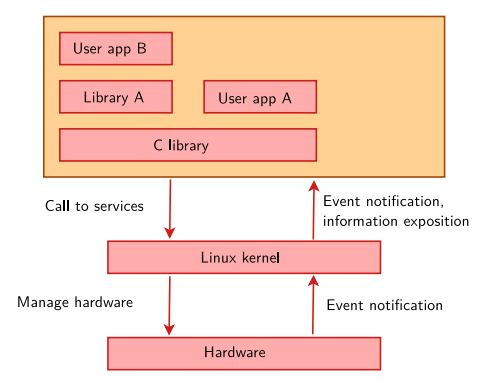
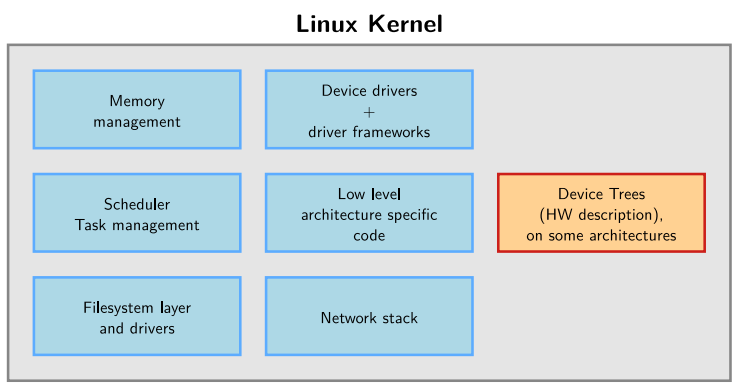
Driver



The Linux kernel manages the hardware hardware: CPU, memory, I/O. Besides, it also provides an API to handle that hardware and also handles access from hardware coming from different software

* Inside linux kernel



**Linux code and Device Drivers**

* Use C (not user space code)(same with all UNIX system), no C++, compile by gcc. Contain a little Assembly
* Fixed file stack (4 or 8KB) -> stack overflow -> no recursion
* User space device driver can be written in any language, using API to access hardware. Can use accross kernel (API takes the hardest part). More secure due to less permission. Drawbacks are hard to handle interrupt (take step to call sysfs, syscall) (needed for low-latency task)

> In summary, user space device drivers are generally easier to deploy and test, while kernel space device drivers generally have higher performance and reliability. The choice between these two types of drivers depends on the specific requirements of the application and device.

Cscope, ctags, elixir??

**Building the kernel**

**Kernel configuration**

Create a sequence of drivers, protocols and related components for kernel compiled using Makefile

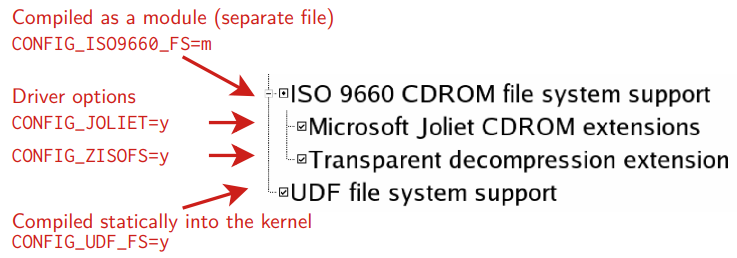
To build ARM on linux, you need

* Set the environment variable **ARCH=arm -> so**that the kernel will build on ARM instead of x86
* Use cross-compilers such as mips-linux-gcc, arm-linux-gnueabi-gcc,... If we code on Linux but deploy in ARM or MIPS,...
* We can pass directly to the make ARCH=arm command CROSS\_COMPILE=arm-linux- ... or export to an environment variable
* FIle .config can be opened as graphical (xconfig, gconfig) or text (menuconfig, nconfig)

**Initial config**: to find info about current kernel that we will work on (kernel config : Kconfig)

Kernel or module?

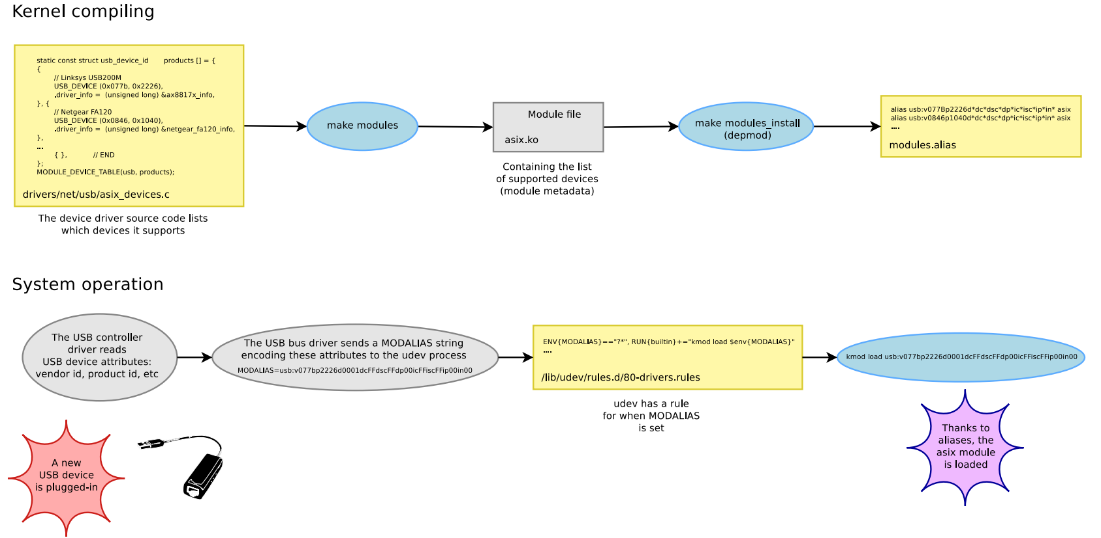
* Kernel as mentioned above is for system administration, resources, memory
* Features such as connecting peripherals, filesystem (not root filesystem) are compiled in module form
* Các Kernel option types, select dependency ???
* Phân biệt compiled as module or into kernel



**Compiling and installing kernel**

* Run multiple compiles if there are multiple cpu cores
* Use the ccache compiler cache: export CROSS\_COMPILE="ccache riscv64-linux-" to recompile faster (compile, not the first time)
* All kernel modules, spread over the kernel source tree, as .ko (Kernel Object) files.
* vmlinux to debug the ELF form ??
* **Kernel installation**: native case (sudo make install) and embedded case (use script depend on kernel image, in embedded kernerl image is a single file -> easy to handle)
* **Module installation**:
  + native case (sudo make modules\_install) run as root

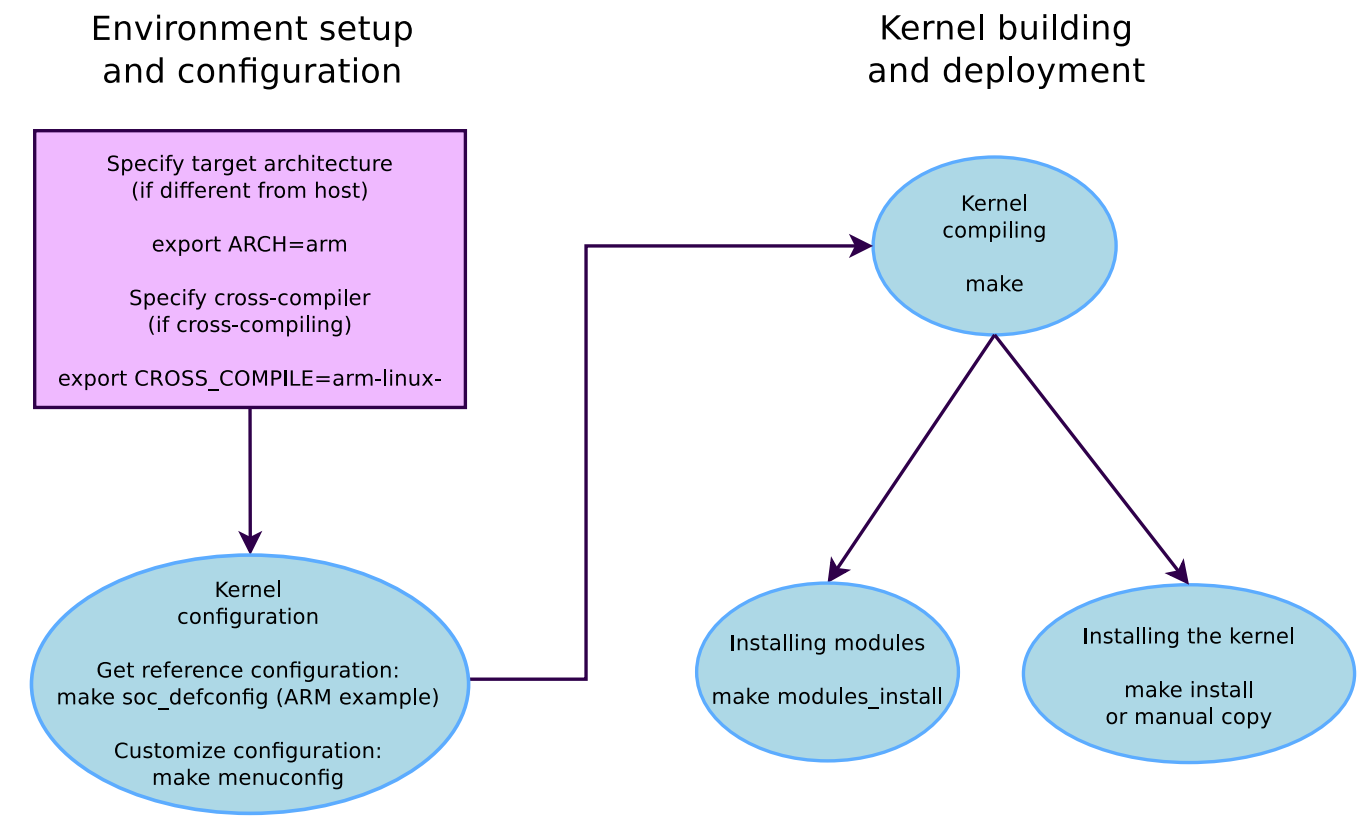
Automatic module loading with module aliases



Viết source code cho module -> make modules ra file .ko -> make modules\_install (depmod???) tạo 'modules alias' (rule cho MODALIAS)

When attaching an external device to -> send MODALIAS to udev process -> udev check rule -> load module

* Embedded case: due to the code for another device's module (ARM for example), we must point the output module to the INSTALL\_MOD\_PATH of that system (if using make module\_install, we will put the module into the host machine)
* **Kernel Cleaner target**: creates PHONY clean for make to clean files
* **Kernel building overview**



**Booting the kernel**

**Device tree**

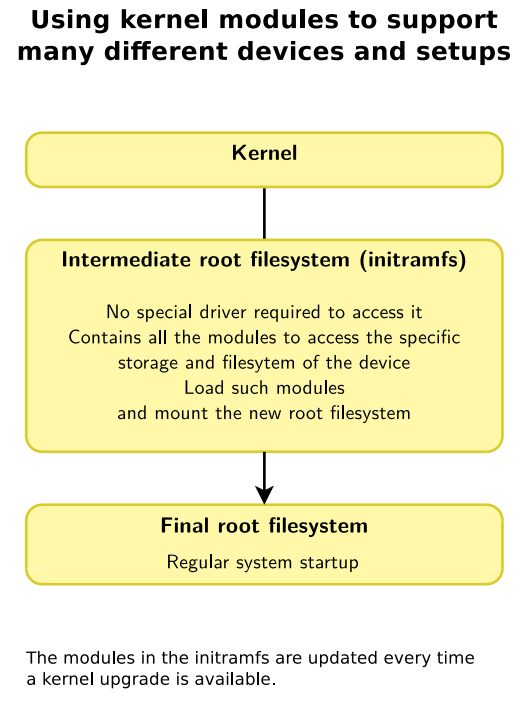
* There will be non-discoverable devices (serial, ethernet, I2C, Nand flash, USB controllers,..) clocks, interrupts, DMA channels, battery muxing,... needs to be declared directly to the kernel (difficult to reuse config file) -> using device tree (DT)
* DTS (device tree source) -> compiled thành DTB (Device tree blob)
* The DTB and kernel image must be loaded into memory before booting the -> kernel so that the kernel knows which SoC and device are running

**Submarine???**

The command line can be used to control kernel behavior without recompilation

**Using kernel module**

**Advantages**

* 
* Easy to handle drivers without having to reboot, keep kernel image smallest size + reduce boot time. The module has access to full control and privileges -> root to add and remove the module
* Modules can also depend on other modules

**Kernel log**

* The new module is loaded -> the relevant information is written to the kernel log (see with dmesg command – diagnostic message)

(often used to debug with printk)

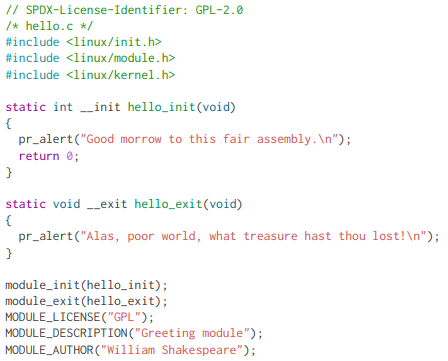
**Module utilities**

* View module information: modinfo <module\_path>.ko
* Load module : sudo insmod <module\_path>.ko!! Insmod does not give full information when load fails -> view in dmesg
* Load module dependencies: sudo modprobe <top\_module\_name>. Delete all modules + dependencies loaded without use: sudo modprobe –r <top\_module\_name>
* See the list of loaded modules: lsmod
* Delete module : sudo rmmod <module\_name>

**Passing parameters to modules + check module parameter values + edit it**

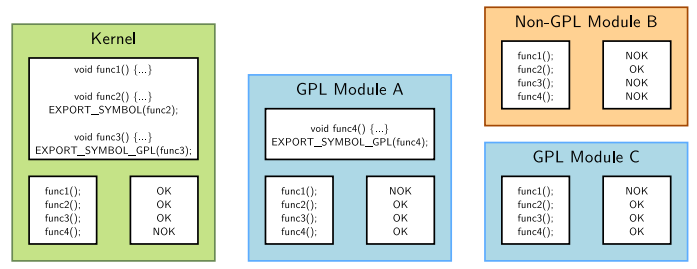
**Developing kernel modules**

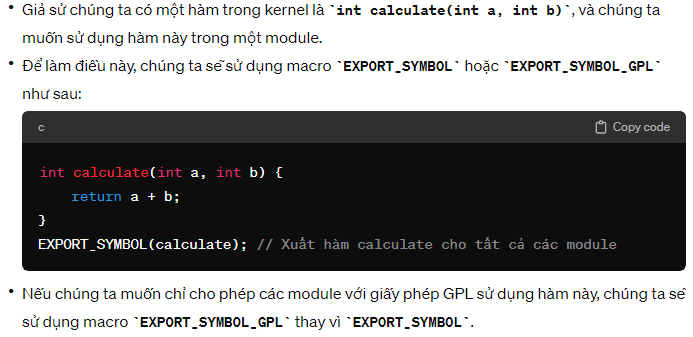
**Hello module**



* Macros \_\_init and \_\_exit are called only 1 time at module load and unload
* Load header in linux / xxx.h because we are coding kernel
* Metadata is declared via MODULE\_LICENSE(), MODULE\_DESCRIPTION() and MODULE\_AUTHOR()

**Symbols exported to module**



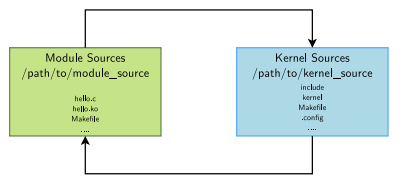


**Module license**

* Use to limit modules that do not have a GPL license
* Is the check system 100% free, is the kernel check tainted ???

**Compile a module**

* **Compiling an out-of-tree module**
  + Ko dc intergrated to kernel, build individually





Note: $$ is a reference to an environment variable

KERNELRELEASE???

* Inside a kernel tree: intergrated wiht kernel config

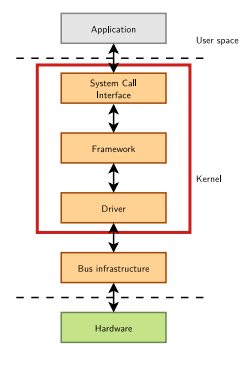
**Modules and kernel version**

* To compile, the module kernel needs kernel headers (containing def of func, types and constants). Full kernel sources or kernel headers only can be accessed, these 2 types must be written in .config
* need kernel makefile, script directory,...
* Error invalid module format due to compile different version of kernel

**New driver in kernel sources**

* Create a source.c file (usually 1 single file), description Kconfig(???)
* wirte makefile (obj-m += source.o)

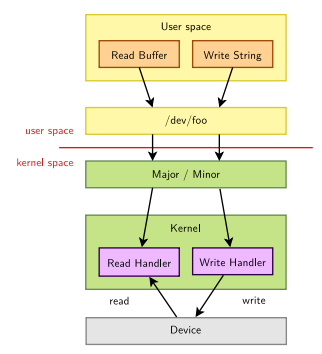
**Kernel and device drivers**



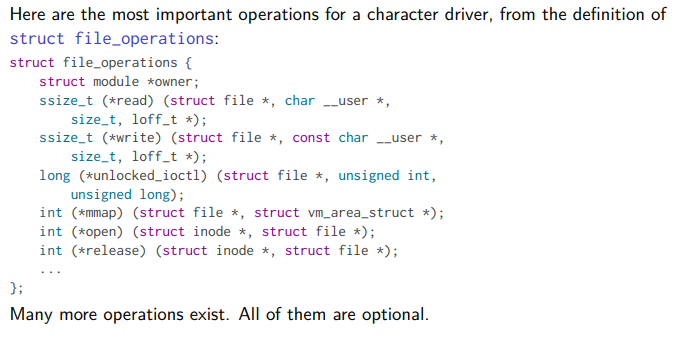
Under linux, there are 3 types:

* + Network devices
  + Block devices in /dev
  + Character devices in /dev

**Character devices driver**



In this part we have struct file\_operations:

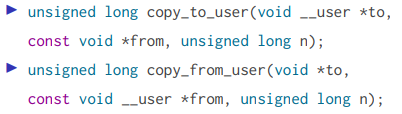


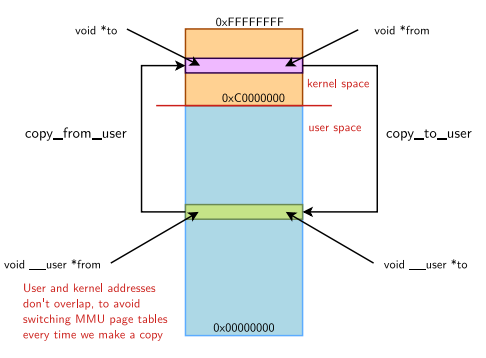
* + Open() and release() : (struct inode \*i, struct file \*f)
  + Read() and write() :(struct file \*f, char \_\_user \*buf, size\_t sz, loff\_t \*off)

**Exchanging data with user space**

Kernel code isn’t allowed to directly access user space mem (so memcpy()/direct pointer dereferencing cannot be used)

* Use a single value get\_user(value\_pointer, user\_space\_pointer) and put\_user(value\_pointer, user\_space\_pointer)
* Or buffer copy\_to\_user and copy\_from\_user:



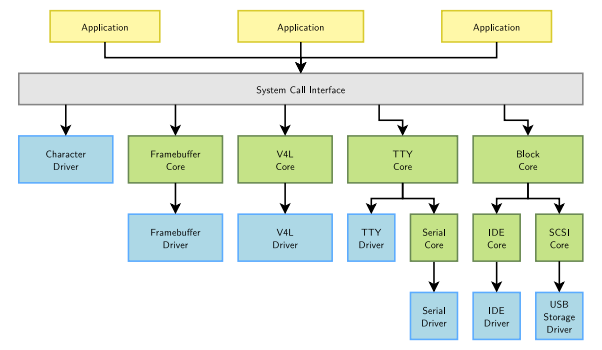


But what happen if buffer cannot store and tranfer efficiently (for ex large video)

* User zero copy access to user mem : mmap() (open a bridge directly from that driver to data), get\_user\_pages() (open a bridge from kernel space to user space )

Unlocked\_ioctl() -> release Big Kernel Lock to do things beyond read/write API

**Kernel frameworks**



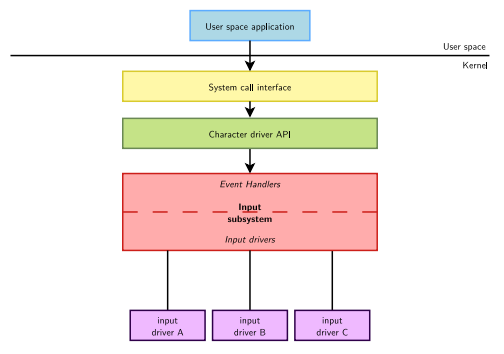
This is library for kernel to control system

* As we know, probe() and remove() are used to allocate memory for driver manually. Now we can use “**device managed allocations**” to automatically manage and allocate memory. Just use prefix “devm\_”. For ex:



* + We can link between structure and subclassing, reference or including part of framework structure

**Input subsystem diagram**

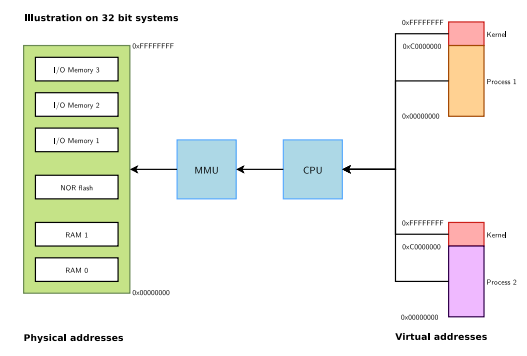


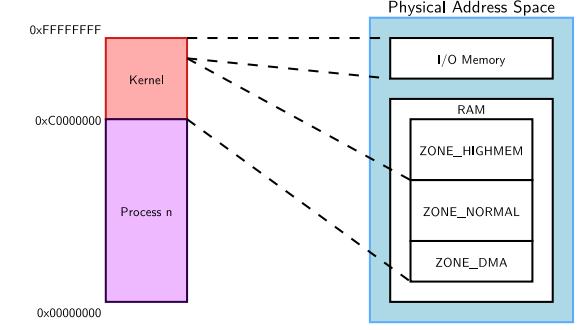
This is a small system that handle input events from human user, include: Devicer drivers and Event handlers

**Memory management – page allocator**

Physical memory : in working space such as RAM

Virtual memory: swap area or storage disk



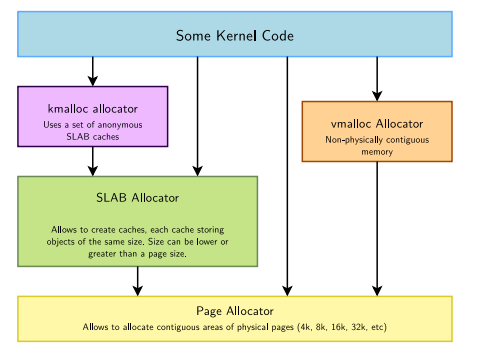


In 32bit, kernel usually has 1GB memory address and the rest 3GB is for user space to do process (program, stack,…). In some case, if we want more mem in kernel, we can take from 3GB or use highmem support (if available)

Some note: new user space mem is allocated from already allocated process mem or using the mmap call. If we call a mem that is not existed or [cannot allocate mem (running out of mem) -> OOM killer will kill some task to retrieve some memory]

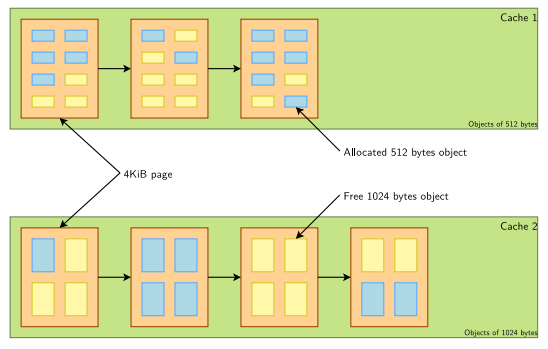
* Cause page fault when access to virtual address ( this means system copy data from swap/storage into RAM). Sometimes this lead to be swapped out -> page fault

**SLAB Allocators in kernel**



Min is 4KB and max 8122KB (depend on kernel config) and is contigous inn both virtual and physical space -> large area may not be available -> The contigous memory allocator is sollution by storing parts of system memory for this request

**SLAB allocator** find, allocate and store components that have same size -> optimize system performance



By investigating SLAB caches for small size and Page allocator for large size, we know how to manage memory efficiently in kernel. That’s how **Kmalloc** works in general, Maximum sizes, on x86 and arm

- Per allocation: 4 MB

- Total allocations: 128 MB

We should use kmalloc as a main allocator for kernel

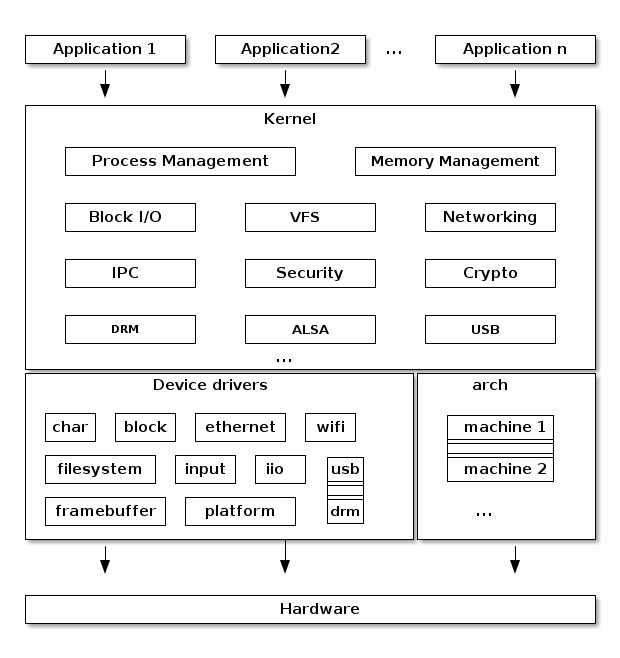
**Virtual malloc vmalloc()**

This func let us allocate large area (not contigous required) in virtual mem (can’t be use for direct memory access)

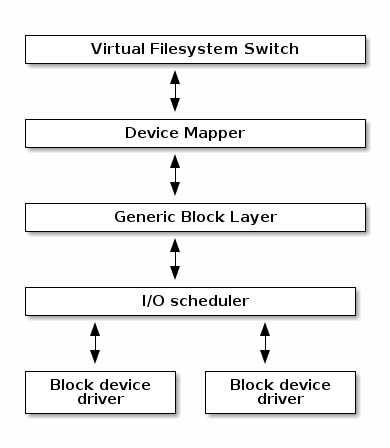
Kernel memory debugging

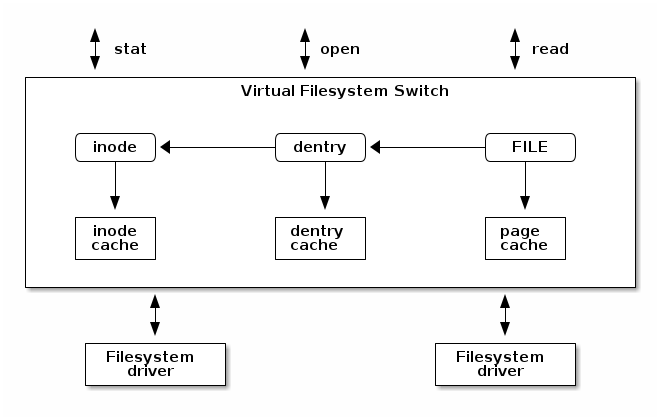
* + KASAN (Kernel Address Sanitizer): check use-after-free and out-of-bounds
  + KFENCE (Kernel Electric Fence): low overhead than KASAN, trading performance for precision used in production systems
  + Kmemleak: Dynamic checker for memory leaks

[**Linux kernel architecture**](https://linux-kernel-labs.github.io/refs/heads/master/lectures/intro.html#linux-kernel-architecture)**¶**

[](https://linux-kernel-labs.github.io/refs/heads/master/_images/ditaa-b9ffae65be16d30be11b5eca188a7a143b1b8227.png)

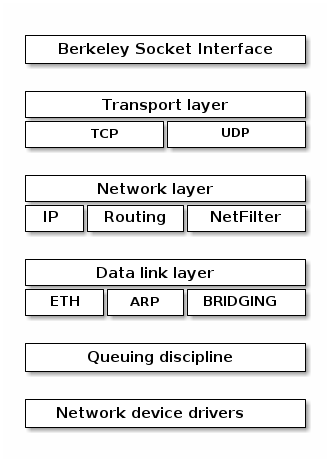
* + **Arch**: source code for specific harware (to help system access hardware for every single achitecture)
  + **Device drivers**: this help system can communicate with hardware outside
  + **Process management**: have fork(), exec(), wait(), POSIX threads in Unix standard. Each kernel has its own task\_struct -> task on same process can point to same resources. Clone() and unshare() system also help implementing new features such as namespaces (which is use with cgroup to implement OS virtualization)
  + **Memory management**: Linux memory management is a complex subsystem that deals with:
    - Management of the physical memory: allocating and freeing memory
    - Management of the virtual memory: paging, swapping, demand paging, copy on write
    - User services: user address space management (e.g. mmap(), brk(), shared memory)
    - Kernel services: SL\*B allocators, vmalloc
  + **Block I/O management**: The Linux Block I/O subsystem deals with reading and writing data from or to block devices: creating block I/O requests, transforming block I/O requests (e.g. for software RAID or LVM), merging and sorting the requests and scheduling them via various I/O schedulers to the block device drivers
  + **Virtual FS Switch**: The Linux Virtual Filesystem Switch implements common / generic filesystem code to reduce duplication in filesystem drivers. It introduces certain filesystem abstractions such as:
    - inode - describes the file on disk (attributes, location of data blocks on disk)
    - dentry - links an inode to a name
    - file - describes the properties of an opened file (e.g. file pointer)
    - superblock - describes the properties of a formatted filesystem (e.g. number of blocks, block size, location of root directory on disk, encryption, etc.)



[](https://linux-kernel-labs.github.io/refs/heads/master/_images/ditaa-afa57a07e21b1b842554278abe30fea575278452.png)

The Linux VFS also implements a complex caching mechanism which includes the following:

* the inode cache - caches the file attributes and internal file metadata
* the dentry cache - caches the directory hierarchy of a filesystem
* the page cache - caches file data blocks in memory
  + Networking stack



* + Linux security modules
    - Hooks to extend the default Linux security model
    - Used by several Linux security extensions:
      * Security Enhancened Linux
      * AppArmor
      * Tomoyo
      * Smack