do_fault(vmf);
}
```

```c
static vm_fault_t do_anonymous_page(struct vm_fault *vmf)
{
      bool uffd_wp = vmf_orig_pte_uffd_wp(vmf);
      struct vm_area_struct *vma = vmf->vma;
      unsigned long addr = vmf->address;
      struct folio *folio;
      vm_fault_t ret = 0;
      int nr_pages = 1;
      pte_t entry;
      int i;

      /* File mapping without ->vm_ops ? */
      if (vma->vm_flags & VM_SHARED)
            return VM_FAULT_SIGBUS;

      /*
       * Use pte_alloc() instead of pte_alloc_map(), so that OOM can
       * be distinguished from a transient failure of pte_offset_map().
       */
      if (pte_alloc(vma->vm_mm, vmf->pmd))
            return VM_FAULT_OOM;

      /* Use the zero-page for reads */
      if (!(vmf->flags & FAULT_FLAG_WRITE) &&
                  !mm_forbids_zeropage(vma->vm_mm)) {
            entry = pte_mkspecial(pfn_pte(my_zero_pfn(vmf->address),
                                    vma->vm_page_prot));
            vmf->pte = pte_offset_map_lock(vma->vm_mm, vmf->pmd,
                        vmf->address, &vmf->ptl);
            if (!vmf->pte)
                  goto unlock;
            if (vmf_pte_changed(vmf)) {
                  update_mmu_tlb(vma, vmf->address, vmf->pte);
                  goto unlock;
            }
            ret = check_stable_address_space(vma->vm_mm);
            if (ret)
                  goto unlock;
            /* Deliver the page fault to userland, check inside PT lock */
            if (userfaultfd_missing(vma)) {
                  pte_unmap_unlock(vmf->pte, vmf->ptl);
                  return handle_userfault(vmf, VM_UFFD_MISSING);
            }
            goto setpte;
      }

      /* Allocate our own private page. */
      if (unlikely(anon_vma_prepare(vma)))
            goto oom;
      /* Returns NULL on OOM or ERR_PTR(-EAGAIN) if we must retry the fault */
      folio = alloc_anon_folio(vmf);
      if (IS_ERR(folio))
            return 0;
      if (!folio)
            goto oom;

      nr_pages = folio_nr_pages(folio);
      addr = ALIGN_DOWN(vmf->address, nr_pages * PAGE_SIZE);
```

```c
      if (mem_cgroup_charge(folio, vma->vm_mm, GFP_KERNEL))
            goto oom_free_page;
      folio_throttle_swaprate(folio, GFP_KERNEL);

      /*
       * The memory barrier inside __folio_mark_uptodate makes sure that
       * preceding stores to the page contents become visible before
       * the set_pte_at() write.
       */
      __folio_mark_uptodate(folio);

      entry = mk_pte(&folio->page, vma->vm_page_prot);
      entry = pte_sw_mkyoung(entry);
      if (vma->vm_flags & VM_WRITE)
            entry = pte_mkwrite(pte_mkdirty(entry), vma);

      vmf->pte = pte_offset_map_lock(vma->vm_mm, vmf->pmd, addr, &vmf->ptl);
      if (!vmf->pte)
            goto release;
      if (nr_pages == 1 && vmf_pte_changed(vmf)) {
            update_mmu_tlb(vma, addr, vmf->pte);
            goto release;
      } else if (nr_pages > 1 && !pte_range_none(vmf->pte, nr_pages)) {
            for (i = 0; i < nr_pages; i++)
                  update_mmu_tlb(vma, addr + PAGE_SIZE * i, vmf->pte + i);
            goto release;
      }

      ret = check_stable_address_space(vma->vm_mm);
      if (ret)
            goto release;

      /* Deliver the page fault to userland, check inside PT lock */
      if (userfaultfd_missing(vma)) {
            pte_unmap_unlock(vmf->pte, vmf->ptl);
            folio_put(folio);
            return handle_userfault(vmf, VM_UFFD_MISSING);
      }

      folio_ref_add(folio, nr_pages - 1);
      add_mm_counter(vma->vm_mm, MM_ANONPAGES, nr_pages);
      folio_add_new_anon_rmap(folio, vma, addr);
      folio_add_lru_vma(folio, vma);
setpte:
      if (uffd_wp)
            entry = pte_mkuffd_wp(entry);
      set_ptes(vma->vm_mm, addr, vmf->pte, entry, nr_pages);

      /* No need to invalidate - it was non-present before */
      update_mmu_cache_range(vmf, vma, addr, vmf->pte, nr_pages);
unlock:
      if (vmf->pte)
            pte_unmap_unlock(vmf->pte, vmf->ptl);
      return ret;
release:
      folio_put(folio);
      goto unlock;
oom_free_page:
      folio_put(folio);
oom:
      return VM_FAULT_OOM;
}
```

woven by TOYOTA



```
/* Use the zero-page for reads */
if (!(vmf→flags & FAULT_FLAG_WRITE) &&
    !mm_forbids_zeropage(vma→vm_mm)) {
  entry = pte_mkspecial(pfn_pte(my_zero_pfn(vmf→address),
                vma→vm_page_prot));
```

```
static vm_fault_t do_anonymous_page(struct vm_fault *vmf)
{
      bool uffd_wp = vmf_orig_pte_uffd_wp(vmf);
      struct vm_area_struct *vma = vmf→vma;
      unsigned long addr = vmf→address;
      struct folio *folio;
      vm_fault_t ret = 0;
      int nr_pages = 1;
      pte_t entry;
      int i;

      /* File mapping without →vm_ops ? */
      if (vma→vm_flags & VM_SHARED)
            return VM_FAULT_SIGBUS;

      /*
       * Use pte_alloc() instead of pte_alloc_map(), so that OOM can
       * be distinguished from a transient failure of pte_offset_map().
       */
      if (pte_alloc(vma→vm_mm, vmf→pmd))
            return VM_FAULT_OOM;

      /* Use the zero-page for reads */
      if (!(vmf→flags & FAULT_FLAG_WRITE) &&
                  !mm_forbids_zeropage(vma→vm_mm)) {
            entry = pte_mkspecial(pfn_pte(my_zero_pfn(vmf→address),
                                    vma→vm_page_prot));
            vmf→pte = pte_offset_map_lock(vma→vm_mm, vmf→pmd,
                        vmf→address, &vmf→ptl);
            if (!vmf→pte)
                  goto unlock;
            if (vmf_pte_changed(vmf)) {
                  update_mmu_tlb(vma, vmf→address, vmf→pte);
                  goto unlock;
            }
            ret = check_stable_address_space(vma→vm_mm);
            if (ret)
                  goto unlock;
            /* Deliver the page fault to userland, check inside PT lock */
            if (userfaultfd_missing(vma)) {
                  pte_unmap_unlock(vmf→pte, vmf→ptl);
                  return handle_userfault(vmf, VM_UFFD_MISSING);
            }
            goto setpte;
      }

      /* Allocate our own private page. */
      if (unlikely(anon_vma_prepare(vma)))
            goto oom;
      /* Returns NULL on OOM or ERR_PTR(-EAGAIN) if we must retry the fault */
      folio = alloc_anon_folio(vmf);
      if (IS_ERR(folio))
            return 0;
      if (!folio)
            goto oom;

      nr_pages = folio_nr_pages(folio);
      addr = ALIGN_DOWN(vmf→address, nr_pages * PAGE_SIZE);

      if (mem_cgroup_charge(folio, vma→vm_mm, GFP_KERNEL))
            goto oom_free_page;
      folio_throttle_swaprate(folio, GFP_KERNEL);

      /*
       * The memory barrier inside __folio_mark_uptodate makes sure that
       * preceding stores to the page contents become visible before
       * the set_pte_at() write.
       */
      __folio_mark_uptodate(folio);

      entry = mk_pte(&folio→page, vma→vm_page_prot);
      entry = pte_sw_mkyoung(entry);
      if (vma→vm_flags & VM_WRITE)
            entry = pte_mkwrite(pte_mkdirty(entry), vma);

      vmf→pte = pte_offset_map_lock(vma→vm_mm, vmf→pmd, addr, &vmf→ptl);
      if (!vmf→pte)
            goto release;
      if (nr_pages == 1 && vmf_pte_changed(vmf)) {
            update_mmu_tlb(vma, addr, vmf→pte);
            goto release;
      } else if (nr_pages > 1 && !pte_range_none(vmf→pte, nr_pages)) {
            for (i = 0; i < nr_pages; i++)
                  update_mmu_tlb(vma, addr + PAGE_SIZE * i, vmf→pte + i);
            goto release;
      }

      ret = check_stable_address_space(vma→vm_mm);
      if (ret)
            goto release;

      /* Deliver the page fault to userland, check inside PT lock */
      if (userfaultfd_missing(vma)) {
            pte_unmap_unlock(vmf→pte, vmf→ptl);
            folio_put(folio);
            return handle_userfault(vmf, VM_UFFD_MISSING);
      }

      folio_ref_add(folio, nr_pages - 1);
      add_mm_counter(vma→vm_mm, MM_ANONPAGES, nr_pages);
      folio_add_new_anon_rmap(folio, vma, addr);
      folio_add_lru_vma(folio, vma);
setpte:
      if (uffd_wp)
            entry = pte_mkuffd_wp(entry);
      set_ptes(vma→vm_mm, addr, vmf→pte, entry, nr_pages);

      /* No need to invalidate - it was non-present before */
      update_mmu_cache_range(vmf, vma, addr, vmf→pte, nr_pages);
unlock:
      if (vmf→pte)
            pte_unmap_unlock(vmf→pte, vmf→ptl);
      return ret;
release:
      folio_put(folio);
      goto unlock;
oom_free_page:
      folio_put(folio);
oom:
      return VM_FAULT_OOM;
}
```

```c
folio = alloc_anon_folio(vmf);
```

```c
static vm_fault_t do_anonymous_page(struct vm_fault *vmf)
{
	bool uffd_wp = vmf_orig_pte_uffd_wp(vmf);
	struct vm_area_struct *vma = vmf->vma;
	unsigned long addr = vmf->address;
	struct folio *folio;
	vm_fault_t ret = 0;
	int nr_pages = 1;
	pte_t entry;
	int i;

	/* File mapping without ->vm_ops ? */
	if (vma->vm_flags & VM_SHARED)
		return VM_FAULT_SIGBUS;

	/*
	 * Use pte_alloc() instead of pte_alloc_map(), so that OOM can
	 * be distinguished from a transient failure of pte_offset_map().
	 */
	if (pte_alloc(vma->vm_mm, vmf->pmd))
		return VM_FAULT_OOM;

	/* Use the zero-page for reads */
	if (!(vmf->flags & FAULT_FLAG_WRITE) &&
			!mm_forbids_zeropage(vma->vm_mm)) {
		entry = pte_mkspecial(pfn_pte(my_zero_pfn(vmf->address),
						vma->vm_page_prot));
		vmf->pte = pte_offset_map_lock(vma->vm_mm, vmf->pmd,
				vmf->address, &vmf->ptl);
		if (!vmf->pte)
			goto unlock;
		if (vmf_pte_changed(vmf)) {
			update_mmu_tlb(vma, vmf->address, vmf->pte);
			goto unlock;
		}
		ret = check_stable_address_space(vma->vm_mm);
		if (ret)
			goto unlock;
		/* Deliver the page fault to userland, check inside PT lock */
		if (userfaultfd_missing(vma)) {
			pte_unmap_unlock(vmf->pte, vmf->ptl);
			return handle_userfault(vmf, VM_UFFD_MISSING);
		}
		goto setpte;
	}

	/* Allocate our own private page. */
	if (unlikely(anon_vma_prepare(vma)))
		goto oom;
	/* Returns NULL on OOM or ERR_PTR(-EAGAIN) if we must retry the fault */
	folio = alloc_anon_folio(vmf);
	if (IS_ERR(folio))
		return 0;
	if (!folio)
		goto oom;

	nr_pages = folio_nr_pages(folio);
	addr = ALIGN_DOWN(vmf->address, nr_pages * PAGE_SIZE);

	if (mem_cgroup_charge(folio, vma->vm_mm, GFP_KERNEL))
		goto oom_free_page;
	folio_throttle_swaprate(folio, GFP_KERNEL);

	/*
	 * The memory barrier inside __folio_mark_uptodate makes sure that
	 * preceding stores to the page contents become visible before
	 * the set_pte_at() write.
	 */
	__folio_mark_uptodate(folio);

	entry = mk_pte(&folio->page, vma->vm_page_prot);
	entry = pte_sw_mkyoung(entry);
	if (vma->vm_flags & VM_WRITE)
		entry = pte_mkwrite(pte_mkdirty(entry), vma);

	vmf->pte = pte_offset_map_lock(vma->vm_mm, vmf->pmd, addr, &vmf->ptl);
	if (!vmf->pte)
		goto release;
	if (nr_pages == 1 && vmf_pte_changed(vmf)) {
		update_mmu_tlb(vma, addr, vmf->pte);
		goto release;
	} else if (nr_pages > 1 && !pte_range_none(vmf->pte, nr_pages)) {
		for (i = 0; i < nr_pages; i++)
			update_mmu_tlb(vma, addr + PAGE_SIZE * i, vmf->pte + i);
		goto release;
	}

	ret = check_stable_address_space(vma->vm_mm);
	if (ret)
		goto release;

	/* Deliver the page fault to userland, check inside PT lock */
	if (userfaultfd_missing(vma)) {
		pte_unmap_unlock(vmf->pte, vmf->ptl);
		folio_put(folio);
		return handle_userfault(vmf, VM_UFFD_MISSING);
	}

	folio_ref_add(folio, nr_pages - 1);
	add_mm_counter(vma->vm_mm, MM_ANONPAGES, nr_pages);
	folio_add_new_anon_rmap(folio, vma, addr);
	folio_add_lru_vma(folio, vma);
setpte:
	if (uffd_wp)
		entry = pte_mkuffd_wp(entry);
	set_ptes(vma->vm_mm, addr, vmf->pte, entry, nr_pages);

	/* No need to invalidate - it was non-present before */
	update_mmu_cache_range(vmf, vma, addr, vmf->pte, nr_pages);
unlock:
	if (vmf->pte)
		pte_unmap_unlock(vmf->pte, vmf->ptl);
	return ret;
release:
	folio_put(folio);
	goto unlock;
oom_free_page:
	folio_put(folio);
oom:
	return VM_FAULT_OOM;
}
```

```c
struct page *__alloc_pages(gfp_t gfp, unsigned int order, int preferred_nid,
                              nodemask_t *nodemask)
{
    struct page *page;
    unsigned int alloc_flags = ALLOC_WMARK_LOW;
    gfp_t alloc_gfp; /* The gfp_t that was actually used for allocation */
    struct alloc_context ac = { };

    /*
     * There are several places where we assume that the order value is sane
     * so bail out early if the request is out of bound.
     */
    if (WARN_ON_ONCE_GFP(order > MAX_PAGE_ORDER, gfp))
        return NULL;

    gfp &= gfp_allowed_mask;
    /*
     * Apply scoped allocation constraints. This is mainly about GFP_NOFS
     * resp. GFP_NOIO which has to be inherited for all allocation requests
     * from a particular context which has been marked by
     * memalloc_no{fs,io}_{save,restore}. And PF_MEMALLOC_PIN which ensures
     * movable zones are not used during allocation.
     */
    gfp = current_gfp_context(gfp);
    alloc_gfp = gfp;
    if (!prepare_alloc_pages(gfp, order, preferred_nid, nodemask, &ac,
             &alloc_gfp, &alloc_flags))
        return NULL;

    /*
     * Forbid the first pass from falling back to types that fragment
     * memory until all local zones are considered.
     */
    alloc_flags |= alloc_flags_nofragment(ac.preferred_zoneref->zone, gfp);

    /* First allocation attempt */
    page = get_page_from_freelist(alloc_gfp, order, alloc_flags, &ac);
    if (likely(page))
        goto out;

    alloc_gfp = gfp;
    ac.spread_dirty_pages = false;

    /*
     * Restore the original nodemask if it was potentially replaced with
     * &cpuset_current_mems_allowed to optimize the fast-path attempt.
     */
    ac.nodemask = nodemask;

    page = __alloc_pages_slowpath(alloc_gfp, order, &ac);

out:
    if (memcg_kmem_online() && (gfp & __GFP_ACCOUNT) && page &&
        unlikely(__memcg_kmem_charge_page(page, gfp, order) != 0)) {
        __free_pages(page, order);
        page = NULL;
    }

    trace_mm_page_alloc(page, order, alloc_gfp, ac.migratetype);
    kmsan_alloc_page(page, order, alloc_gfp);

    return page;
}
```

```
struct page *__alloc_pages(gfp_t gfp, unsigned int order, int preferred_nid,
                            nodemask_t *nodemask)
{
    struct page *page;
    unsigned int alloc_flags = ALLOC_WMARK_LOW;
    gfp_t alloc_gfp; /* The gfp_t that was actually used for allocation */
    struct alloc_context ac = { };

    /*
     * There are several places where we assume that the order value is sane
     * so bail out early if the request is out of bound.
     */
    if (WARN_ON_ONCE_GFP(order > MAX_PAGE_ORDER, gfp))
        return NULL;

    gfp &= gfp_allowed_mask;
    /*
     * Apply scoped allocation constraints. This is mainly about GFP_NOFS
     * resp. GFP_NOIO which has to be inherited for all allocation requests
     * from a particular context which has been marked by
     * memalloc_no{fs,io}_{save,restore}. And PF_MEMALLOC_PIN which ensures
     * movable zones are not used during allocation.
     */
    gfp = current_gfp_context(gfp);
    alloc_gfp = gfp;
    if (!prepare_alloc_pages(gfp, order, preferred_nid, nodemask, &ac,
            &alloc_gfp, &alloc_flags))
        return NULL;

    /*
     * Forbid the first pass from falling back to types that fragment
     * memory until all local zones are considered.
     */
    alloc_flags |= alloc_flags_nofragment(ac.preferred_zoneref->zone, gfp);

    /* First allocation attempt */
    page = get_page_from_freelist(alloc_gfp, order, alloc_flags, &ac);
    if (likely(page))
        goto out;

    alloc_gfp = gfp;
    ac.spread_dirty_pages = false;

    /*
     * Restore the original nodemask if it was potentially replaced with
     * &cpuset_current_mems_allowed to optimize the fast-path attempt.
     */
    ac.nodemask = nodemask;

    page = __alloc_pages_slowpath(alloc_gfp, order, &ac);

out:
    if (memcg_kmem_online() && (gfp & __GFP_ACCOUNT) && page &&
        unlikely(__memcg_kmem_charge_page(page, gfp, order) != 0)) {
        __free_pages(page, order);
        page = NULL;
    }

    trace_mm_page_alloc(page, order, alloc_gfp, ac.migratetype);
    kmsan_alloc_page(page, order, alloc_gfp);

    return page;
}
```

```
/* First allocation attempt */
page = get_page_from_freelist(alloc_gfp, order, alloc_flags, &ac);
if (likely(page))
  goto out;

/* ... */

page = __alloc_pages_slowpath(alloc_gfp, order, &ac);
```

```c
static void prep_new_page(struct page *page,
              unsigned int order, gfp_t gfp_flags,
              unsigned int alloc_flags)
{
  post_alloc_hook(page, order, gfp_flags);

  if (order && (gfp_flags & __GFP_COMP))
    prep_compound_page(page, order);

  /*
   * page is set pfmemalloc when ALLOC_NO_WATERMARKS was
   * necessary to allocate the page. The expectation is that
   * the caller is taking steps that will free more memory.
   * The caller should avoid the page being used for
   * !PFMEMALLOC purposes.
   */
  if (alloc_flags & ALLOC_NO_WATERMARKS)
    set_page_pfmemalloc(page);
  else
    clear_page_pfmemalloc(page);
}
```

```
inline void post_alloc_hook(struct page *page, unsigned int order,
                    gfp_t gfp_flags)
{
     bool init = !want_init_on_free() && want_init_on_alloc(gfp_flags) &&
               !should_skip_init(gfp_flags);
     bool zero_tags = init && (gfp_flags & __GFP_ZEROTAGS);
     int i;

     set_page_private(page, 0);
     set_page_refcounted(page);

     arch_alloc_page(page, order);
     debug_pagealloc_map_pages(page, 1 << order);

     /*
      * Page unpoisoning must happen before memory initialization.
      * Otherwise, the poison pattern will be overwritten for __GFP_ZERO
      * allocations and the page unpoisoning code will complain.
      */
     kernel_unpoison_pages(page, 1 << order);

     /*
      * As memory initialization might be integrated into KASAN,
      * KASAN unpoisoning and memory initialization code must be
      * kept together to avoid discrepancies in behavior.
      */

     /*
      * If memory tags should be zeroed
      * (which happens only when memory should be initialized as well).
      */
     if (zero_tags) {
          /* Initialize both memory and memory tags. */
          for (i = 0; i != 1 << order; ++i)
               tag_clear_highpage(page + i);

          /* Take note that memory was initialized by the loop above. */
          init = false;
     }
     if (!should_skip_kasan_unpoison(gfp_flags) &&
         kasan_unpoison_pages(page, order, init)) {
          /* Take note that memory was initialized by KASAN. */
          if (kasan_has_integrated_init())
               init = false;
     } else {
          /*
           * If memory tags have not been set by KASAN, reset the page
           * tags to ensure page_address() dereferencing does not fault.
           */
          for (i = 0; i != 1 << order; ++i)
               page_kasan_tag_reset(page + i);
     }
     /* If memory is still not initialized, initialize it now. */
     if (init)
          kernel_init_pages(page, 1 << order);

     set_page_owner(page, order, gfp_flags);
     page_table_check_alloc(page, order);
}
```

```c
inline void post_alloc_hook(struct page *page, unsigned int order,
                            gfp_t gfp_flags)
{
	bool init = !want_init_on_free() && want_init_on_alloc(gfp_flags) &&
			!should_skip_init(gfp_flags);
	bool zero_tags = init && (gfp_flags & __GFP_ZEROTAGS);
	int i;

	set_page_private(page, 0);
	set_page_refcounted(page);

	arch_alloc_page(page, order);
	debug_pagealloc_map_pages(page, 1 << order);

	/*
	 * Page unpoisoning must happen before memory initialization.
	 * Otherwise, the poison pattern will be overwritten for __GFP_ZERO
	 * allocations and the page unpoisoning code will complain.
	 */
	kernel_unpoison_pages(page, 1 << order);

	/*
	 * As memory initialization might be integrated into KASAN,
	 * KASAN unpoisoning and memory initializion code must be
	 * kept together to avoid discrepancies in behavior.
	 */

	/*
	 * If memory tags should be zeroed
	 * (which happens only when memory should be initialized as well).
	 */
	if (zero_tags) {
		/* Initialize both memory and memory tags. */
		for (i = 0; i != 1 << order; ++i)
			tag_clear_highpage(page + i);

		/* Take note that memory was initialized by the loop above. */
		init = false;
	}
	if (!should_skip_kasan_unpoison(gfp_flags) &&
	    kasan_unpoison_pages(page, order, init)) {
		/* Take note that memory was initialized by KASAN. */
		if (kasan_has_integrated_init())
			init = false;
	} else {
		/*
		 * If memory tags have not been set by KASAN, reset the page
		 * tags to ensure page_address() dereferencing does not fault.
		 */
		for (i = 0; i != 1 << order; ++i)
			page_kasan_tag_reset(page + i);
	}
	/* If memory is still not initialized, initialize it now. */
	if (init)
		kernel_init_pages(page, 1 << order);

	set_page_owner(page, order, gfp_flags);
	page_table_check_alloc(page, order);
}
```

```c
/* If memory is still not initialized, initialize it now. */
if (init)
  kernel_init_pages(page, 1 << order);
```

```
static void kernel_init_pages(struct page *page,
                int numpages)
{
  int i;
  /* s390's use of memset() could
     override KASAN redzones. */
  kasan_disable_current();
  for (i = 0; i < numpages; i++)
    clear_highpage_kasan_tagged(page + i);
  kasan_enable_current();
}

static inline void clear_highpage_kasan_tagged(
                struct page *page)
{
  void *kaddr = kmap_local_page(page);

  clear_page(kasan_reset_tag(kaddr));
  kunmap_local(kaddr);
}
```

```
SYM_FUNC_START(__pi_clear_page)
  mrs x1, dczid_el0
  tbnz  x1, #4, 2f /* Branch if DC ZVA is prohibited */
  and w1, w1, #0xf
  mov x2, #4
  lsl x1, x2, x1

1:  dc  zva, x0
  add x0, x0, x1
  tst x0, #(PAGE_SIZE - 1)
  b.ne  1b
  ret

2:  stnp  xzr, xzr, [x0]
  stnp  xzr, xzr, [x0, #16]
  stnp  xzr, xzr, [x0, #32]
  stnp  xzr, xzr, [x0, #48]
  add x0, x0, #64
  tst x0, #(PAGE_SIZE - 1)
  b.ne  2b
  ret
```

```
This is the Linux-penguin again...

Originally drawn by Larry Ewing (http://www.isc.tamu.edu/~lewing/)
(with the GIMP) the Linux Logo has been vectorized by me (Simon Budig,
http://www.home.unix-ag.org/simon/).

This happened quite some time ago with Corel Draw 4. But luckily
meanwhile there are tools available to handle vector graphics with
Linux. Bernhard Herzog (bernhard@users.sourceforge.net) deserves kudos
for creating Sketch (http://sketch.sourceforge.net), a powerful free
tool for creating vector graphics. He converted the Corel Draw file to
the Sketch native format. Since I am unable to maintain the Corel Draw
file any longer, the Sketch version now is the "official" one.

Anja Gerwinski (anja@gerwinski.de) has created an alternate version of
the penguin (penguin-variant.sk) with a thinner mouth line and slightly
altered gradients. It also features a nifty drop shadow.

The third bird (penguin-flat.sk) is a version reduced to three colors
(black/white/yellow) for e.g. silk screen printing. I made this version
for a mug, available at the friendly folks at
http://www.kernelconcepts.de/ - they do good stuff, mail Petra
(pinguin@kernelconcepts.de) if you need something special or don't
understand the german  :-)

These drawings are copyrighted by Larry Ewing and Simon Budig
(penguin-variant.sk also by Anja Gerwinski), redistribution is free but
has to include this README/Copyright notice.

The use of these drawings is free. However I am happy about a sample of
your mug/t-shirt/whatever with this penguin on it...

Have fun
    Simon Budig


Simon.Budig@unix-ag.org
http://www.home.unix-ag.org/simon/

Simon Budig
Am Hardtkoeppel 2
D-61279 Graevenwiesbach
```

```
unsigned long do_mmap(struct file *file,
     unsigned long addr,
     unsigned long len,
     unsigned long prot,
     unsigned long flags,
     vm_flags_t vm_flags,
     unsigned long pgoff,
     unsigned long *populate,
     struct list_head *uf)
{
  //  ...
  /* clear anonymous mappings that don't ask
     for uninitialized data */
  if (!vma->vm_file &&
      (!IS_ENABLED(CONFIG_MMAP_ALLOW_UNINITIALIZED) ||
       !(flags & MAP_UNINITIALIZED)))
   memset((void *)region->vm_start, 0,
          region->vm_end - region->vm_start);
  //  ...
}
```

“but what about MAP_UNINITIALIZED?”

woven by TOYOTA

Why not just zero the free store?

```
calloc();

     new char[128]();
```

Why not just zero the free store?

jemalloc

opt.junk (const char *) r- [--enable-fill]
Junk filling. If set to “alloc”, each byte of uninitialized allocated memory will be initialized to 0xa5. If set to “free”, all deallocated memory will be initialized to 0x5a. If set to “true”, both allocated and deallocated memory will be initialized, and if set to “false”, junk filling be disabled entirely. This is intended for debugging and will impact performance negatively. This option is “false” by default unless --enable-debug is specified during configuration, in which case it is “true” by default.

opt.zero (bool) r- [--enable-fill]
Zero filling enabled/disabled. If enabled, each byte of uninitialized allocated memory will be initialized to 0. Note that this initialization only happens once for each byte, so realloc() and rallocx() calls do not zero memory that was previously allocated. This is intended for debugging and will impact performance negatively. This option is disabled by default.

Why not just zero the free store?

```
unsafe fn alloc_zeroed(&self, layout: Layout) -> *mut u8
```

Behaves like alloc, but also ensures that the contents are set to zero before being returned.

This function is unsafe for the same reasons that alloc is. However the allocated block of memory is guaranteed to be initialized.

Why even have a free store?

Could I use these values?

```c
void srandomdev() {
        int fd, done;
        size_t len;

        if (rand_type == TYPE_0)
                len = sizeof state[0];
        else
                len = rand_deg * sizeof state[0];

        done = 0;
        fd = _open("/dev/random", O_RDONLY, 0);
        if (fd >= 0) {
                if (_read(fd, (void *) state, len) == (ssize_t) len)
                        done = 1;
                _close(fd);
        }

        if (!done) {
                struct timeval tv;
                unsigned long junk;

                gettimeofday(&tv, NULL);
                srandom((getpid() << 16) ^ tv.tv_sec ^ tv.tv_usec ^ junk);
                return;
        }

        if (rand_type != TYPE_0) {
                fptr = &state[rand_sep];
                rptr = &state[0];
        }
}
```

Could I use this?

```c
int get_hw_address(struct device *dev,
    struct user *usr) {

  unsigned char addr[MAX_ADDR_LEN]; // Leak this
  if (!dev->has_address)
    return -EOPNOTSUPP;
  dev->get_hw_address(addr); // doesn’t fill addr
  return copy_out(usr, addr, sizeof(addr));
}
```

```
int queue_manage() {
  struct async_request *backlog;       // Uninitialized

  if (engine->state == IDLE)           // Usually true
    backlog = get_backlog(&engine->queue);

  if (backlog)
    backlog->complete(backlog, -EINPROGRESS); // Oops

  return 0;
}
```

```
-ftrivial-auto-var-init=zero
```

HEAP

UaF

GC?

borrow checker?

GWP-asan

secret allocator

not secret allocator

Could we do more?

- PartitionAlloc
- isoalloc
- Gigacage
- isoheap
- XNU memory safety
- CHERI
- MTE

write your own allocator

woven by TOYOTA

https://woven.toyota

→ we make mobility software
→ we're hiring

Tokyo
Palo Alto, CA
Seattle, WA
Ann Arbor, MI
Brooklyn, NY

どうもありがとうございました

The Bytes Before the Types

Unveiling Uninitialized Uses

JF Bastien
Distinguished Engineer, Woven by Toyota
Chair, WG21 C++ evolution working group

jfb@woven.toyota
jfbastien.com
@jfbastien

ジェイエフ バスティエン
Distinguished エンジニア、ウーブン・バイ・トヨタ
議長、WG21 C++ 進化作業グループ