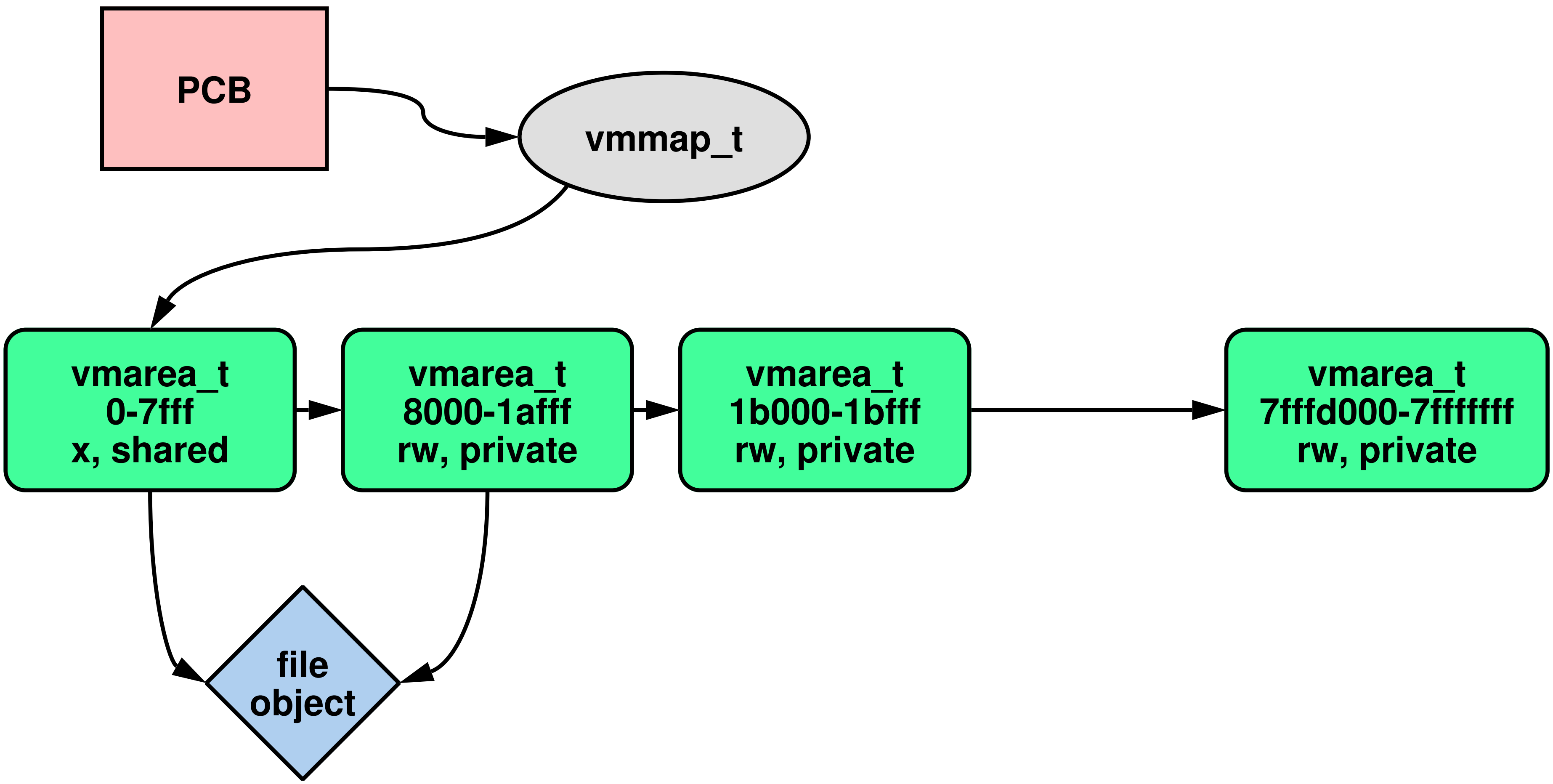
More On Kernel 3

Bill Cheng

http://merlot.usc.edu/william/usc/

Address Space Implementation

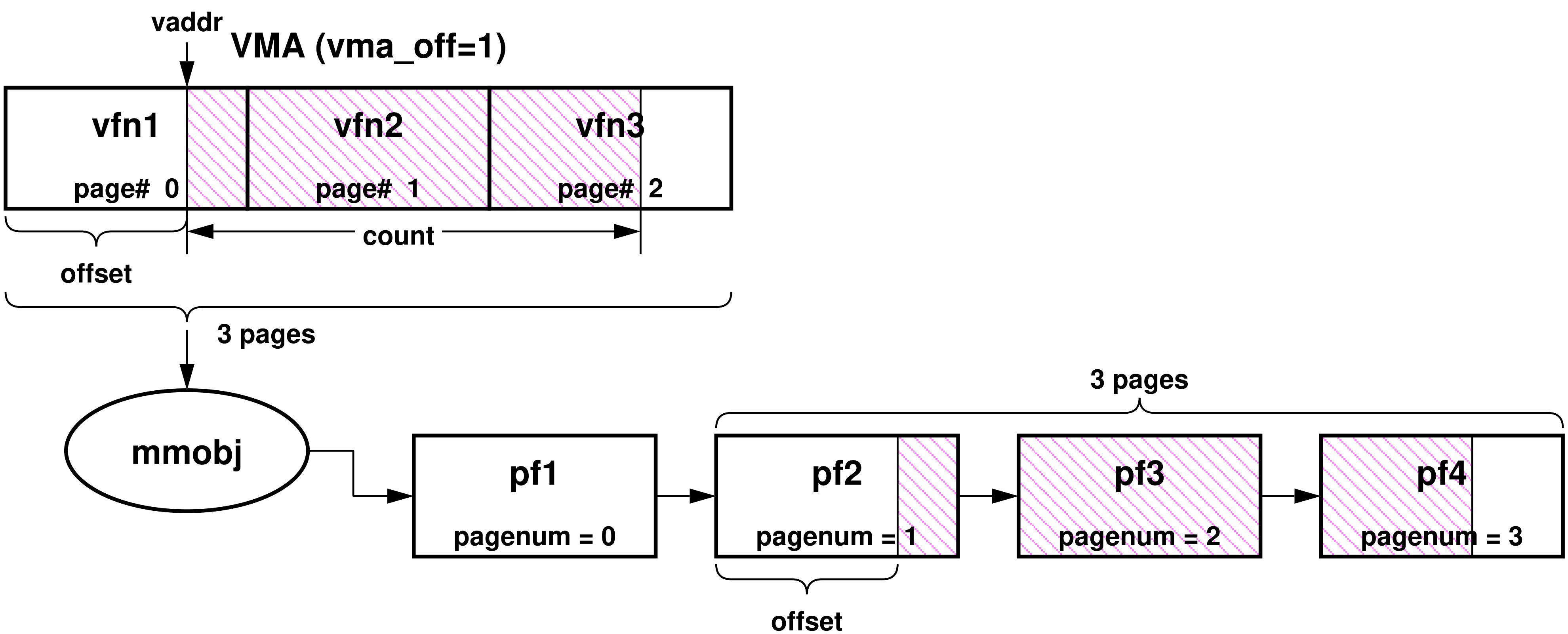
- *Address Space* is implemented using *Virtual Memory Map* (`vmmap`)
  - in lectures, sometimes I used the term "memory map"



  - we will use `vma` or `vmarea` to refer to `vmarea_t`
  - the values in `vmas` above are not what's in weenix

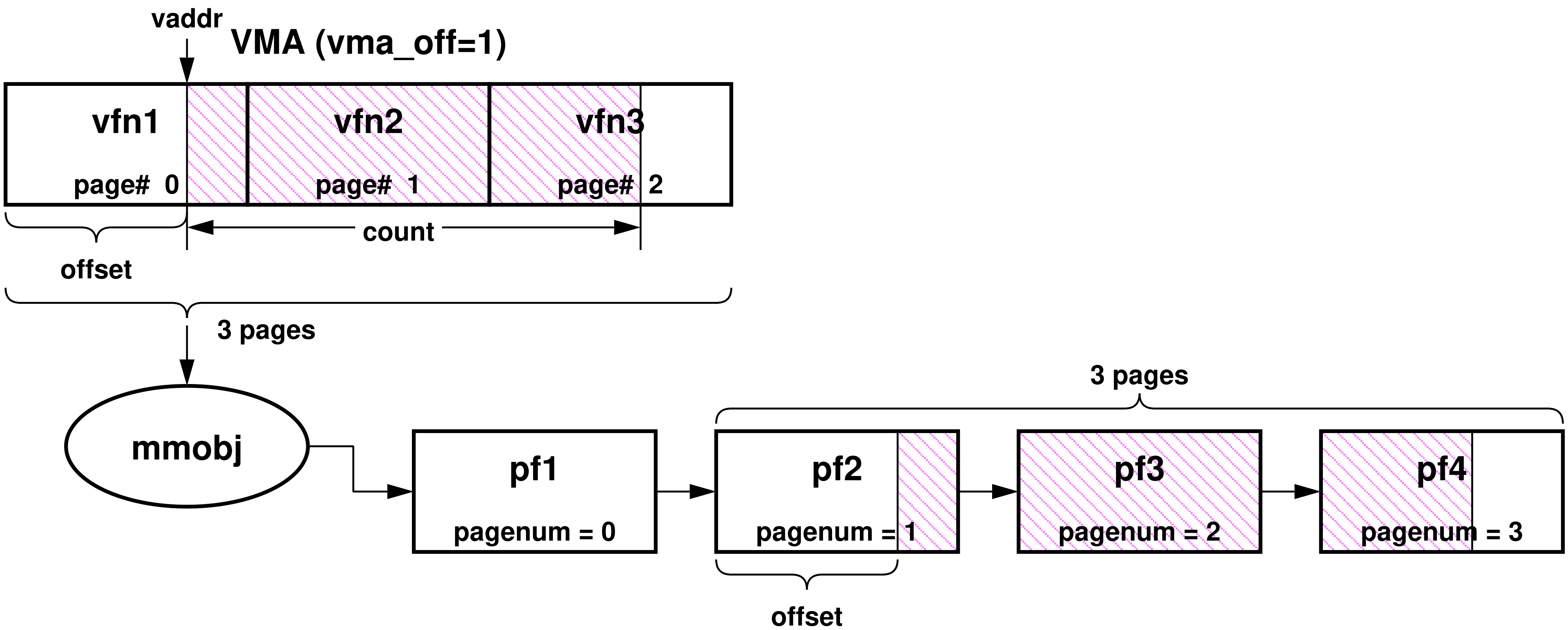
mmobj and pagenum

- This is a very important picture in the kernel FAQ to understand



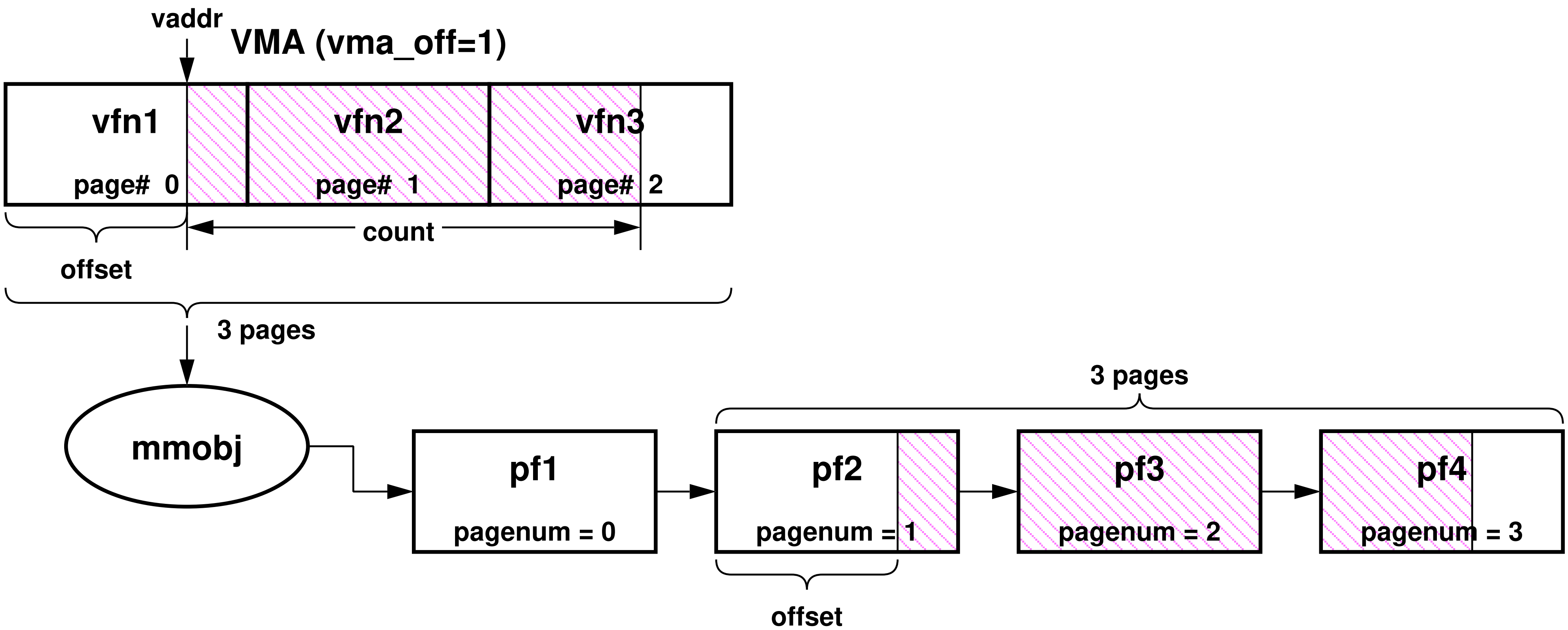
- read the kernel FAQ about "pagenum" and the next few FAQ items that follow it
- these are crucial in understanding how to build an address space for a user process

mmobj and pagenum



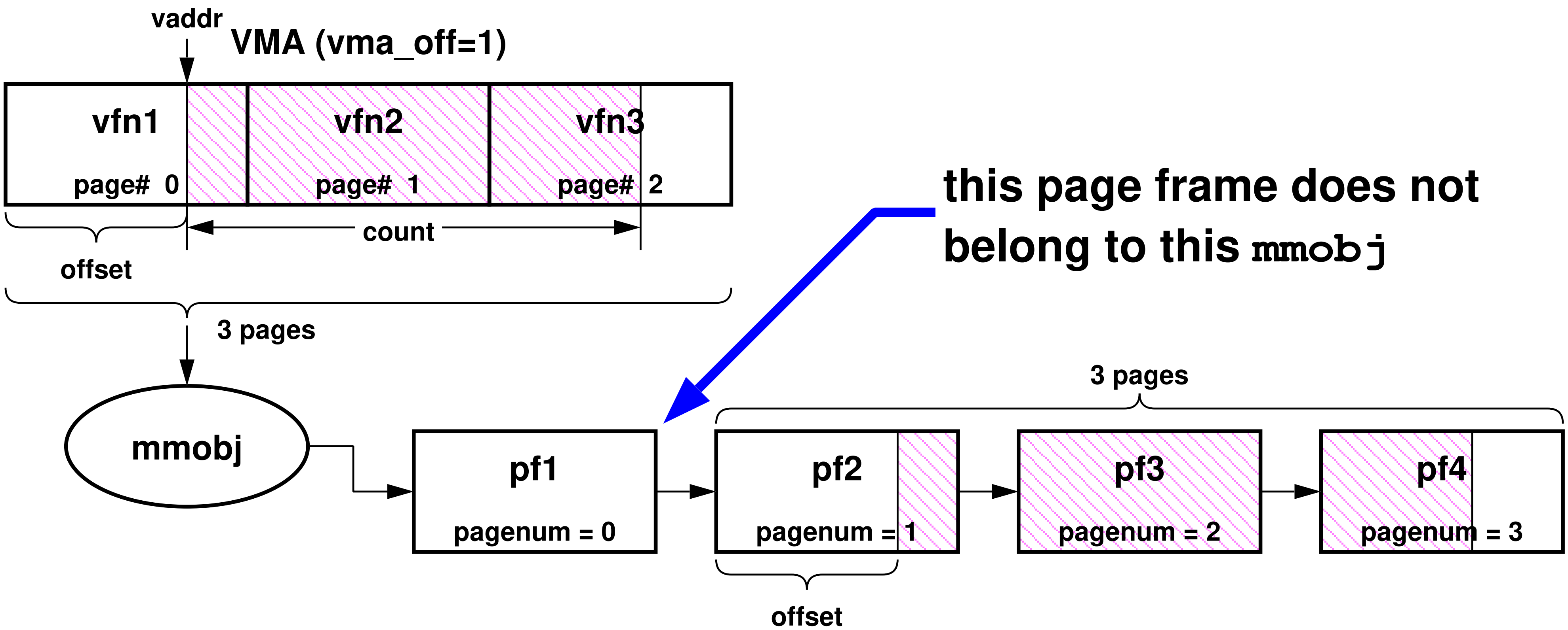
1) The address space is made up of a list of *non-overlapping* `vmareas`
2) Each `vmarea` is a memory segment (contiguous virtual memory locations)
   - in the above, it's shown as `[vaddr,vaddr+count)`
     - typically `vaddr` (first virtual address of a memory segment) is *page aligned*

mmobj and pagenum



3) The kernel manages VM in "pages" (not "bytes"), it allocates enough pages so that a memory segment can fit inside
   - above, it takes 3 virtual pages to cover this memory segment
   - each virtual page need to be mapped into a physical page
     - `vfn` = `vpn` = virtual page number (20-bits long)
     - you need to get used to printing these numbers in *hex*

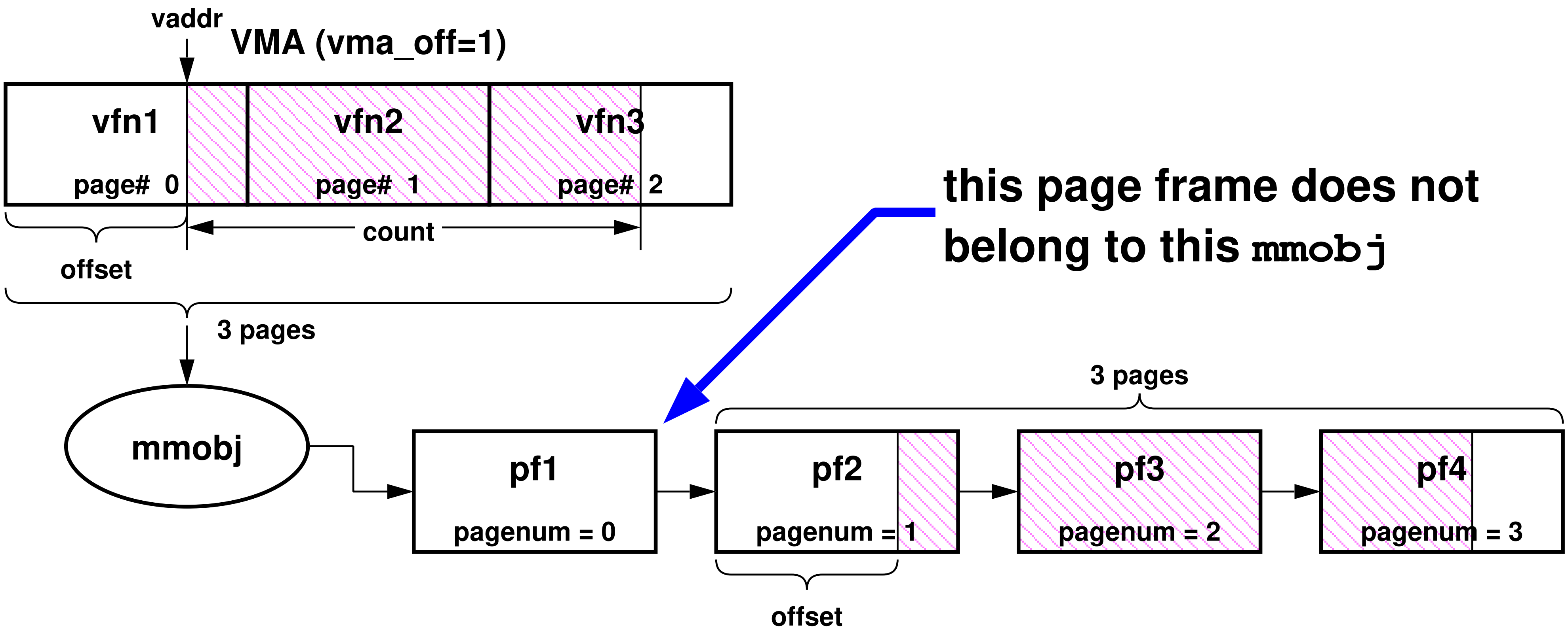
mmobj and pagenum



4) Page frames are managed by `mmobjs`

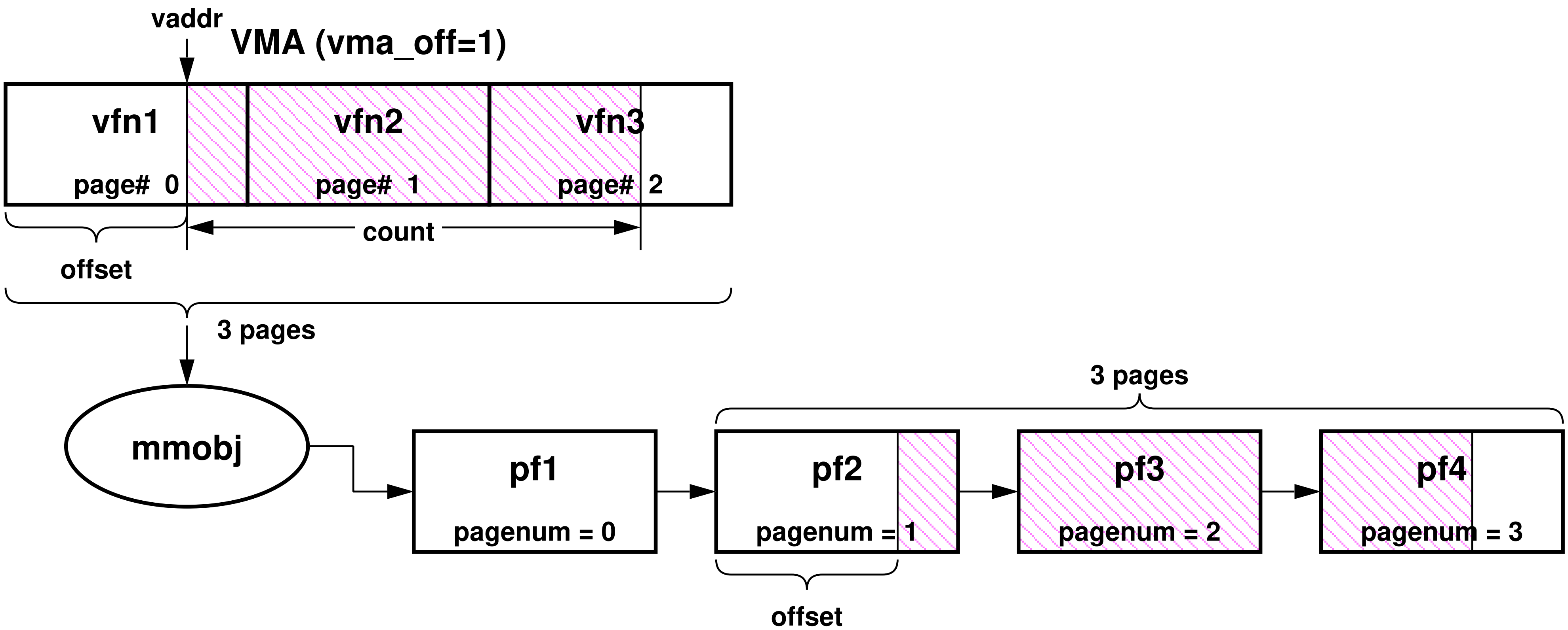
- there is a physical page (hidden) inside each page frame
- if an `mmobj` manages `N` page frames, their `pagenums` are `[vma_off, vma_off+N)` for that `mmobj`
- *not really implemented as a linked list* as shown above
  - a *hash table* is used

`mmobj` and `pagenum`



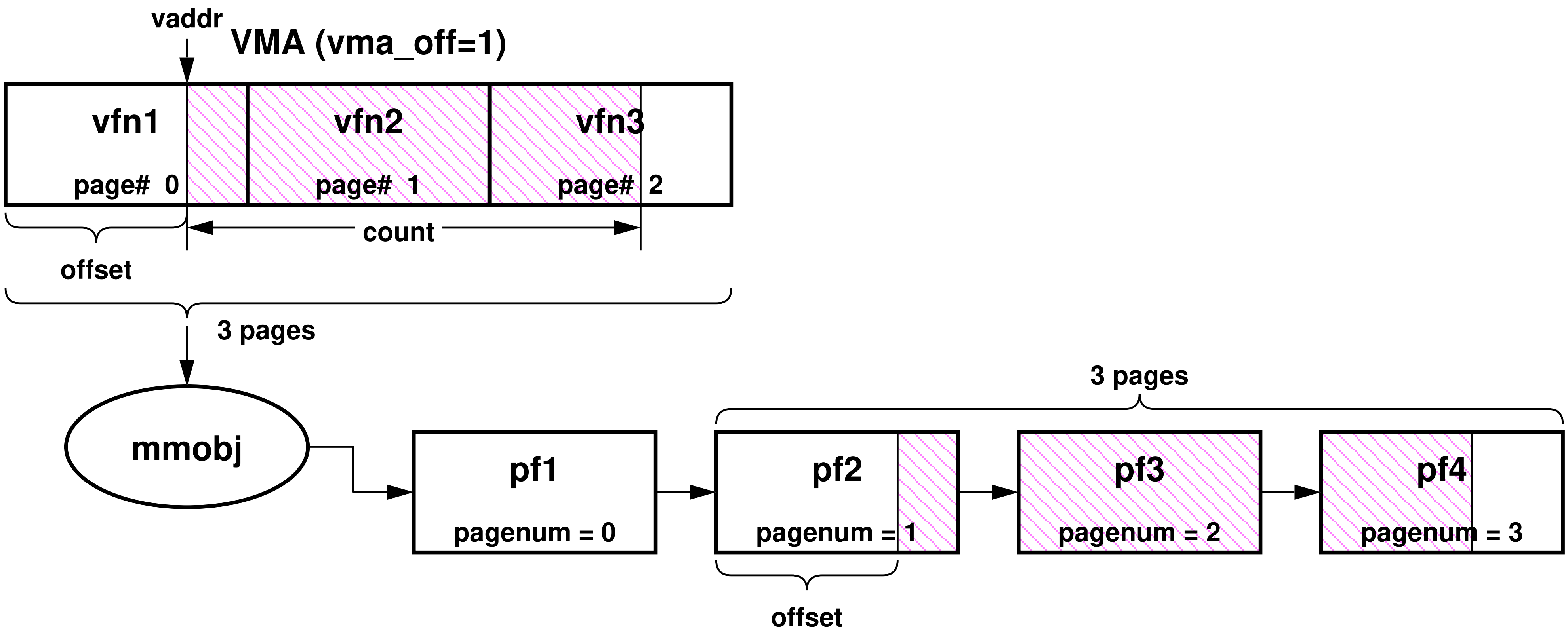
5) You can create multiple memory segments by mapping different pages of a file into your address space
   - conceptually, a file is divided into pages
   - `vma_off` in a memory segment gives you the starting page number of the file

`mmobj` and `pagenum`



6) A page frame may not be present for an `mmobj`
- since we are doing *demand paging*, in the beginning, none of the page frames are present for an `mmobj`
- as page frames are brought in, they become *"resident"*
  - i.e., they are *cached* inside the corresponding `mmobj`
  - if modified, the page frame becomes *"dirty"*

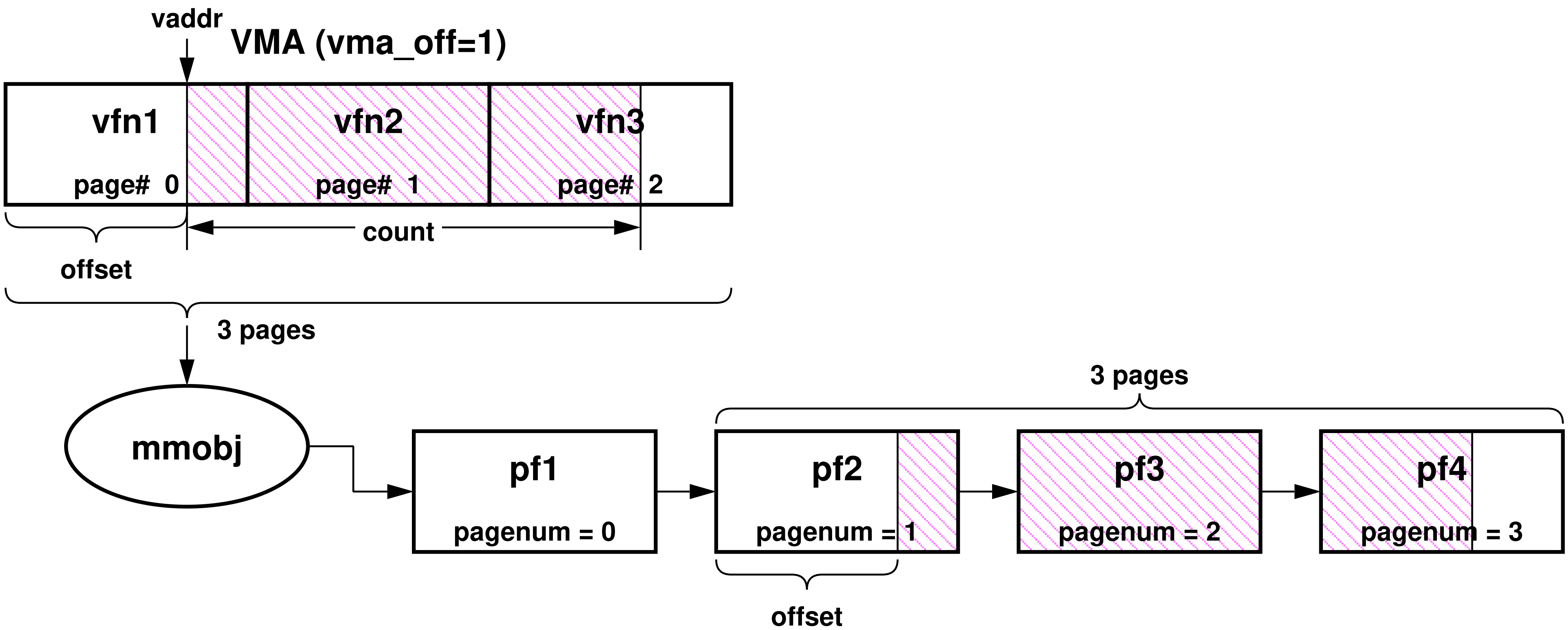
mmobj and pagenum



7) A page frame is identified by the `mmobj` that manages it and the `pagenum` of the page frame

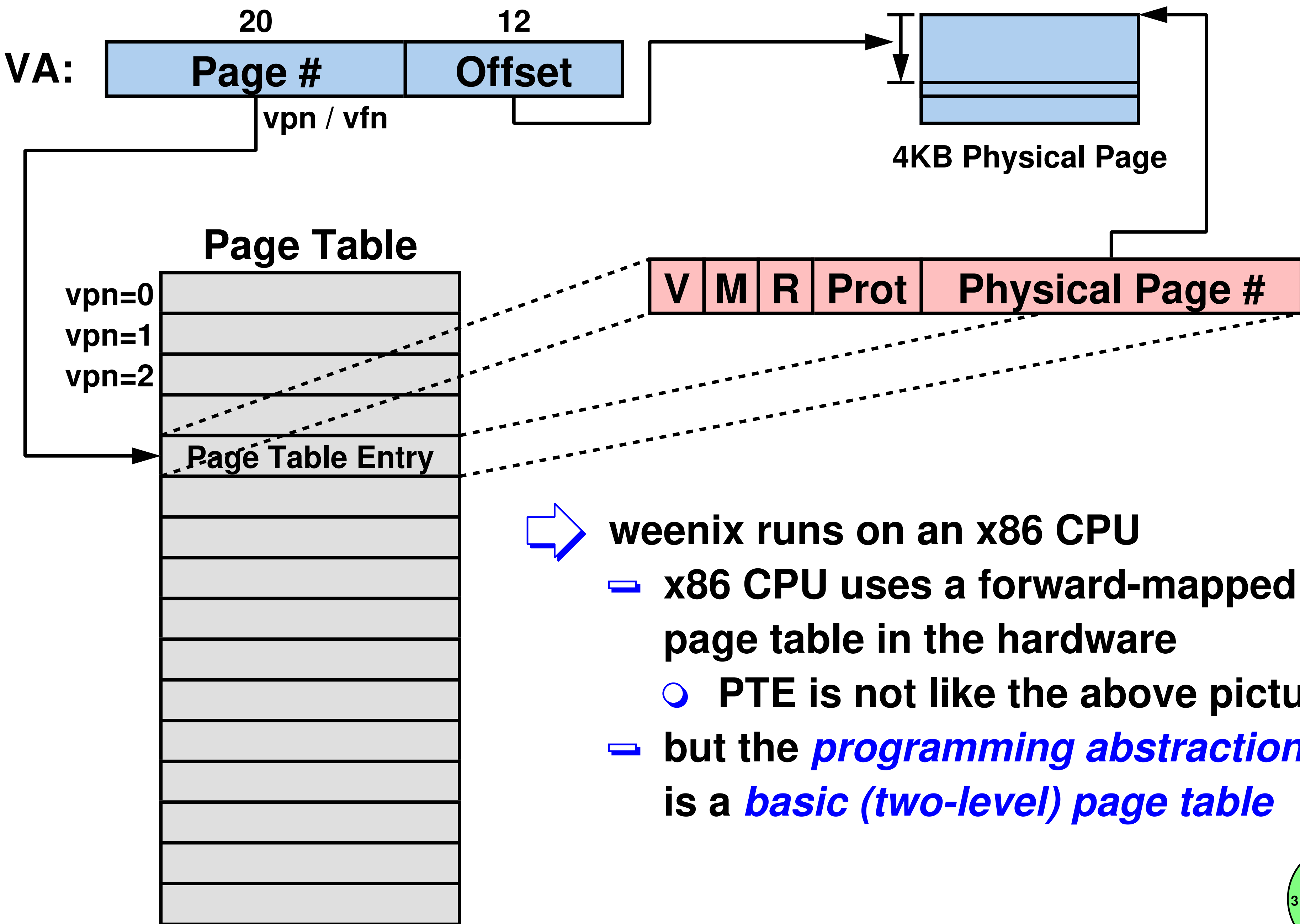
- in the kernel FAQ, it uses the notation *(o,n)* where *o* is an `mmobj` and *n* is a `pagenum`
- a *linked list* of `mmobjs` must be used to make *copy-on-write* work correctly with `fork()`

`mmobj` and `pagenum`



8) Page table maps a *virtual page number* to a *physical page number*
   - in the above picture, `vfn1`/`vfn2`/`vfn3` needs to be mapped to the physical page that lives inside `pf2`/`pf3`/`pf4`, respectively
   - `pf_addr` of a page frame contains a *kernel virtual address* for the corresponding "physical page"
     - `pt_virt_to_phys()` converts it to physical address

Forward-Mapped (Multilevel) Page Table



- weenix runs on an x86 CPU
  - x86 CPU uses a forward-mapped page table in the hardware
    - PTE is not like the above picture
  - but the *programming abstraction* is a *basic (two-level) page table*

Very Useful gdb Commands

- Address space

```
kernel info vmmap_mapping_info curproc->p_vmmap
```

  - the *loader* builds address space for a user-space program
  - after your user-space program is "loaded", what does the address space looks like?
  - when you get your first legitimate page fault in `handle_pagefault()`, look at your address space
    - if it's is wrong, is there point proceeding?!
    - ask your classmates in the class Google Group if they are seeing the same thing
      - don't just ask others to share, that's not nice!

- Page table (not that useful)

```
kernel info pt_mapping_info curproc->p_pagedir
```

Read Some Code

- **User**
  - "hello.c"

```
int main(int argc, char **argv)
{
  open("/dev/tty0", O_RDONLY, 0);
  open("/dev/tty0", O_WRONLY, 0);
  write(1, "Hello, world!\n", 14);
  return 0;
}
```

- **Kernel**
  - "elf32.c" - the loader
  - "access.c" - need to understand `copy_from_user()` and `copy_to_user()`
  - "exec.c" - how to go into user space
  - "pagefault.c" - handle page fault
  - "vn_mmobj_ops.c" - code for the `mmobj` inside a `vnode`
  - "syscall.c" - to see how to implement `sys_write()`, look at how other `sys_*()` functions are implemented