3. Driver sequence

NXKR_YoungSikYang

Exported on 03/06/2024

Table of Contents

1. Kernel initialization 3
1.1 Kernel initialization level........ 3
1.2 Kernel initialization in the code........ 3
1.3 Diagram showing how modules are loaded in the kernel initialization........ 7
2. Module driver 8
2.1 Registration and loading of a module driver........ 8
2.2 Registering a module driver in 'postcore' 9
2.2.1 Check by time 9
2.2.2 Check by modifying do_initcalls() 9
Reference 11

1. Kernel initialization

The Linux kernel initialization is a highly structured process that ensures system components and drivers are started in a specific order, allowing dependencies to be resolved correctly.

The initialization sequence consists of several levels.

1.1 Kernel initialization level

Each phase is called by **initcall()**

1. **pure**: Architecture-specific, dealing with setting up essential hardware configurations.
2. **core**: Involves setting up essential kernel services and infrastructure, scheduling, interrupt handling, and basic memory management.
3. **postcore**: Initializes additional core subsystems that depend on the very basic services set up during the core phase.
4. **arch**: Since the Linux kernel supports multiple hardware architectures (such as x86, ARM, MIPS, etc.), this phase customizes the initialization process to the specific requirements of the current architecture.
5. **subsys**: The subsystems initializes various kernel subsystems that are not directly tied to the core kernel functionality. This can include driver frameworks, networking, filesystem support, and other subsystems that provide higher-level services.
6. **fs**: The filesystem initialization phase sets up the kernel's filesystem infrastructure, allowing the kernel to access file systems on disk, which is critical for the rest of the system's operation.
7. **rootfs**: The root filesystem is mounted during this phase.
8. **device**: This stage involves the initialization of device drivers and the device model, which allows the kernel to interact with the hardware components of the system.
9. **late**: The late initialization phase is the final step, handling any remaining initialization. This might include starting up user-space applications and services that are essential for the system's operation.

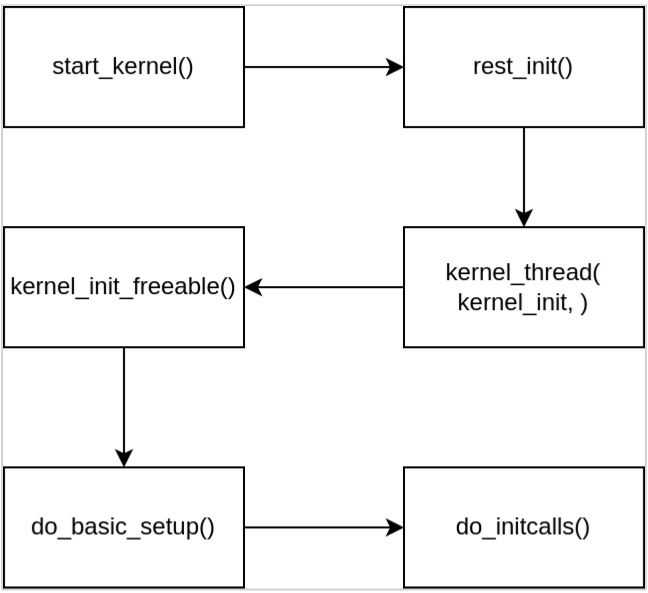
1.2 Kernel initialization in the code

`head.S` runs `init/main.c` : start_kernel()

kernel-4.4.x > arch > arm64 > kernel > head.S

```
#ifdef CONFIG_KASAN
    bl  kasan_early_init
#endif
#ifdef CONFIG_RANDOMIZE_BASE
    tst x23, ~(MIN_KIMG_ALIGN - 1)  // already running randomized?
    b.ne    0f
    mov x0, x21             // pass FDT address in x0
    mov x1, x23             // pass modulo offset in x1
    bl  kaslr_early_init        // parse FDT for KASLR options
    cbz x0, 0f              // KASLR disabled? just proceed
    orr x23, x23, x0            // record KASLR offset
    ret x28             // we must enable KASLR, return
                        // to __enable_mmu()
0:
#endif
    b   start_kernel
ENDPROC(__mmap_switched)
```

start_kernel() initializes various components and leads to do_initcalls(). do_initcalls() runs each init stage.

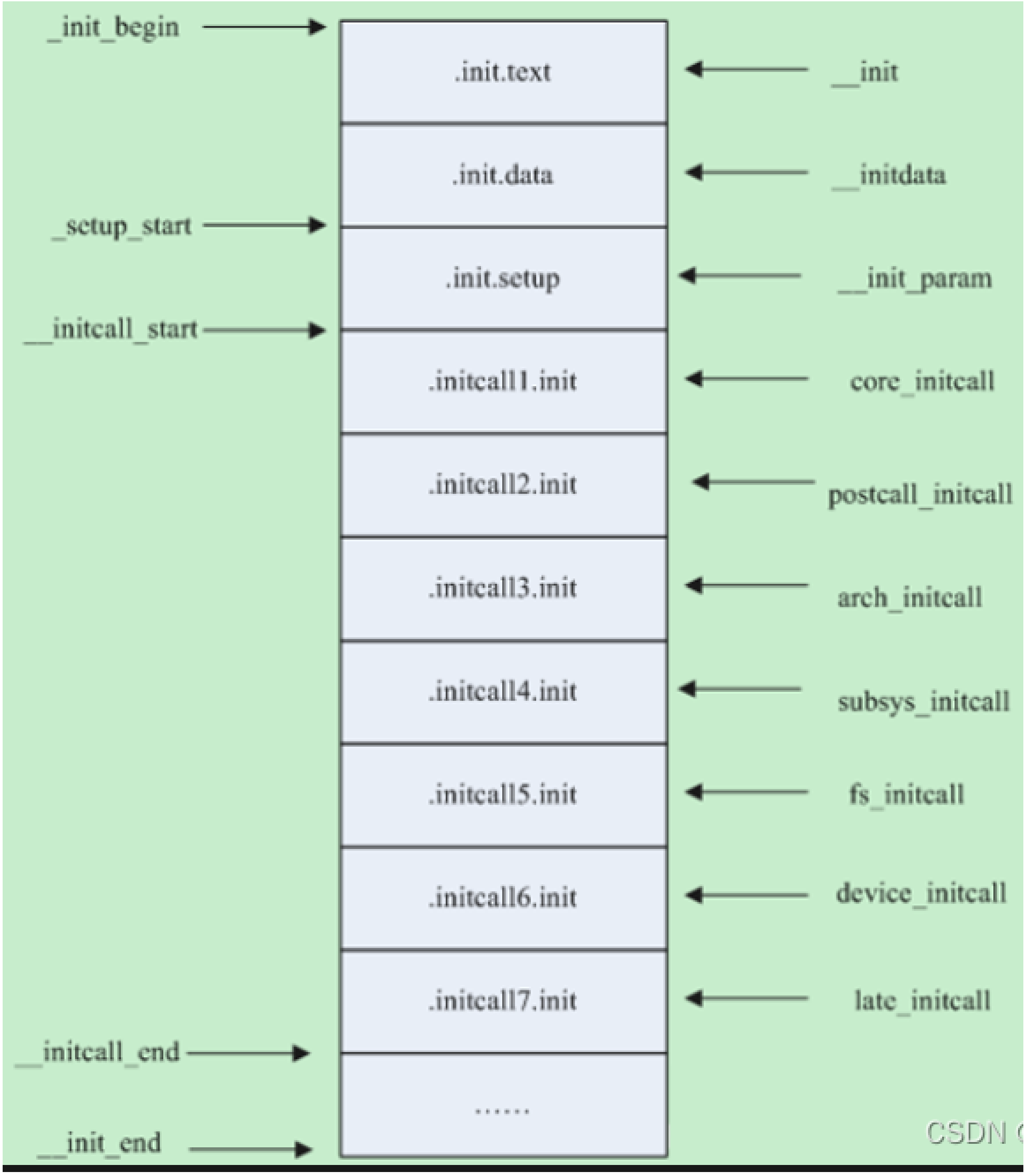


Macros defined in `include/linux/init.h` register function pointers in specific sections of the kernel binary so that they can be run in specific init stages.

```
#define pure_initcall(fn)          __define_initcall(fn, 0)

#define core_initcall(fn)          __define_initcall(fn, 1)
#define core_initcall_sync(fn)         __define_initcall(fn, 1s)
#define postcore_initcall(fn)          __define_initcall(fn, 2)
#define postcore_initcall_sync(fn)     __define_initcall(fn, 2s)
#define arch_initcall(fn)          __define_initcall(fn, 3)
#define arch_initcall_sync(fn)         __define_initcall(fn, 3s)
#define subsys_initcall(fn)        __define_initcall(fn, 4)
#define subsys_initcall_sync(fn)       __define_initcall(fn, 4s)
#define fs_initcall(fn)            __define_initcall(fn, 5)
#define fs_initcall_sync(fn)           __define_initcall(fn, 5s)
#define rootfs_initcall(fn)        __define_initcall(fn, rootfs)
#define device_initcall(fn)        __define_initcall(fn, 6)
#define device_initcall_sync(fn)       __define_initcall(fn, 6s)
#define late_initcall(fn)          __define_initcall(fn, 7)
#define late_initcall_sync(fn)         __define_initcall(fn, 7s)
```

The linker script(`vmlinux.lds`) organizes these special sections so that the function pointers are registered well in their correct section.



Registered functions can be seen using 'nm vmlinux | grep'

```
es/kernel/kernel-4.4.x$ nm vmlinux | grep mydevice_debug_init
ffffff8008add530 d __initcall_mydevice_debug_init2
ffffff8008aad9f4 t mydevice_debug_init
```

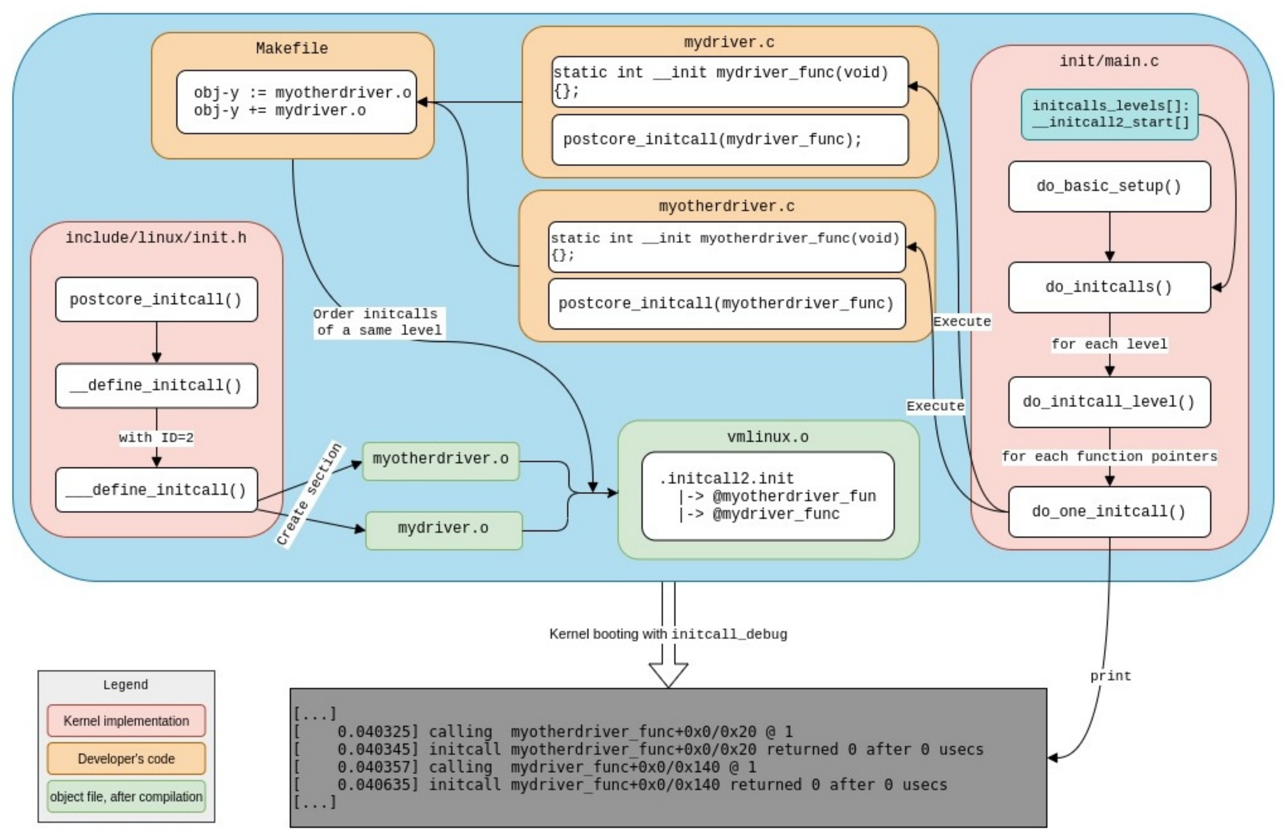
During the kernel's boot process, do_initcalls() iterates over these sections and executes the function pointers contained within them.

```
static void __init do_initcalls(void)
{
    int level;

#ifdef CONFIG_INITCALLS_THREAD
    for (level = 0; level < CONFIG_INITCALLS_THREAD_LEVEL; level++)
        do_initcall_level(level);

    init_thread_level = level;
    kthread_run(do_initcalls_kth,
        (void *)&init_thread_level, "initcalls:thread");
#else
    for (level = 0; level < ARRAY_SIZE(initcall_levels) - 1; level++)
        do_initcall_level(level);
#endif
}
```

1.3 Diagram showing how modules are loaded in the kernel initialization



2. Module driver

2.1 Registration and loading of a module driver

- module_init() registers a module driver

```
static int __init misc_init(void)
{
    int error;

    error = misc_register(&etx_misc_device);
    if (error) {
        pr_err("misc_register failed!!!\n");
        return error;
    }

    pr_info("misc_register init done!!!\n");
    return 0;
}
module_init(misc_init)
```

- module_init() is defined in `include/linux/module.h`

```
#define module_init(x)  __initcall(x);
```

- __initcall() is defined in `include/linux/init.h`

```
#define core_initcall(fn)           __define_initcall(fn, 1)
#define core_initcall_sync(fn)          __define_initcall(fn, 1s)
#define postcore_initcall(fn)           __define_initcall(fn, 2)
#define postcore_initcall_sync(fn)      __define_initcall(fn, 2s)
#define arch_initcall(fn)           __define_initcall(fn, 3)
#define arch_initcall_sync(fn)          __define_initcall(fn, 3s)
#define subsys_initcall(fn)         __define_initcall(fn, 4)
#define subsys_initcall_sync(fn)        __define_initcall(fn, 4s)
#define fs_initcall(fn)             __define_initcall(fn, 5)
#define fs_initcall_sync(fn)            __define_initcall(fn, 5s)
#define rootfs_initcall(fn)         __define_initcall(fn, rootfs)
#define device_initcall(fn)         __define_initcall(fn, 6)
#define device_initcall_sync(fn)        __define_initcall(fn, 6s)
#define late_initcall(fn)           __define_initcall(fn, 7)
#define late_initcall_sync(fn)          __define_initcall(fn, 7s)

#define __initcall(fn) device_initcall(fn)
```

This shows module_init() registers a module driver in the **device** phase.

If it is a built-in module, it is automatically loaded in the **device phase at boot time**.

2.2 Registering a module driver in 'postcore'

- Registering the driver in the **late** phase

```
late_initcall(mydevice_debug_init);
```

- Registering the driver in the **postcore** phase

```
postcore_initcall(mydevice_debug_init);
```

2.2.1 Check by time

The pictures below show how device driver messages are recorded at different times on different kernel initialization levels.

- Device driver registered in the '**late**' phase using 'late_initcall()'.

```
root@s5p6818:~# dmesg | grep yang
[    2.488000] ===yang's misc device[mydevice_debug_init][L:34]===
[    2.492000] ===yang's misc device[mydevice_probe][L:19]
```

- Device driver registered in the '**postcore**' phase using 'postcore_initcall()'.

```
root@s5p6818:~# dmesg | grep yang
[    0.184000] ===yang's misc device[mydevice_debug_init][L:34]===
[    0.208000] ===yang's misc device[mydevice_probe][L:19]
```

The driver init is recorded at an earlier time on **postcore** as shown above.

2.2.2 Check by modifying do_initcalls()

Add a print function to display the current initcall level

```c
static void __init do_initcalls(void)
{
    int level;

#ifdef CONFIG_INITCALLS_THREAD
    for (level = 0; level < CONFIG_INITCALLS_THREAD_LEVEL; level++)
        do_initcall_level(level);

    init_thread_level = level;
    kthread_run(do_initcalls_kth,
        (void *)&init_thread_level, "initcalls:thread");
#else
    for (level = 0; level < ARRAY_SIZE(initcall_levels) - 1; level++) {
        pr_info("# Initcall level: %s-----\n", initcall_level_names[level]);
        do_initcall_level(level);
    }
#endif
}
```

You can see the driver is loaded in the **postcore phase**

```
[    0.184000] # Initcall level: postcore-----
[    0.184000] ===yang's misc device[mydevice_debug_init][L:34]===
```

Reference

- https://kkslinuxinfo.wordpress.com/2015/12/11/kernel-initialization/
- https://blog.csdn.net/weixin_47491758/article/details/131157608
- https://www.collabora.com/news-and-blog/blog/2020/09/25/initcalls-part-2-digging-into-implementation/