VIRTUAL MEMORY IN STOS

Pierre 'Pimzero' Marsais
pierre.marsais@lse.epita.fr

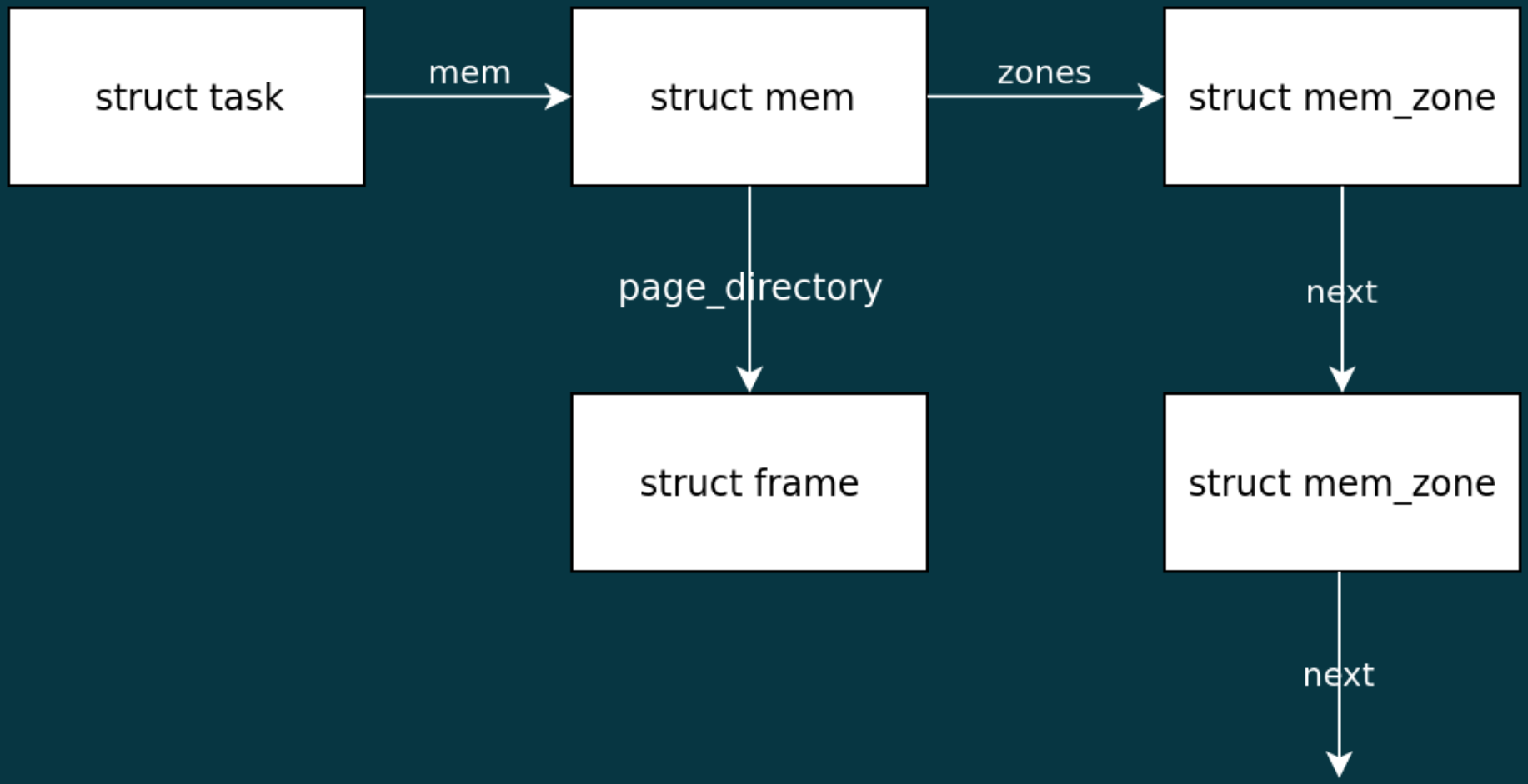
LSE Security System

WHAT DO WE WANT

- Specify rights on memory
- Allow private or shared mapping
- `mmap`(2) should be able to map anonymous and file backed memory
- `fork`(2) implies copy on write

WHAT ABOUT STOS ?

TASK'S MEMORY IN STOS



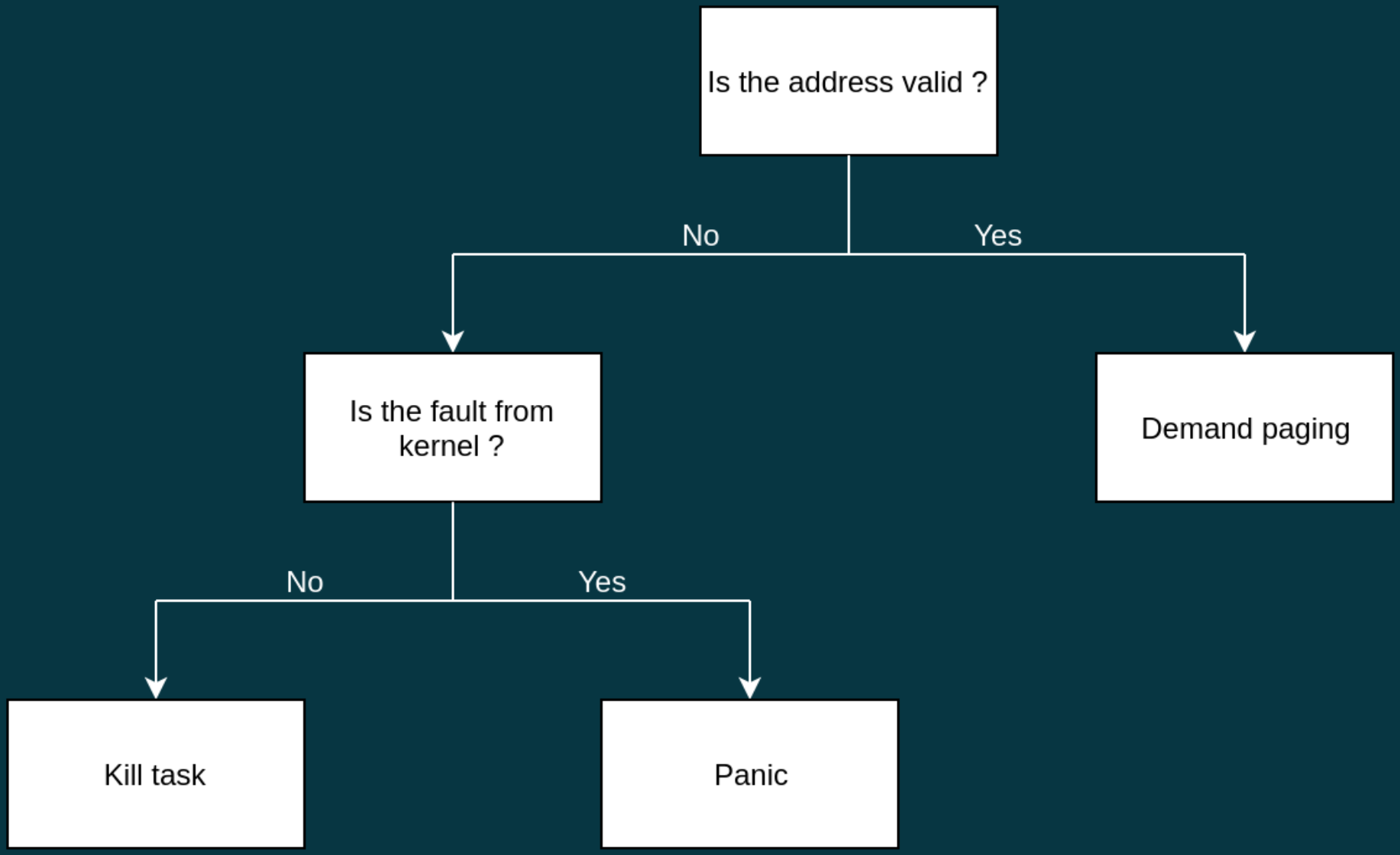
ONE STEP CLOSER

```
struct mem_zone {
    void* beg;
    void* end;

    /* TODO: Access right */

    struct list_node next;
};
```

MMAP IN STOS

```
long __syscall sys_mmap(void* addr, size_t len, int prot, int flags,
                        int filedes, off_t off)
{
    assert(filedes == -1);

    len = align2_up(len, PAGE_SIZE);

    addr = find_free_mem_range(get_current(), len);

    int ret = add_mem_zone(get_current(), addr, addr + len);
    if (ret < 0)
        return ret;

    return (long)addr;
}
```

FAULT HANDLER IN STOS



PREVIOUS STATE

STOS has a rather rudimentary memory management

- All memory is RW
- Only private mapping
- `mmap`(2) can only map anonymous memory
- `fork`(2) copies all pages from the parent to the child

MEMORY PROTECTION

MEMORY PROTECTION

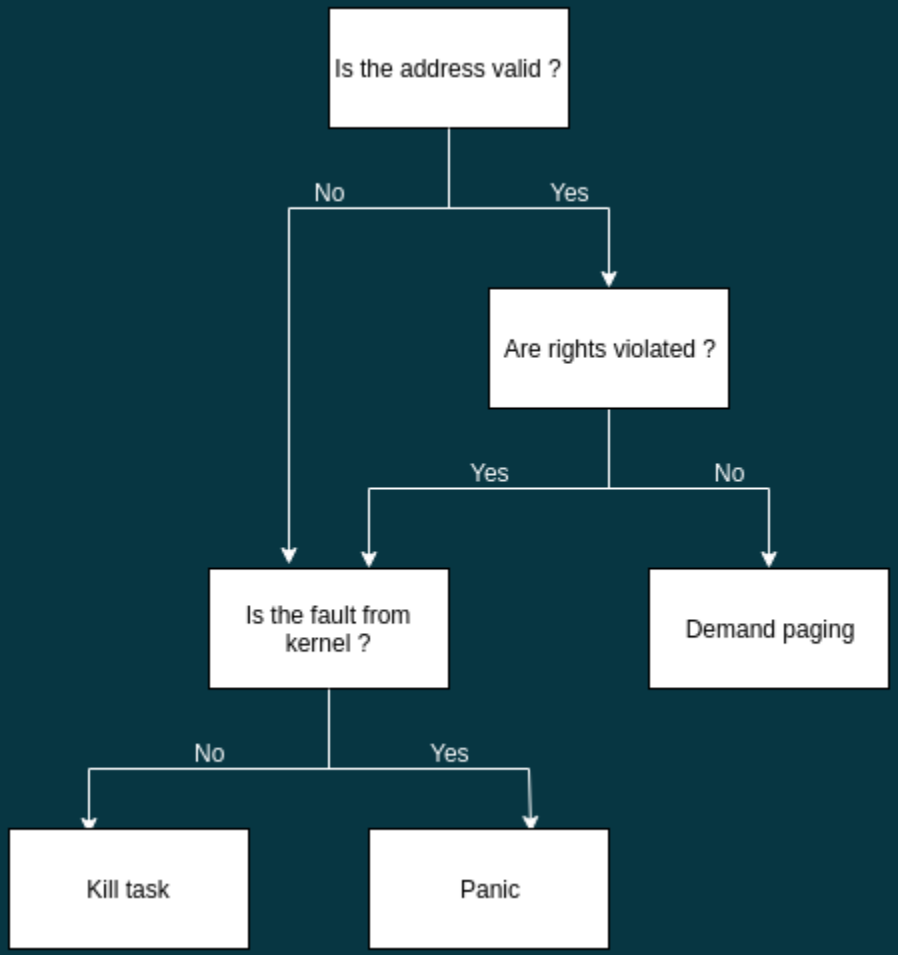
In userland: `mprotect`(2) changes the protection for a `mem_zone` in the address space

- `PROT_NONE`
- `PROT_READ`
- `PROT_WRITE`
- `PROT_EXEC`

MEMORY PROTECTION HOW-TO

- Memory protection in enforced by hardware with flags in page table entries (PTE)
- Demand paging implies we have to keep protection even when a page is not loaded
- So, a protection flag is present in `struct mem_zone`
- In the `page_fault_handler()`, get the reason of the page fault
- Check if the rights in the `mem_zone` are violated
- If so, then do as if the address was not valid, else, it's on demand paging

FAULT HANDLER WITH MEMORY PROTECTION



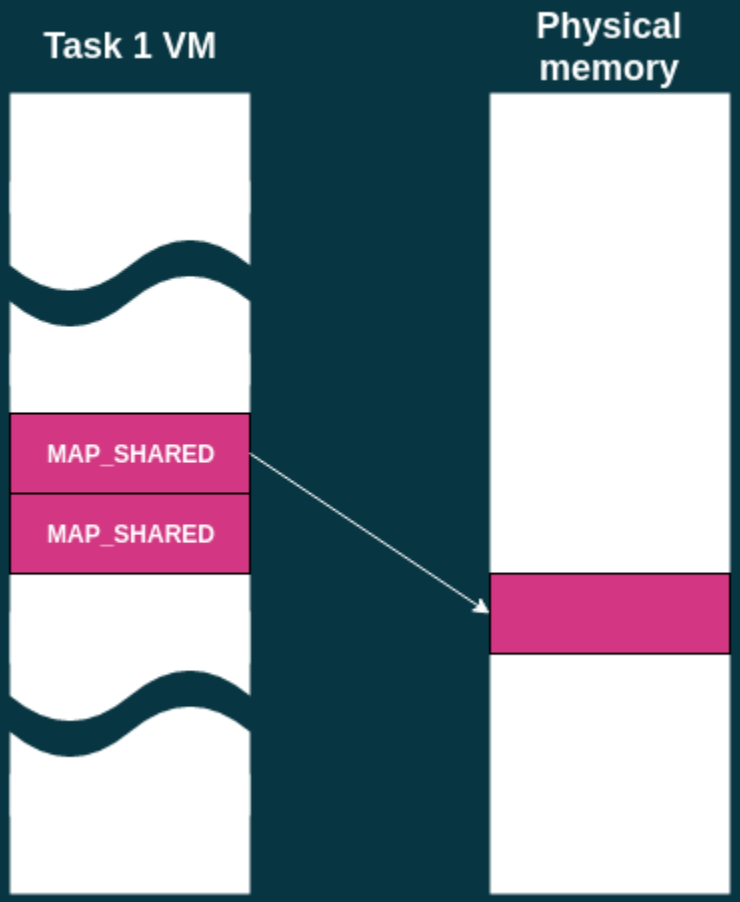
SHARED MAPPING

MMAP(2) FLAGS

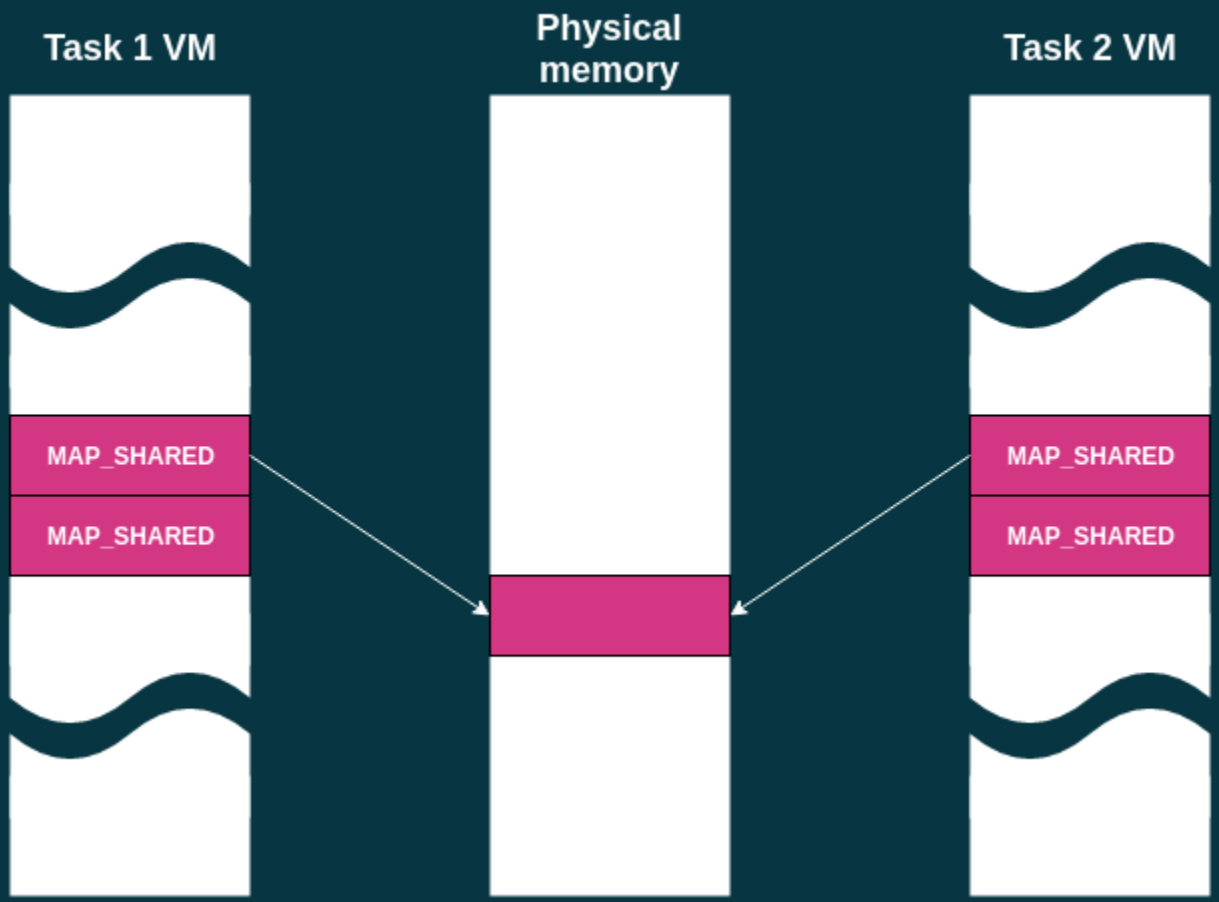
`man 2 mmap`

- `MAP_PRIVATE` updates are not visible to other processes mapping the same file, and are not carried through to the underlying file
- `MAP_SHARED` updates are visible to other processes that map this file, and are carried through to the underlying file

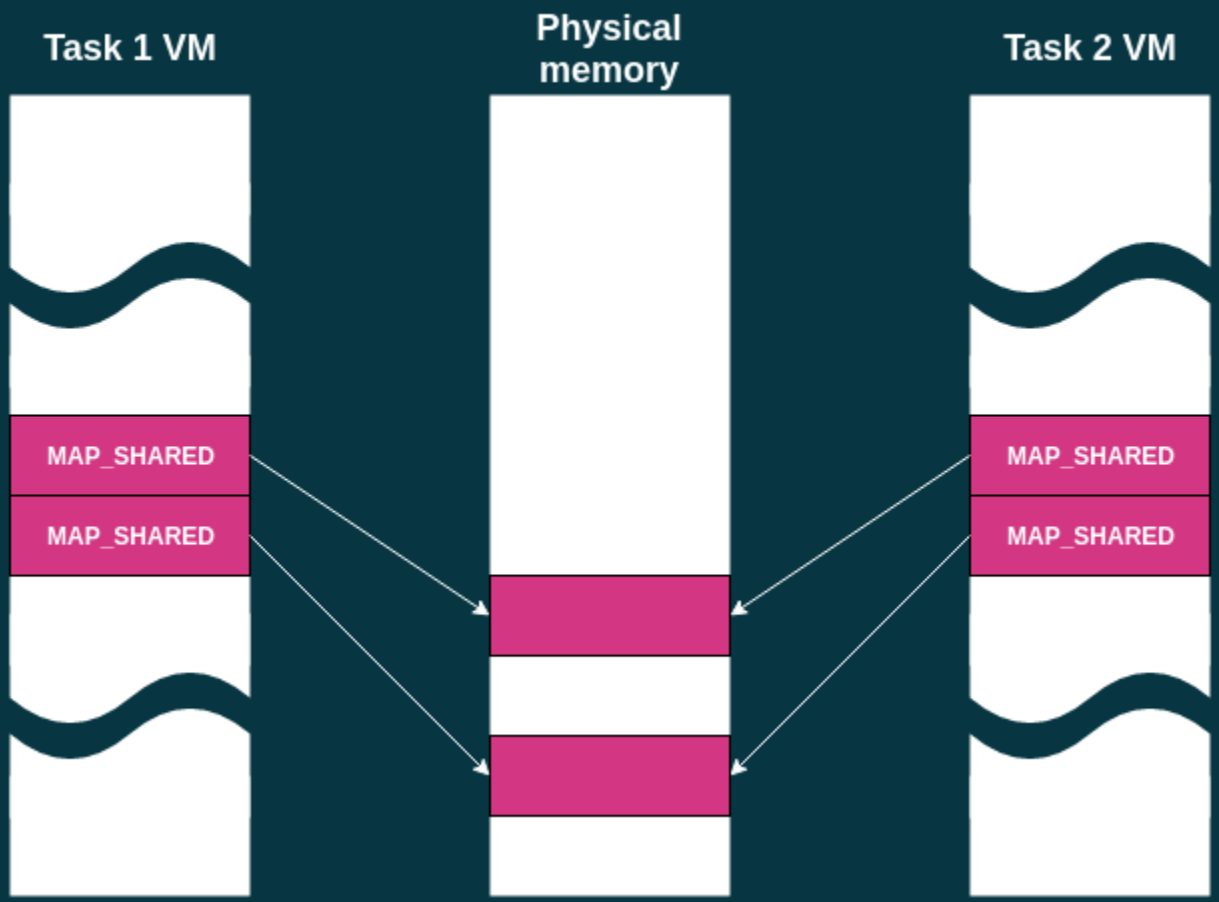
SHARED MAPPING (1/3)



SHARED MAPPING (2/3)



SHARED MAPPING (3/3)



LET'S MAP PRIVATE MEMORY

```
mmap(NULL, 4096, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANON, -1, 0);
fork();
```

- Find free space in the address space
- Add a new `mem_zone` marked as private
- During the fork, clone every pages marked as present in private mappings for the new task

LET'S MAP SHARED MEMORY

```
mmap(NULL, 4096, PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANON, -1, 0);
fork();
```

- Find free space in the address space
- Add a new `mem_zone` marked as shared
- During the fork, copy every page table entries in shared mappings for the new task
- When demand paging occurs, propagate the newly mapped page in other tasks' memory

HOW TO DO THIS ?

- We need a way to find `mem_zone` from the same mapping
- When we need to do demand paging, add the new page to all the shared memory zones

FILE MAPPING

FILE MAPPING IN USERLAND

```c
int fd = open("my_file", O_RDONLY);
char* file_ptr = mmap(NULL, 4096, PROT_READ, MAP_PRIVATE, fd, 0);
/* Access to the file's content through 'file_ptr' */
```

LET'S USE THE PAGE FAULT HANDLER TO READ THE FILE

OH WAIT...

NOT ALL FILES ARE CREATED EQUAL

Some files that have a custom behaviour when they are mmapped like:

- Unseekable files (like pipes)
- Many files in `/dev/*` shouldn't be mmappable
- But some are (e.g. `/dev/zero`)

AND IN KERNEL ? (1/3)

- Create a new `mem_zone` for the mapping
- In the `mem_zone`, store we store the file and the offset
- Use the file's `mmap` file operation on the `mem_zone`.
- The `mmap` file operation fills `mem_operations` of the `mem_zone`
- We let the task continue

AND IN KERNEL ? (2/3)

- If the task tries to access an unloaded page, we go in `page_fault_handler`
- In `page_fault_handler`, we retrieve the `mem_zone` of the fault
- Call the `load` memory operation of the `mem_zone`
- In the `load` memory operation, we fill the page with the data

AND IN KERNEL ? (3/3)

- When the task munmap the mapping, we call the `release` memory operation
- On standard files, the `release` memory operation writes dirty pages back to the file

COPY ON WRITE

MAN TO THE RESCUE (1/2)

`man 2 mmap`

- MAP_PRIVATE: Create a private copy-on-write mapping.

MAN TO THE RESCUE (2/2)

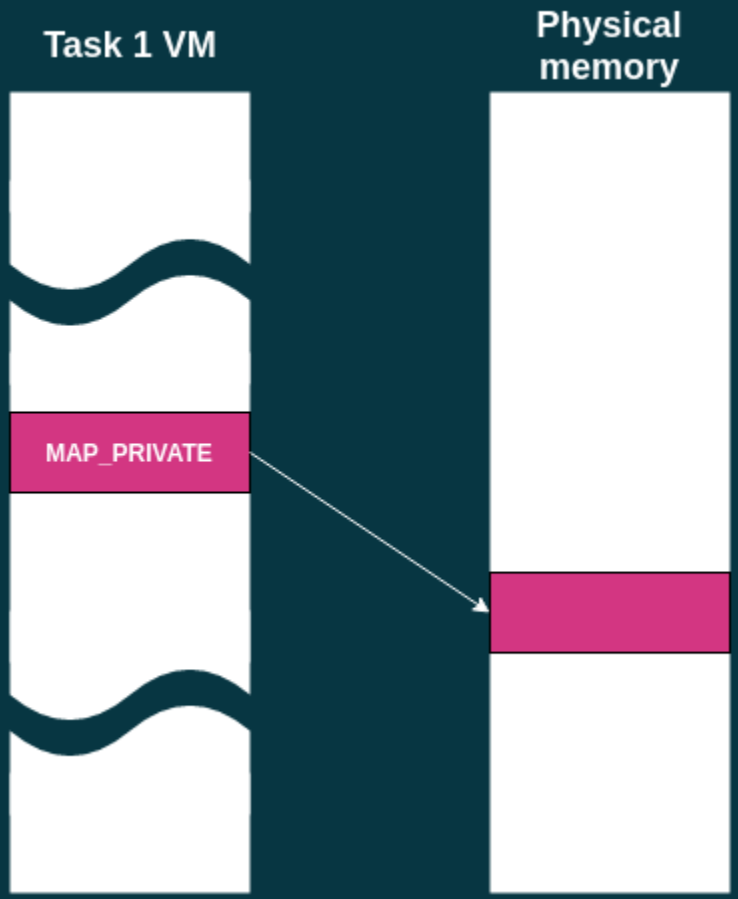
```
man 2 fork
```

- fork() is implemented using copy-on-write

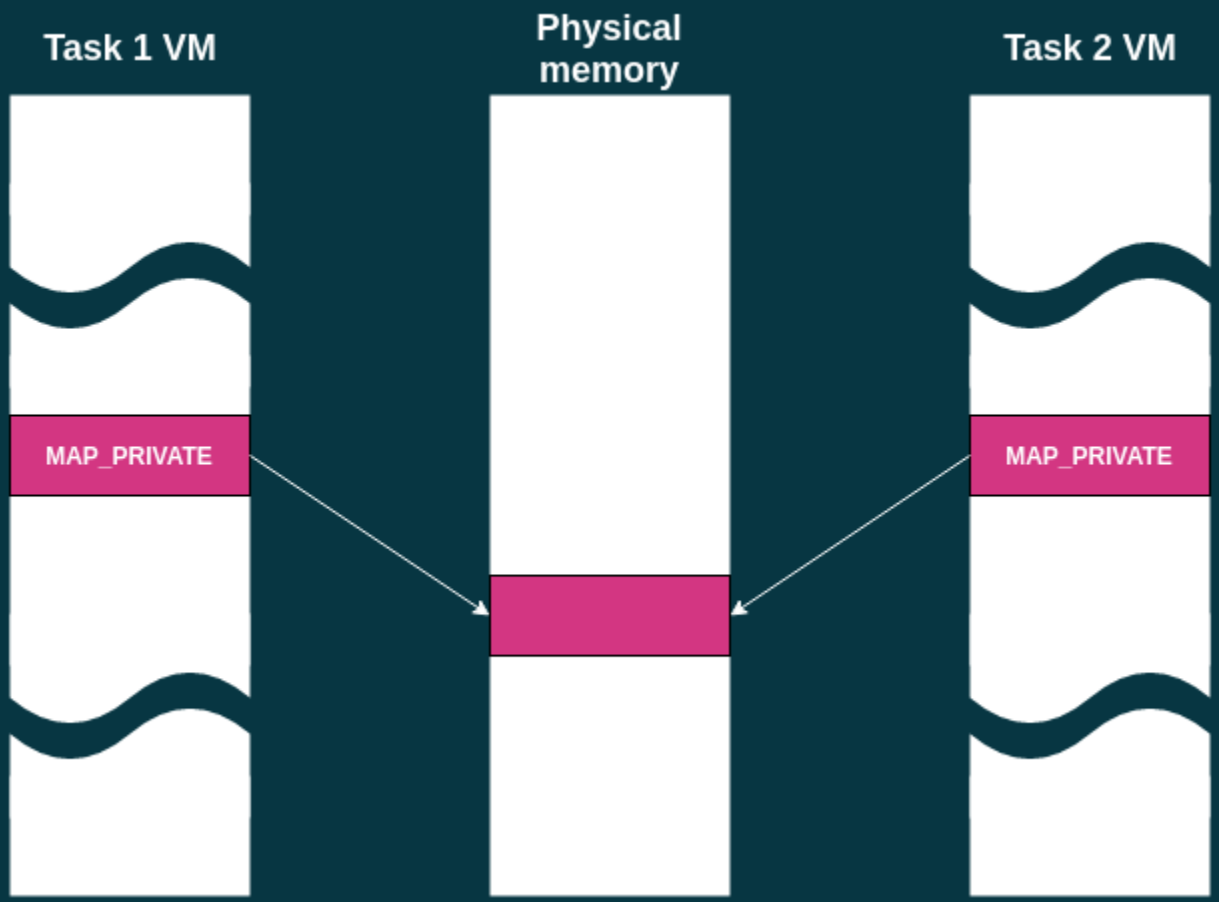
COPY ON WUT?

- After a call to `fork(2)`, a new task is created, identical to its parent, appart from the return value of fork
- We don't want to copy every pages from the parent to the new task
- Copy on write lets us copy the data at the last moment

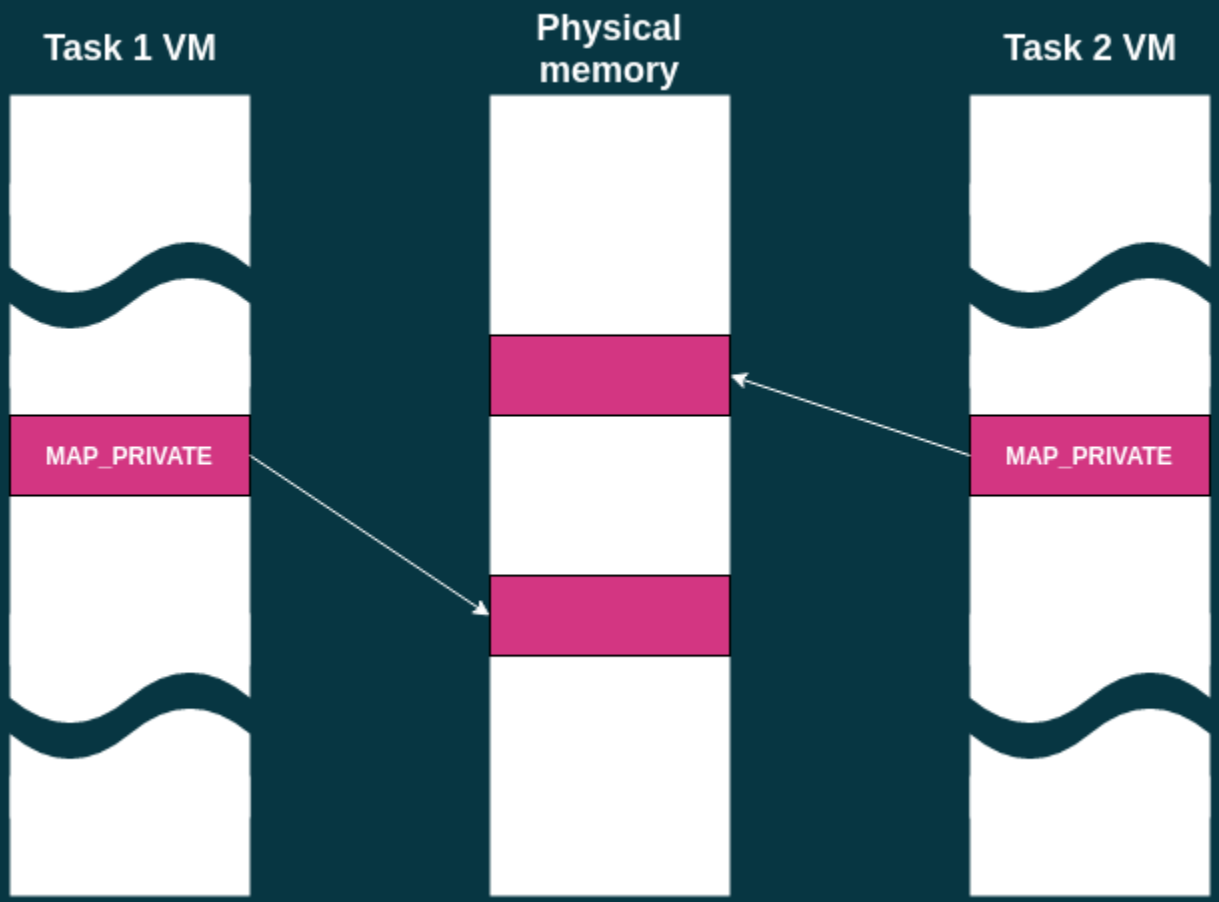
COW (1/3)



COW (2/3)



COW (3/3)



MAKE FORK(2) GREAT AGAIN!

When forking:

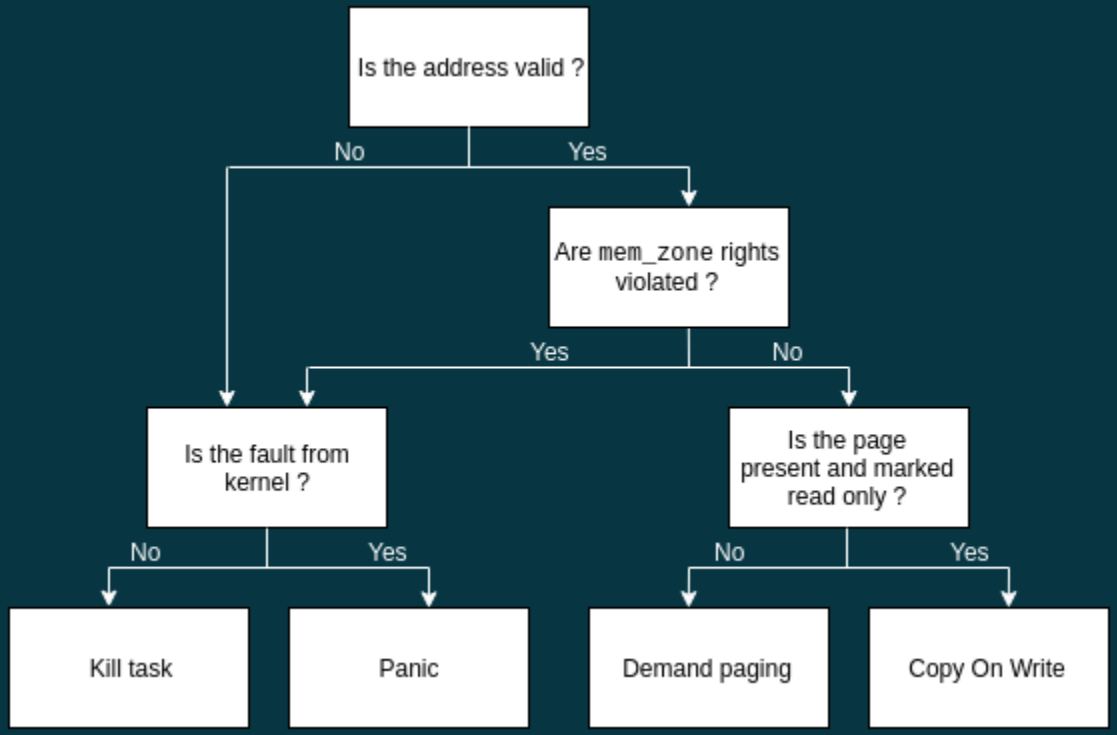
- Mark every PTE in `MAP_PRIVATE` mappings as read only
- Copy the content of the PTE from the parent to the child page table
- They now point to the same physical page

COPY ON WRITE

When writing:

- A page fault happens
- In the fault handler, we can see when we try to write in a read only page, but in a read/write `mem_zone`
- In the fault handler:
  - allocate a new frame
  - copy the content of the read only frame in the new frame
  - set the PTE to be read/write and point it to the new frame

FAULT


Is the address valid ?
No
Yes
Are mem_zone rights violated ?
Yes
No
Is the fault from kernel ?
Is the page present and marked read only ?
No
Yes
No
Yes
Kill task
Panic
Demand paging
Copy On Write

NEXT ?

QUESTIONS ?