Dirty COW

Race condition vulnerability in Linux memory management kernel code

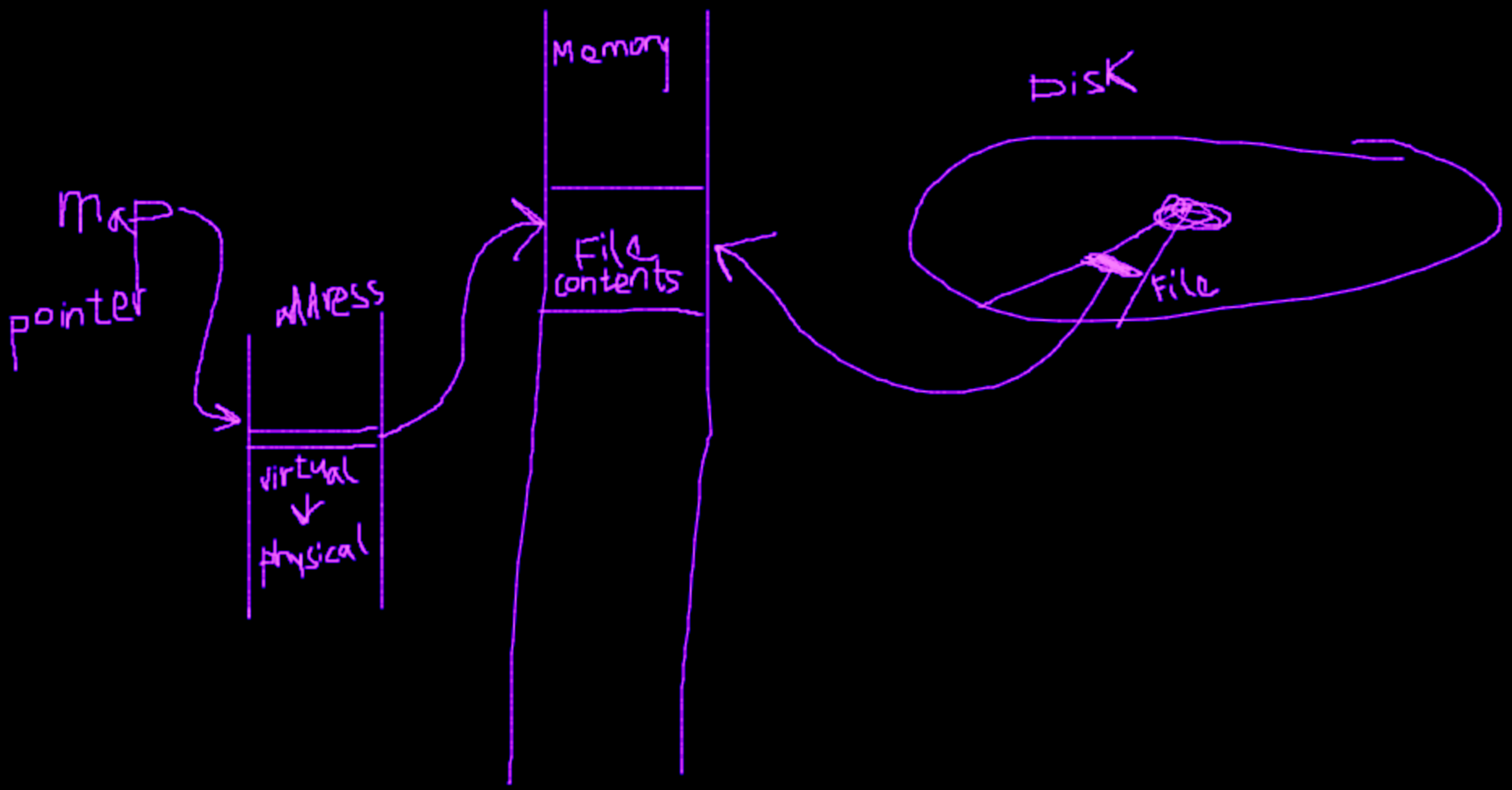
Jakub Hladik
CprE 431
Fall 2017

Dirty COW?

- Dirty = page in memory has been modified
- COW = copy-on-write...
- Patched and fixed in October 2016 in majority of Linux distributions and Linux kernel
- But... Android...

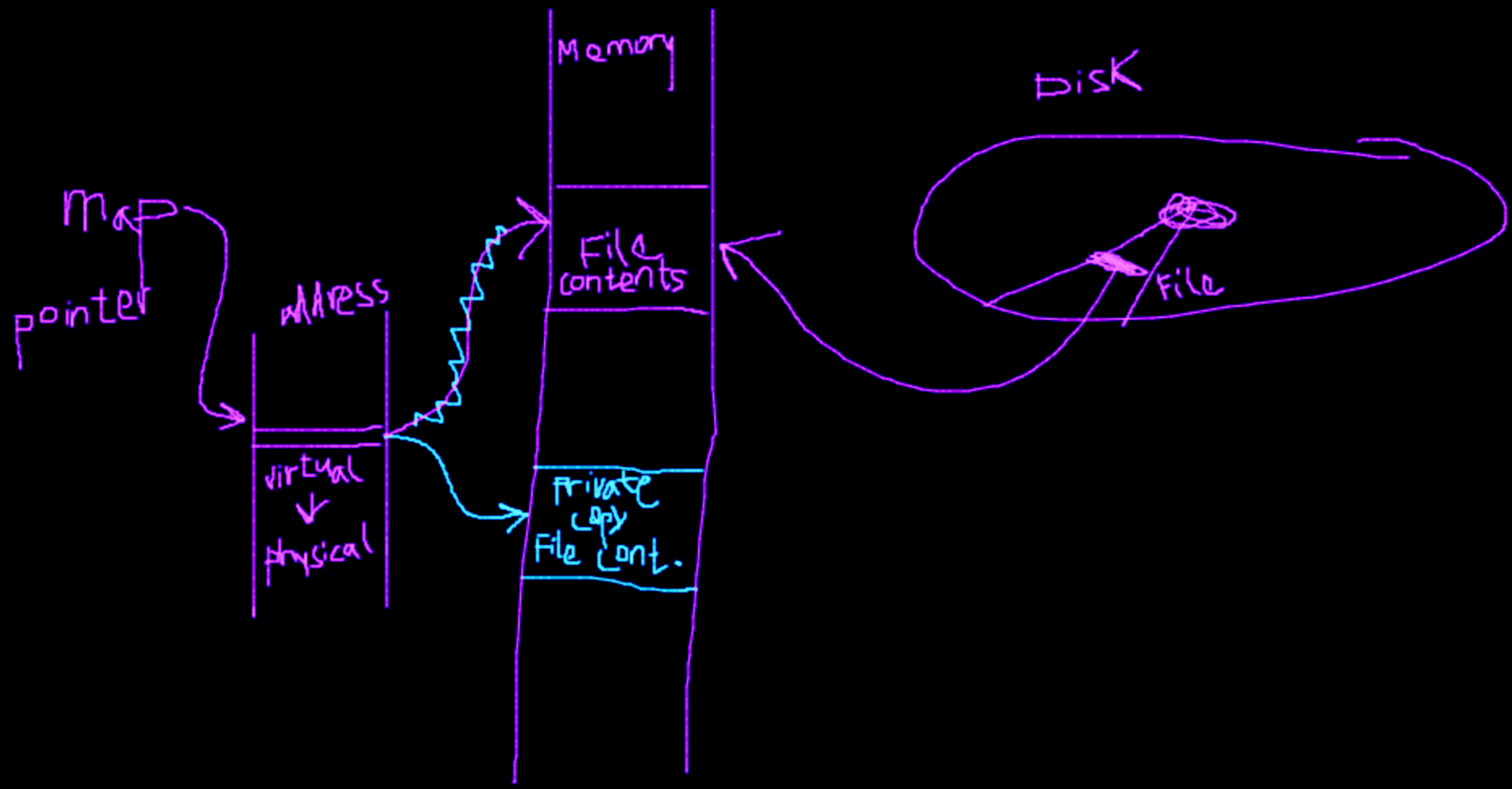
Copy-on-write?

- User can directly map a file into memory using a system call **mmap()**
- User changes file in the memory, the page gets marked "dirty" and the kernel stores changes back to the disk upon getting rid of the page


Memory
Disk
Map
pointer
address
File contents
File
virtual
↓
physical

So what?

- User can also map read-only files from the disk to the memory
- We can write into the mapping but the kernel will provide us with a private copy **(hence, copy-on-write)** with the same virtual address but different physical address
- If we made changed to this private page and the kernel whats to get rid off it, nothing happens because this page is not directly mapped to the read-only file

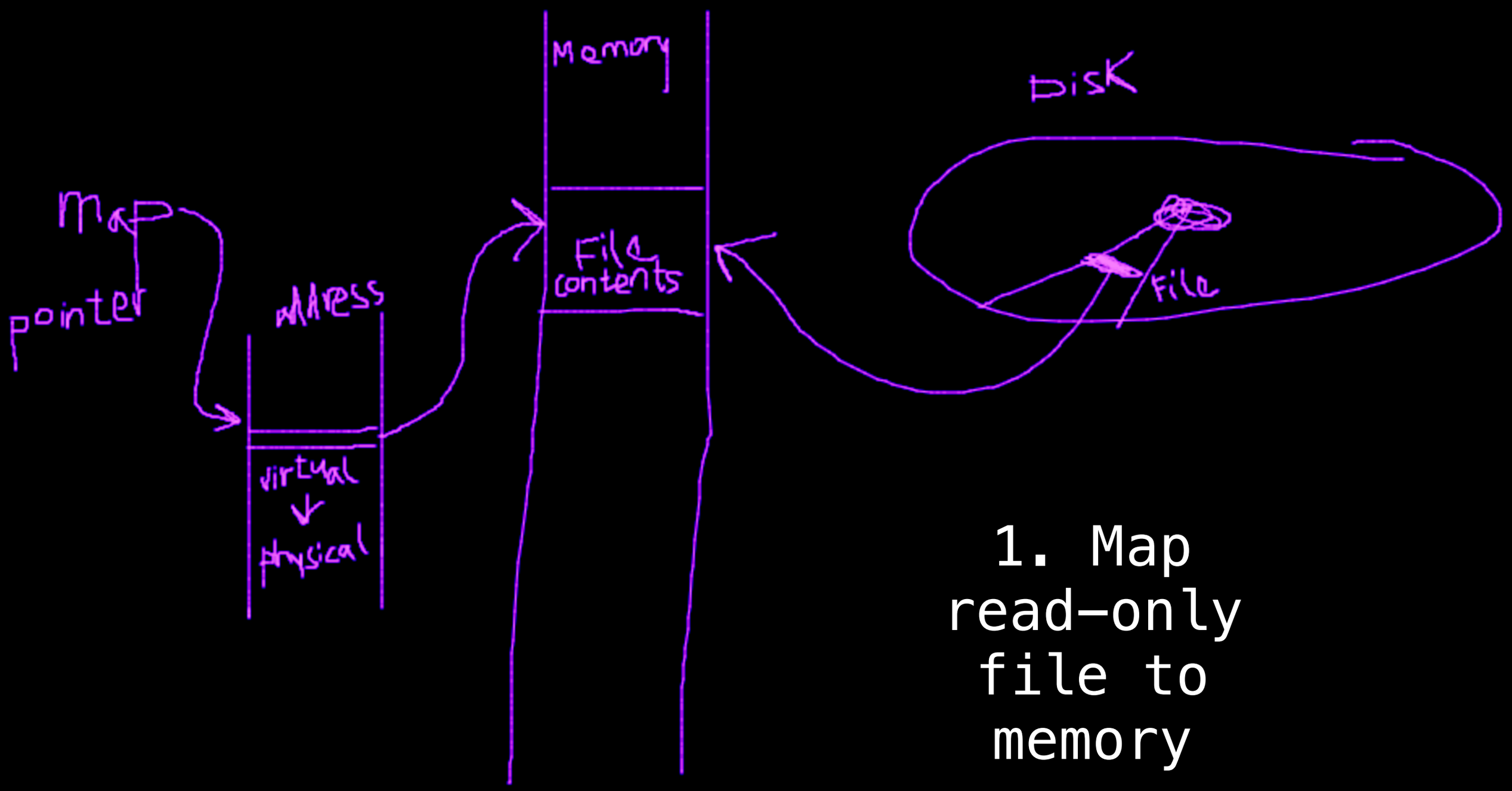

Memory
Disk
File contents
File
map
pointer
address
virtual
physical
Private copy
File Cont.

Race condition

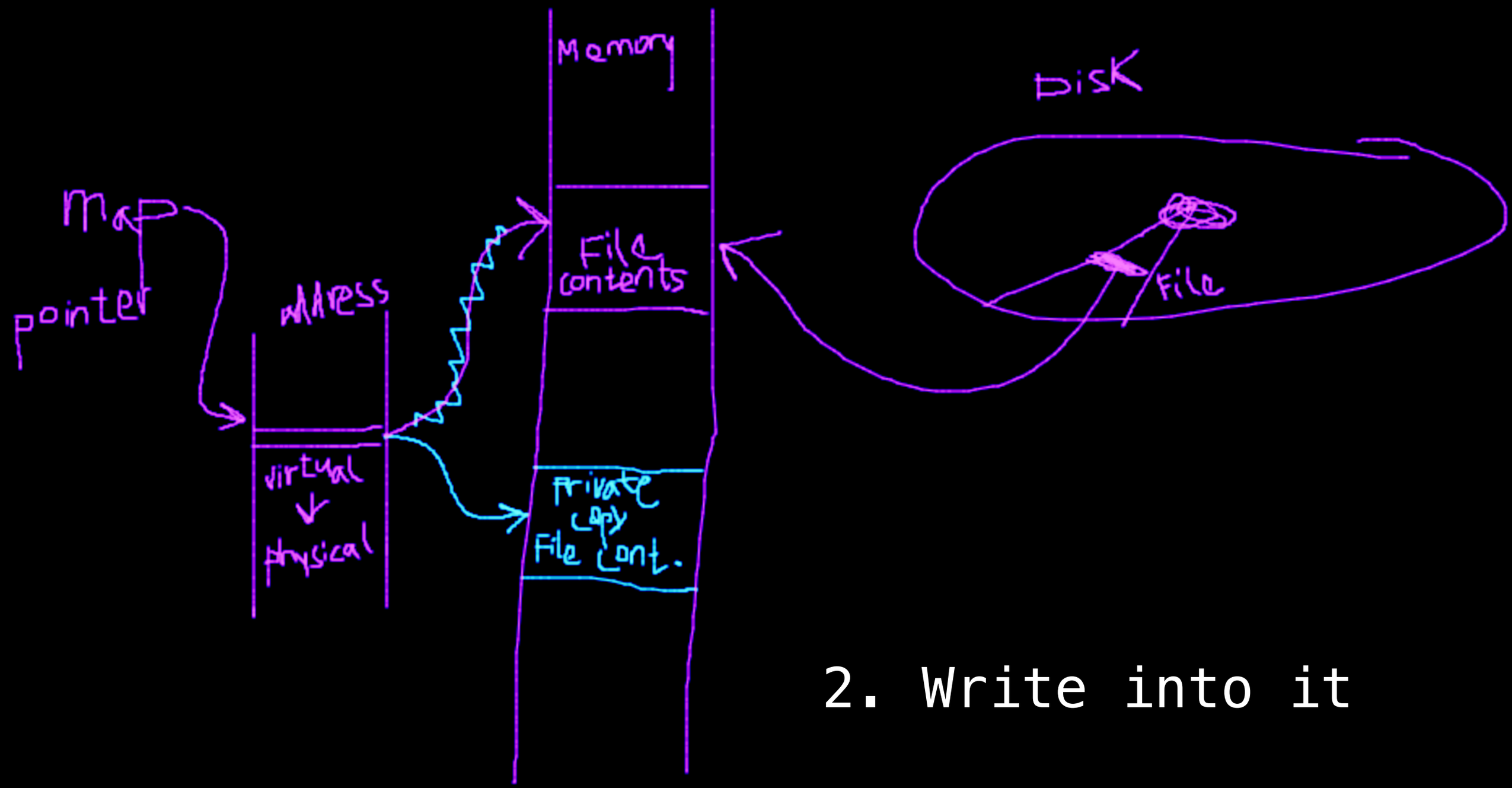
- We can also utilize a system call **madvise()** to let the kernel know what we intend to do with the memory mapping
- This system call aids the page replacement algorithm make better decisions
- **madvise(MADV_DONTNEED):** tell kernel the memory map is no longer needed (get rid off the page in the future)
- If we call **madvise()** on a read-only mapping, it first deallocates the private copy, then changes the mapping to the original mapping and then deallocates it

Two threads

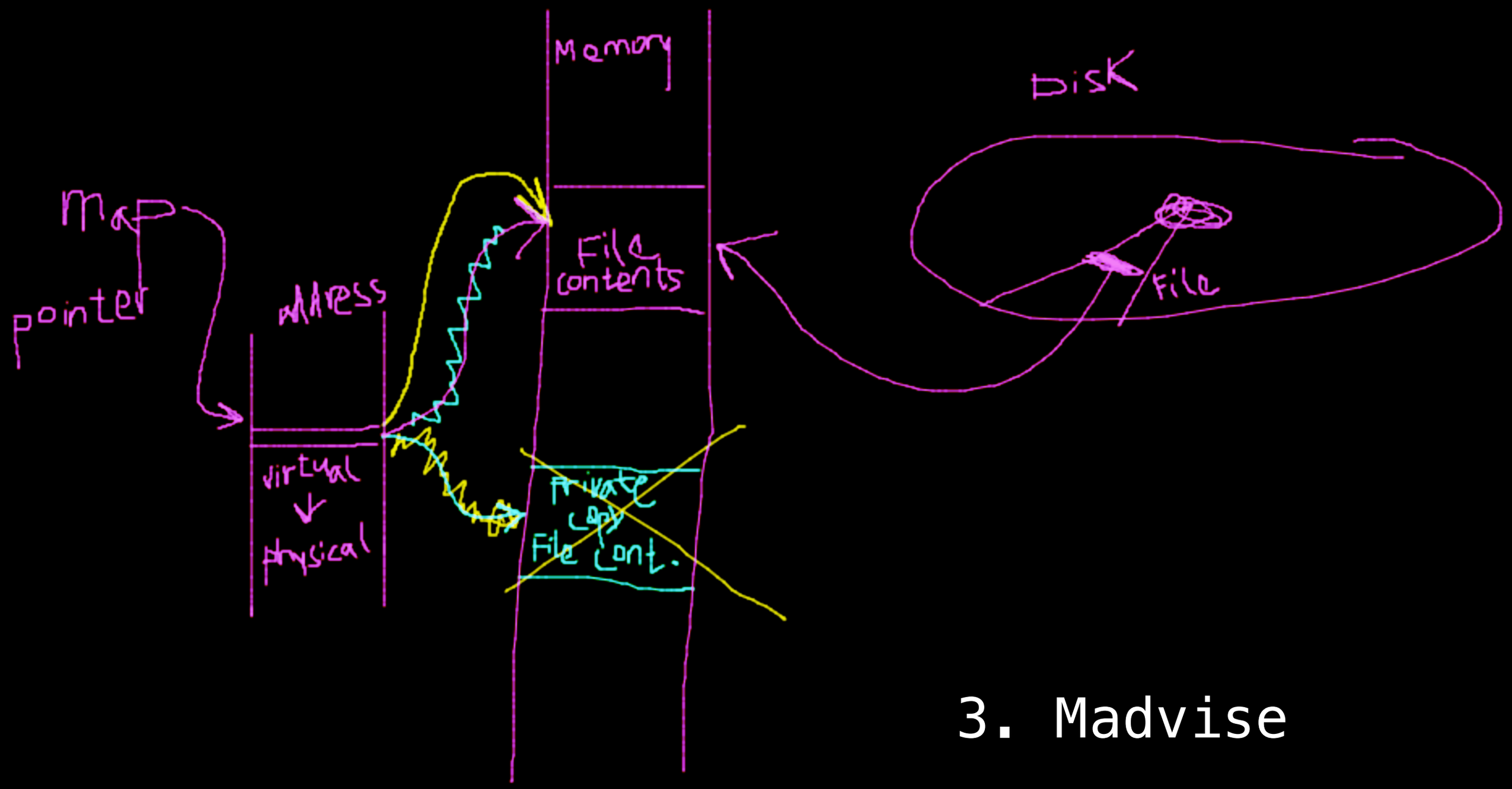
- Map a read-only file to the memory
- Create 2 threads:
  - **Write thread**: constantly writes into the mapped memory
  - **Madvise thread**: constantly tells the kernel the mapped memory will not be needed in the future



1. Map read-only file to memory



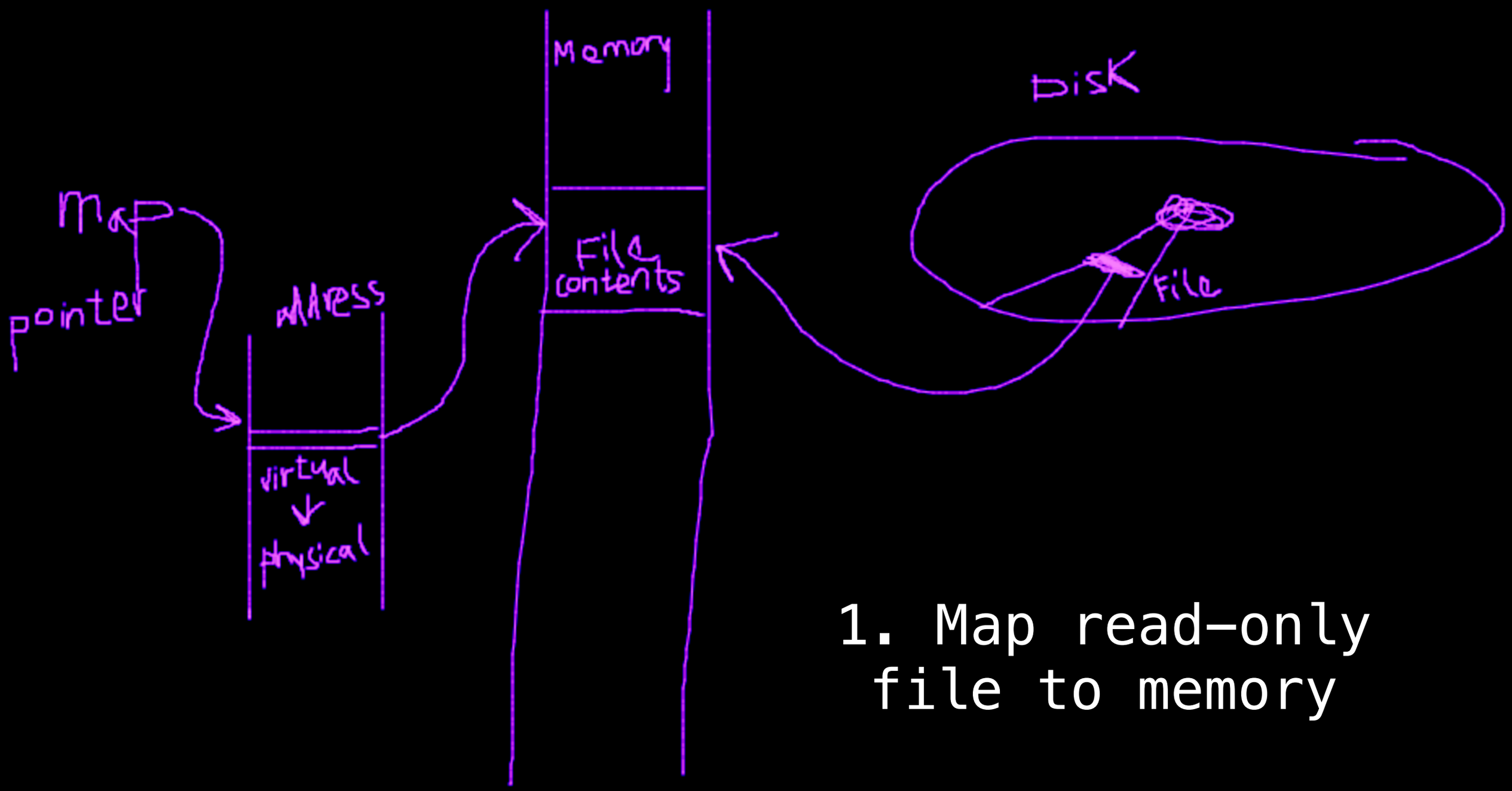
2. Write into it


Memory
Disk
File contents
File
map
pointer
address
virtual
physical
private
copy
File Cont.

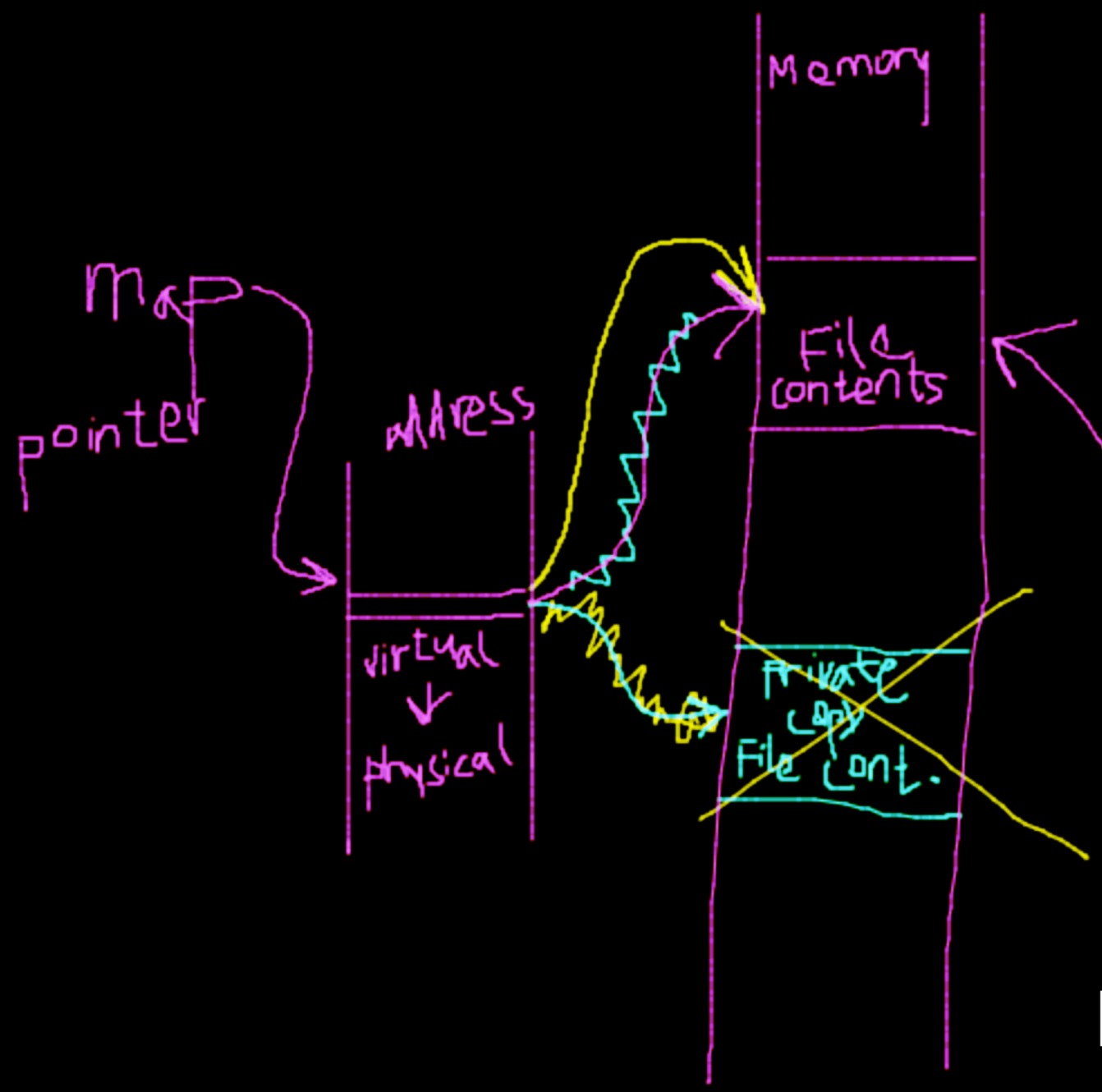
3. Madvise

So what's the problem?

- Well, since the **write()** and **madvise()** operations are not atomic, we can get a race condition where **madvise()** gets executed after we create a private copy and we end up writing into the original mapped memory which should have been read-only
- Later, the kernel sees the dirty flag on the original page and writes changes to the disk
- Effectively, we eliminate the copy-on-write operation and we write into the original read-only file

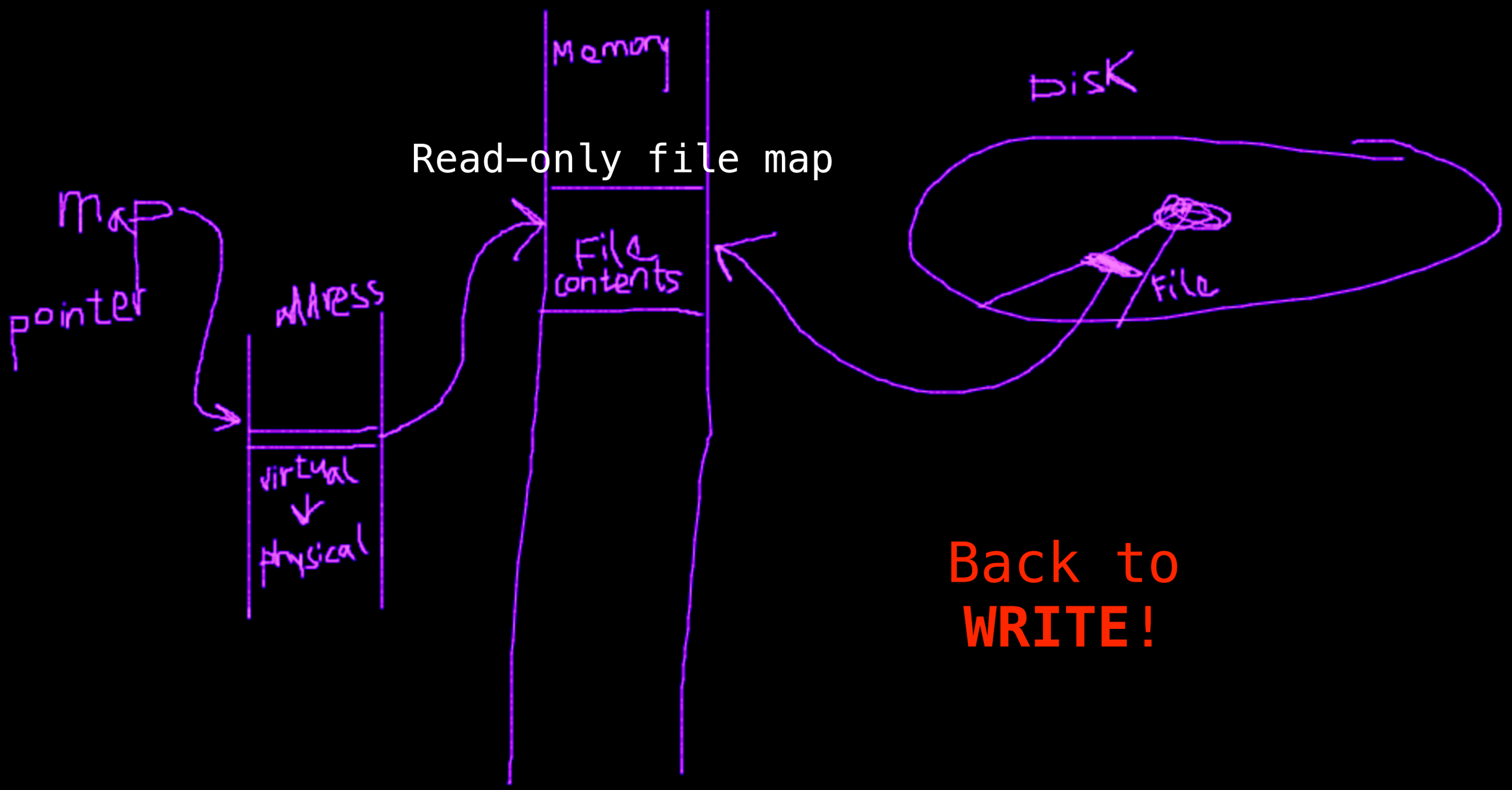


1. Map read-only file to memory



Write + Context Switch after creating private copy but before write + Madvise (deallocate the copy) + Context Switch

Read-only file map



Back to
WRITE!

How to carry out the attack?

- Need access to the device as a regular user and execute code which runs the two racing threads
  - Malicious software that the target willingly downloads (including Android apps)
  - Other attacks that will get us into the device as a regular user

Demo

https://youtu.be/_e6FU_NaM9U