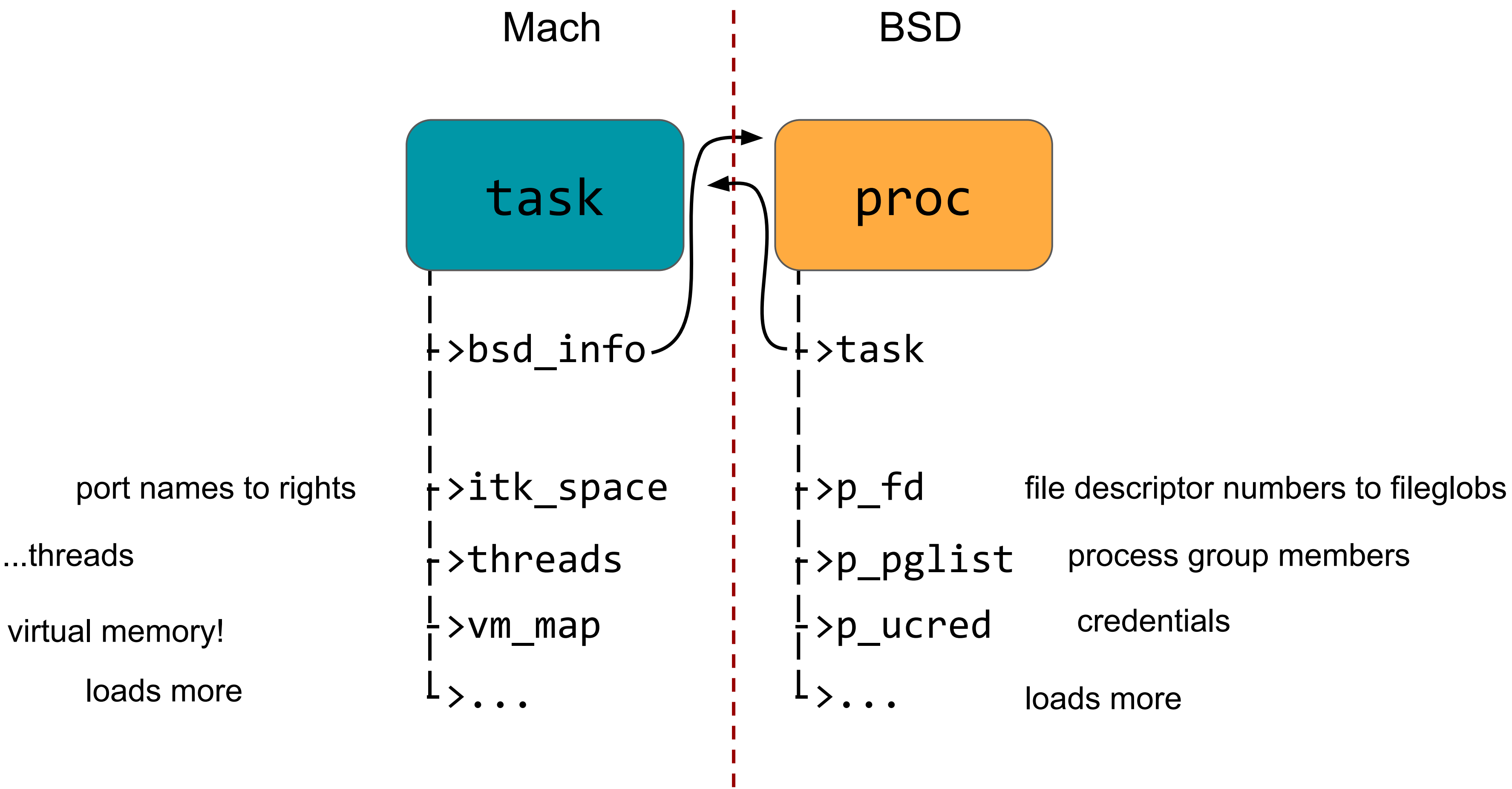
vm_map'ing out XNU Virtual Memory

@i41nbeer

XNU process

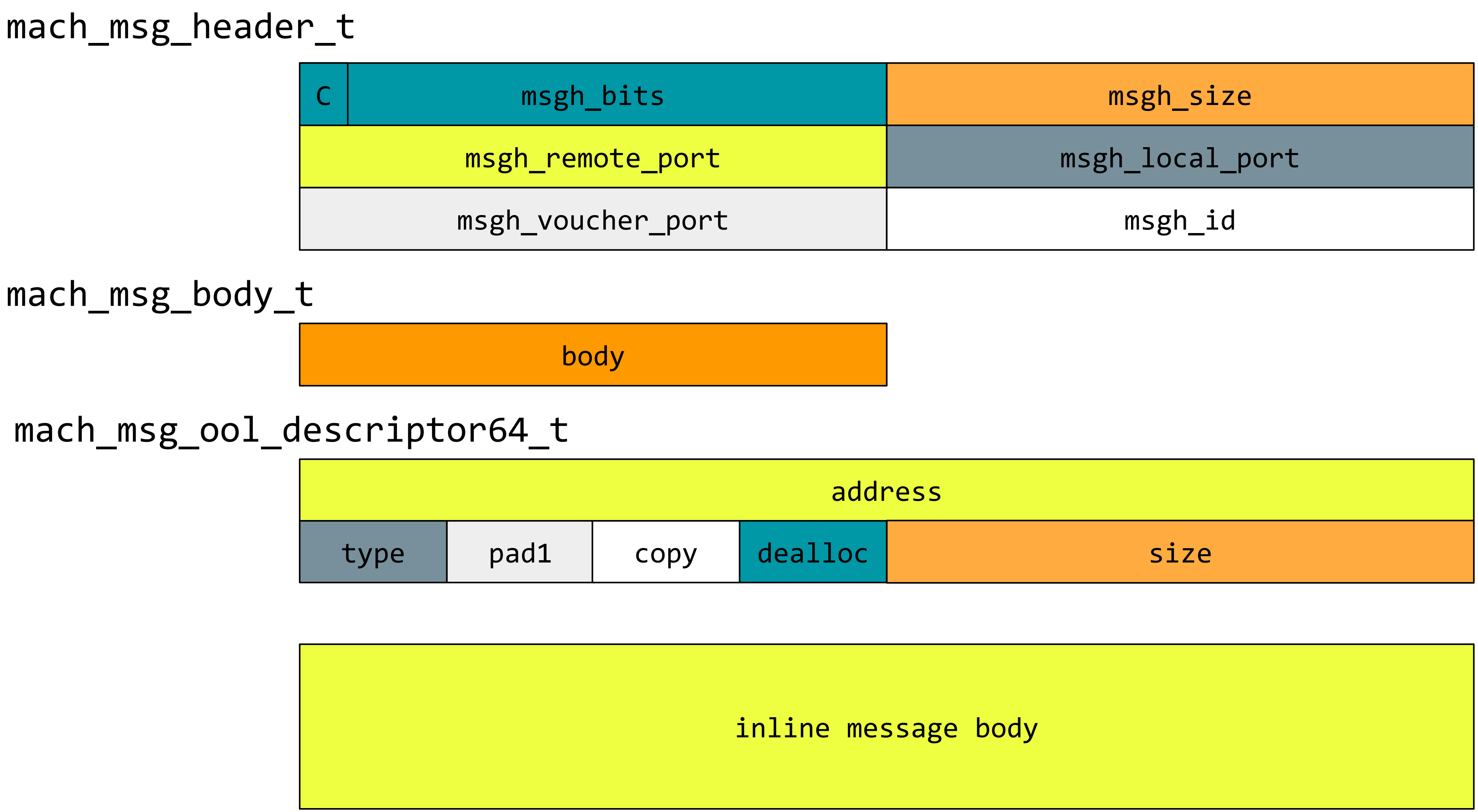

Mach
BSD
task
proc
->bsd_info
->task
port names to rights
->itk_space
->p_fd
file descriptor numbers to fileglobs
...threads
->threads
->p_pglist
process group members
virtual memory!
->vm_map
->p_ucred
credentials
loads more
->...
->...
loads more

Message Passing

fundamental microkernel paradigm; split up functionality and send messages between low-privilege tasks, as independently as possible

Thought to be major performance bottleneck for microkernel architectures; XNU is no microkernel but still retains Mach's solution to this problem

This talk: how Mach uses virtual memory tricks to make passing large messages fast, and how it was broken


mach_msg_header_t
C
msgh_bits
msgh_size
msgh_remote_port
msgh_local_port
msgh_voucher_port
msgh_id
mach_msg_body_t
body
mach_msg_ool_descriptor64_t
address
type
pad1
copy
dealloc
size
inline message body

mach_msg_header_t

This entire structure is copied into kernel memory each time a message is sent, then copied out to a userspace process when a message is received

Lots of memory copying, and OS written in an era when memory copying was slooooow

Wanted to avoid copying large amounts of data
-> move it to the ool_desc!
-> use virtual memory magic to move it for free!

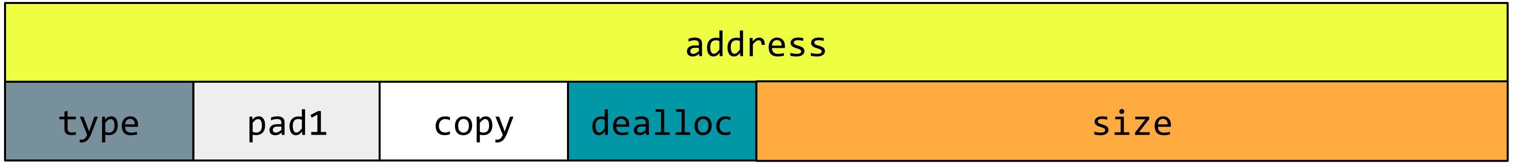
mach_msg

mach_ms

inline message body

A Mach virtual memory trick:

`mach_msg_ool_descriptor64_t`



setting:

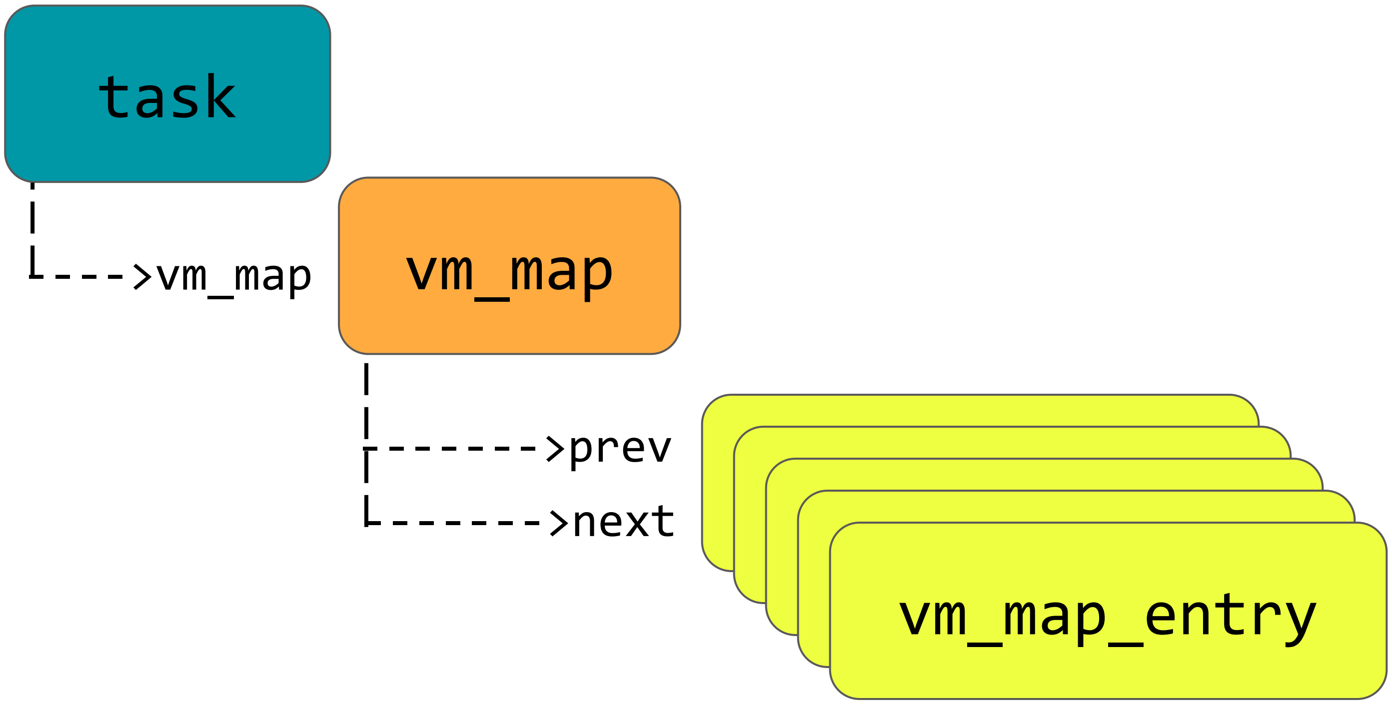
```
copy    = 0;
dealloc = 1;
```

implies:

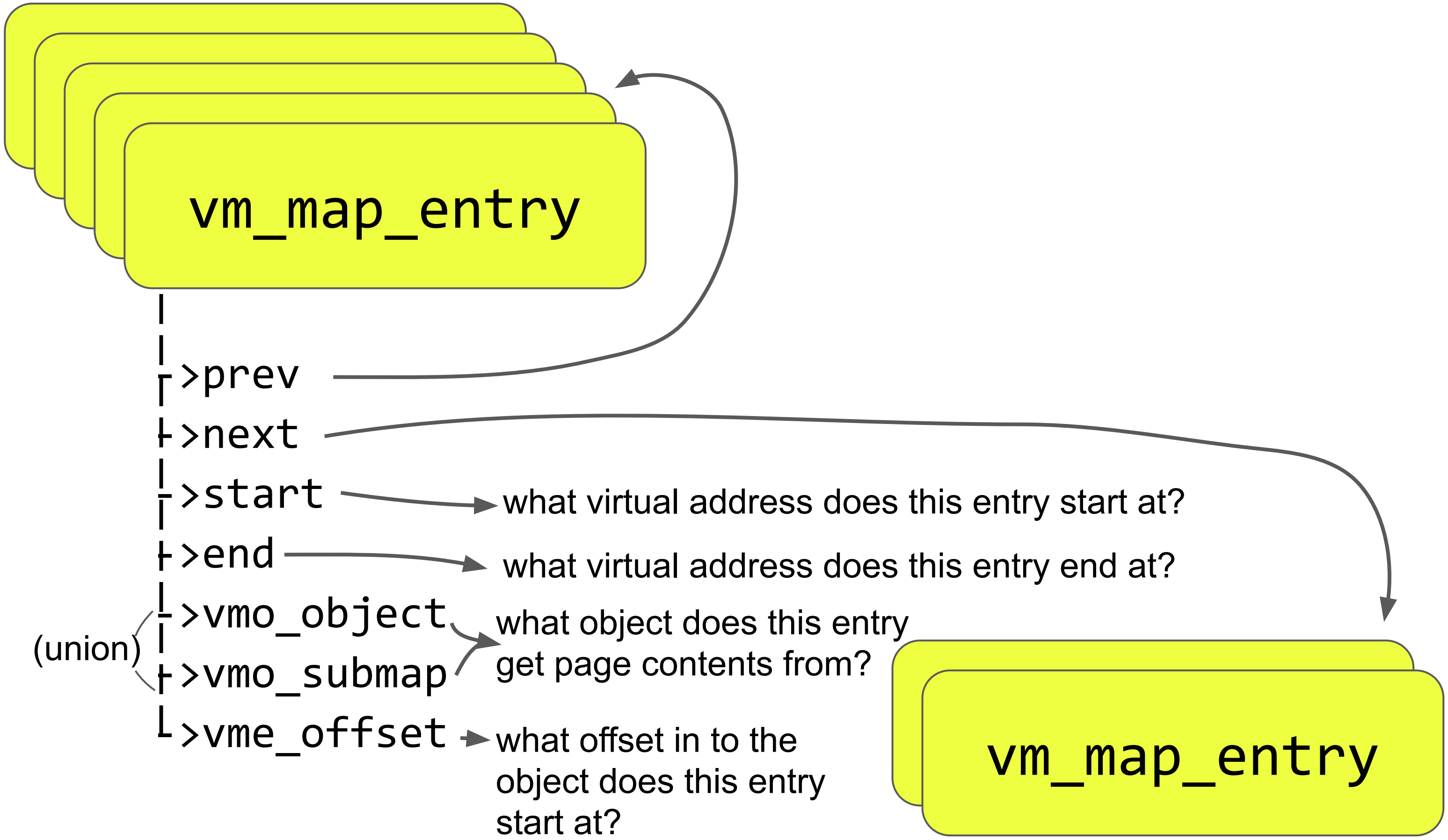
MOVE the virtual memory region at address to address+size from the current process to the recipient

Idea is to speed up sending and receiving large messages by replacing memory copies with virtual memory manipulation to move pages between processes

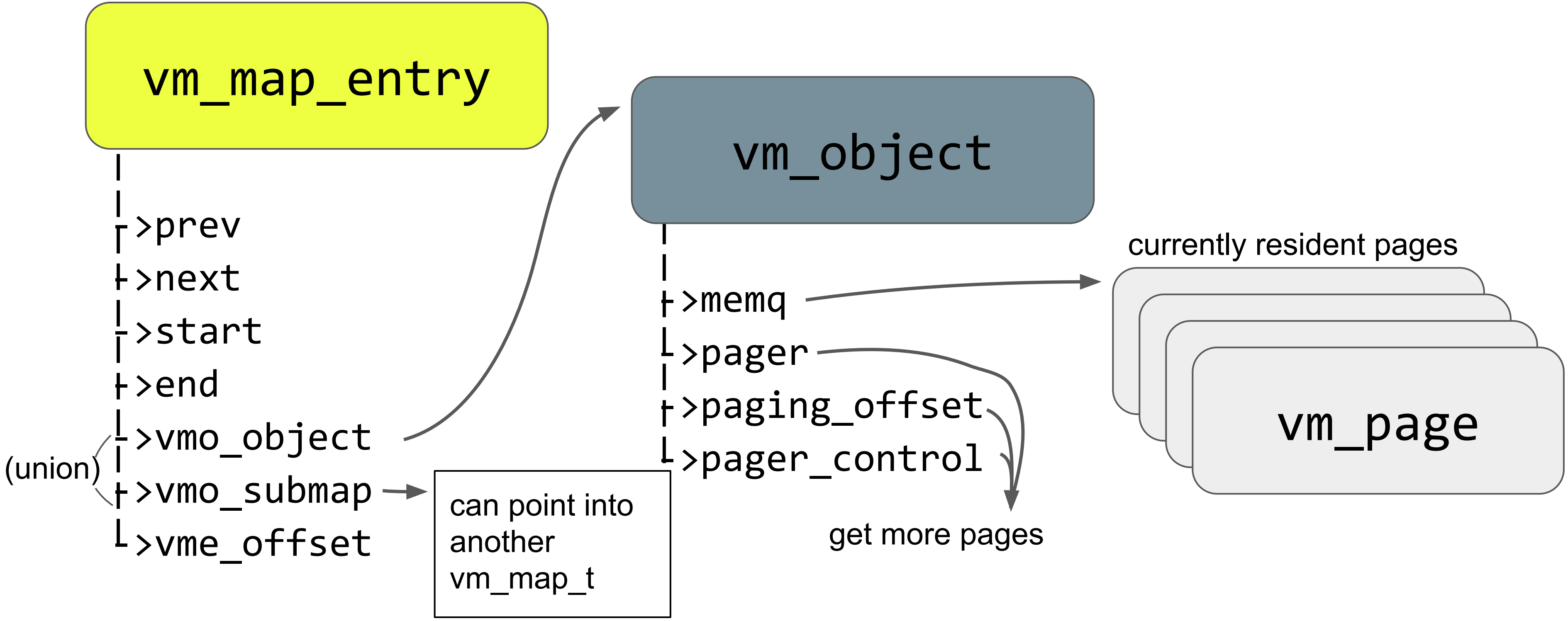
Mach VM zoo:



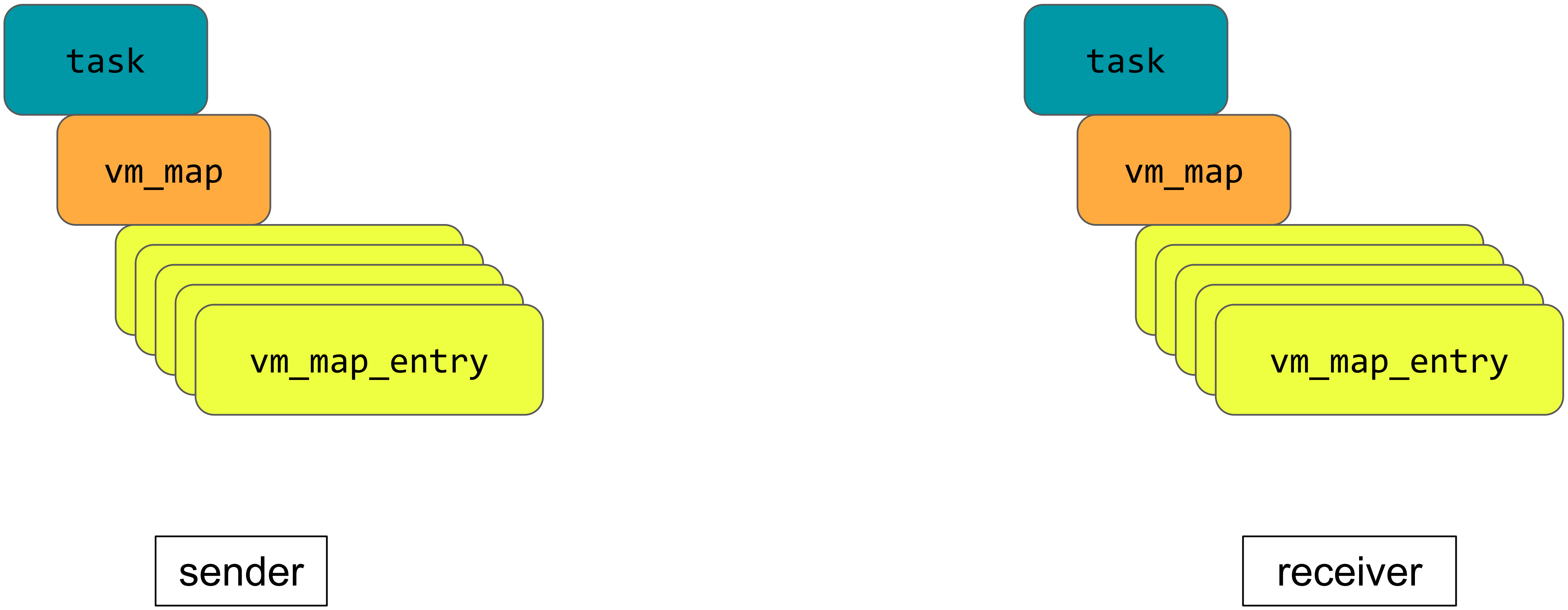
struct vm_map_entry



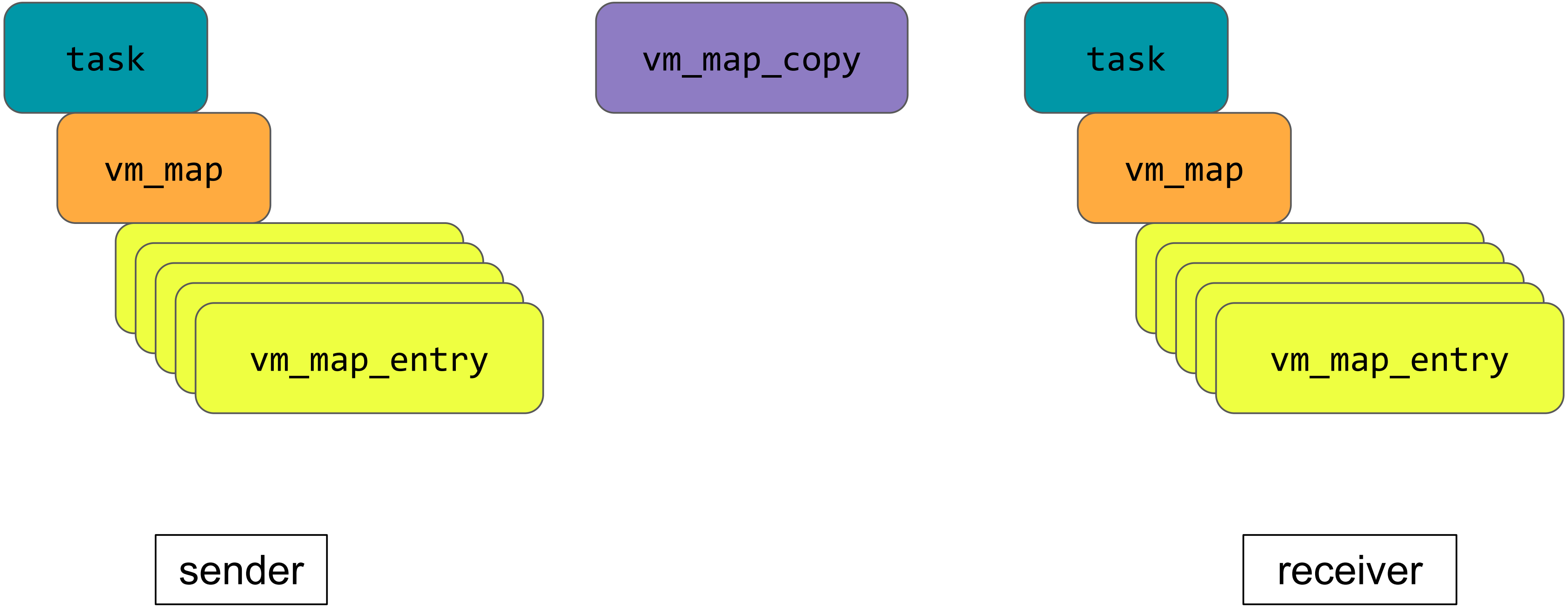
struct vm_object



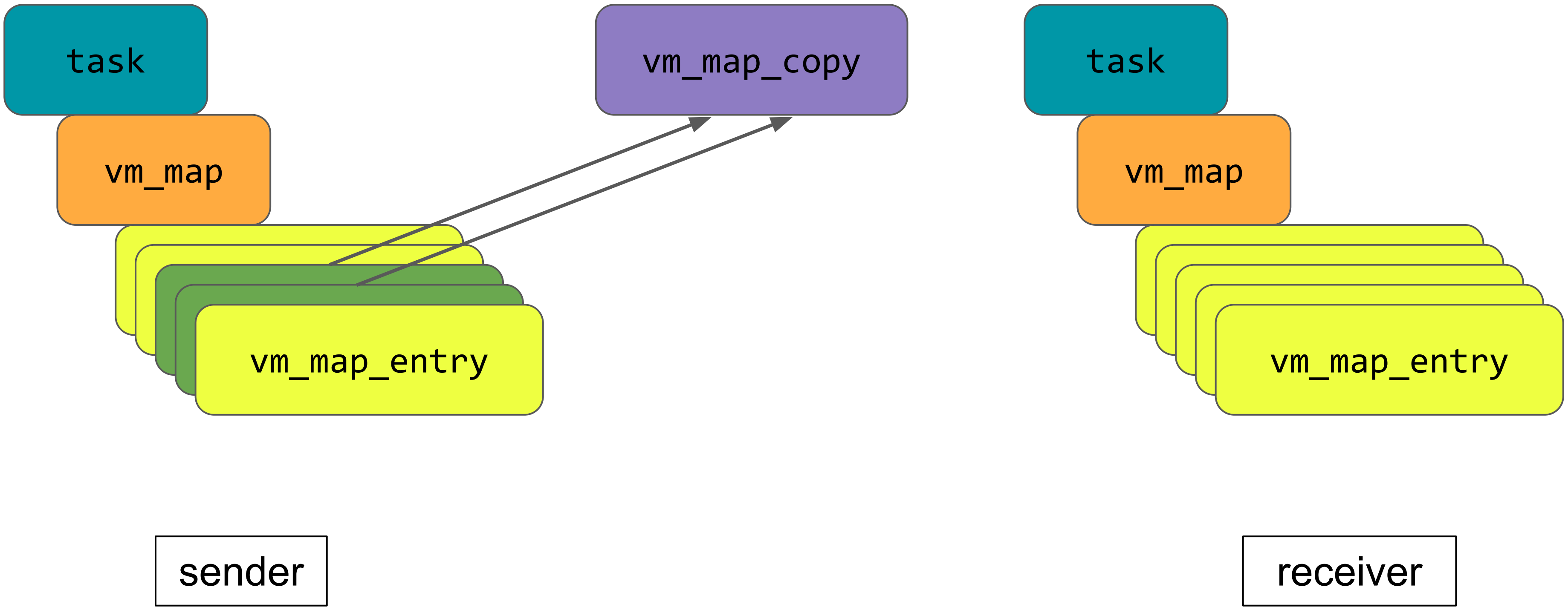
optimized entry MOVE:



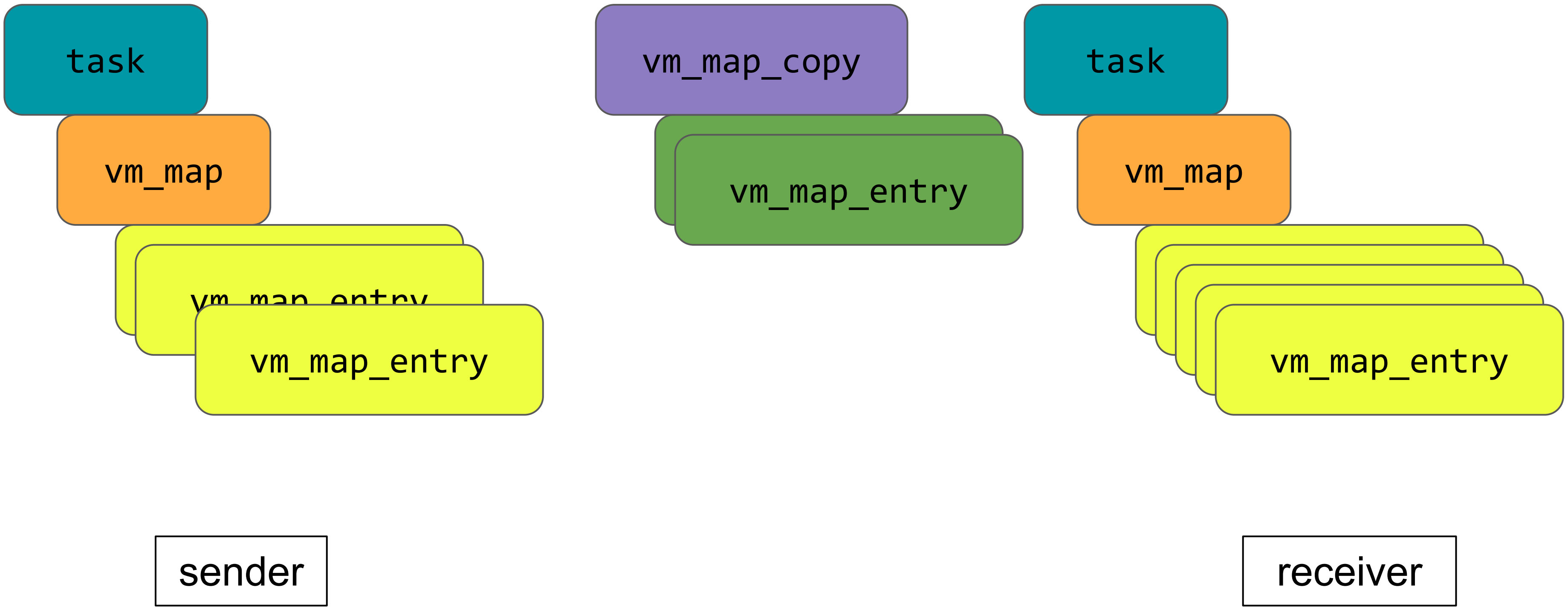
optimized entry MOVE:



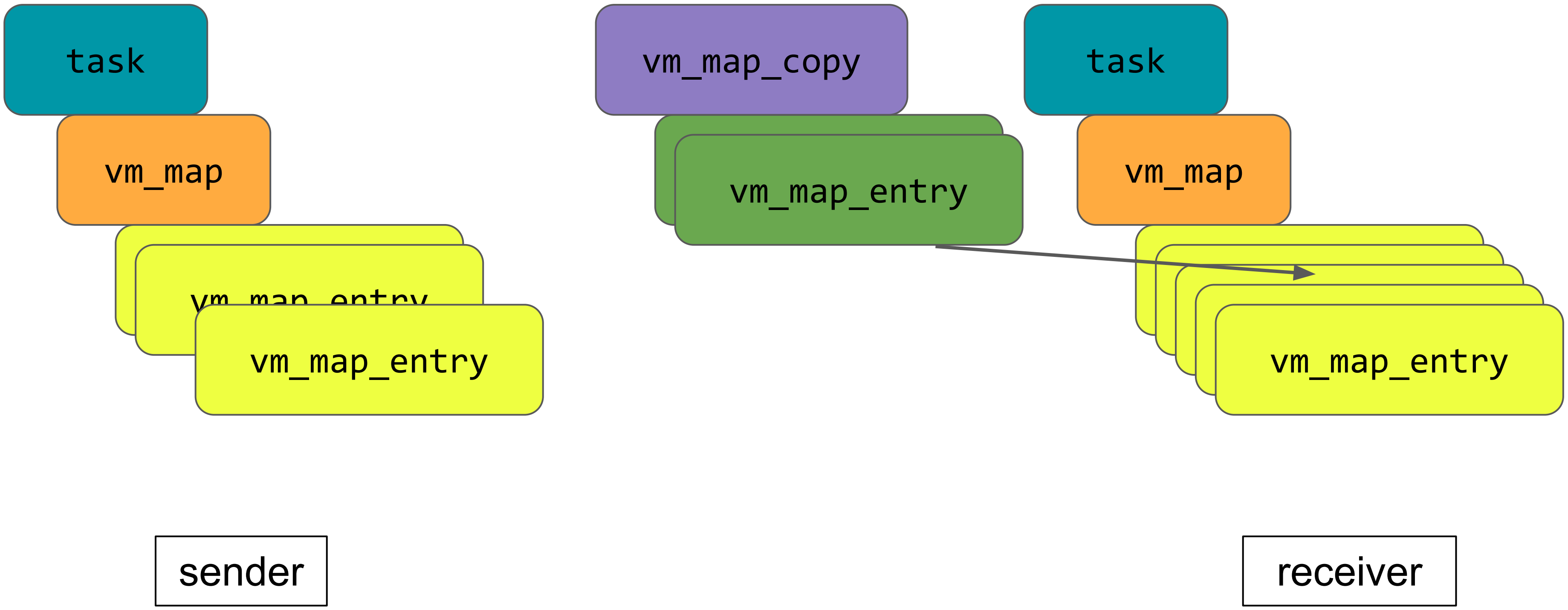
optimized entry MOVE:



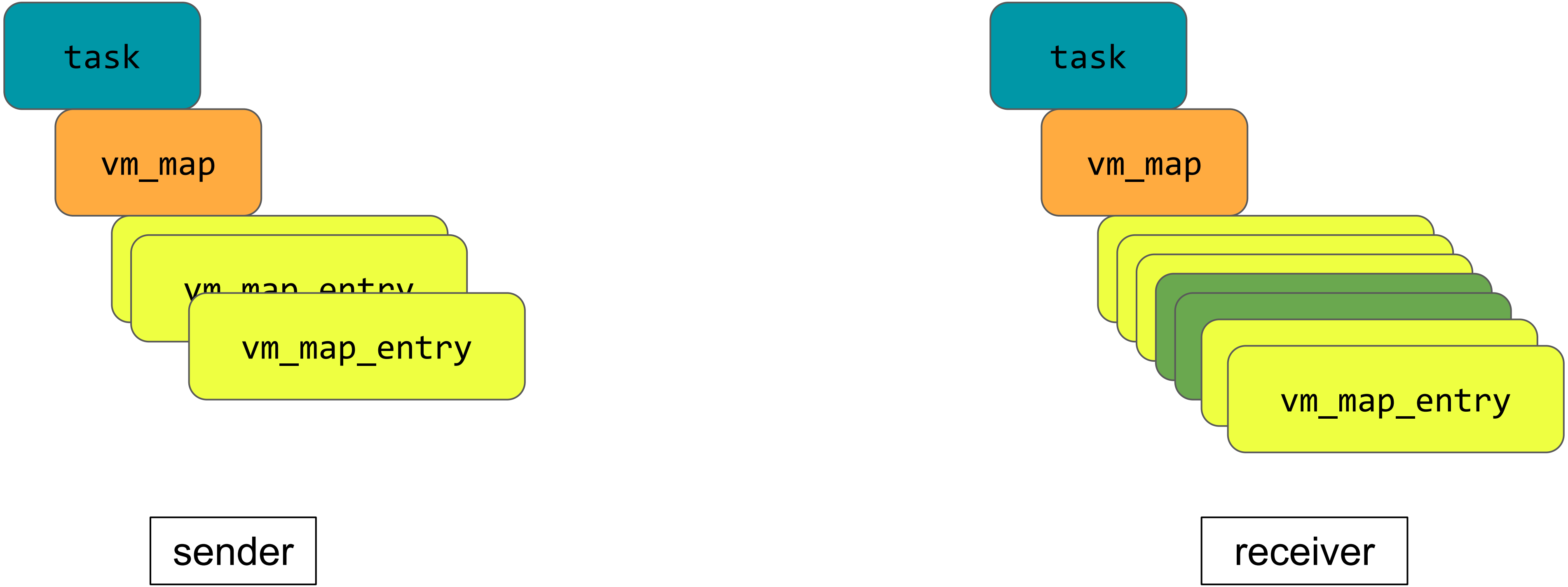
optimized entry MOVE:



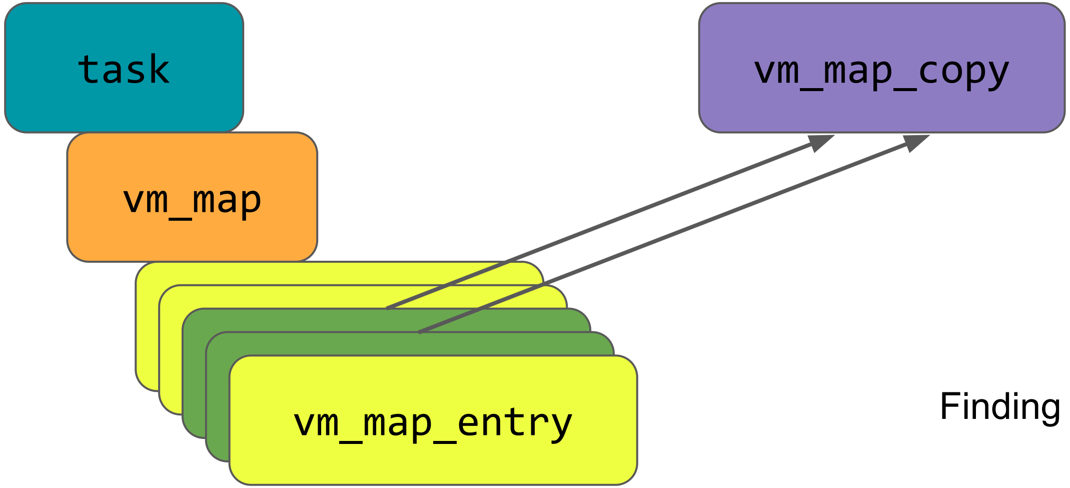
optimized entry MOVE:



optimized entry MOVE:

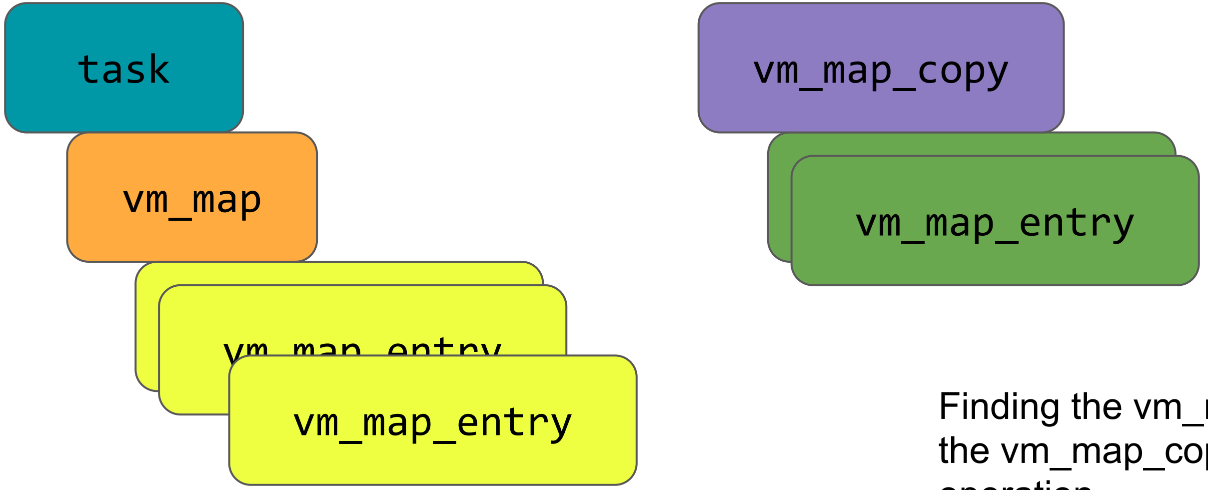


The bug:



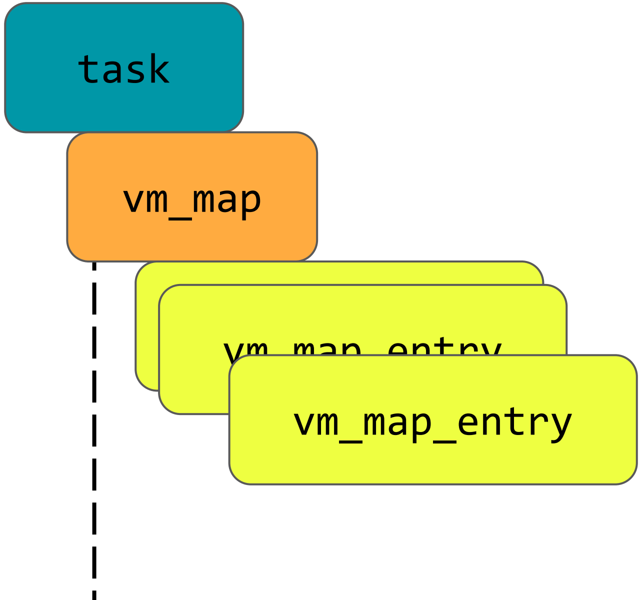
Finding the vm_map_entry's

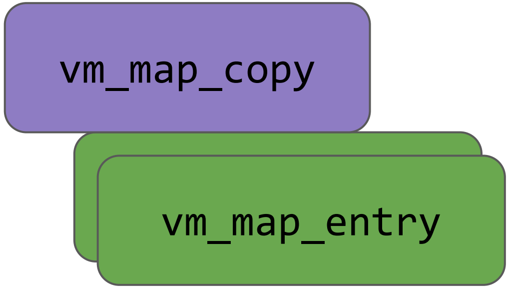
The bug:



Finding the vm_map_entry's, and copying them to the vm_map_copy structure needs to be an atomic operation.

The bug:



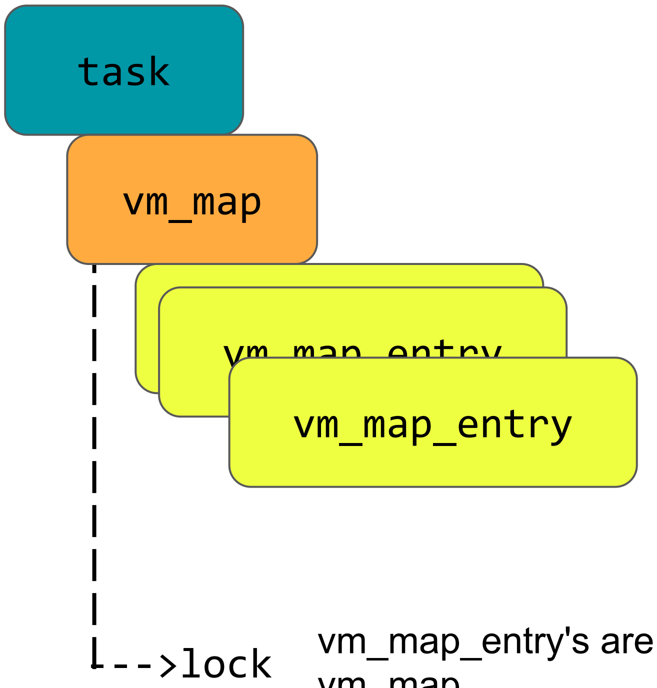


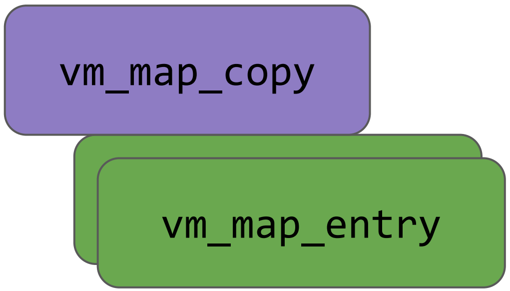
Finding the vm_map_entry's, and copying them to the vm_map_copy structure needs to be an atomic operation.

--->lock

vm_map_entry's are owned by the vm_map,
ownership enforced by vm_map->lock

The bug:





vm_map_entry's are owned by the vm_map,
ownership enforced by vm_map->lock

Finding the vm_map_entry's, and copying them to the vm_map_copy structure needs to be an atomic operation.

doing ANYTHING with a vm_map_entry without holding its vm_map lock is almost certainly wrong in a very bad way

(Reading the code I get the feeling someone at Apple audited for this anti-pattern, good job!)

An aside on locking in the VM subsystem...

Avoiding deadlocks is a hard problem...

```
#define vm_map_lock(map)    lck_rw_lock_exclusive(&(map)->lock)

#define vm_map_unlock(map)  \
  ((map)->timestamp++ ,   lck_rw_done(&(map)->lock))
```

`uint32_t..`

err...

Example use of vm_map.timestamp:

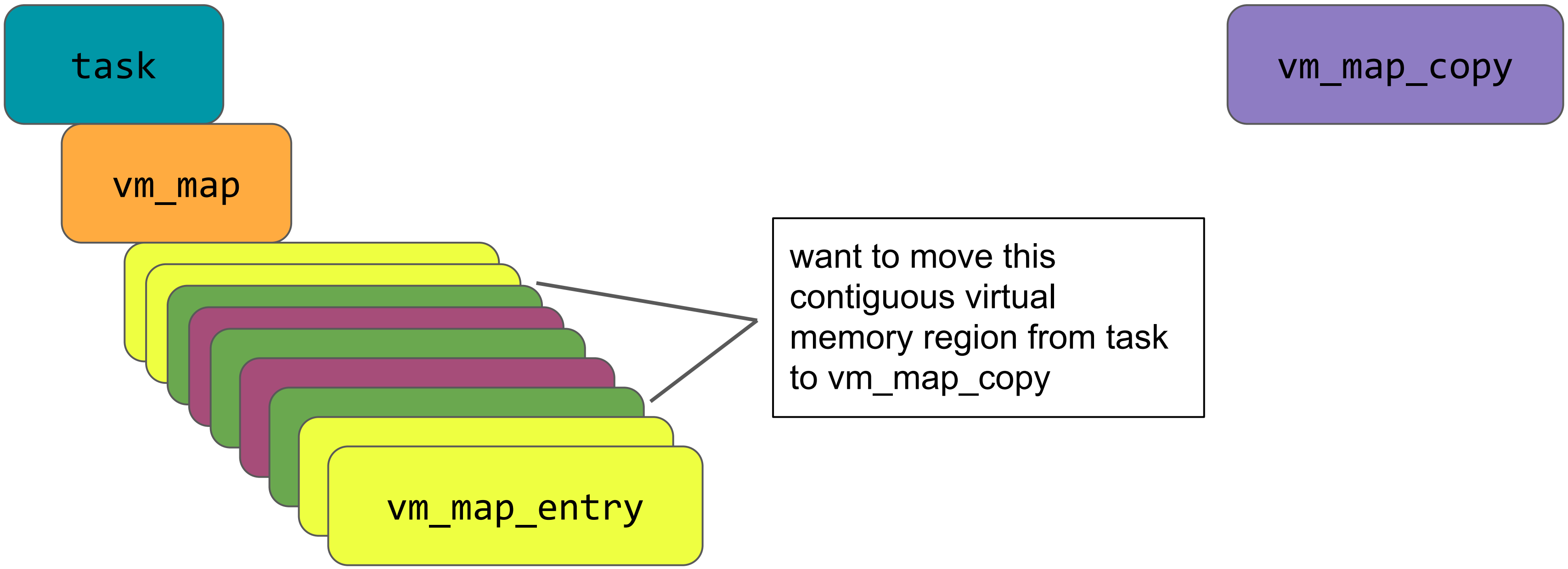
```
last_timestamp = map->timestamp;
...
vm_map_unlock(map);

...

vm_map_lock(map);

if (last_timestamp+1 != map->timestamp) {
  /*
   * Find the entry again.  It could have
   * been clipped after we unlocked the map.
   */
   if (!vm_map_lookup_entry(map, s, &first_entry)){
    ...
```

In this window, do stuff which requires map to be unlocked (eg kalloc allocation)

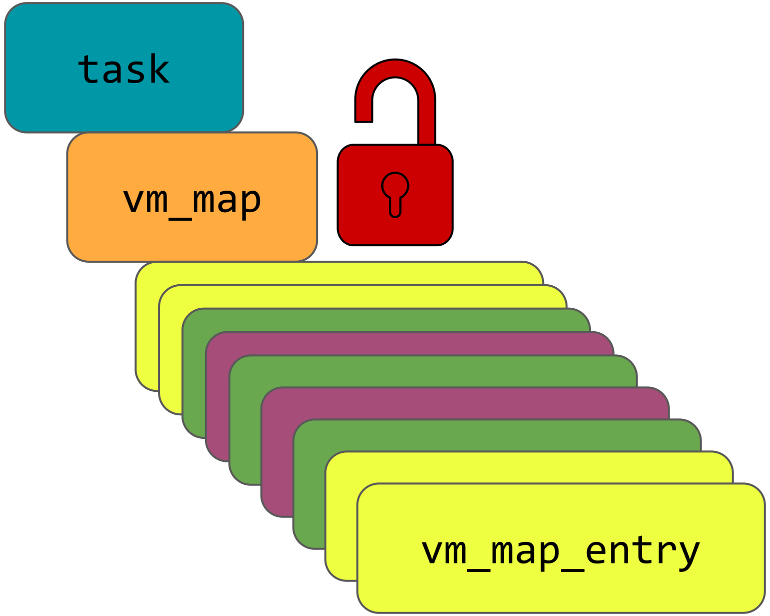
Did someone else take and drop the vm_map's lock while we dropped it?

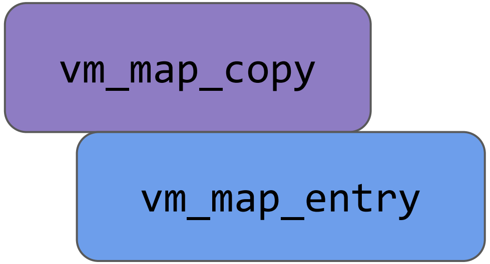
Yes? let's reset our expectations about the state of the world then...

More accurately...



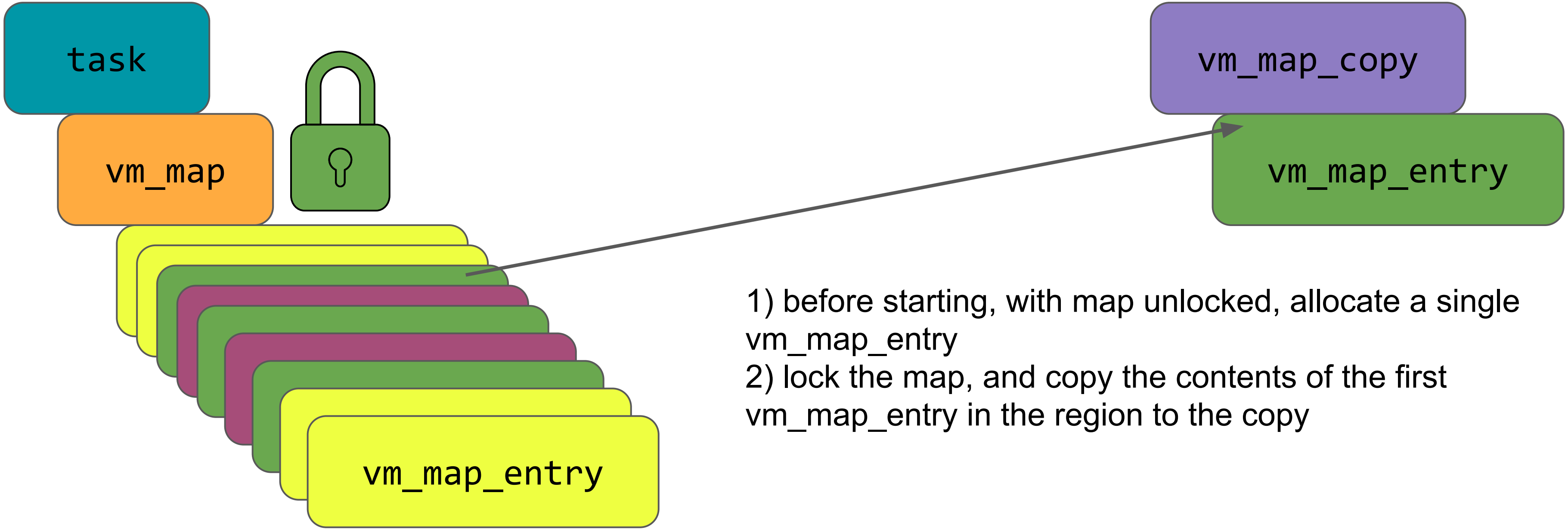
More accurately...





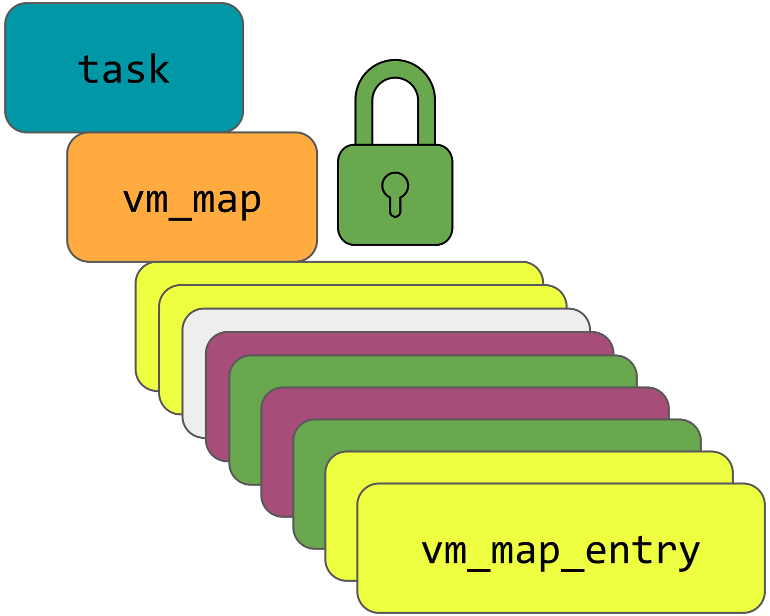
1) before starting, with map unlocked, allocate a single vm_map_entry

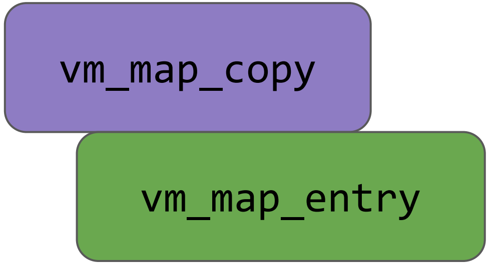
More accurately...



1) before starting, with map unlocked, allocate a single vm_map_entry
2) lock the map, and copy the contents of the first vm_map_entry in the region to the copy

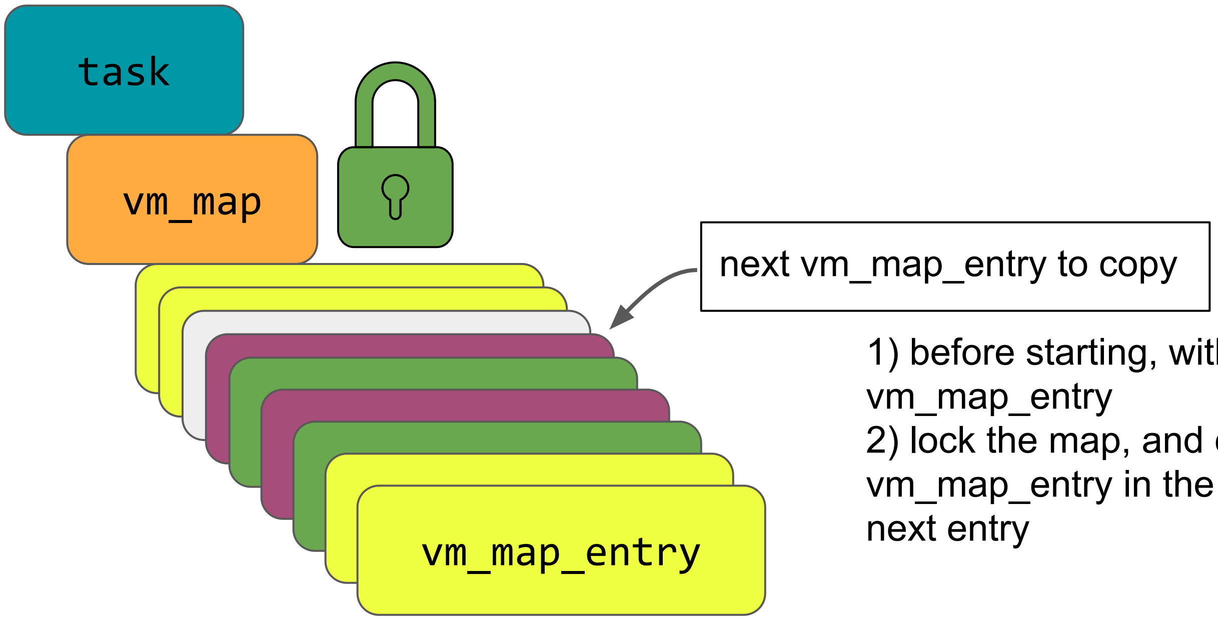
More accurately...

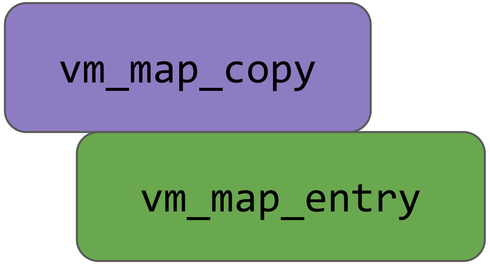




1) before starting, with map unlocked, allocate a single vm_map_entry
2) lock the map, and copy the contents of the first vm_map_entry in the region to the copy

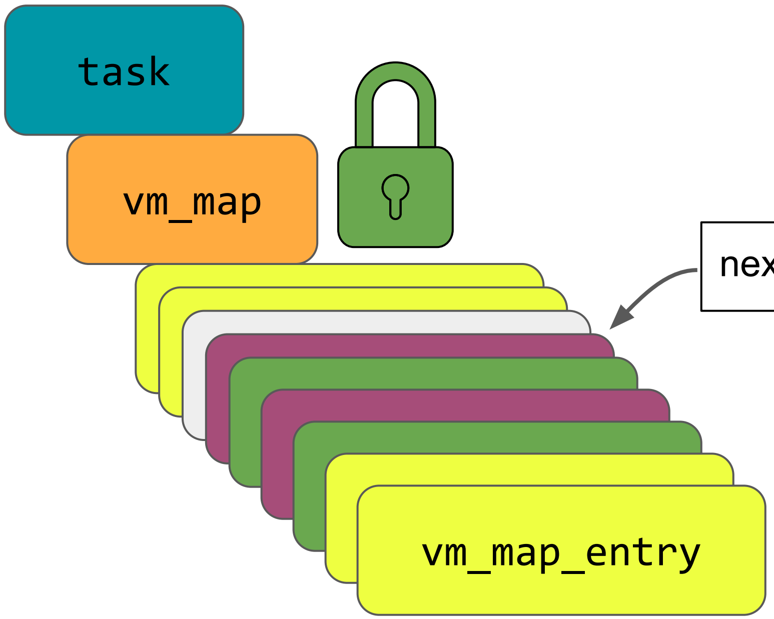
More accurately...

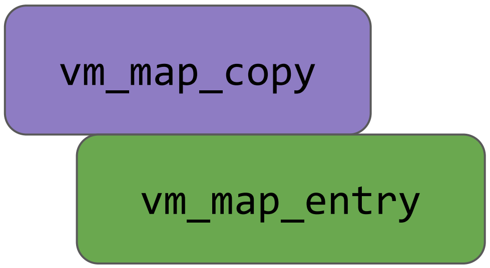




1) before starting, with map unlocked, allocate a single vm_map_entry
2) lock the map, and copy the contents of the first vm_map_entry in the region to the copy, and move to the next entry

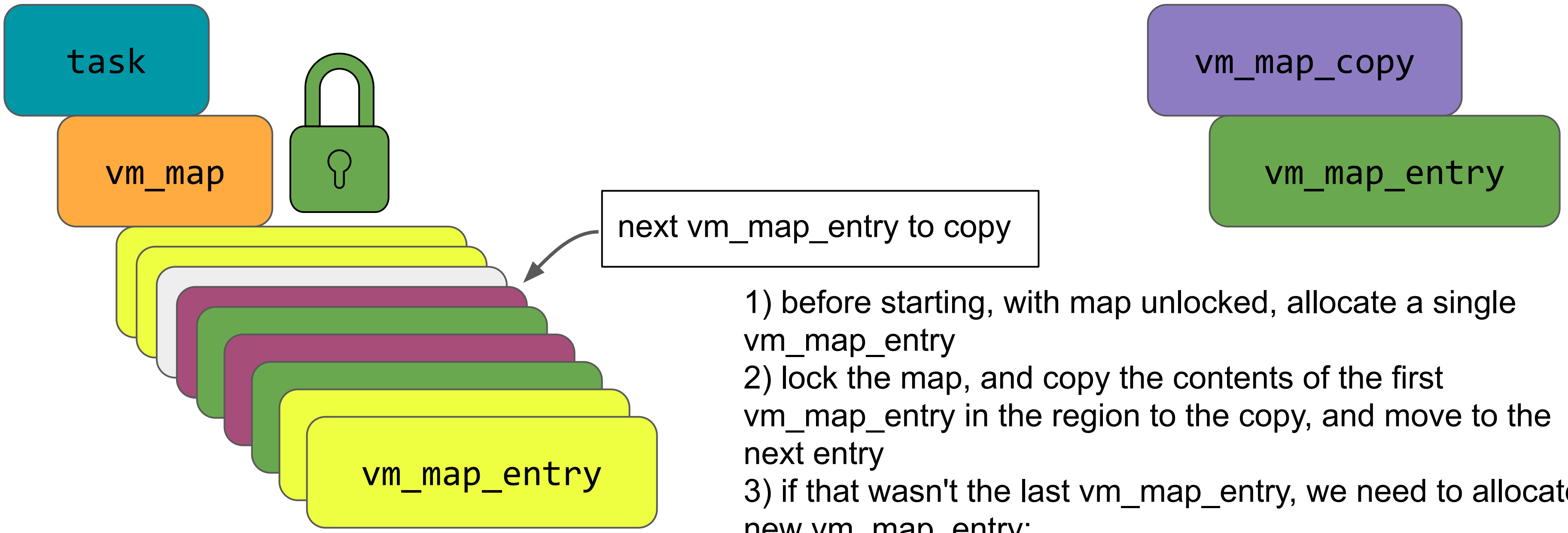
More accurately...





1) before starting, with map unlocked, allocate a single vm_map_entry
2) lock the map, and copy the contents of the first vm_map_entry in the region to the copy, and move to the next entry
3) if that wasn't the last vm_map_entry, we need to allocate a new vm_map_entry:

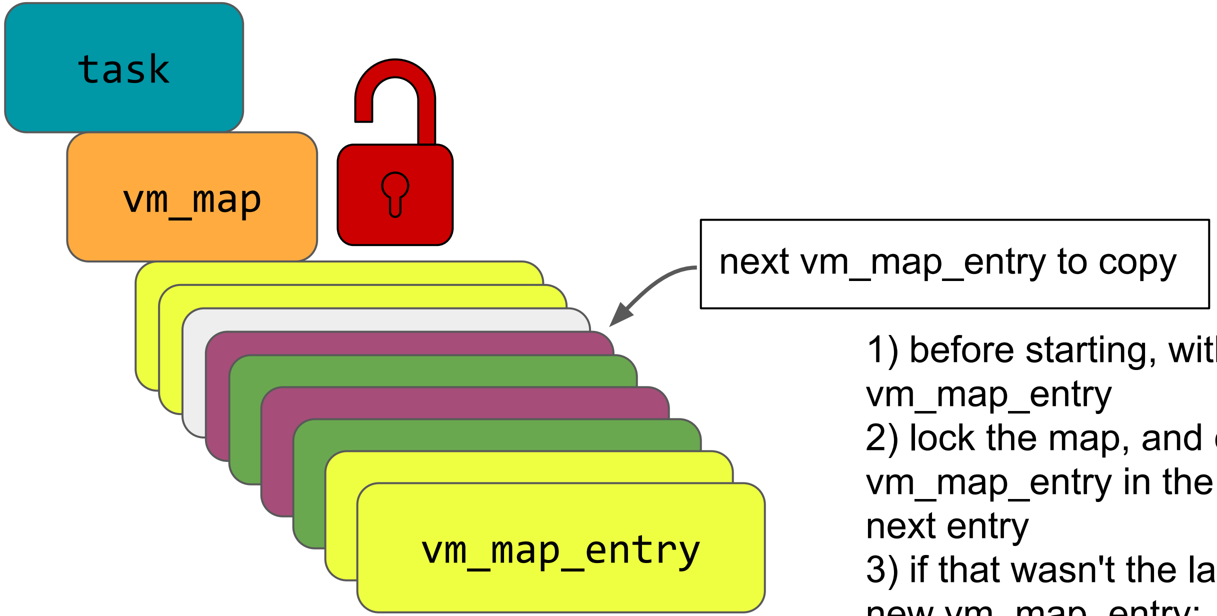
More accurately...

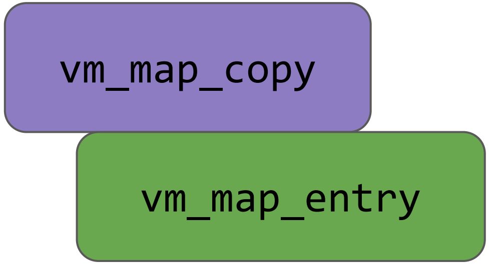


1) before starting, with map unlocked, allocate a single vm_map_entry
2) lock the map, and copy the contents of the first vm_map_entry in the region to the copy, and move to the next entry
3) if that wasn't the last vm_map_entry, we need to allocate a new vm_map_entry:
  3a) record the timestamp

```
ts = 100;
```

More accurately...

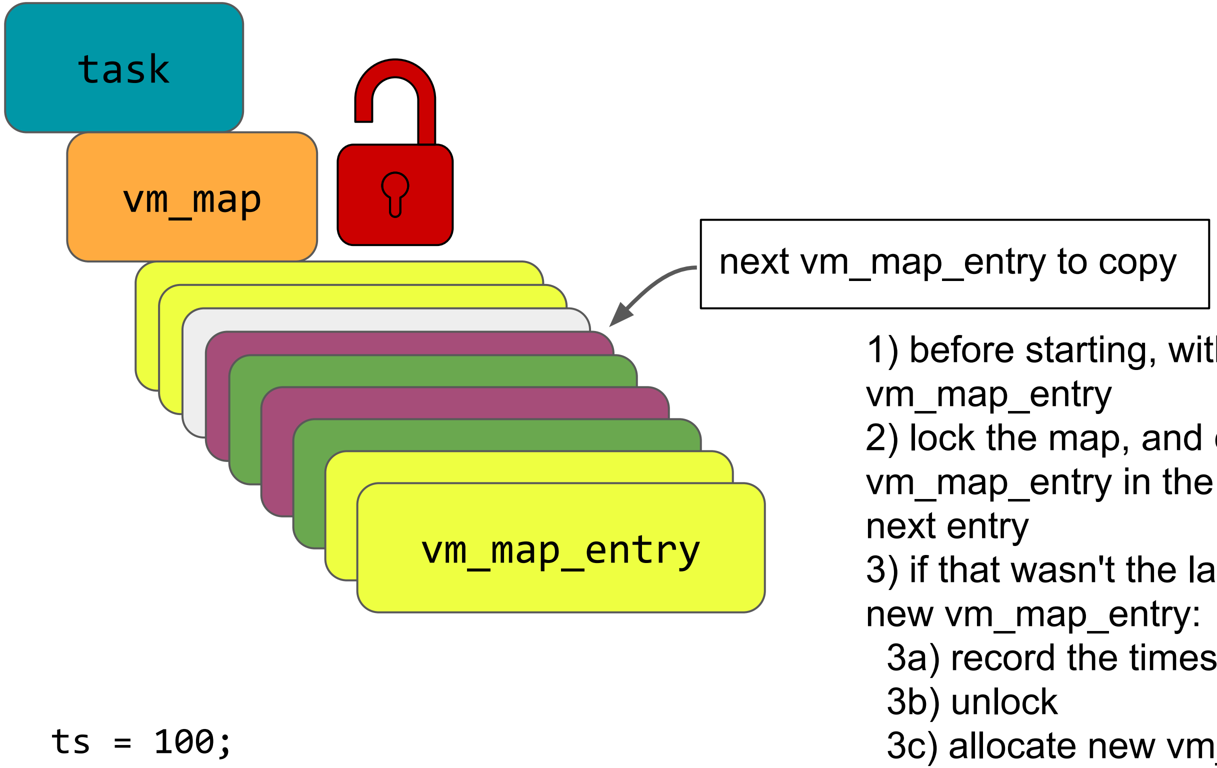


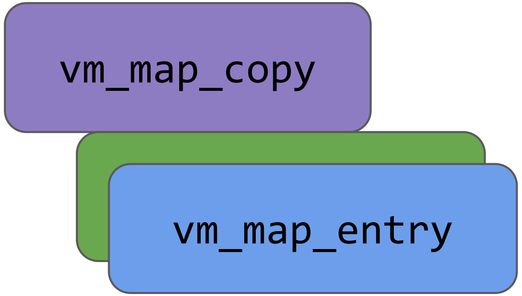


1) before starting, with map unlocked, allocate a single vm_map_entry
2) lock the map, and copy the contents of the first vm_map_entry in the region to the copy, and move to the next entry
3) if that wasn't the last vm_map_entry, we need to allocate a new vm_map_entry:
  3a) record the timestamp
  3b) unlock

```
ts = 100;
```

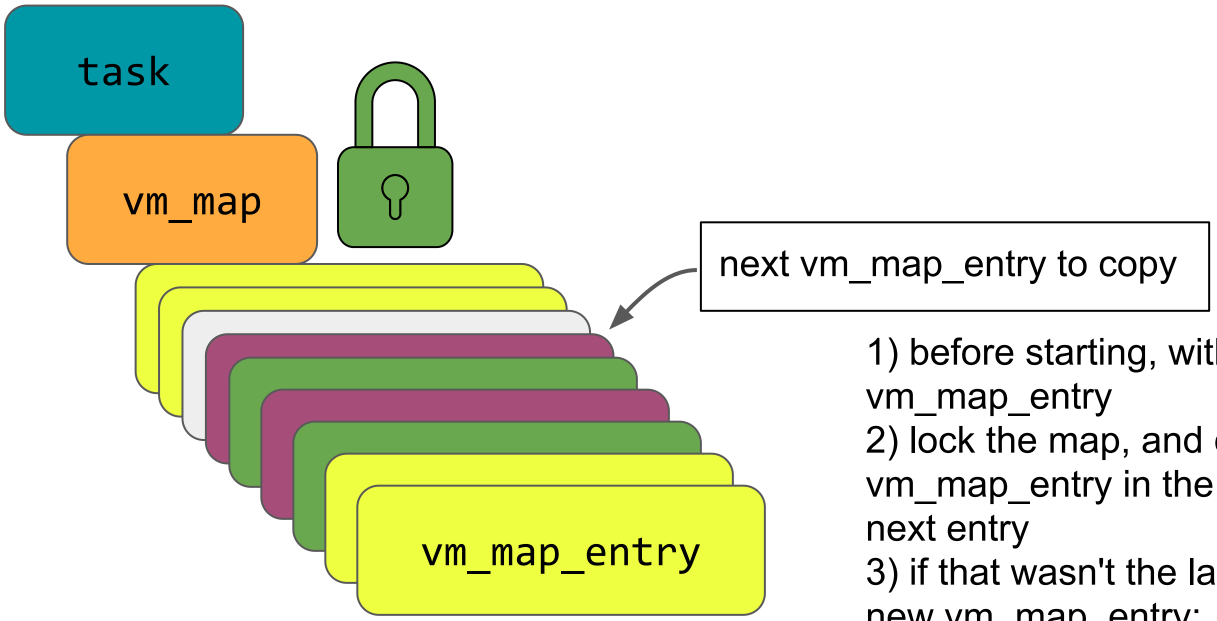
More accurately...

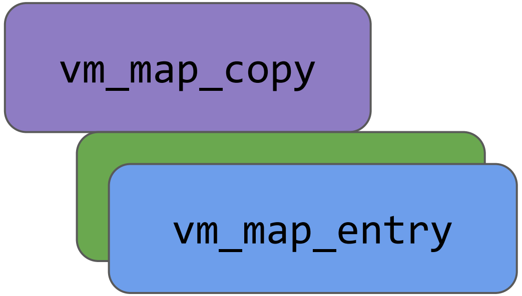




1) before starting, with map unlocked, allocate a single vm_map_entry
2) lock the map, and copy the contents of the first vm_map_entry in the region to the copy, and move to the next entry
3) if that wasn't the last vm_map_entry, we need to allocate a new vm_map_entry:
  3a) record the timestamp
  3b) unlock
  3c) allocate new vm_map_entry for the copy

More accurately...

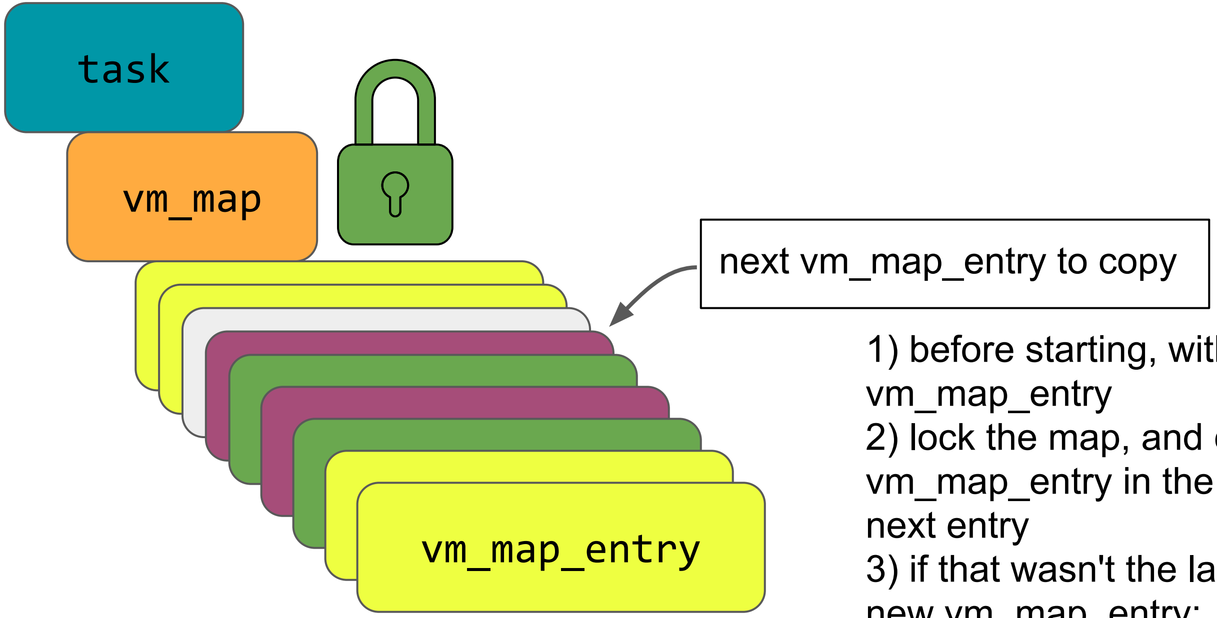


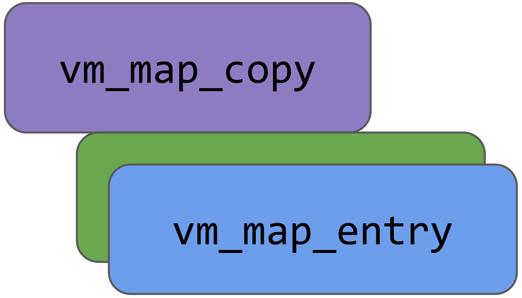


1) before starting, with map unlocked, allocate a single vm_map_entry
2) lock the map, and copy the contents of the first vm_map_entry in the region to the copy, and move to the next entry
3) if that wasn't the last vm_map_entry, we need to allocate a new vm_map_entry:
  3a) record the timestamp
  3b) unlock
  3c) allocate new vm_map_entry for the copy
  3d) lock

```
ts = 100;
```

More accurately...

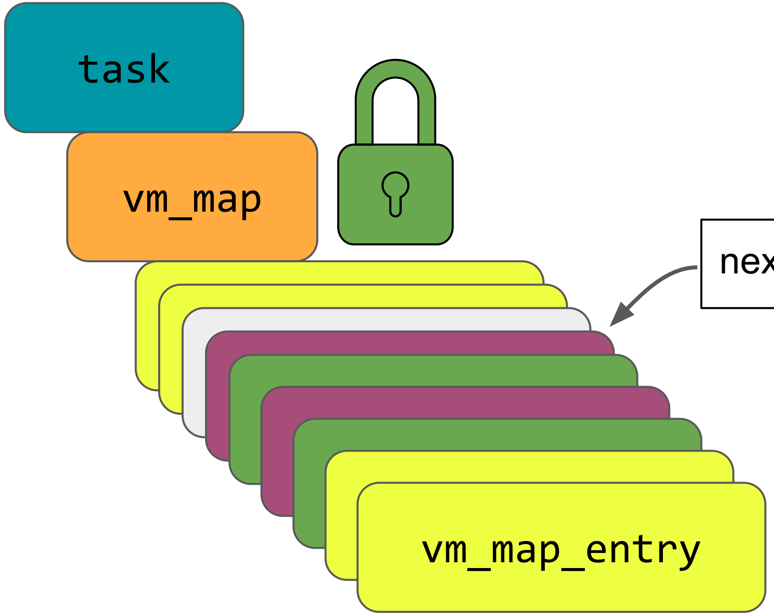




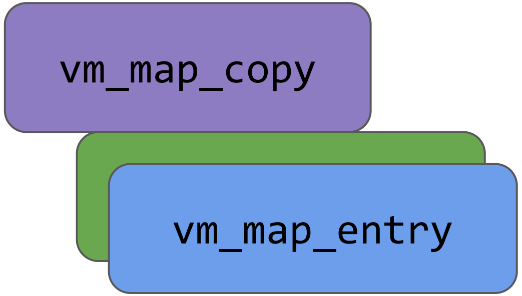

```
ts = 100;
```

1) before starting, with map unlocked, allocate a single vm_map_entry
2) lock the map, and copy the contents of the first vm_map_entry in the region to the copy, and move to the next entry
3) if that wasn't the last vm_map_entry, we need to allocate a new vm_map_entry:
  3a) record the timestamp
  3b) unlock
  3c) allocate new vm_map_entry for the copy
  3d) lock
4) does ts + 1 == current vm_map timestamp?

More accurately...



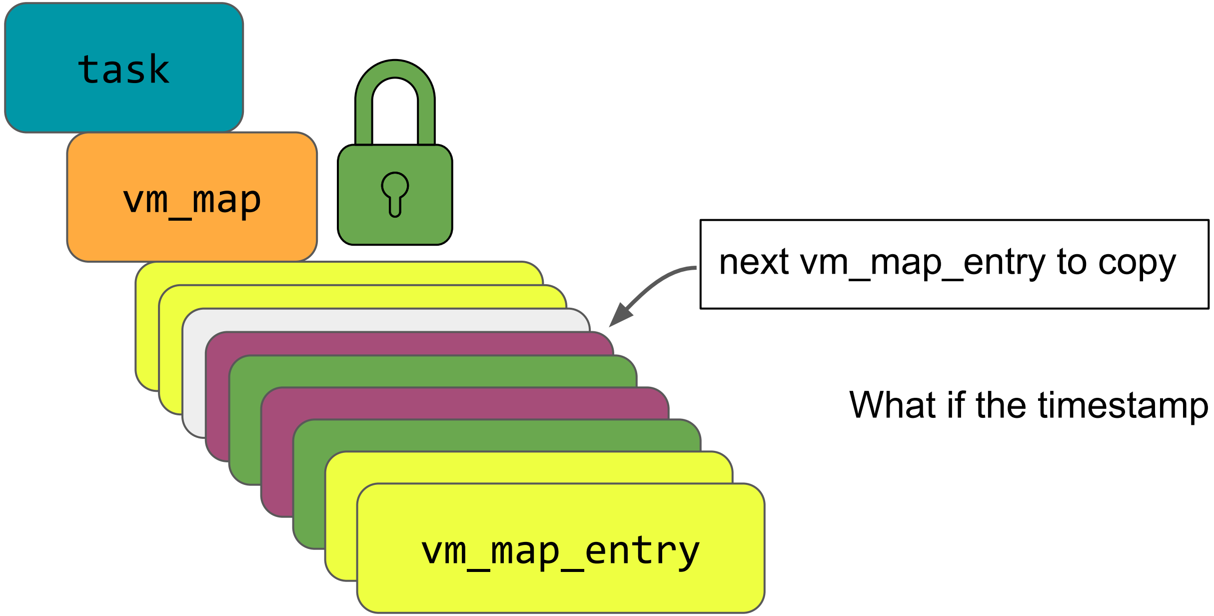
next vm_map_entry to copy

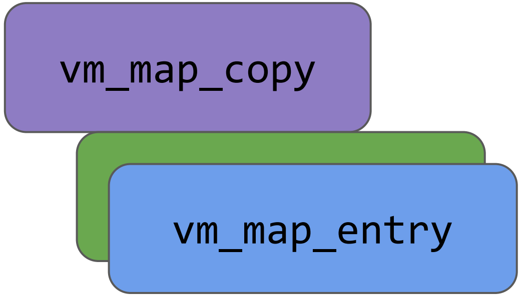


1) before starting, with map unlocked, allocate a single vm_map_entry
2) lock the map, and copy the contents of the first vm_map_entry in the region to the copy, and move to the next entry
3) if that wasn't the last vm_map_entry, we need to allocate a new vm_map_entry:
  3a) record the timestamp
  3b) unlock
  3c) allocate new vm_map_entry for the copy
  3d) lock
4) does ts + 1 == current vm_map timestamp?
  4a) Yes -> we were not raced, continue the loop here

```
ts = 100;
```

More accurately...

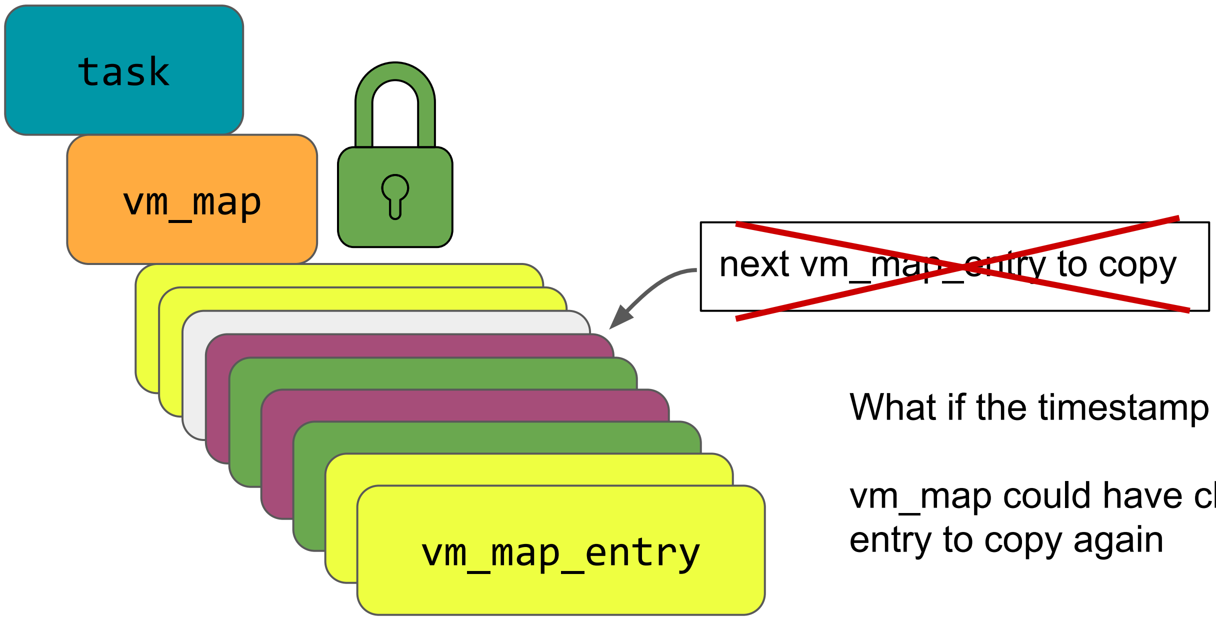


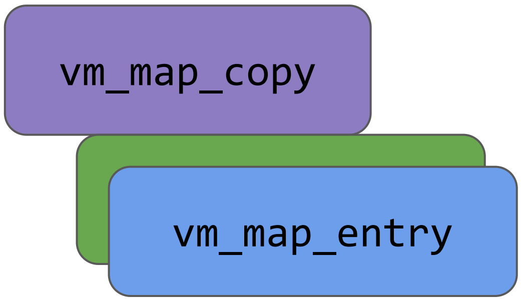


What if the timestamp doesn't match?

```
ts = 100;
```

More accurately...



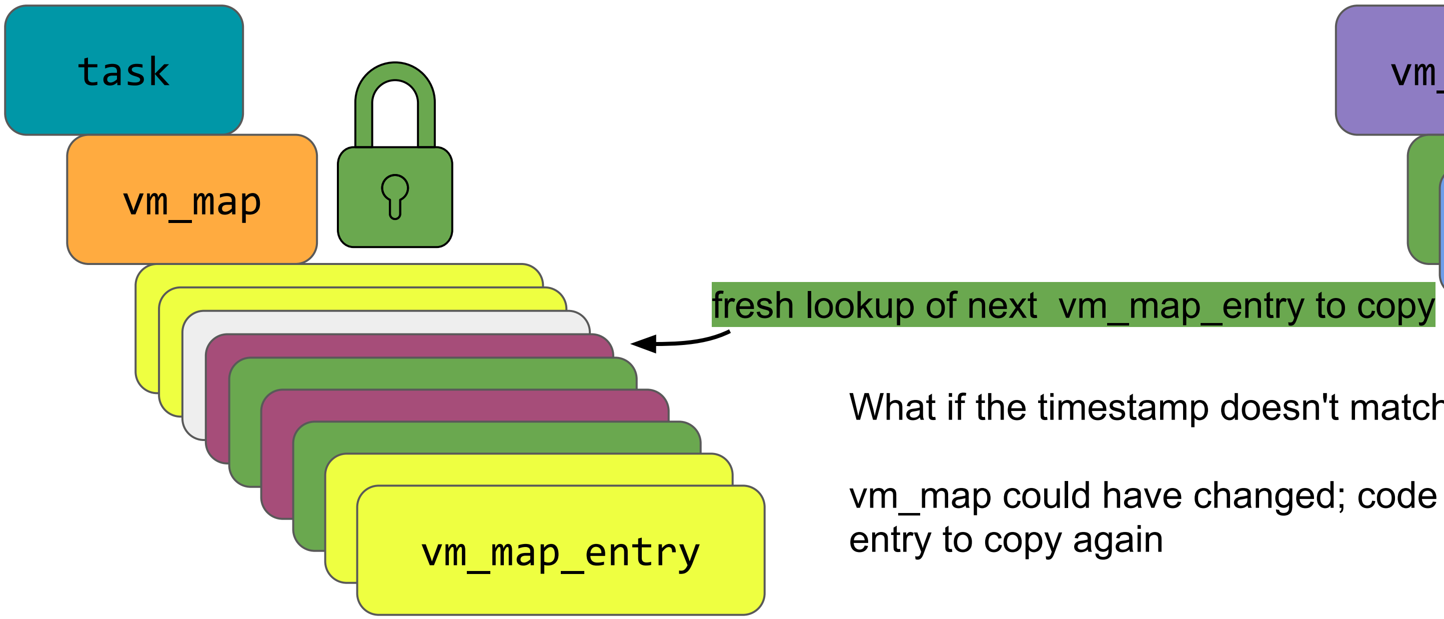


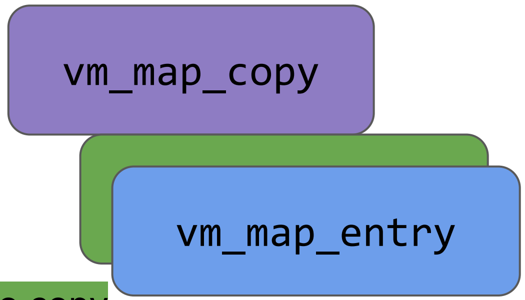
What if the timestamp doesn't match?

vm_map could have changed; code looks up the next entry to copy again

```
ts = 100;
```

More accurately...



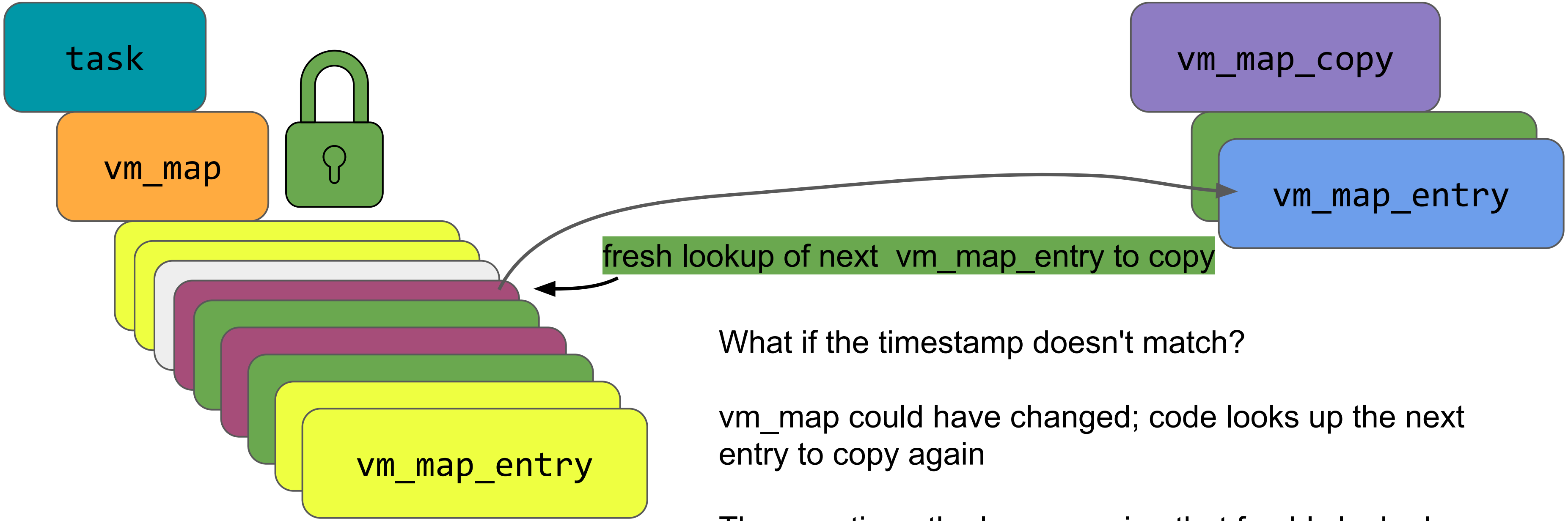


What if the timestamp doesn't match?

vm_map could have changed; code looks up the next entry to copy again

```
ts = 100;
```

More accurately...



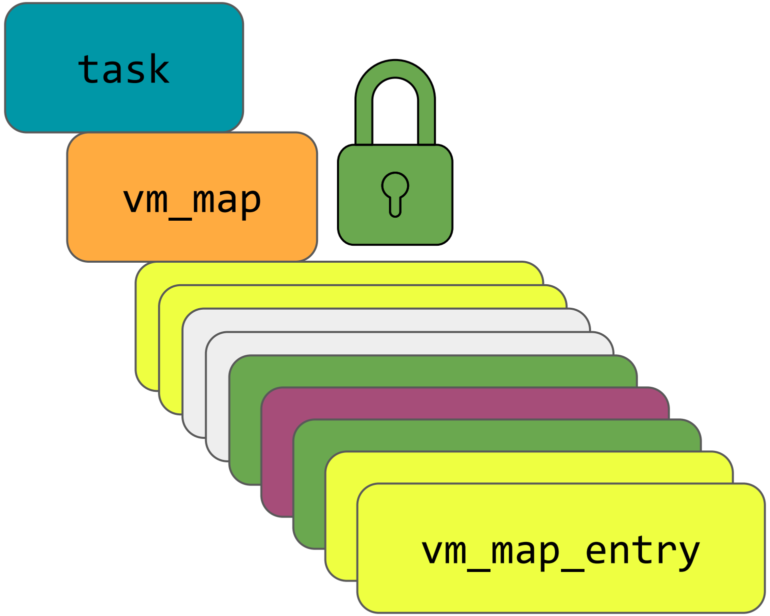
What if the timestamp doesn't match?

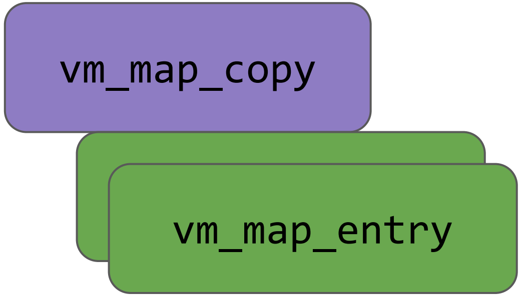
vm_map could have changed; code looks up the next entry to copy again

Then continue the loop, copying that freshly looked up vm_map_entry into the next vm_map_entry in the vm_map_copy chain

`ts = 100;`

More accurately...





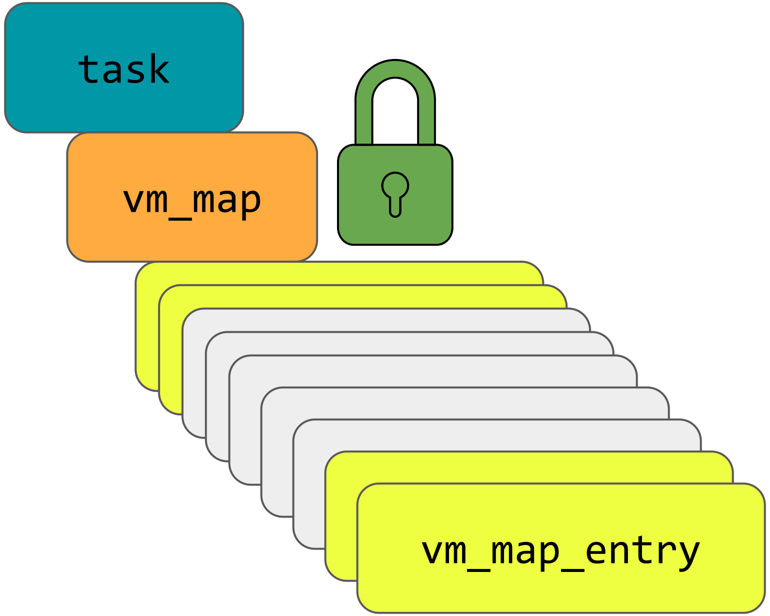

```
ts = 100;
```

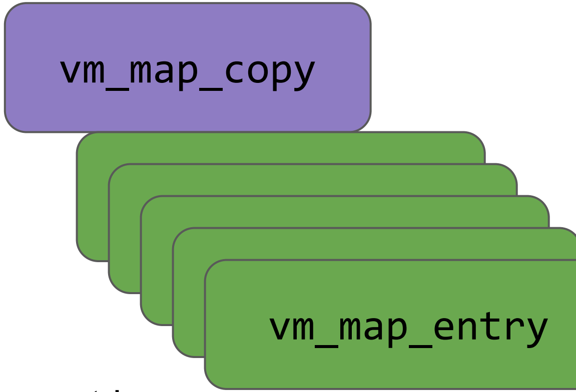
What if the timestamp doesn't match?

vm_map could have changed; code looks up the next entry to copy again

Then continue the loop, copying that freshly looked up vm_map_entry into the next vm_map_entry in the vm_map_copy chain

More accurately...

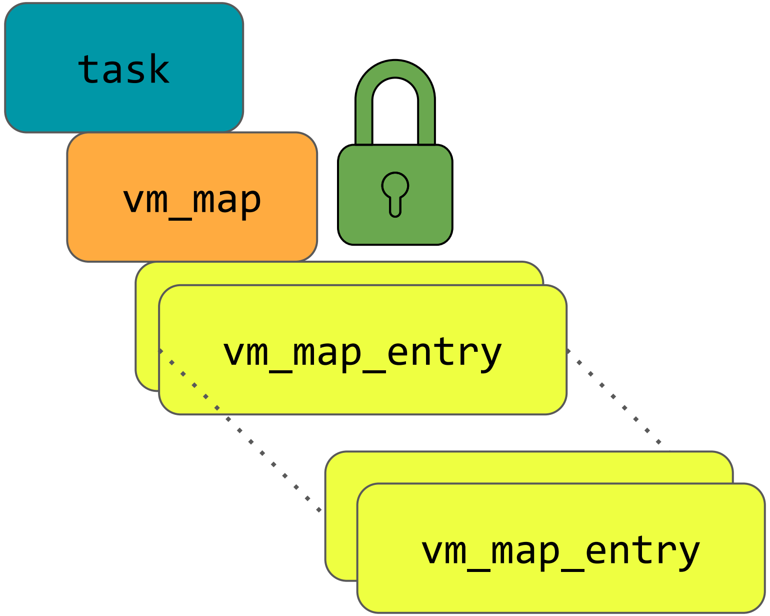


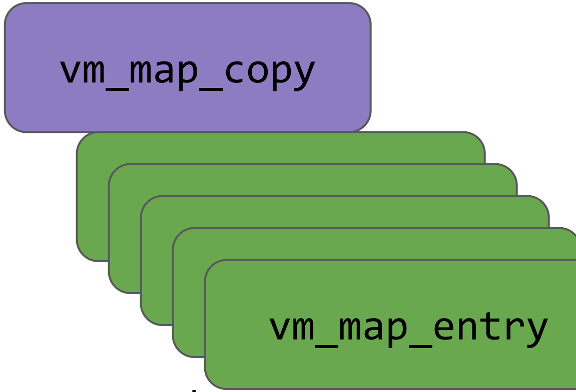


this repeats for all the vm_map_entries in the region

```
ts = 100;
```

More accurately...

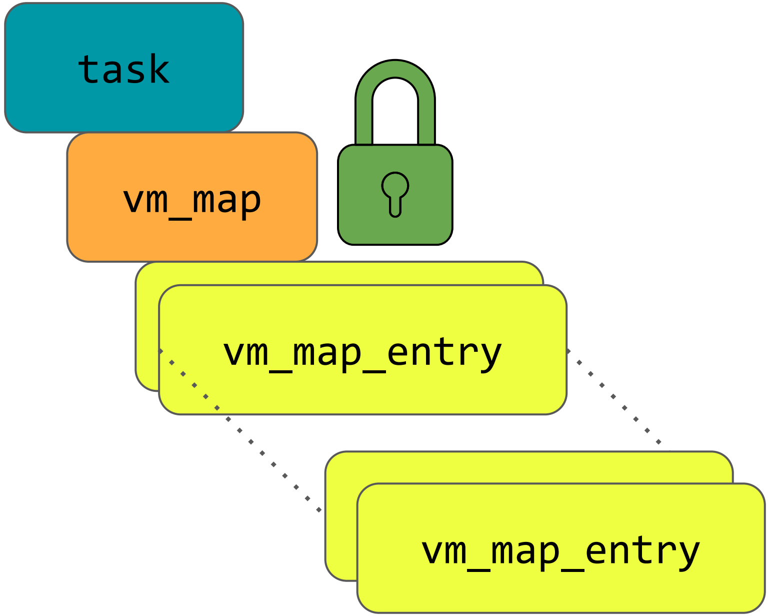


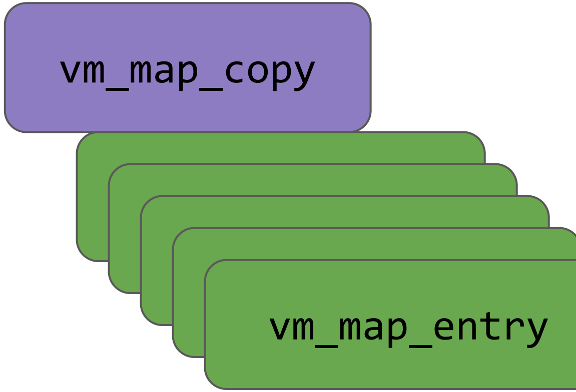


Finally, the vm_map_entry's are removed from the source vm_map

```
ts = 100;
```

What's wrong?






```
ts = 100;
```

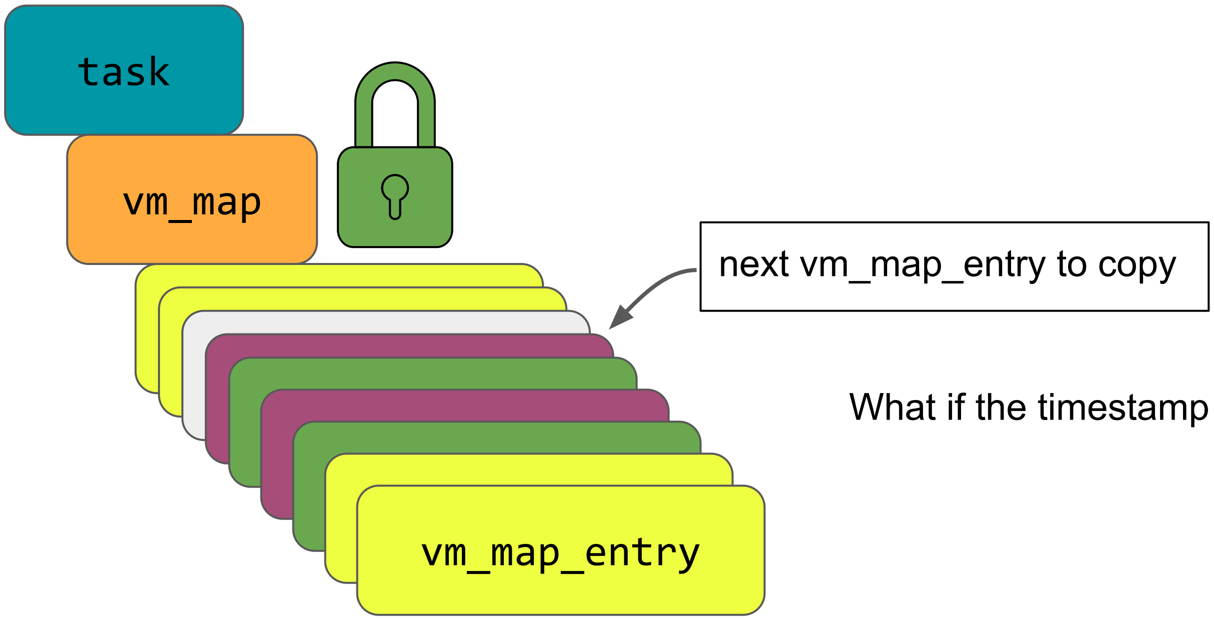
Code doesn't consider the **full semantics** of the whole operation

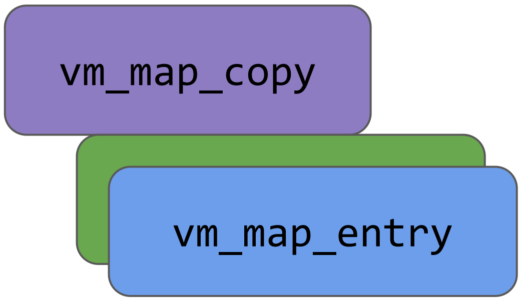
supposed to be an ATOMIC MOVE relative to the vm_map

observers should only be able to see full region in map, or full region not in map

What does that mean?

More accurately...

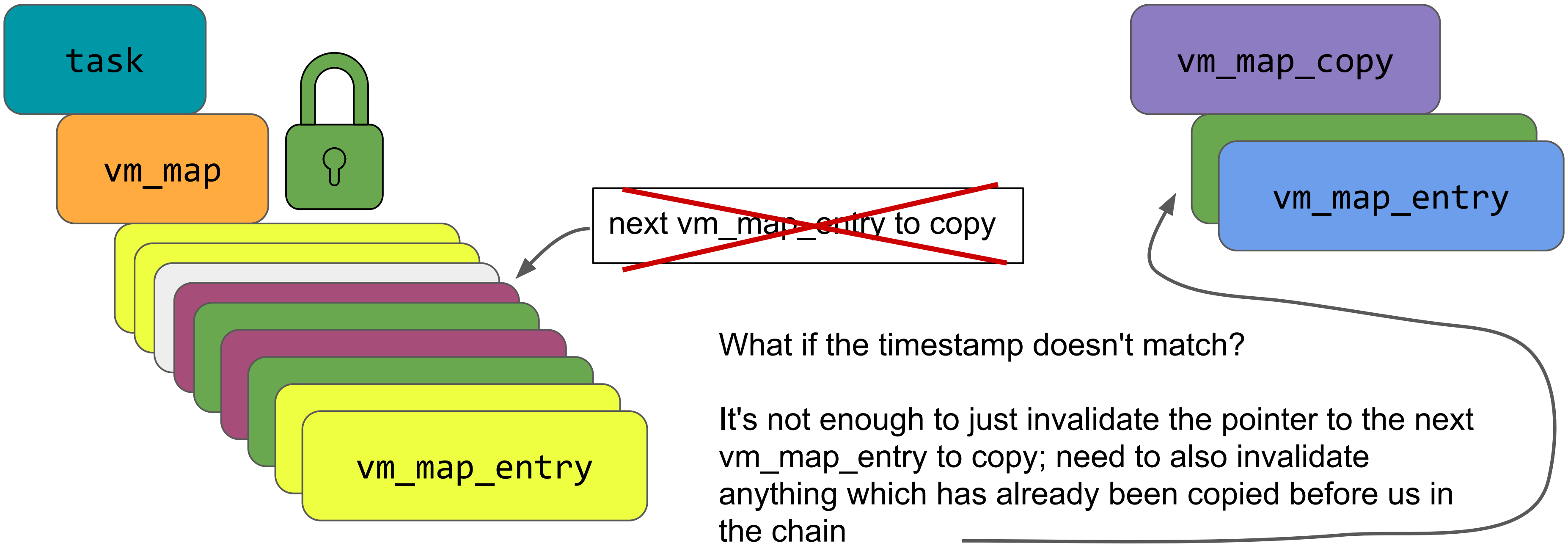




What if the timestamp doesn't match?

```
ts = 100;
```

More accurately...

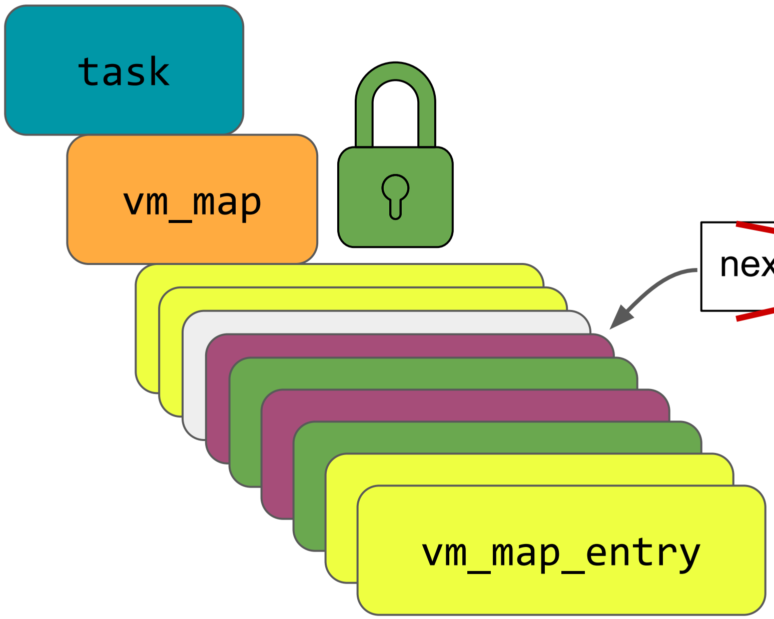


What if the timestamp doesn't match?

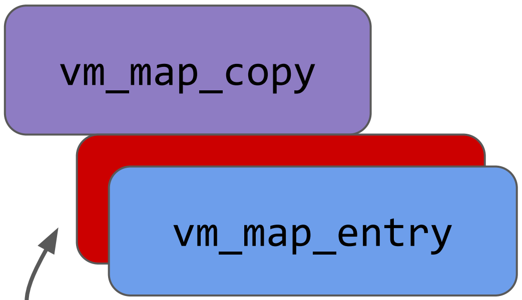
It's not enough to just invalidate the pointer to the next vm_map_entry to copy; need to also invalidate anything which has already been copied before us in the chain

```
ts = 100;
```

More accurately...



~~next vm_map_entry to copy~~



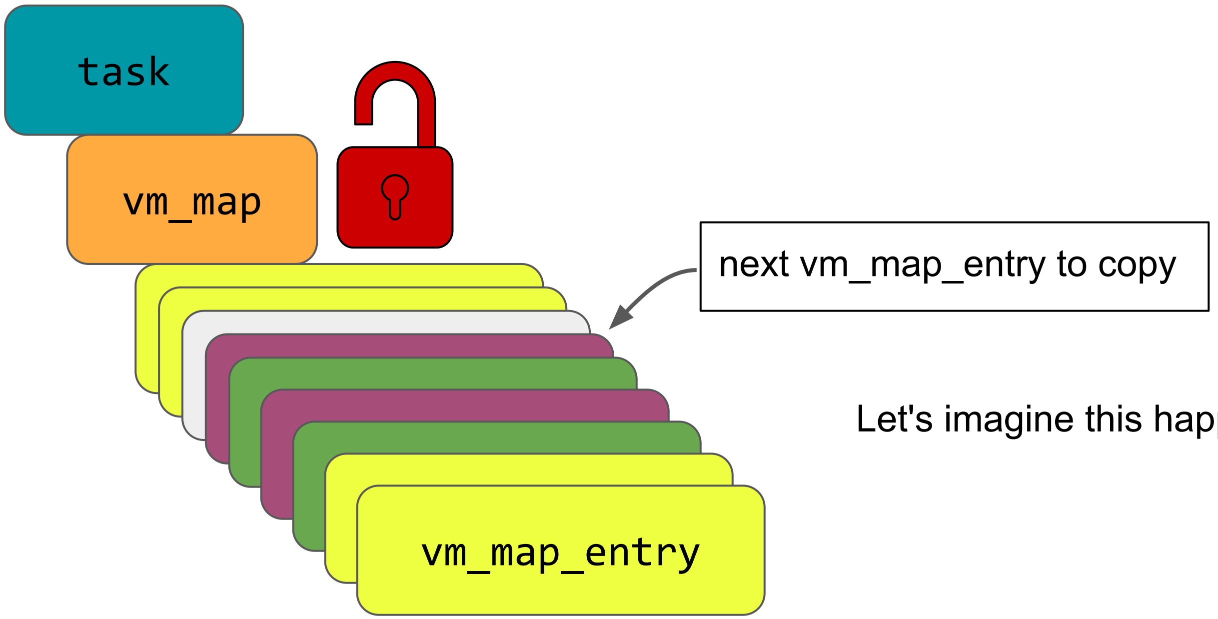
What if the timestamp doesn't match?

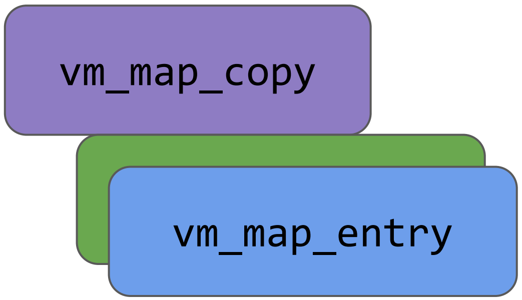
It's not enough to just invalidate the pointer to the next vm_map_entry to copy; need to also invalidate anything which has already been copied before us in the chain

```
ts = 100;
```

Why?

More accurately...

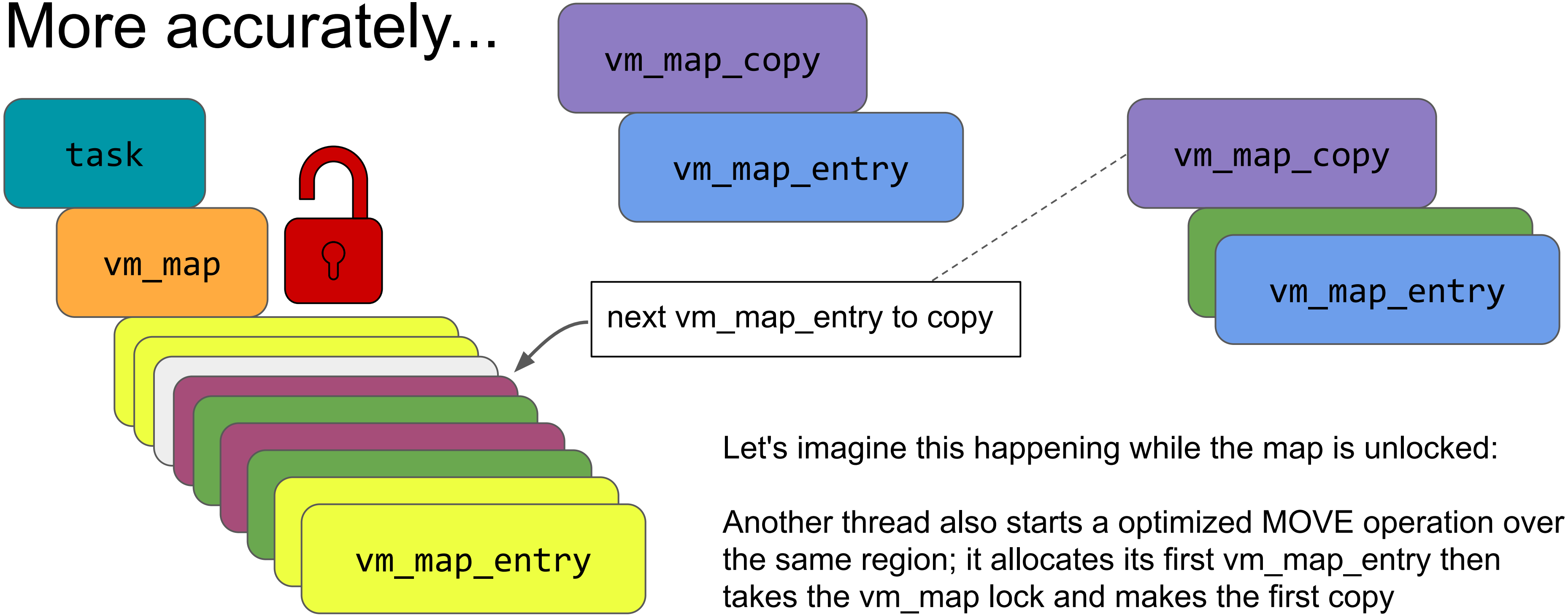




Let's imagine this happening while the map is unlocked:

```
ts = 100;
```

More accurately...

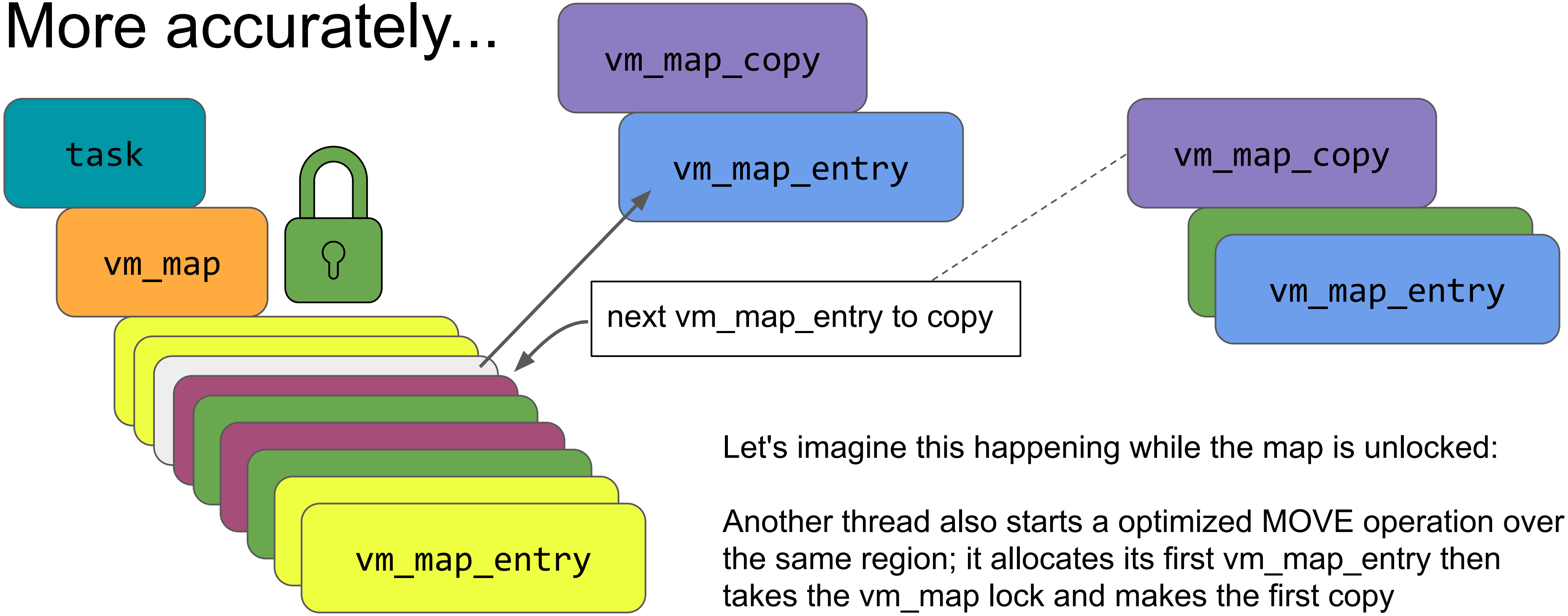


Let's imagine this happening while the map is unlocked:

Another thread also starts a optimized MOVE operation over the same region; it allocates its first vm_map_entry then takes the vm_map lock and makes the first copy

```
ts = 100;
```

More accurately...

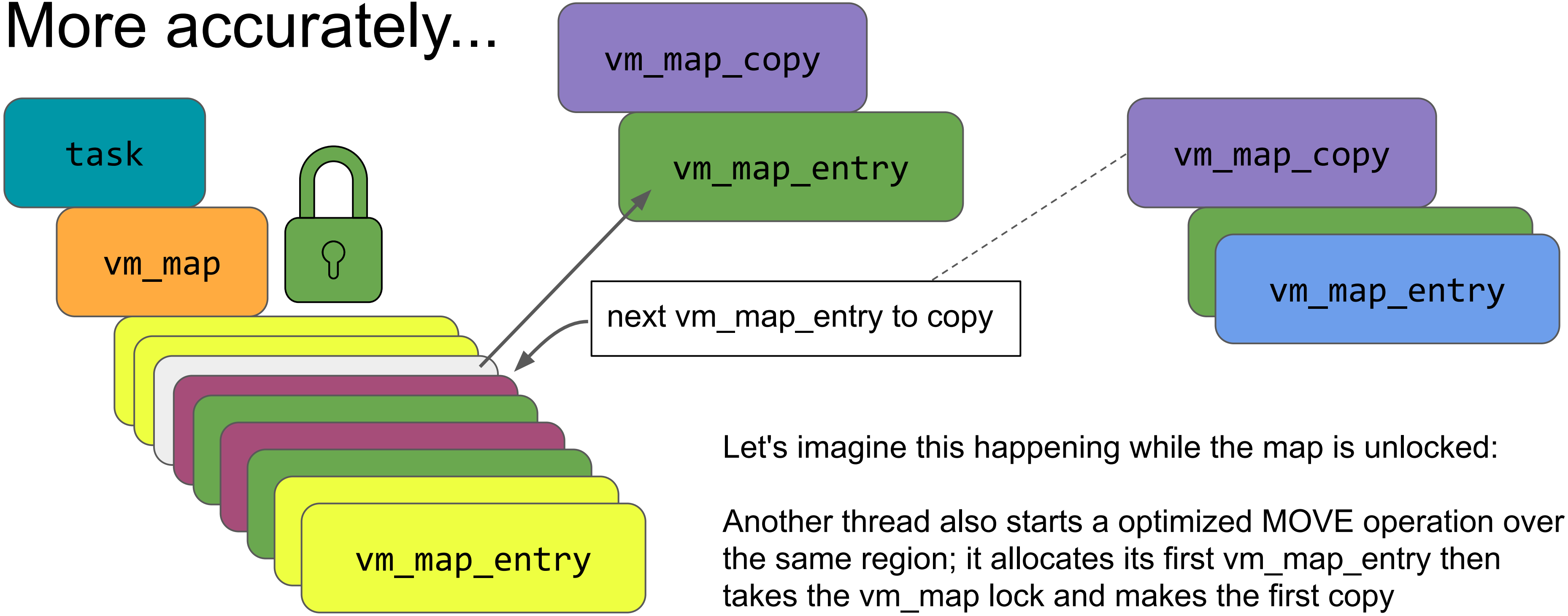


Let's imagine this happening while the map is unlocked:

Another thread also starts a optimized MOVE operation over the same region; it allocates its first vm_map_entry then takes the vm_map lock and makes the first copy

```
ts = 100;
```

More accurately...



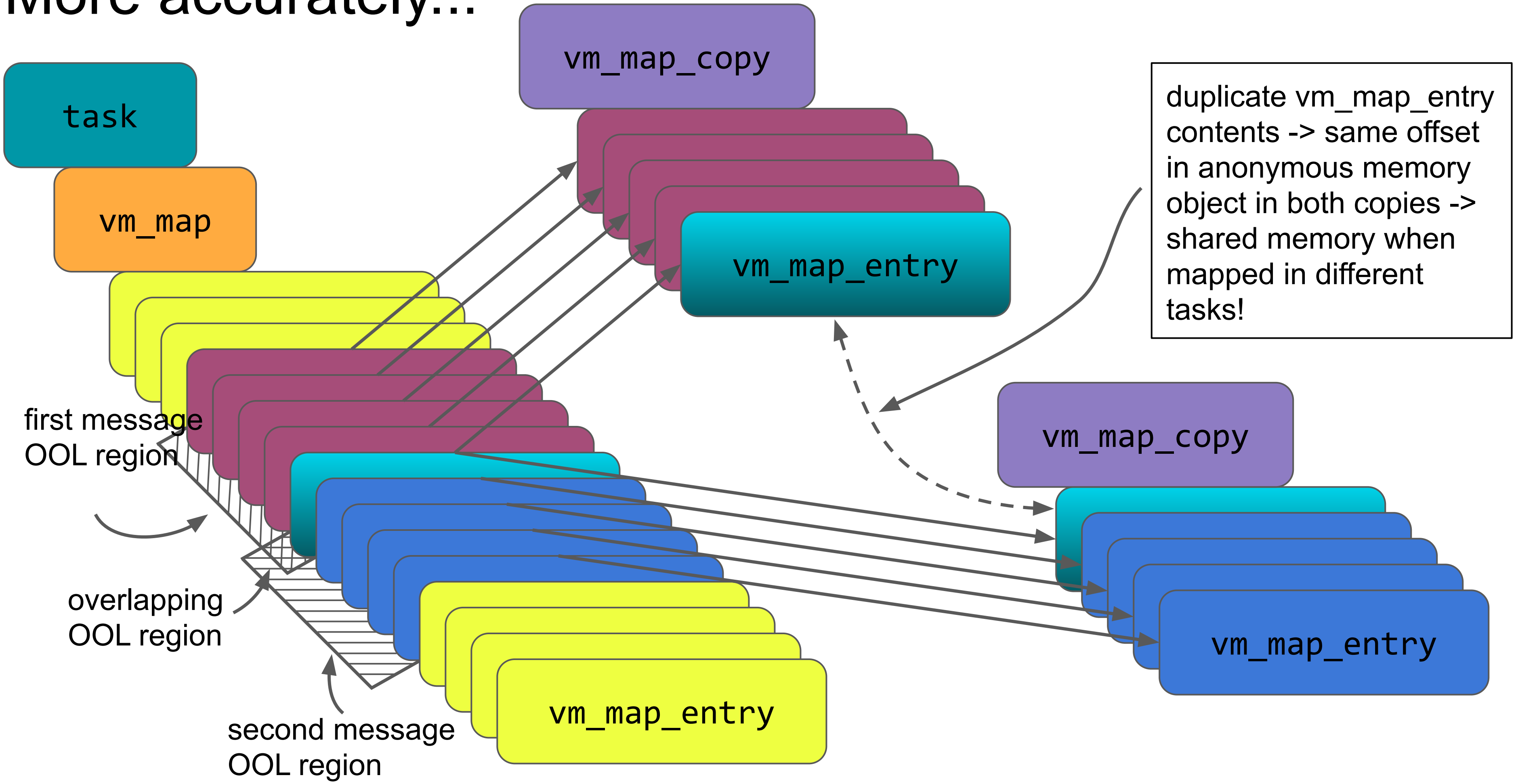
Let's imagine this happening while the map is unlocked:

Another thread also starts a optimized MOVE operation over the same region; it allocates its first vm_map_entry then takes the vm_map lock and makes the first copy

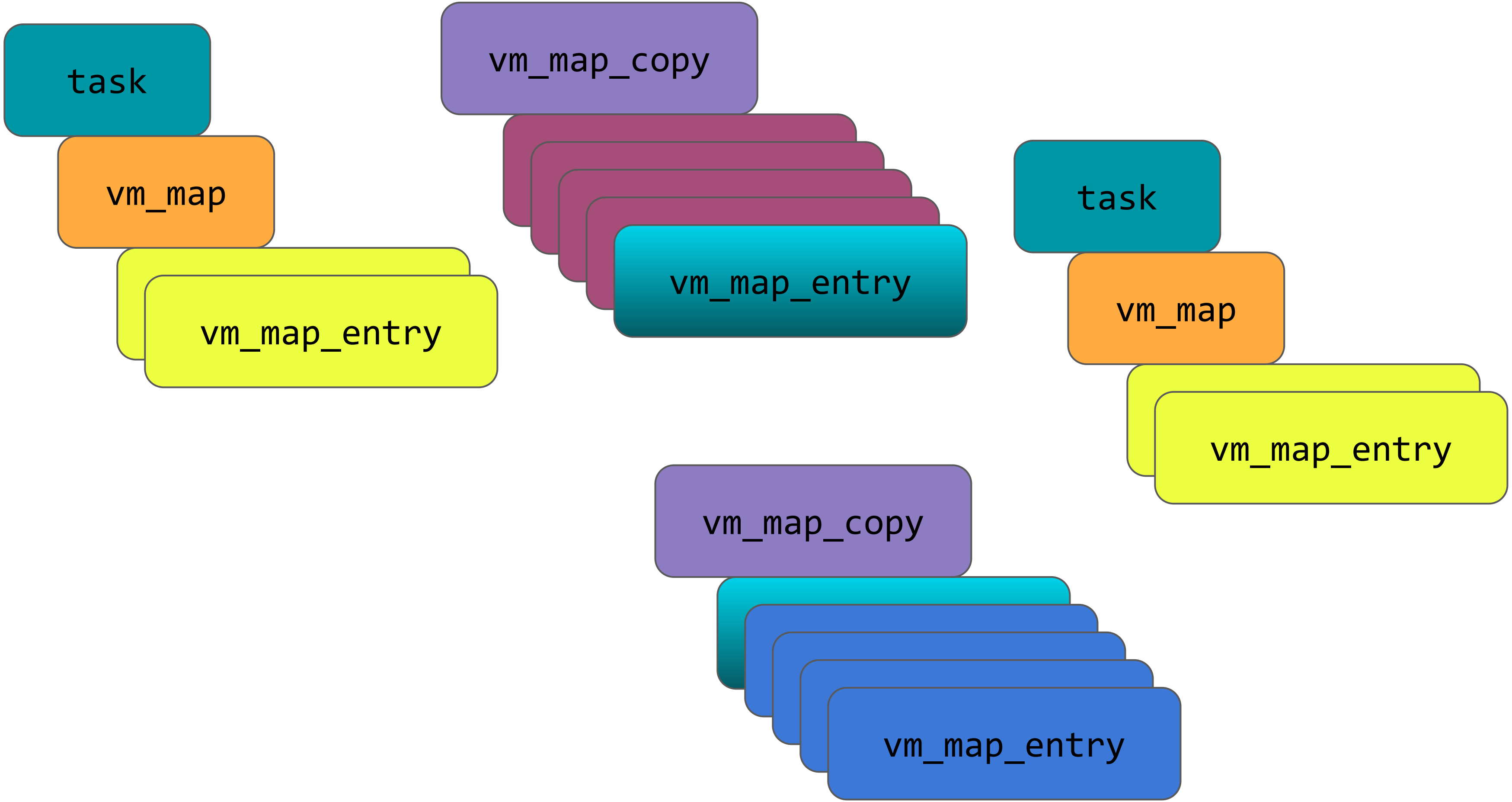
The two threads will thrash about with the vm_map lock, but if you're careful about how to structure the regions you can get two vm_map_copy with the same vm_map_entry CONTENTS (not the same actual vm_map_entry) when the intended semantics imply this shouldn't be possible

```
ts = 100;
```

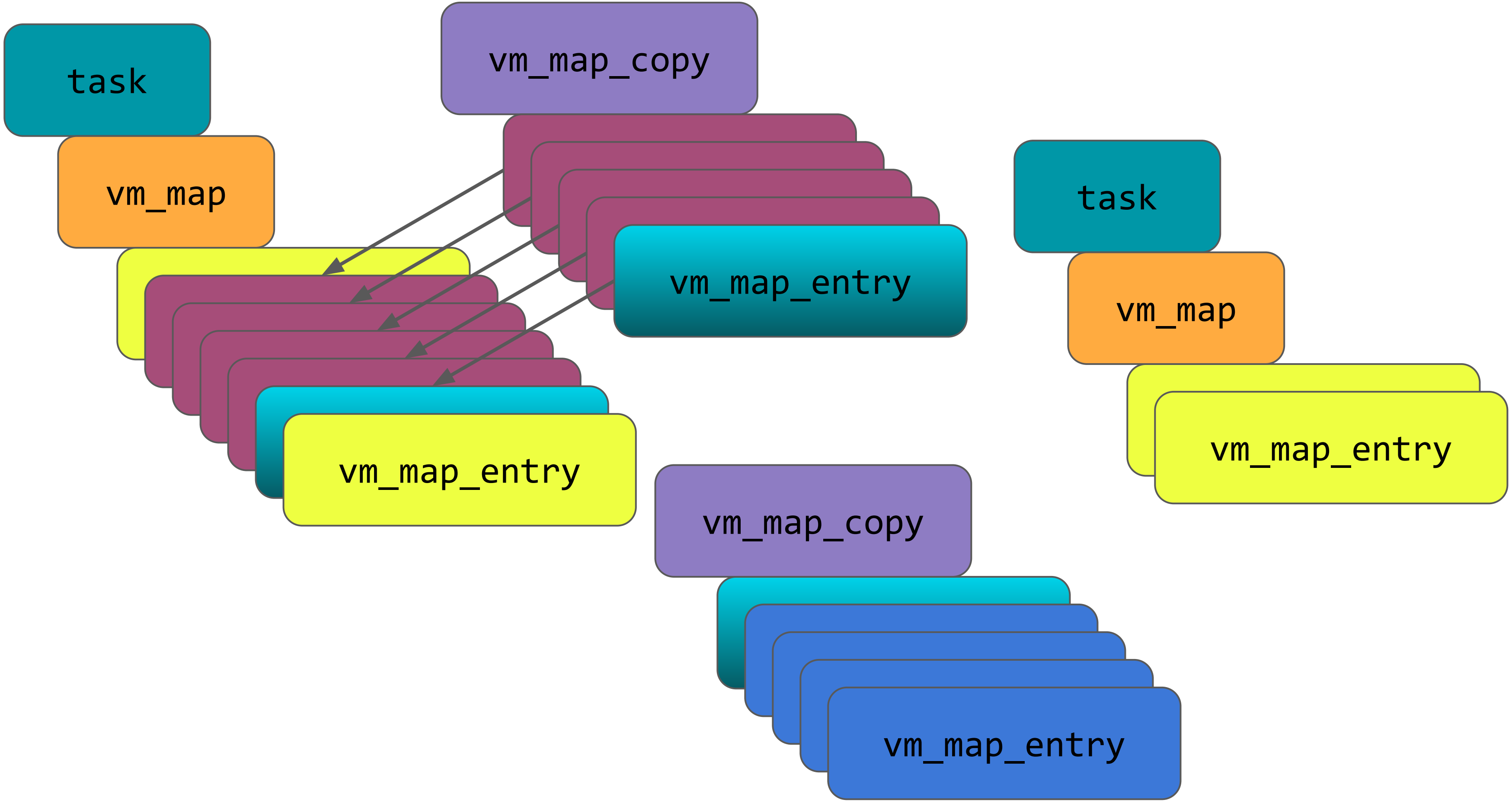
More accurately...



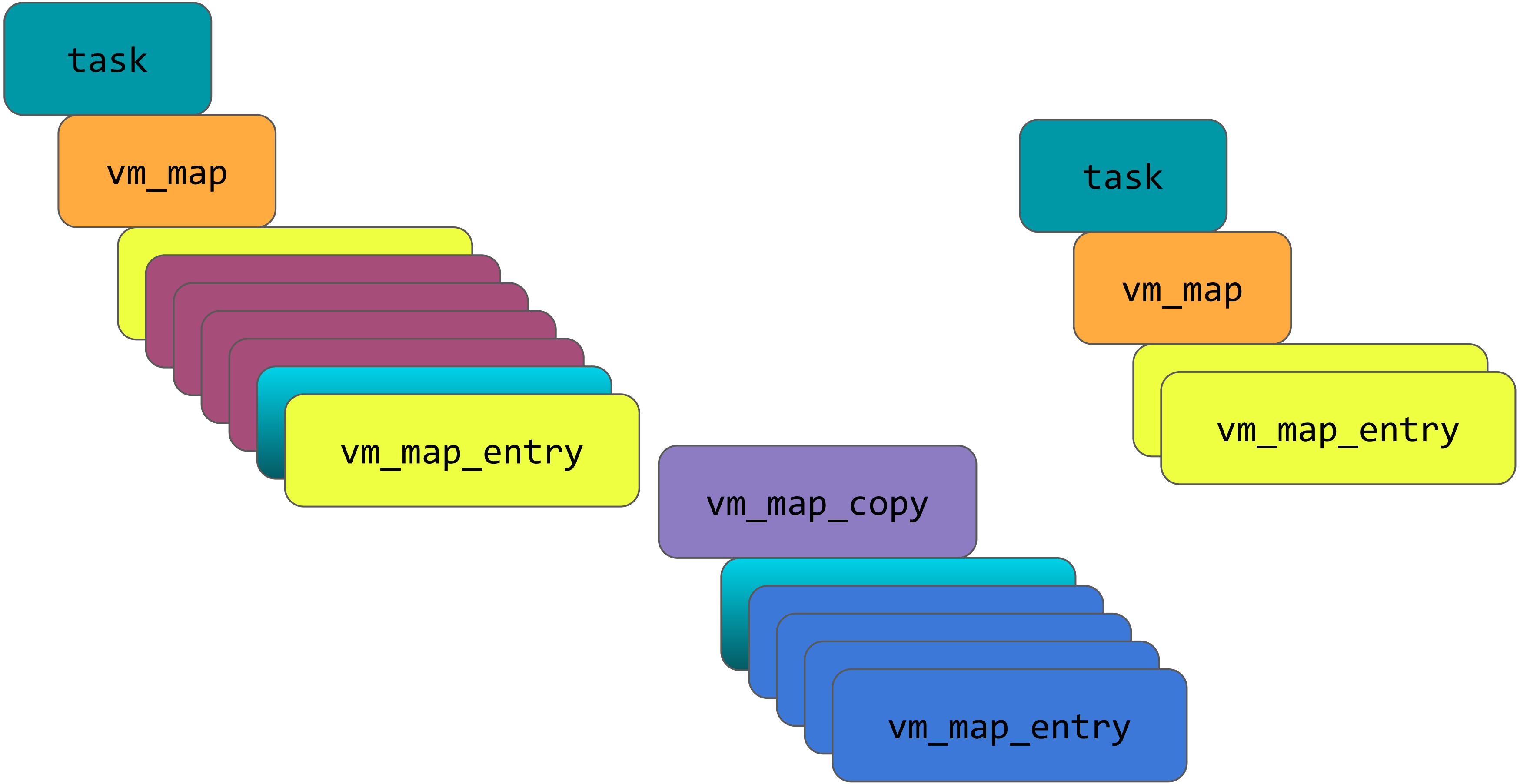
message receive:



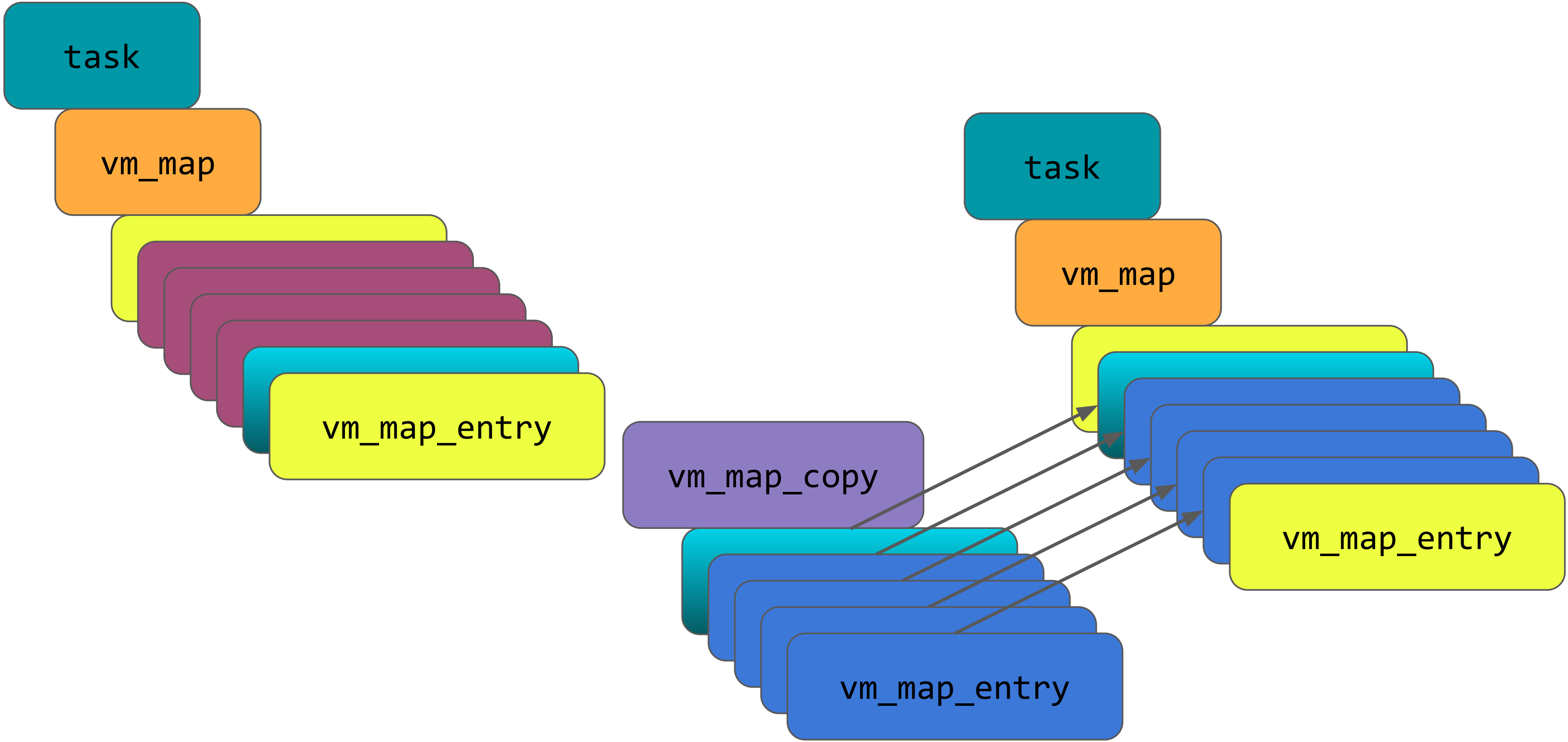
message receive:



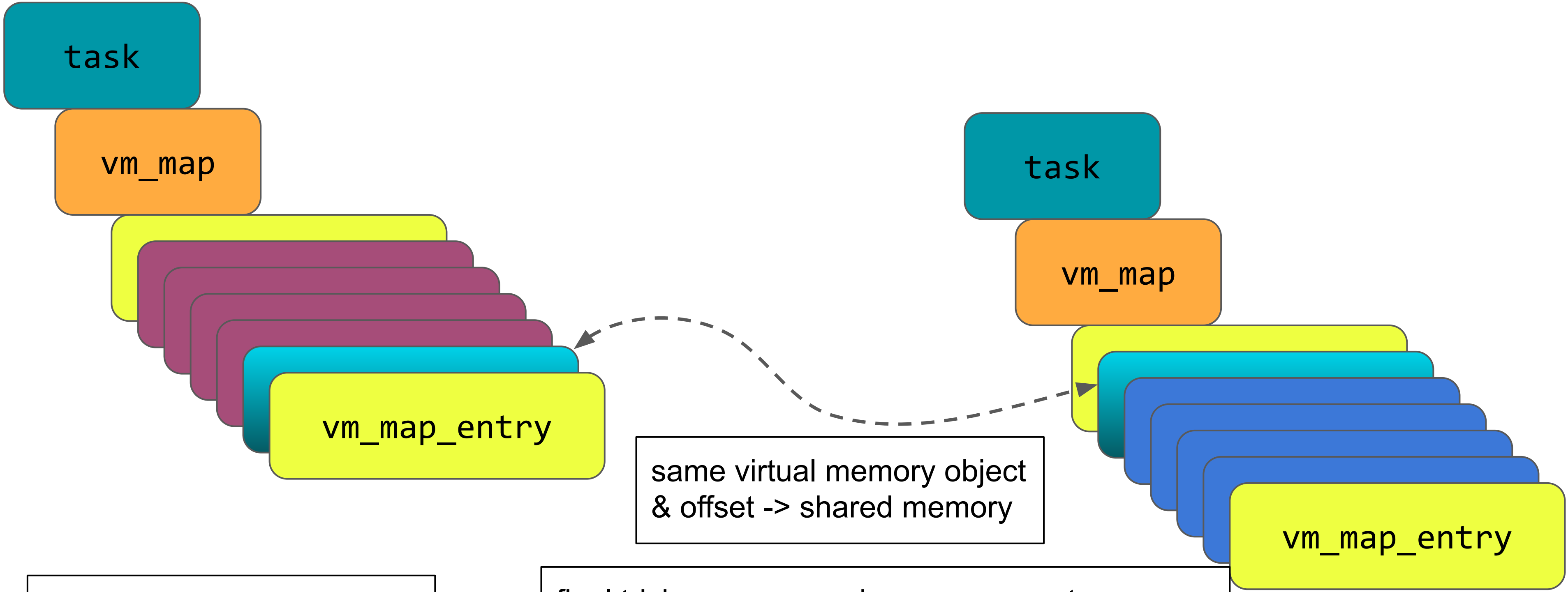
message receive:



message receive:



message receive:



same virtual memory object & offset -> shared memory

when mapped to physical page, writes in task on the left seen on the right, and the other way round

final trick: we can send one message to ourselves, and one to the real target, this lets us then modify the contents of the OOL descriptor received by the target as it's parsing it

Exploiting *unexpectedly shared memory*

This bug breaks "mach message as an envelope" semantics:

When you're reading a letter you took out of an envelope, you don't expect it to change while you're reading it. A mach message is meant to be like that.

But: we need to find somewhere where breaking those semantics across a trust boundary leads to "something with security impact"

Aside: what is "security impact"?

Surprisingly difficult to concisely define.

Memory corruption is the most boring yet widely accepted thing with security impact:

- Decades of public research should help you convince yourself that it's almost always possible to turn memory corruption in a target context into the ability to perform arbitrary system interactions with the trust level of the target context

Far more interesting things possible when you dig more deeply in to target-specific code:

- Time-Of-Check-Time-Of-Use in signature checking?
- TOCTOU in selector validation? (NSXPC?)
- TOCTOU in bytecode verification? (BPF?, hello Luca & PS4 ;))
- Weird allocator that reuses the pages for internal heap rather than returning them?
- Endless possibilities... (compiler bugdoors causing unnecessary double fetches? ;)

I am boring and lazy, lets just cause memory corruption..

Also gives an opportunity to play with pointer auth on A12, see the blog post at https://googleprojectzero.blogspot.com/2019/04/splitting-atoms-in-xnu.html for more details

Shared memory to memory corruption

In 2017 lokihardt found CVE-2017-2456 which lead to a similar style of bug

In that case, the unexpected shared memory was caused by sending OOL descriptors where the memory was backed by memory from a memory entry object (via mach_make_memory_entry_64)

His exploit targeted a particular construct in libxpc...

libxpc TOCTOU

Serialized XPC message bodies can be either inline (after any descriptors) or be in a single OOL descriptor, which must be the first descriptor in the message*

This means any part of the XPC deserialization code is a target for an "unexpected TOCTOU"

Note that these by themselves aren't bugs; that's why Loki's technique still worked

XPC dictionary key deserialization

```
loc_2335:
mov     [rbp+var_29], bl
mov     rax, [rbp+var_48]
lea     r13, [rdx+rax*8+70h]
mov     rdi, r12           ; char *
call    _strlen
lea     rdi, [rax+29h]     ; size_t
call    __xpc_malloc
mov     rbx, rax
mov     dword ptr [rbx+18h], 0
lea     rdi, [rbx+20h]     ; char *
xor     eax, eax
mov     [rbx+8], rax
mov     [rbx], rax
mov     rsi, r12           ; char *
call    _strcpy
mov     rdi, r15
call    _xpc_retain
mov     [rbx+10h], rax
mov     rax, [r13+0]
mov     [rbx], rax
test    rax, rax
jz      short loc_238D
```

```
loc_180B53230             ; char *
MOV     X0, X21
BL      j__strlen_18
ADD     X0, X0, #0x29 ; ')' ; size_t
BL      __xpc_malloc
MOV     X23, X0
STR     WZR, [X23,#0x18]
ADD     X0, X23, #0x20 ; ' ' ; char *
STP     XZR, XZR, [X23]
MOV     X1, X21 ; char *
BL      j__strcpy_0
MOV     X0, X20
BL      _xpc_retain_0
STR     X0, [X23,#0x10]
LDR     X8, [X24]
STR     X8, [X23]
CBZ     X8, loc_180B53274
```

enjoy the X64 version while it's still relevant, and then learn ARM64 ;)

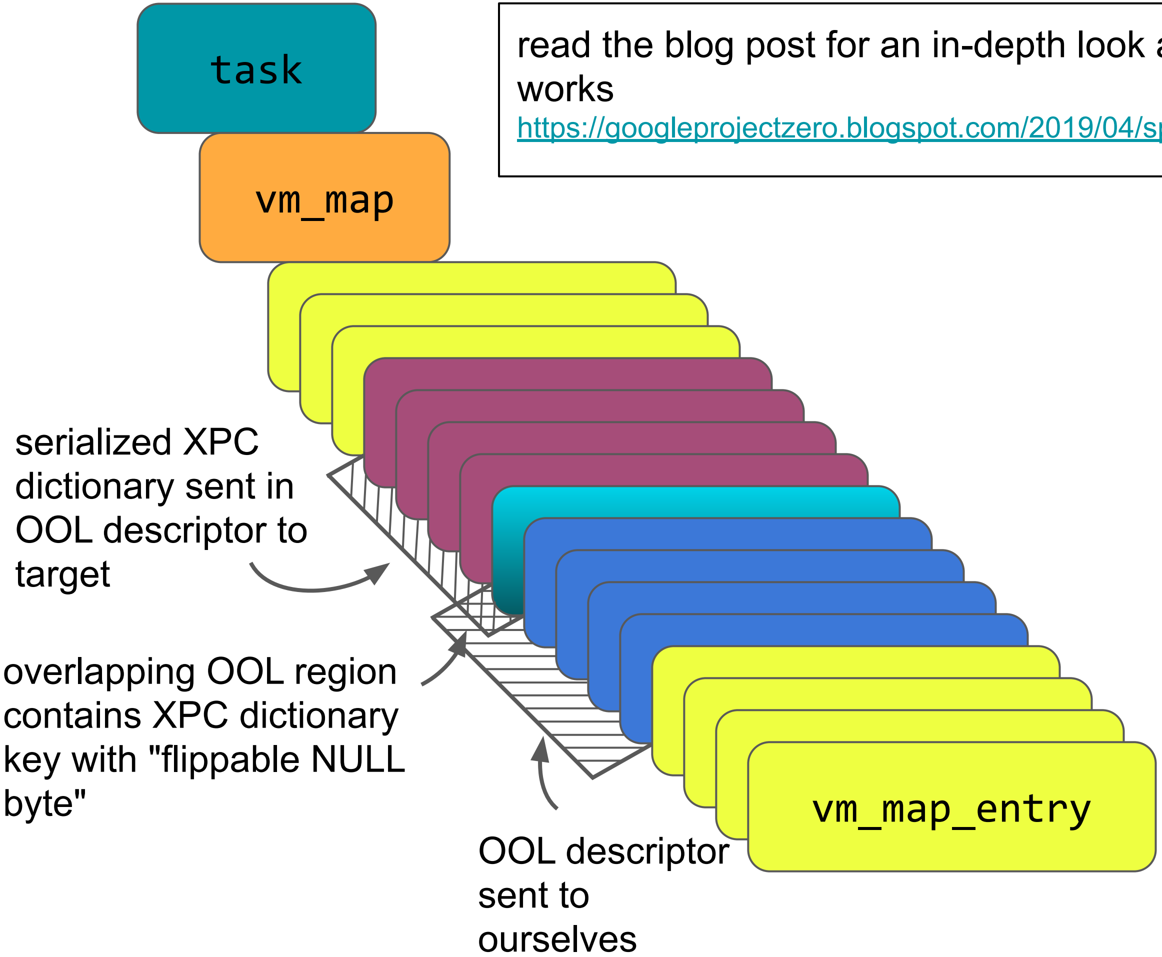
XPC dictionary key deserialization:

```
char* key_len = strlen(key_ptr);

xpc_dict_entry_t* dict_entry =
  xpc_malloc(key_len + 0x29);

dict_entry->flags = 0;

dict_entry->prev = NULL;

dict_entry->next = NULL;

strcpy(dict_entry->key_buf, key_ptr);
```

points into unexpected shared memory

30 second overview of exploit:

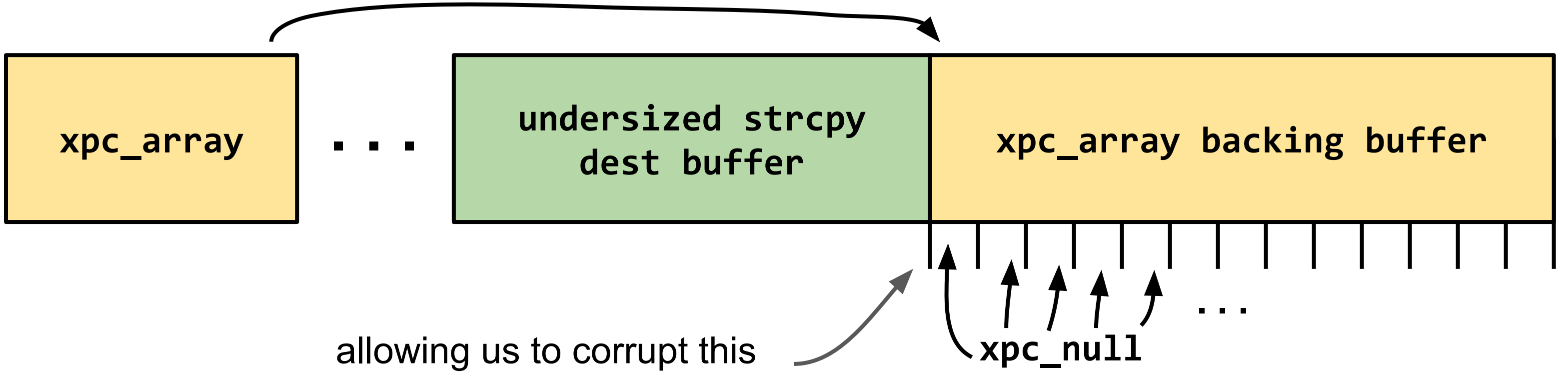
read the blog post for an in-depth look at how this actually works
https://googleprojectzero.blogspot.com/2019/04/splitting-atoms-in-xnu.html



30 second overview of exploit:

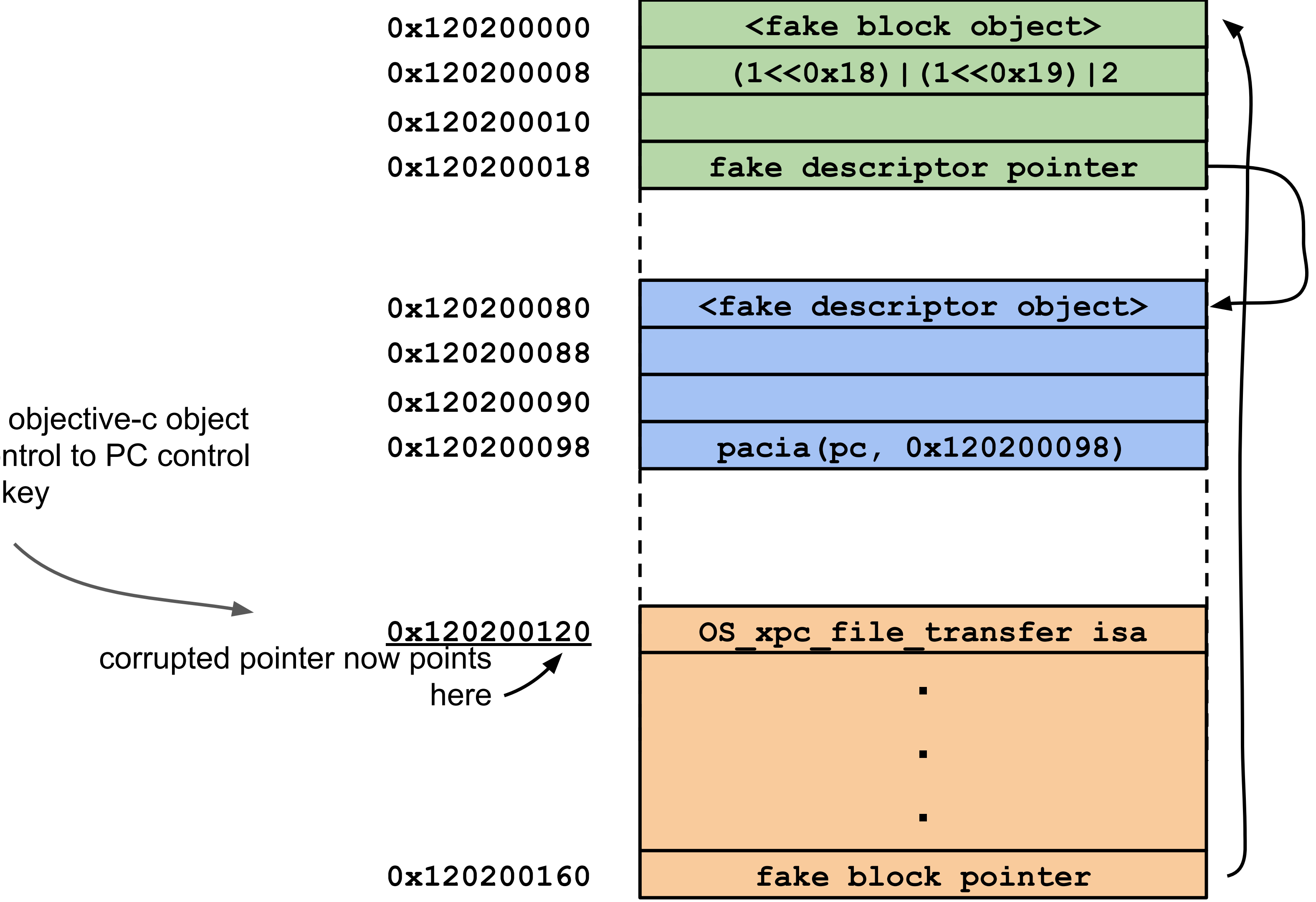
this ends up in shared memory (it's part of the serialized XPC dictionary)

deserialization while sender is flipping the flipper bit hopefully leads to this memory layout in the target:



allowing us to corrupt this objective-c object pointer here

Path from objective-c object pointer control to PC control without B key



Some final thoughts

- XNU Virtual Memory code is:
    - very old
    - very complex
    - very hard to read
    - very hard to reason about
    - very keen on multi-thousand line functions
    - very critical to almost every security boundary

Thanks

Prior XNU VM research and documentation:

Jonathan Levin [http://www.newosxbook.com]

Amit Singh [http://osxbook.com/about/]