Linux Kernel Compilation

make bzImage

- create vmlinux.bin in ksrc/arch/x86/boot
- further compressed into vmlinuz

binary file
(self extracting file)

make modules

- creates all .ko files into respective directories

sudo make modules_install

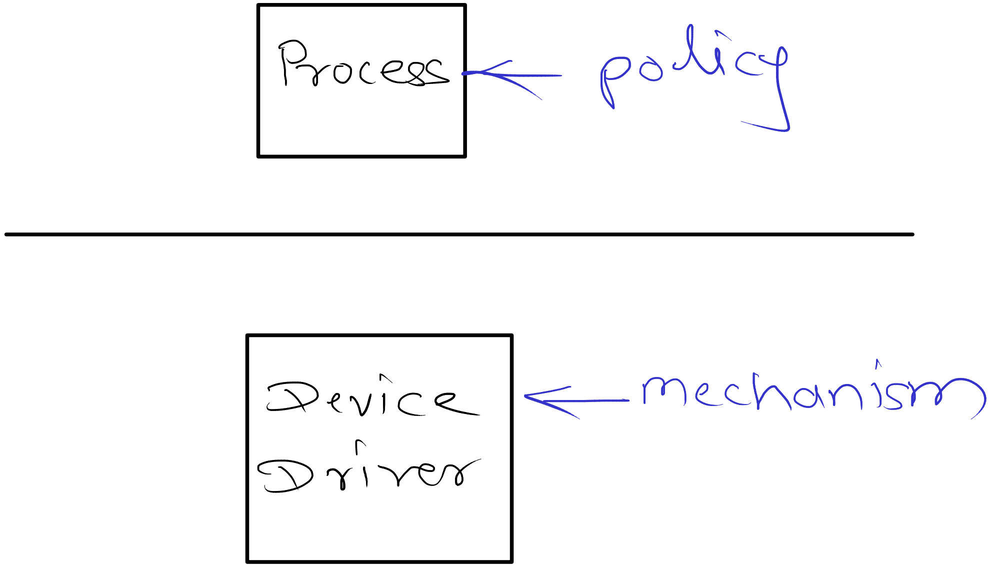
- create directory in /lib/modules with name of kernel version

mkdir /lib/modules/6.1.9-desd

- copy all kernel modules (*.ko) into this directory

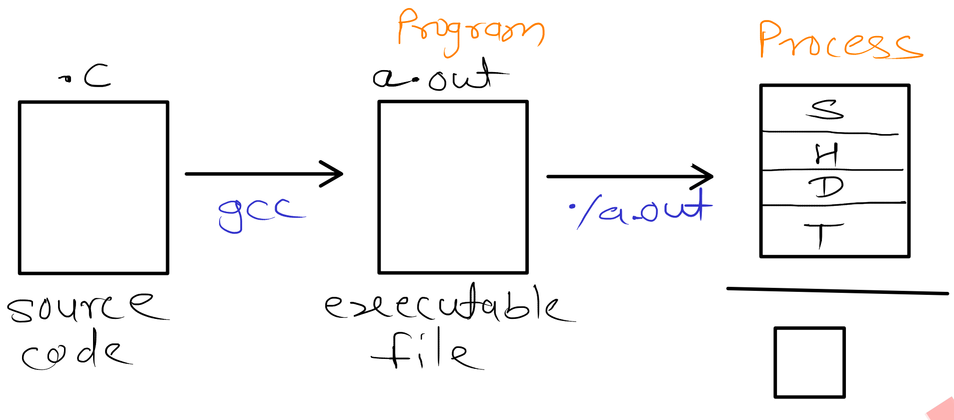
sudo make install

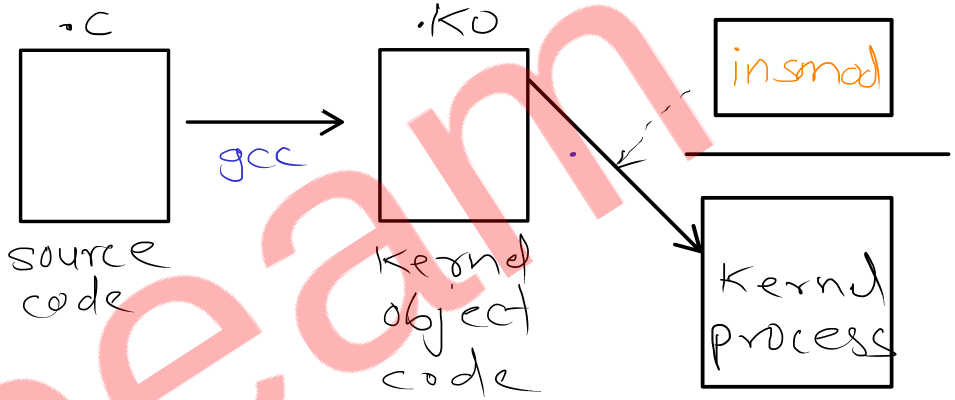
- copy vmlinuz into /boot directory
- update the grub to add entry for new kernel


Process
policy
Device Driver
mechanism

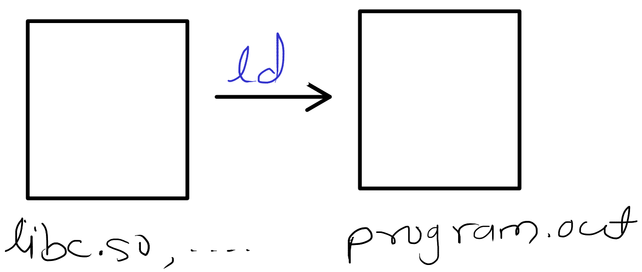
Linux Kernel module programming vs User space programming

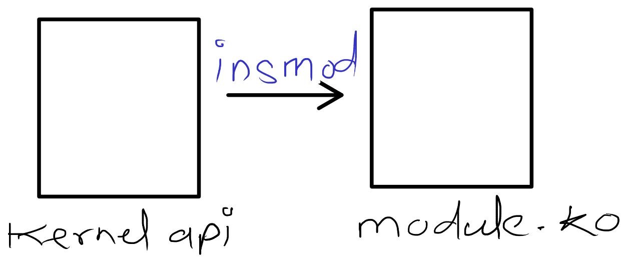
- User programs are self-executable and independent process is created for each program at runtime. However kernel modules are compiled into object files and at runtimeloaded into the kernel process.





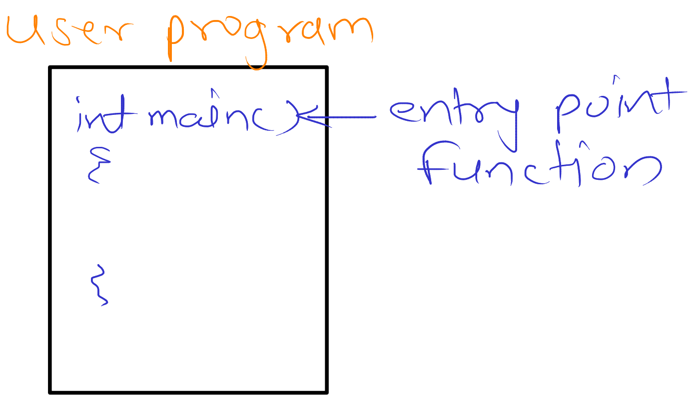
- Standard linker is not used to link kernel modules. Modules are linked dynamically with kernel using insmod like utilities. Since standard linker is not used to link kernel modules, user-space libraries (including C library) cannot be used in kernel modules. Instead kernel modules can access only functions exported by the kernel, called as Kernel APIs. Note that few commonly needed user-space functions are re-implemented in kernel space e.g. memset, strcpy, etc.



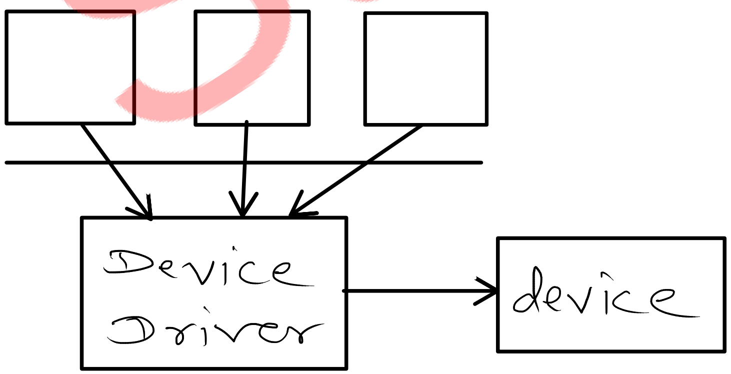


Linux Kernel module programming vs User space programming

- Typical user programs have single entry point i.e. main(). Program is terminated when main() is completed. Kernel modules have multiple entry-points. Also kernel modules are not terminated, when entry point function is finished.



- User space applications are not multi-threaded and hence rarely concurrency aware (synchronization usage). However kernel modules can be used by multiple Linux applications at the same time, so they must be concurrency aware programs.



Linux Kernel module programming vs User space programming

- If resources like memory, file or network connections are not released by user space applications, they are automatically released when process terminates. Any resource leakage in kernel module is never recovered (until reboot), because modules are loaded into system process itself.

Resource
1) Memory → kmalloc(), get_free_page()
2) synchronisation → semaphore, mutex, spin locks

- Programming mistakes in user space programs are ignored, produce exception or terminate process. However mistakes in kernel module may lead to kernel panic and system crash. Errors will be logged under /var/log/messages (as hexdump).

- kernel logs / oops → dmesg
↳ deamon messages

- User space applications may use FPU heavily. Resetting FPU for each operation doesn't hamper whole system performance. However kernel space code should run in real time, so using FPU in kernel module is not recommended.

Linux Kernel module programming vs User space programming

- Kernel module code is re-entrant i.e. while one thread is executing a kernel module function, another thread can also begin execution of the same. Here multiple copies of the variables will be created on kernel stack of threads.

Thread1

FAR is created on thread1 stack

function( )
{

}

Thread2

FAR is created on thread2 stack

ISR( )
{

}

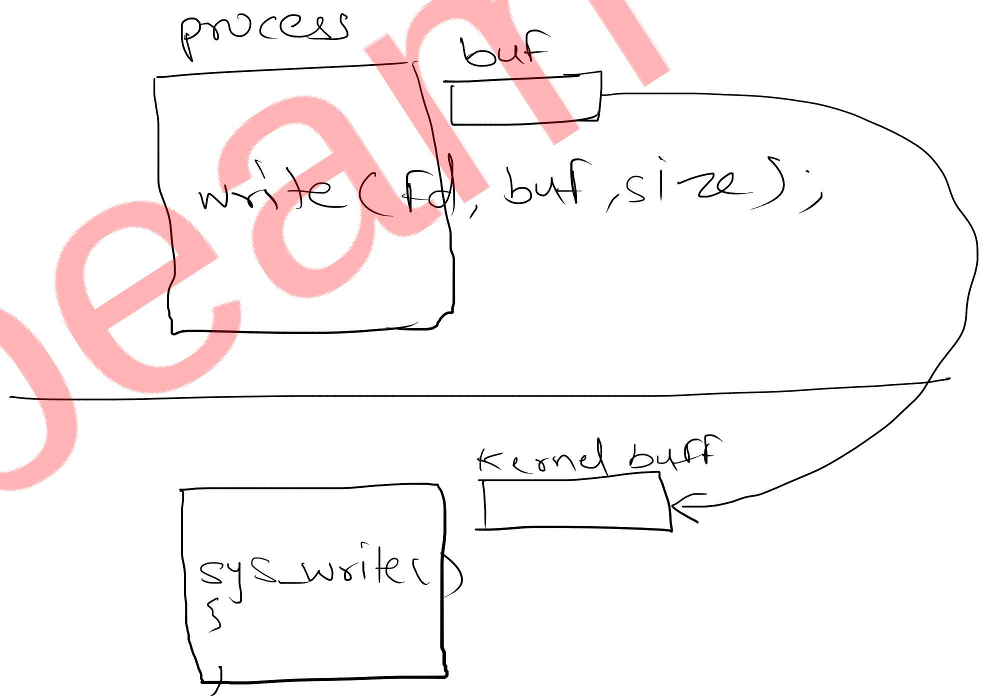
(8kb)

- Size of kernel stack much smaller than user-space process stack. It is recommended not to create huge local arrays, structure variables in kernel module functions.

Linux Kernel module programming vs User space programming

- If user space application need to pass data to/from kernel module, then user space buffer should not be accessed directly from kernel space and vice-versa. The data should be copied using architecture dependent code.

write(fd, buf, size);
read (fd, buf, size);



Kernel module implementation

- Kernel modules are binary files containing code & data (like user-space applications) which are dynamically linked to the kernel at runtime.
- Each kernel module have at least two entry point functions i.e. init and exit.
  - Traditionally their names as init_module() and cleanup_module().
  - Programmer may choose different names using module_init() & module_exit() macros
  - These functions are marked with __init and __exit attributes.
- Each module also have information associated with it using MODULE_XYZ() macros.
  - These macros will expands to MODULE_INFO() macros.
  - All this metadata is added into .modinfo section of .ko file, which can be inspected using modinfo command.
  - It also stores kernel version (of kernel against which module is built). This version is verified while loading it into the kernel. If version mismatch, module loading fails.
- Kernel modules can access functions exported by the kernel or other kernel module.

Kernel module compilation

* Create Makefile for compiling kernel module.

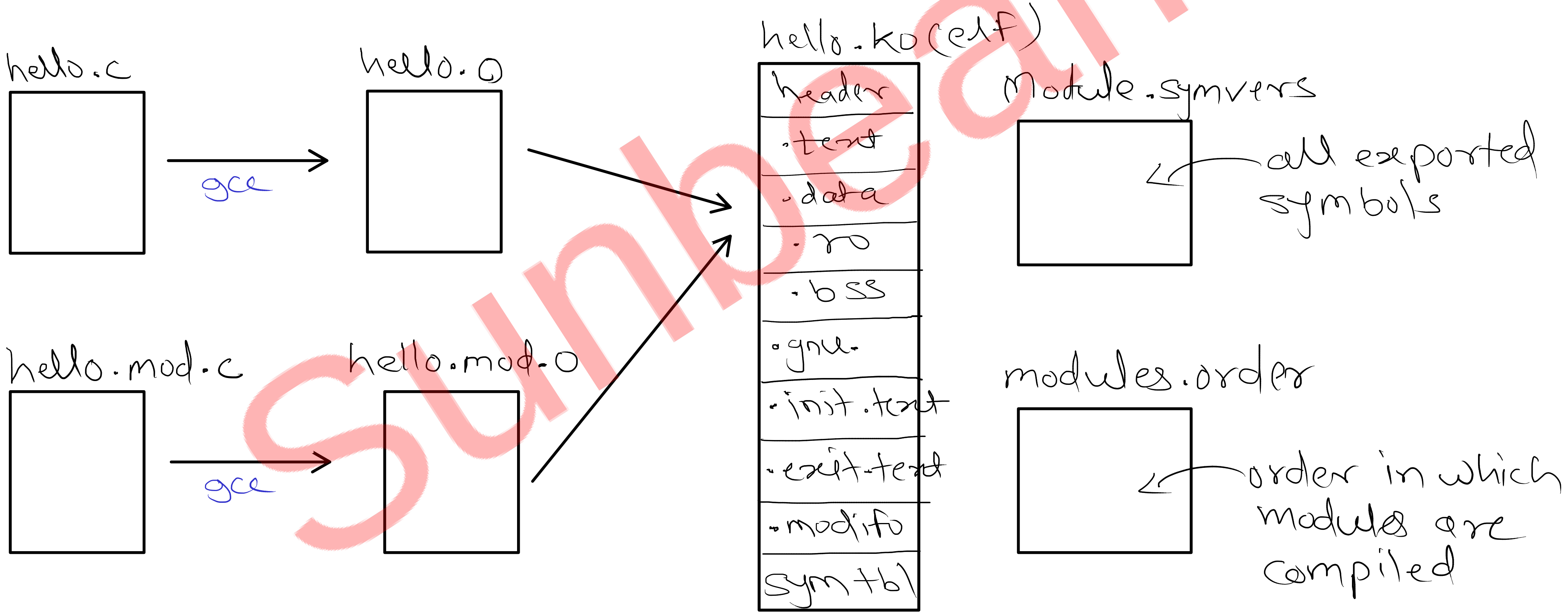
obj-m := hello.o

* Compile the kernel module using kernel Makefile.

make -C /lib/modules/\`uname -r\`/build M=\`pwd\` modules

* Generated files

hello.mod.c, hello.o, hello.mod.o, Module.symvers, modules.order



Kernel module structure

- Kernel module must be compiled against the kernel in which it is to be loaded. For this we should have access to kernel headers and kernel build system (Makefile, ...)

- Compiled kernel modules (.ko) are sectioned binary like ELF.

- terminal> objdump -f hello.ko

- terminal> objdump -h hello.ko
    - .text, .data, .rodata, .bss
    - .init.text
    - .exit.text
    - .gnu.linkonce.this_module
- terminal> objdump -t hello.ko
    - All unresolved symbols (e.g. printk()) are resolved at the time of loading that module (i.e. insmod) from kernel symbol table. This table can be viewed via /proc/kallsyms.

Kernel module internals

- Kernel module is represented by struct module in the Linux kernel.
    - Variable of struct module is created & initialized in .mod.c file, with name __this_module. This can be accessed in the module source code using macro THIS_MODULE.

    - After module is loaded kernel keep this variable in a kernel linked list. All kernel modules info can be accessed via /sys/module or /proc/modules or "lsmod" command.

    - struct module members
        - enum module_state state; — comming – insmod/init_module() / live / going – rmmod/cleanup_module()
        - struct list_head list; — next* | prev*
        - char name[MODULE_NAME_LEN]; — hello
        - int (*init)(void); — init_module()
        - void (*exit)(void); — cleanup_module()
        - void *module_init; — info needed to initialise module
        - void *module_core; — info needed to run the module
        - atomic_t refcnt; — number of other modules dependent on this module