Android Kernel (2) - Kernel Bootstrapping

AUGUST 13, 2014

android (/tags/android.html)

---

3. Bootstrapping

Staring Process

1. power up, execute bootloader program. **Bootloader** providers minimum required hardware environment.
2. Load kernel to memory, bootstrap kernel, at last execute `start_kernel` to start kernel. `start_kernel` will start **init** program in user space.
3. **init** program reads `init.rc` config file, starting a guard process. Two most important guard processes are **zygote** and **ServiceManager**. **zygote** is the first starting Dalvik VM in Android, used to start a Java environment. **ServiceManager** is required by **binder** communication.
4. zygote starts sub-routine `system_server`. `system_server` starts the Android core system services, then adds these services to **ServiceManager**. Then Android goes into systemReady state.
5. **ActivityManagerService** communicates with zygote socket, starts the **Home** app via zygote, which starting the system desktop.

Step 1 is hardware-depend process. It will not be discussed here.

We start from step 2.

Kernel Bootstrap

Android Kernel bootstrapping process is almost the same as Linux Kernel. Most of the code is in `kernel` folder.

Kernel startup process:

1. Boot loader execution: examine the compatibility of hardware. Source code is in `kernel/arch/arm/kernel/head.S` and `kernel/arch/arm/kernel/head-common.S`, programmed using assembly language.
2. Kernel bootstrap: after loading, call `start_kernel()` to boot the kernel. Source code in `kernel/init/main.c`.

1. Boot Loading

Here is the code from `head-common.S`:

```
1 /*
2  * The following fragment of code is executed with the MMU on in MMU mode,
3  * and uses absolute addresses; this is not position independent.
4  *
5  *  r0  = cp#15 control register
6  *  r1  = machine ID
7  *  r2  = atags/dtb pointer
8  *  r9  = processor ID
9  */
10  __INIT
11 __mmap_switched:
12  adr r3, __mmap_switched_data
13
14  ldmia   r3!, {r4, r5, r6, r7}
15  cmp r4, r5               @ Copy data segment if needed
16 1:  cmpne   r5, r6
17  ldrne   fp, [r4], #4
18  strne   fp, [r5], #4
19  bne 1b
20
21  mov fp, #0               @ Clear BSS (and zero fp)
22 1:  cmp r6, r7
23  strcc   fp, [r6],#4
24  bcc 1b
25
26  ARM( ldmia   r3, {r4, r5, r6, r7, sp})
27  THUMB(   ldmia   r3, {r4, r5, r6, r7}  )
28  THUMB(   ldr sp, [r3, #16]        )
29  str r9, [r4]           @ Save processor ID
30  str r1, [r5]           @ Save machine type
31  str r2, [r6]           @ Save atags pointer
32  cmp r7, #0
33  bicne   r4, r0, #CR_A         @ Clear 'A' bit
34  stmneia r7, {r0, r4}            @ Save control register values
35  b   start_kernel
36 ENDPROC(__mmap_switched)
```

In line 35, it called `start_kernel`. Where do we called `__mmap_switched`? It is inside `head.S`:

```
1 /*
2   * The following calls CPU specific code in a position independent
3   * manner.  See arch/arm/mm/proc-*.S for details.  r10 = base of
4   * xxx_proc_info structure selected by __lookup_processor_type
5   * above.  On return, the CPU will be ready for the MMU to be
6   * turned on, and r0 will hold the CPU control register value.
7   */
8  ldr r13, =__mmap_switched       @ address to jump to after
9                          @ mmu has been enabled
10  adr lr, BSYM(1f)           @ return (PIC) address
11  mov r8, r4                 @ set TTBR1 to swapper_pg_dir
12  ARM( add pc, r10, #PROCINFO_INITFUNC   )
13  THUMB(   add r12, r10, #PROCINFO_INITFUNC  )
14  THUMB(   mov pc, r12               )
15 1:  b    __enable_mmu
16 ENDPROC(stext)
```

Line 8: `__mmap_switched` is stored at address `r13`. When does program counter point to `r13`? After searching from source code, I found:

```
 1 /*
 2  * Enable the MMU.  This completely changes the structure of the visible
 3  * memory space.  You will not be able to trace execution through this.
 4  * If you have an enquiry about this, *please* check the linux-arm-kernel
 5  * mailing list archives BEFORE sending another post to the list.
 6  *
 7  *  r0  = cp#15 control register
 8  *  r1  = machine ID
 9  *  r2  = atags or dtb pointer
10  *  r9  = processor ID
11  *  r13 = *virtual* address to jump to upon completion
12  *
13  * other registers depend on the function called upon completion
14  */
15  .align  5
16  .pushsection    .idmap.text, "ax"
17 ENTRY(__turn_mmu_on)
18  mov r0, r0
19  instr_sync
20  mcr p15, 0, r0, c1, c0, 0        @ write control reg
21  mrc p15, 0, r3, c0, c0, 0        @ read id reg
22  instr_sync
23  mov r3, r3
24  mov r3, r13
25  mov pc, r3
26 __turn_mmu_on_end:
27 ENDPROC(__turn_mmu_on)
28  .popsection
```

Line 24, `mov r3, r13` and `mov pc, r3`, `@pc` points to `r13`, which means `r13` instruction is the one going to be executed. `__turn_mmu_on` is called in a method `__enable_mmu`:

```
/*
 * Setup common bits before finally enabling the MMU.  Essentially
 * this is just loading the page table pointer and domain access
 * registers.
 *
 *  r0  = cp#15 control register
 *  r1  = machine ID
 *  r2  = atags or dtb pointer
 *  r4  = page table pointer
 *  r9  = processor ID
 *  r13 = *virtual* address to jump to upon completion
 */
__enable_mmu:
#if defined(CONFIG_ALIGNMENT_TRAP) && __LINUX_ARM_ARCH__ < 6
    orr r0, r0, #CR_A
#else
    bic r0, r0, #CR_A
#endif
#ifdef CONFIG_CPU_DCACHE_DISABLE
    bic r0, r0, #CR_C
#endif
#ifdef CONFIG_CPU_BPREDICT_DISABLE
    bic r0, r0, #CR_Z
#endif
#ifdef CONFIG_CPU_ICACHE_DISABLE
    bic r0, r0, #CR_I
#endif
#ifdef CONFIG_ARM_LPAE
    mov r5, #0
    mcrr    p15, 0, r4, r5, c2      @ load TTBR0
#else
    mov r5, #(domain_val(DOMAIN_USER, DOMAIN_MANAGER) | \
              domain_val(DOMAIN_KERNEL, DOMAIN_MANAGER) | \
              domain_val(DOMAIN_TABLE, DOMAIN_MANAGER) | \
              domain_val(DOMAIN_IO, DOMAIN_CLIENT))
    mcr p15, 0, r5, c3, c0, 0       @ load domain access register
    mcr p15, 0, r4, c2, c0, 0       @ load page table pointer
#endif
    b   __turn_mmu_on
ENDPROC(__enable_mmu)
```

The program using instruction `b` to jump to `__turn_mmu_on`.

Summary of bootloading

When system starts MMU, `__enable_mmu` calls `__turn_mmu_on`. And it use `mov r3, r13` and `mov pc, r3` to call `__mmap_switched`. `__mmap_switched` calls `start_kernel`.

- `__enable_mmu`
  - -> `__turn_mmu_on`
  - -> `__mmap_switched`
    - -> `start_kernel`

`start_kernel` is the common Linux method to start kernel. This is the first C method gets called after the assembly code.

2. Kernel Start

`start_kernel` method is inside `kernel/init/main.c`:

```
asmlinkage void __init start_kernel(void)
{
    char * command_line;
    extern const struct kernel_param __start___param[], __stop___param[];

// ...

    /*
     * Need to run as early as possible, to initialise the
     * lockdep hash:
     */

    /*
     * Set up the the initial canary ASAP:
     */

    /*
     * Interrupts are still disabled. Do necessary setups, then
     * enable them
     */

    /*
     * These use large bootmem allocations and must precede
     * kmem_cache_init()
     */

    /*
     * Set up the scheduler prior starting any interrupts (such as the
     * timer interrupt). Full topology setup happens at smp_init()
     * time - but meanwhile we still have a functioning scheduler.
     */

    /*
     * Disable preemption - early bootup scheduling is extremely
     * fragile until we cpu_idle() for the first time.
     */

    /*
     * HACK ALERT! This is early. We're enabling the console before
     * we've done PCI setups etc, and console_init() must be aware of
     * this. But we do want output early, in case something goes wrong.
     */

    /*
```

```
     * Need to run this when irqs are enabled, because it wants
     * to self-test [hard/soft]-irqs on/off lock inversion bugs
     * too:
     */

    /* Do the rest non-__init'ed, we're now alive */
    rest_init();
}
```

I leave the comments in `start_kernel` there to illustrate the process of this method. On the last line, there is a `rest_init` method, which starts the **init** process. **init** process has pid **1** in Android User space, responsible for starting the Android runtime environment.

```
static noinline void __init_refok rest_init(void)
{
    int pid;

    rcu_scheduler_starting();
    /*
     * We need to spawn init first so that it obtains pid 1, however
     * the init task will end up wanting to create kthreads, which, if
     * we schedule it before we create kthreadd, will OOPS.
     */
    kernel_thread(kernel_init, NULL, CLONE_FS | CLONE_SIGHAND);
    numa_default_policy();
    pid = kernel_thread(kthreadd, NULL, CLONE_FS | CLONE_FILES);
    rcu_read_lock();
    kthreadd_task = find_task_by_pid_ns(pid, &init_pid_ns);
    rcu_read_unlock();
    complete(&kthreadd_done);

    /*
     * The boot idle thread must execute schedule()
     * at least once to get things moving:
     */
    init_idle_bootup_task(current);
    schedule_preempt_disabled();
    /* Call into cpu_idle with preempt disabled */
    cpu_startup_entry(CPUHP_ONLINE);
}
```

`rest_init` uses `kernel_thread` to start two kernel thread.

```
pid_t kernel_thread(int (*fn)(void *), void *arg, unsigned long flags)
```

The first `kernel_thread` calls a `kernel_init` method.

```c
static int __ref kernel_init(void *unused)
{
    kernel_init_freeable();
    /* need to finish all async __init code before freeing the memory */
    async_synchronize_full();
    free_initmem();
    mark_rodata_ro();
    system_state = SYSTEM_RUNNING;
    numa_default_policy();

    flush_delayed_fput();

    if (ramdisk_execute_command) {
        if (!run_init_process(ramdisk_execute_command))
            return 0;
        pr_err("Failed to execute %s\n", ramdisk_execute_command);
    }

    /*
     * We try each of these until one succeeds.
     *
     * The Bourne shell can be used instead of init if we are
     * trying to recover a really broken machine.
     */
    if (execute_command) {
        if (!run_init_process(execute_command))
            return 0;
        pr_err("Failed to execute %s.  Attempting defaults...\n",
            execute_command);
    }
    if (!run_init_process("/sbin/init") ||
        !run_init_process("/etc/init") ||
        !run_init_process("/bin/init") ||
        !run_init_process("/bin/sh"))
        return 0;

    panic("No init found.  Try passing init= option to kernel. "
          "See Linux Documentation/init.txt for guidance.");
}
```

Important part is the bottom half. `execute_command` is an argument from bootloader to kernel. Normally its' value is `/init`. Then after this command, it runs `run_init_process`, which is a wrapper to `do_execve`.

`do_execve` is the Linux interface to create user process.

```
static int run_init_process(const char *init_filename)
{
    argv_init[0] = init_filename;
    return do_execve(init_filename,
        (const char __user *const __user *)argv_init,
        (const char __user *const __user *)envp_init);
}
```

3. `init` process execution

Code of **init** process, which is the `execute_command` in previous section, is inside `/system/core/init`. Here is the `Android.mk` file of `/system/core/init` module:

```
LOCAL_SRC_FILES:= \
    builtins.c \
    init.c \
    devices.c \
    property_service.c \
    util.c \
    parser.c \
    logo.c \
    keychords.c \
    signal_handler.c \
    init_parser.c \
    ueventd.c \
    ueventd_parser.c \
    watchdogd.c

LOCAL_MODULE:= init
```

The `main()` method of this module is inside `init.c`. By reading the source code (attached at the end of this article), I found that `init` process contains four stages:

1. initialise file system and log system

2. parse `init.rc` and `init.<hardware>.rc` file
3. start Action and Service
4. inittialize event listener loop.

3.1 initialise file system and log system

Inside `main()` of `init.c`:

```
/* Get the basic filesystem setup we need put
     * together in the initramdisk on / and then we'll
     * let the rc file figure out the rest.
     */
    mkdir("/dev", 0755);
    mkdir("/proc", 0755);
    mkdir("/sys", 0755);

    mount("tmpfs", "/dev", "tmpfs", MS_NOSUID, "mode=0755");
    mkdir("/dev/pts", 0755);
    mkdir("/dev/socket", 0755);
    mount("devpts", "/dev/pts", "devpts", 0, NULL);
    mount("proc", "/proc", "proc", 0, NULL);
```

This part is standard Linux method calling to prepare for file system and log system

3.2 Parse `init.rc`

`init.rc` is defined using **Android Init Language**.

Android Init Language

`on` and `service` are keywords. `on` is used to declare Action, and `service` is used to declare Service.

Documentation of **AIL** is in `/system/core/init/readme.txt`, keywords in `keyword.h`.

1. Action

```
on <trigger>
    <command>
    <command>
    ...
```

On `<trigger>` condition satisfied, run `<command>`s.

2. Command

Linux shell commands.

3. Service

```
service <name> <pathname> [ <argument ]*
    <option>
    <option>
    ...
```

4. Option

5. Section

6. Trigger

Inside `main()`, I can see:

```
/* These directories were necessarily created before initial policy load
     * and therefore need their security context restored to the proper value.
     * This must happen before /dev is populated by ueventd.
     */
    restorecon("/dev");
    restorecon("/dev/socket");
    restorecon("/dev/__properties__");
    restorecon_recursive("/sys");

    is_charger = !strcmp(bootmode, "charger");

    INFO("property init\n");
    if (!is_charger)
        property_load_boot_defaults();

    INFO("reading config file\n");
    init_parse_config_file("/init.rc");
```

Thus, to parse the `init.rc`, I read the `init_parse_config_file()` in `/system/core/init/init_parser.c`:

```
int init_parse_config_file(const char *fn)
{
    char *data;
    data = read_file(fn, 0);
    if (!data) return -1;

    parse_config(fn, data);
    DUMP();
    return 0;
}

// /system/core/init/init.h
// * reads a file, making sure it is terminated with \n \0 */
// void *read_file(const char *fn, unsigned *_sz)
```

`read_file` reads file buffer to `data`. `init_parse_config_file` reads, parses and debugs the config file. Important method here is the `parse_config(fn, data)`. This method is defined in `init_parser.c`:

```
static void parse_config(const char *fn, char *s)
{
    struct parse_state state;
    struct listnode import_list;
    struct listnode *node;
    char *args[INIT_PARSER_MAXARGS];
    int nargs;

    nargs = 0;
    state.filename = fn;
    state.line = 0;
    state.ptr = s;
    state.nexttoken = 0;
    state.parse_line = parse_line_no_op;

    list_init(&import_list);
    state.priv = &import_list;

    for (;;) {
        switch (next_token(&state)) {
        case T_EOF:
            state.parse_line(&state, 0, 0);
            goto parser_done;
        case T_NEWLINE:
            state.line++;
            if (nargs) {
                int kw = lookup_keyword(args[0]);
                if (kw_is(kw, SECTION)) {
                    state.parse_line(&state, 0, 0);
                    parse_new_section(&state, kw, nargs, args);
                } else {
                    state.parse_line(&state, nargs, args);
                }
                nargs = 0;
            }
            break;
        case T_TEXT:
            if (nargs < INIT_PARSER_MAXARGS) {
                args[nargs++] = state.text;
            }
            break;
        }
    }
```

```
parser_done:
    list_for_each(node, &import_list) {
         struct import *import = node_to_item(node, struct import, list);
         int ret;

         INFO("importing '%s'", import->filename);
         ret = init_parse_config_file(import->filename);
         if (ret)
             ERROR("could not import file '%s' from '%s'\n",
                   import->filename, fn);
    }
```

The parsing is using line-by-line strategy. `parse_state` stores the parsing state, defined in `parser.h`. When encountering a keyword, it calls `lookup_keyword` to match keywords, and then call other method to parse, such as `parse_new_section`.

```
struct parse_state
{
    char *ptr;
    char *text;
    int line;
    int nexttoken;
    void *context;
    void (*parse_line)(struct parse_state *state, int nargs, char **arg
s);
    const char *filename;
    void *priv;
};
```

```c
void parse_new_section(struct parse_state *state, int kw,
                       int nargs, char **args)
{
    printf("[ %s %s ]\n", args[0],
           nargs > 1 ? args[1] : "");
    switch(kw) {
    case K_service:
        state->context = parse_service(state, nargs, args);
        if (state->context) {
            state->parse_line = parse_line_service;
            return;
        }
        break;
    case K_on:
        state->context = parse_action(state, nargs, args);
        if (state->context) {
            state->parse_line = parse_line_action;
            return;
        }
        break;
    case K_import:
        parse_import(state, nargs, args);
        break;
    }
    state->parse_line = parse_line_no_op;
}
```

To parse `Service` and `Action`, it calls `parse_service`, when encounter keyword `service`, and `parse_action` when encounter keyword `on` respectively.

`parse_service`

```c
    case K_service:
        state->context = parse_service(state, nargs, args);
        if (state->context) {
            state->parse_line = parse_line_service;
            return;
        }
        break;
```

```
static void *parse_service(struct parse_state *state, int nargs, char **args)
{
    struct service *svc;
    if (nargs < 3) {
        parse_error(state, "services must have a name and a program\n");
        return 0;
    }
    if (!valid_name(args[1])) {
        parse_error(state, "invalid service name '%s'\n", args[1]);
        return 0;
    }

    svc = service_find_by_name(args[1]);
    if (svc) {
        parse_error(state, "ignored duplicate definition of service '%s'\n", args[1]);
        return 0;
    }

    nargs -= 2;
    svc = calloc(1, sizeof(*svc) + sizeof(char*) * nargs);
    if (!svc) {
        parse_error(state, "out of memory\n");
        return 0;
    }
    svc->name = args[1];
    svc->classname = "default";
    memcpy(svc->args, args + 2, sizeof(char*) * nargs);
    svc->args[nargs] = 0;
    svc->nargs = nargs;
    svc->onrestart.name = "onrestart";
    list_init(&svc->onrestart.commands);
    list_add_tail(&service_list, &svc->slist);
    return svc;
}
```

It mainly does 3 jobs: **1)** allocation space for new Service, **2)** initialise Service, **3)** put Service into a `service_list`.

Here are some code necessary known:

```
// /system/core/include/cutiks.list.h
#define list_declare(name) \
    struct listnode name = { \
        .next = &name, \
        .prev = &name, \
    }


// /system/core/libcutils/list.c
void list_init(struct listnode *node)
{
    node->next = node;
    node->prev = node;
}

void list_add_tail(struct listnode *head, struct listnode *item)
{
    item->next = head;
    item->prev = head->prev;
    head->prev->next = item;
    head->prev = item;
}
```

It is the standard double linked list.

Most important, is the `struct service` in line 3, defined in `/system/core/init/init.h`:

```c
struct service {
        /* list of all services */
    struct listnode slist;

    const char *name;
    const char *classname;

    unsigned flags;
    pid_t pid;
    time_t time_started;    /* time of last start */
    time_t time_crashed;    /* first crash within inspection window */
    int nr_crashed;         /* number of times crashed within window */

    uid_t uid;
    gid_t gid;
    gid_t supp_gids[NR_SVC_SUPP_GIDS];
    size_t nr_supp_gids;

    char *seclabel;

    struct socketinfo *sockets;
    struct svcenvinfo *envvars;

    struct action onrestart;  /* Actions to execute on restart. */

    /* keycodes for triggering this service via /dev/keychord */
    int *keycodes;
    int nkeycodes;
    int keychord_id;

    int ioprio_class;
    int ioprio_pri;

    int nargs;
    /* "MUST BE AT THE END OF THE STRUCT" */
    char *args[1];
}; /*     ^-------'args' MUST be at the end of this struct! */
```

`parse_service` just initialise Service basic information. A lot of details are filled by `parse_line_service`.

In `parse_new_section`, we change `state->parse_line = parse_line_service;`. Previous value in `parse_config()` was `parse_line_no_op`.

`parse_line_service`

```
static void parse_line_service(struct parse_state *state, int nargs, char
**args)
{
    struct service *svc = state->context;
    struct command *cmd;
    int i, kw, kw_nargs;

    if (nargs == 0) {
        return;
    }

    svc->ioprio_class = IoSchedClass_NONE;

    kw = lookup_keyword(args[0]);
    switch (kw) {
    // ...
    case K_onrestart:
        nargs--;
        args++;
        kw = lookup_keyword(args[0]);
        if (!kw_is(kw, COMMAND)) {
            parse_error(state, "invalid command '%s'\n", args[0]);
            break;
        }
        kw_nargs = kw_nargs(kw);
        if (nargs < kw_nargs) {
            parse_error(state, "%s requires %d %s\n", args[0], kw_nargs -
1,
                kw_nargs > 2 ? "arguments" : "argument");
            break;
        }

        cmd = malloc(sizeof(*cmd) + sizeof(char*) * nargs);
        cmd->func = kw_func(kw);
        cmd->nargs = nargs;
        memcpy(cmd->args, args, sizeof(char*) * nargs);
        list_add_tail(&svc->onrestart.commands, &cmd->clist);
        break;
    case K_user:
        if (nargs != 2) {
            parse_error(state, "user option requires a user id\n");
        } else {
            svc->uid = decode_uid(args[1]);
        }
```

```
        break;
    default:
        parse_error(state, "invalid option '%s'\n", args[0]);
    }
}
```

This is the end of Service parsing

`parse_action`

Remember in `parse_new_section()`:

```
    case K_on:
        state->context = parse_action(state, nargs, args);
        if (state->context) {
            state->parse_line = parse_line_action;
            return;
        }
        break;
```

We call `parse_action()`.

```c
static void *parse_action(struct parse_state *state, int nargs, char **ar
gs)
{
    struct action *act;
    if (nargs < 2) {
        parse_error(state, "actions must have a trigger\n");
        return 0;
    }
    if (nargs > 2) {
        parse_error(state, "actions may not have extra parameters\n");
        return 0;
    }
    act = calloc(1, sizeof(*act));
    act->name = args[1];
    list_init(&act->commands);
    list_init(&act->qlist);
    list_add_tail(&action_list, &act->alist);
        /* XXX add to hash */
    return act;
}
```

Similar to Service, `parse_action` does **1)** allocation space for Action, **2)** put Action into a `action_list`.

The definition of `action` is inside `/system/core/init/init.h`:

```c
struct action {
        /* node in list of all actions */
    struct listnode alist;
        /* node in the queue of pending actions */
    struct listnode qlist;
        /* node in list of actions for a trigger */
    struct listnode tlist;

    unsigned hash;
    const char *name;

    struct listnode commands;
    struct command *current;
};
```

Core of parsing action is inside `parse_line_action`:

```
static void parse_line_action(struct parse_state* state, int nargs, char
**args)
{
    struct command *cmd;
    struct action *act = state->context;
    int (*func)(int nargs, char **args);
    int kw, n;

    if (nargs == 0) {
        return;
    }

    kw = lookup_keyword(args[0]);
    if (!kw_is(kw, COMMAND)) {
        parse_error(state, "invalid command '%s'\n", args[0]);
        return;
    }

    n = kw_nargs(kw);
    if (nargs < n) {
        parse_error(state, "%s requires %d %s\n", args[0], n - 1,
            n > 2 ? "arguments" : "argument");
        return;
    }
    cmd = malloc(sizeof(*cmd) + sizeof(char*) * nargs);
    cmd->func = kw_func(kw);
    cmd->nargs = nargs;
    memcpy(cmd->args, args, sizeof(char*) * nargs);
    list_add_tail(&act->commands, &cmd->clist);
}
```

The definition of `struct command`:

```
struct command
{
    /* list of commands in an action */
    struct listnode clist;

    int (*func)(int nargs, char **args);
    int nargs;
    char *args[1];
};
```

3.3 Start Action and Service

Start Action

Go back to `main()`.

```
INFO("reading config file\n");
    init_parse_config_file("/init.rc");

    action_for_each_trigger("early-init", action_add_queue_tail);

    queue_builtin_action(wait_for_coldboot_done_action, "wait_for_coldboot_done");
    queue_builtin_action(mix_hwrng_into_linux_rng_action, "mix_hwrng_into_linux_rng");
    queue_builtin_action(keychord_init_action, "keychord_init");
    queue_builtin_action(console_init_action, "console_init");

    /* execute all the boot actions to get us started */
    action_for_each_trigger("init", action_add_queue_tail);

    /* skip mounting filesystems in charger mode */
    if (!is_charger) {
        action_for_each_trigger("early-fs", action_add_queue_tail);
        action_for_each_trigger("fs", action_add_queue_tail);
        action_for_each_trigger("post-fs", action_add_queue_tail);
        action_for_each_trigger("post-fs-data", action_add_queue_tail);
    }

    /* Repeat mix_hwrng_into_linux_rng in case /dev/hw_random or /dev/random
     * wasn't ready immediately after wait_for_coldboot_done
     */
    queue_builtin_action(mix_hwrng_into_linux_rng_action, "mix_hwrng_into_linux_rng");

    queue_builtin_action(property_service_init_action, "property_service_init");
    queue_builtin_action(signal_init_action, "signal_init");
    queue_builtin_action(check_startup_action, "check_startup");

    if (is_charger) {
        action_for_each_trigger("charger", action_add_queue_tail);
    } else {
        action_for_each_trigger("early-boot", action_add_queue_tail);
        action_for_each_trigger("boot", action_add_queue_tail);
    }

        /* run all property triggers based on current state of the pr
```

```
operties */
    queue_builtin_action(queue_property_triggers_action, "queue_property_triggers");

#if BOOTCHART
    queue_builtin_action(bootchart_init_action, "bootchart_init");
#endif

    for(;;) {
        int nr, i, timeout = -1;

        execute_one_command();
        restart_processes();

        if (!property_set_fd_init && get_property_set_fd() > 0) {

        // ...
```

After parsing `init.rc`, Android runs some `action_for_each_trigger()` and `queue_builtin_action()`.

```c
void action_for_each_trigger(const char *trigger,
                             void (*func)(struct action *act))
{
    struct listnode *node;
    struct action *act;
    list_for_each(node, &action_list) {
        act = node_to_item(node, struct action, alist);
        if (!strcmp(act->name, trigger)) {
            func(act);
        }
    }
}

void queue_builtin_action(int (*func)(int nargs, char **args), char *nam
e)
{
    struct action *act;
    struct command *cmd;

    act = calloc(1, sizeof(*act));
    act->name = name;
    list_init(&act->commands);
    list_init(&act->qlist);

    cmd = calloc(1, sizeof(*cmd));
    cmd->func = func;
    cmd->args[0] = name;
    list_add_tail(&act->commands, &cmd->clist);

    list_add_tail(&action_list, &act->alist);
    action_add_queue_tail(act);
}
```

For `list_for_each` and `node_to_item`:

```c
#define list_for_each(node, list) \
    for (node = (list)->next; node != (list); node = node->next)

#define node_to_item(node, container, member) \
    (container *) (((char*) (node)) - offsetof(container, member))
```

`offsetof` is a IMPORTANT C macro, used to calculate the offset of a member in struct. It is defined in Android kernel project `/include/linux/stddef.h`:

```
#undef offsetof
#ifdef __compiler_offsetof
#define offsetof(TYPE,MEMBER) __compiler_offsetof(TYPE,MEMBER)
#else
#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
#endif
```

`action_add_queue_tail()` adds action to the end of the queue:

```
void action_add_queue_tail(struct action *act)
{
    if (list_empty(&act->qlist)) {
        list_add_tail(&action_queue, &act->qlist);
    }
}
```

Next step is `execute_one_command` in `init.c`:

```
void execute_one_command(void)
{
    int ret;

    if (!cur_action || !cur_command || is_last_command(cur_action, cur_co
mmand)) {
        cur_action = action_remove_queue_head();
        cur_command = NULL;
        if (!cur_action)
            return;
        INFO("processing action %p (%s)\n", cur_action, cur_action->nam
e);
        cur_command = get_first_command(cur_action);
    } else {
        cur_command = get_next_command(cur_action, cur_command);
    }

    if (!cur_command)
        return;

    ret = cur_command->func(cur_command->nargs, cur_command->args);
    INFO("command '%s' r=%d\n", cur_command->args[0], ret);
}
```

It gets command and executes in `func`. `func` was assigned in `parse_line_action`: `cmd->func = kw_func(kw)`:

What is `kw_func()`?

```c
#include "keywords.h"

#define KEYWORD(symbol, flags, nargs, func) \
    [ K_##symbol ] = { #symbol, func, nargs + 1, flags, },

struct {
    const char *name;
    int (*func)(int nargs, char **args);
    unsigned char nargs;
    unsigned char flags;
} keyword_info[KEYWORD_COUNT] = {
    [ K_UNKNOWN ] = { "unknown", 0, 0, 0 },
#include "keywords.h"
};
#undef KEYWORD

#define kw_func(kw) (keyword_info[kw].func)
```

> **Refer to my another article about C Macro Pre-processing (/c-essence-4-pre-processing/)**

`KEYWORD` is a function-like macro. And `keyword_info[]` is also defined in `keywords.h`. It defines all the execution method for all Command

```c
    // ...
    KEYWORD(capability,  OPTION,  0, 0)
    KEYWORD(chdir,       COMMAND, 1, do_chdir)
    KEYWORD(chroot,      COMMAND, 1, do_chroot)
    KEYWORD(class,       OPTION,  0, 0)

    KEYWORD(write,       COMMAND, 2, do_write)
    KEYWORD(start,       COMMAND, 1, do_start)
    KEYWORD(class_start, COMMAND, 1, do_class_start)


    // ...
```

To illustrate how Action and Service are executed, I use `early-init Action` as example. Back to `init.rc`:

```
on early-init
    # Set init and its forked children's oom_adj.
    write /proc/1/oom_adj -16

    # Set the security context for the init process.
    # This should occur before anything else (e.g. ueventd) is started.
    setcon u:r:init:s0

    start ueventd
```

`write` is mapped to `do_write` method and `start` is mapped to `do_start`, referring to `KEYWORD` mapping. These methods are defined in `builtins.c`.

```
int do_start(int nargs, char **args)
{
    struct service *svc;
    svc = service_find_by_name(args[1]);
    if (svc) {
        service_start(svc, NULL);
    }
    return 0;
}

int do_write(int nargs, char **args)
{
    const char *path = args[1];
    const char *value = args[2];
    char prop_val[PROP_VALUE_MAX];
    int ret;

    ret = expand_props(prop_val, value, sizeof(prop_val));
    if (ret) {
        ERROR("cannot expand '%s' while writing to '%s'\n", value, path);
        return -EINVAL;
    }
    return write_file(path, prop_val);
}
```

In `do_start`, I see a `service_start`, defined in `init.c`:

```c
void service_start(struct service *svc, const char *dynamic_args)
{
    struct stat s;
    pid_t pid;
    int needs_console;
    int n;
    char *scon = NULL;
    int rc;

        /* starting a service removes it from the disabled or reset
         * state and immediately takes it out of the restarting
         * state if it was in there
         */
    svc->flags &= (~(SVC_DISABLED|SVC_RESTARTING|SVC_RESET|SVC_RESTART));
    svc->time_started = 0;

    // ...

        NOTICE("starting '%s'\n", svc->name);

    pid = fork();

    if (pid == 0) {
        struct socketinfo *si;
        struct svcenvinfo *ei;
        char tmp[32];
        int fd, sz;

        umask(077);
        if (properties_inited()) {
            get_property_workspace(&fd, &sz);
            sprintf(tmp, "%d,%d", dup(fd), sz);
            add_environment("ANDROID_PROPERTY_WORKSPACE", tmp);
        }
        for (si = svc->sockets; si; si = si->next) {
            int socket_type = (
                    !strcmp(si->type, "stream") ? SOCK_STREAM :
                        (!strcmp(si->type, "dgram") ? SOCK_DGRAM : SOCK_S
EQPACKET));
            int s = create_socket(si->name, socket_type,
                                  si->perm, si->uid, si->gid);
            if (s >= 0) {
                publish_socket(si->name, s);
            }
```

```
        }

        if (!dynamic_args) {
            if (execve(svc->args[0], (char**) svc->args, (char**) ENV) <
0) {
                ERROR("cannot execve('%s'): %s\n", svc->args[0], strerro
r(errno));
            }
        } else {
            char *arg_ptrs[INIT_PARSER_MAXARGS+1];
            int arg_idx = svc->nargs;
            char *tmp = strdup(dynamic_args);
            char *next = tmp;
            char *bword;

            /* Copy the static arguments */
            memcpy(arg_ptrs, svc->args, (svc->nargs * sizeof(char *)));

            while((bword = strsep(&next, " "))) {
                arg_ptrs[arg_idx++] = bword;
                if (arg_idx == INIT_PARSER_MAXARGS)
                    break;
            }
            arg_ptrs[arg_idx] = '\0';
            execve(svc->args[0], (char**) arg_ptrs, (char**) ENV);
        }
        _exit(127);
    }

    svc->time_started = gettime();
    svc->pid = pid;
    svc->flags |= SVC_RUNNING;

    if (properties_inited())
        notify_service_state(svc->name, "running");
}
```

When starting a Service, it forks a sub-routine from current process, adds Property information to env variables, create a **Socket**, and run Linux system method `execve` to execute Service. Here it is `ueventd`.

Start Service

In the Who calls `service_start()` (in `do_start`) will start a Service.

There are the methods calling `service_start()` directly:

- `do_start()`
- `do_restart()`
- `restart_service_if_needed()`
- `msg_start()`
- `service_start_if_not_disabled()`

So I found `do_class_start()` can call `service_start()` indirectly:

```
int do_class_start(int nargs, char **args)
{
        /* Starting a class does not start services
         * which are explicitly disabled.  They must
         * be started individually.
         */
    service_for_each_class(args[1], service_start_if_not_disabled);
    return 0;
}

static void service_start_if_not_disabled(struct service *svc)
{
    if (!(svc->flags & SVC_DISABLED)) {
        service_start(svc, NULL);
    }
}
```

`do_class_start` runs when `class_start` Command gets executed. In `init.rc`

```
on boot
    ...
    class_start core
    class_start main
```

In `main()`, `action_for_each_trigger("boot", action_add_queue_tail)` connects Service with Command. Service is a process, started by Command. Thus all the Service are sub-routine of `init` process. Some of the services are `ueventd`, `servicemanager`, `vold`, `zygote`, `installd`, `ril-daemon`, `debuggerd`, `bootanim` and so on.

Property Service

`init` also handles some built-in Action, done by `queue_builtin_action`. This include some jobs related to **Property Service**. To store global system information, Android provides a shared memory, to store some key-value pairs.

Go back to the `main` method in `init.c`, before `init_parse_config_file("/init.rc")`:

```
/* We must have some place other than / to create the
     * device nodes for kmsg and null, otherwise we won't
     * be able to remount / read-only later on.
     * Now that tmpfs is mounted on /dev, we can actually
     * talk to the outside world.
     */
    open_devnull_stdio();
    klog_init();
    property_init();

    // ...

    is_charger = !strcmp(bootmode, "charger");

    INFO("property init\n");
    if (!is_charger)
        property_load_boot_defaults();

    // ...

    /* Repeat mix_hwrng_into_linux_rng in case /dev/hw_random or /dev/random
     * wasn't ready immediately after wait_for_coldboot_done
     */
    queue_builtin_action(mix_hwrng_into_linux_rng_action, "mix_hwrng_into_linux_rng");

    queue_builtin_action(property_service_init_action, "property_service_init");
    queue_builtin_action(signal_init_action, "signal_init");
    queue_builtin_action(check_startup_action, "check_startup");

    if (is_charger) {
        action_for_each_trigger("charger", action_add_queue_tail);
    } else {
        action_for_each_trigger("early-boot", action_add_queue_tail);
        action_for_each_trigger("boot", action_add_queue_tail);
    }

    /* run all property triggers based on current state of the properties */
    queue_builtin_action(queue_property_triggers_action, "queue_property_triggers");
```

1. `property_init` method calls `init_property_area()` to init share memory, open `ashmen` device and apply for the memory
2. `property_load_boot_defaults()` loads `/default.prop` file, which contains default properties
3. `queue_builtin_action()` triggers `property_service_init`
4. `queue_builtin_action()` triggers `queue_property_triggers`

`property_service_init_action()`

```
static int property_service_init_action(int nargs, char **args)
{
    /* read any property files on system or data and
     * fire up the property service.  This must happen
     * after the ro.foo properties are set above so
     * that /data/local.prop cannot interfere with them.
     */
    start_property_service();
    return 0;
}

// /system/core/init/property_service.c
void start_property_service(void)
{
    int fd;

    load_properties_from_file(PROP_PATH_SYSTEM_BUILD);
    load_properties_from_file(PROP_PATH_SYSTEM_DEFAULT);
    load_override_properties();
    /* Read persistent properties after all default values have been load
ed. */
    load_persistent_properties();

    fd = create_socket(PROP_SERVICE_NAME, SOCK_STREAM, 0666, 0, 0);
    if(fd < 0) return;
    fcntl(fd, F_SETFD, FD_CLOEXEC);
    fcntl(fd, F_SETFL, O_NONBLOCK);

    listen(fd, 8);
    property_set_fd = fd;
}
```

It **1)** loads property file, **2)** create Socket connection with client, **3)** listent to Socket

`queue_property_triggers_action()`

```c
static int queue_property_triggers_action(int nargs, char **args)
{
    queue_all_property_triggers();
    /* enable property triggers */
    property_triggers_enabled = 1;
    return 0;
}

// in init_parser.c
void queue_all_property_triggers()
{
    struct listnode *node;
    struct action *act;
    list_for_each(node, &action_list) {
        act = node_to_item(node, struct action, alist);
        if (!strncmp(act->name, "property:", strlen("property:"))) {
            /* parse property name and value
               syntax is property:<name>=<value> */
            const char* name = act->name + strlen("property:");
            const char* equals = strchr(name, '=');
            if (equals) {
                char prop_name[PROP_NAME_MAX + 1];
                char value[PROP_VALUE_MAX];
                int length = equals - name;
                if (length > PROP_NAME_MAX) {
                    ERROR("property name too long in trigger %s", act->na
me);
                } else {
                    memcpy(prop_name, name, length);
                    prop_name[length] = 0;

                    /* does the property exist, and match the trigger val
ue? */
                    property_get(prop_name, value);
                    if (!strcmp(equals + 1, value) ||!strcmp(equals + 1,
"*")) {
                        action_add_queue_tail(act);
                    }
                }
            }
        }
    }
}
```

`queue_property_triggers_action` trigers all Action starts with `property:`. **Property Service** is a special Action in Android. They start with `on property:` :

```
on property:ro.debuggable=1
    start console

# adbd on at boot in emulator
on property:ro.kernel.qemu=1
    start adbd

...
```

Why we need set up **Socket** connection with client? When setting system properties, property server calls `property_set()`. There is a `property_set()` in client, defined in `/system/core/libcutils/properties.c`

```
int property_set(const char *key, const char *value)
{
    return __system_property_set(key, value);
}
```

`__system_property_set` is defined in `/bionic/libc/bionic/system_properties.c`:

```
int __system_property_set(const char *key, const char *value)
{
    int err;
    prop_msg msg;

    if(key == 0) return -1;
    if(value == 0) value = "";
    if(strlen(key) >= PROP_NAME_MAX) return -1;
    if(strlen(value) >= PROP_VALUE_MAX) return -1;

    memset(&msg, 0, sizeof msg);
    msg.cmd = PROP_MSG_SETPROP;
    strlcpy(msg.name, key, sizeof msg.name);
    strlcpy(msg.value, value, sizeof msg.value);

    err = send_prop_msg(&msg);
    if(err < 0) {
        return err;
    }

    return 0;
}

static int send_prop_msg(prop_msg *msg)
{
    struct pollfd pollfds[1];
    struct sockaddr_un addr;
    socklen_t alen;
    size_t namelen;
    int s;
    int r;
    int result = -1;

    s = socket(AF_LOCAL, SOCK_STREAM, 0);
    if(s < 0) {
        return result;
    }

    memset(&addr, 0, sizeof(addr));
    namelen = strlen(property_service_socket);
    strlcpy(addr.sun_path, property_service_socket, sizeof addr.sun_pat
h);
    addr.sun_family = AF_LOCAL;
    alen = namelen + offsetof(struct sockaddr_un, sun_path) + 1;
```

```
    if(TEMP_FAILURE_RETRY(connect(s, (struct sockaddr *) &addr, alen)) <
0) {
        close(s);
        return result;
    }

    r = TEMP_FAILURE_RETRY(send(s, msg, sizeof(prop_msg), 0));

    if(r == sizeof(prop_msg)) {
        // We successfully wrote to the property server but now we
        // wait for the property server to finish its work.  It
        // acknowledges its completion by closing the socket so we
        // poll here (on nothing), waiting for the socket to close.
        // If you 'adb shell setprop foo bar' you'll see the POLLHUP
        // once the socket closes.  Out of paranoia we cap our poll
        // at 250 ms.
        pollfds[0].fd = s;
        pollfds[0].events = 0;
        r = TEMP_FAILURE_RETRY(poll(pollfds, 1, 250 /* ms */));
        if (r == 1 && (pollfds[0].revents & POLLHUP) != 0) {
            result = 0;
        } else {
            // Ignore the timeout and treat it like a success anyway.
            // The init process is single-threaded and its property
            // service is sometimes slow to respond (perhaps it's off
            // starting a child process or something) and thus this
            // times out and the caller thinks it failed, even though
            // it's still getting around to it.  So we fake it here,
            // mostly for ctl.* properties, but we do try and wait 250
            // ms so callers who do read-after-write can reliably see
            // what they've written.  Most of the time.
            // TODO: fix the system properties design.
            result = 0;
        }
    }

    close(s);
    return result;
}
```

We can see that, Android Property system communicate via Socket, using `property_set()` and `property_get()`

3.4 Initialise event listening loop

After init triggers all Actions, it executes `execute_one_command()`, which starts Action and Service, and `restart_processes()`, which restarts Action and Service. At the end of `main()` of `init.c`:

```
for(;;) {
    int nr, i, timeout = -1;

    execute_one_command();
    restart_processes();

    if (!property_set_fd_init && get_property_set_fd() > 0) {
      ufds[fd_count].fd = get_property_set_fd();
      ufds[fd_count].events = POLLIN;
      ufds[fd_count].revents = 0;
      fd_count++;
      property_set_fd_init = 1;
    }
    if (!signal_fd_init && get_signal_fd() > 0) {
      ufds[fd_count].fd = get_signal_fd();
      ufds[fd_count].events = POLLIN;
      ufds[fd_count].revents = 0;
      fd_count++;
      signal_fd_init = 1;
    }
    if (!keychord_fd_init && get_keychord_fd() > 0) {
      ufds[fd_count].fd = get_keychord_fd();
      ufds[fd_count].events = POLLIN;
      ufds[fd_count].revents = 0;
      fd_count++;
      keychord_fd_init = 1;
    }

    if (process_needs_restart) {
      timeout = (process_needs_restart - gettime()) * 1000;
      if (timeout < 0)
        timeout = 0;
    }

    if (!action_queue_empty() || cur_action)
      timeout = 0;

  #if BOOTCHART
    if (bootchart_count > 0) {
      if (timeout < 0 || timeout > BOOTCHART_POLLING_MS)
        timeout = BOOTCHART_POLLING_MS;
      if (bootchart_step() < 0 || --bootchart_count == 0) {
        bootchart_finish();
        bootchart_count = 0;
```

```
            }
        }
    #endif

        nr = poll(ufds, fd_count, timeout);
        if (nr <= 0)
            continue;

        for (i = 0; i < fd_count; i++) {
            if (ufds[i].revents == POLLIN) {
                if (ufds[i].fd == get_property_set_fd())
                    handle_property_set_fd();
                else if (ufds[i].fd == get_keychord_fd())
                    handle_keychord();
                else if (ufds[i].fd == get_signal_fd())
                    handle_signal();
            }
        }
    }

    return 0;
}
```

`poll` listens to events on multiple **fd**, defined by `ufds`. It contains 3 fd. `get_property_set_fd` is the Property Service Socket. `get_signal_fd` gets exit signal from sub-routine.

The Socket created by `start_property_service()`, when there is readable event, `poll()` can get the events, and run `handle_property_set_fd()`, defined in `property_service.c`:

```
void handle_property_set_fd()
{
    prop_msg msg;
    int s;
    int r;
    int res;
    struct ucred cr;
    struct sockaddr_un addr;
    socklen_t addr_size = sizeof(addr);
    socklen_t cr_size = sizeof(cr);
    char * source_ctx = NULL;

    if ((s = accept(property_set_fd, (struct sockaddr *) &addr, &addr_siz
e)) < 0) {
        return;
    }

    /* Check socket options here */
    if (getsockopt(s, SOL_SOCKET, SO_PEERCRED, &cr, &cr_size) < 0) {
        close(s);
        ERROR("Unable to receive socket options\n");
        return;
    }

    r = TEMP_FAILURE_RETRY(recv(s, &msg, sizeof(msg), 0));
    if(r != sizeof(prop_msg)) {
        ERROR("sys_prop: mis-match msg size received: %d expected: %d err
no: %d\n",
              r, sizeof(prop_msg), errno);
        close(s);
        return;
    }

    switch(msg.cmd) {
    case PROP_MSG_SETPROP:
        msg.name[PROP_NAME_MAX-1] = 0;
        msg.value[PROP_VALUE_MAX-1] = 0;

        if (!is_legal_property_name(msg.name, strlen(msg.name))) {
            ERROR("sys_prop: illegal property name. Got: \"%s\"\n", msg.n
ame);
            close(s);
            return;
        }
```

```
        getpeercon(s, &source_ctx);

        if(memcmp(msg.name,"ctl.",4) == 0) {
            // Keep the old close-socket-early behavior when handling
            // ctl.* properties.
            close(s);
            if (check_control_perms(msg.value, cr.uid, cr.gid, source_ct
x)) {
                handle_control_message((char*) msg.name + 4, (char*) ms
g.value);
            } else {
                ERROR("sys_prop: Unable to %s service ctl [%s] uid:%d gi
d:%d pid:%d\n",
                        msg.name + 4, msg.value, cr.uid, cr.gid, cr.pid);
            }
        } else {
            if (check_perms(msg.name, cr.uid, cr.gid, source_ctx)) {
                property_set((char*) msg.name, (char*) msg.value);
            } else {
                ERROR("sys_prop: permission denied uid:%d  name:%s\n",
                      cr.uid, msg.name);
            }

            // Note: bionic's property client code assumes that the
            // property server will not close the socket until *AFTER*
            // the property is written to memory.
            close(s);
        }
        freecon(source_ctx);
        break;

    default:
        close(s);
        break;
    }
}
```

It accepts message from client via `accept()` and `recv()`, and then based on message type, call permission checking method `check_perms()` and `property_set` method.

Summary of Kernel Bootstrap

- `start_kernel`
  - -> `rest_init()`: starts two kernel thread
  - -> `kernel_init()`: runs `run_init_process(execute_command)`, execute_command = '/init'
    - -> `main()` in `init.c`
    - **Step 1**: init file system and log system
    - **Extra Step**: Property Service
      - `property_init()`
      - `property_load_boot_defaults()` loads default system properties
      - `property_service_init_action()` & `queue_property_triggers_action()`
      - This is connected by Socket
    - **Step 2**: `init_parse_config_file("/init.rc");`, in `/system/core/init/init_parser.c`
      - -> `parse_config(fn, data)`
      - -> `parse_new_section()`
        - -> `parse_service()`
        - -> `parse_line_service()`
        - -> `parse_action()`
    - **Step 3**: Start Action and Service
      - `action_for_each_trigger()`, `queue_builtin_action()`
      - `execute_one_command()`
      - `service_start()`
    - **Step 4**: Initialise event listening loop
      - `poll(ufds, fd_count, timeout)`
      - `handle_property_set_fd()`

- `property_set()` & `check_perms()`

Complete Source Code of `main` in `init.c`

```c
int main(int argc, char **argv)
{
    int fd_count = 0;
    struct pollfd ufds[4];
    char *tmpdev;
    char* debuggable;
    char tmp[32];
    int property_set_fd_init = 0;
    int signal_fd_init = 0;
    int keychord_fd_init = 0;
    bool is_charger = false;

    if (!strcmp(basename(argv[0]), "ueventd"))
        return ueventd_main(argc, argv);

    if (!strcmp(basename(argv[0]), "watchdogd"))
        return watchdogd_main(argc, argv);

    /* clear the umask */
    umask(0);

        /* Get the basic filesystem setup we need put
         * together in the initramdisk on / and then we'll
         * let the rc file figure out the rest.
         */
    mkdir("/dev", 0755);
    mkdir("/proc", 0755);
    mkdir("/sys", 0755);

    mount("tmpfs", "/dev", "tmpfs", MS_NOSUID, "mode=0755");
    mkdir("/dev/pts", 0755);
    mkdir("/dev/socket", 0755);
    mount("devpts", "/dev/pts", "devpts", 0, NULL);
    mount("proc", "/proc", "proc", 0, NULL);
    mount("sysfs", "/sys", "sysfs", 0, NULL);

        /* indicate that booting is in progress to background fw loaders, etc */
    close(open("/dev/.booting", O_WRONLY | O_CREAT, 0000));

        /* We must have some place other than / to create the
         * device nodes for kmsg and null, otherwise we won't
         * be able to remount / read-only later on.
         * Now that tmpfs is mounted on /dev, we can actually
```

```
         * talk to the outside world.
         */
    open_devnull_stdio();
    klog_init();
    property_init();

    get_hardware_name(hardware, &revision);

    process_kernel_cmdline();

    union selinux_callback cb;
    cb.func_log = klog_write;
    selinux_set_callback(SELINUX_CB_LOG, cb);

    cb.func_audit = audit_callback;
    selinux_set_callback(SELINUX_CB_AUDIT, cb);

    selinux_initialise();
    /* These directories were necessarily created before initial policy load
     * and therefore need their security context restored to the proper value.
     * This must happen before /dev is populated by ueventd.
     */
    restorecon("/dev");
    restorecon("/dev/socket");
    restorecon("/dev/__properties__");
    restorecon_recursive("/sys");

    is_charger = !strcmp(bootmode, "charger");

    INFO("property init\n");
    if (!is_charger)
        property_load_boot_defaults();

    INFO("reading config file\n");
    init_parse_config_file("/init.rc");

    action_for_each_trigger("early-init", action_add_queue_tail);

    queue_builtin_action(wait_for_coldboot_done_action, "wait_for_coldboot_done");
    queue_builtin_action(mix_hwrng_into_linux_rng_action, "mix_hwrng_into_linux_rng");
    queue_builtin_action(keychord_init_action, "keychord_init");
```

```
    queue_builtin_action(console_init_action, "console_init");

    /* execute all the boot actions to get us started */
    action_for_each_trigger("init", action_add_queue_tail);

    /* skip mounting filesystems in charger mode */
    if (!is_charger) {
        action_for_each_trigger("early-fs", action_add_queue_tail);
        action_for_each_trigger("fs", action_add_queue_tail);
        action_for_each_trigger("post-fs", action_add_queue_tail);
        action_for_each_trigger("post-fs-data", action_add_queue_tail);
    }

    /* Repeat mix_hwrng_into_linux_rng in case /dev/hw_random or /dev/random
     * wasn't ready immediately after wait_for_coldboot_done
     */
    queue_builtin_action(mix_hwrng_into_linux_rng_action, "mix_hwrng_into_linux_rng");

    queue_builtin_action(property_service_init_action, "property_service_init");
    queue_builtin_action(signal_init_action, "signal_init");
    queue_builtin_action(check_startup_action, "check_startup");

    if (is_charger) {
        action_for_each_trigger("charger", action_add_queue_tail);
    } else {
        action_for_each_trigger("early-boot", action_add_queue_tail);
        action_for_each_trigger("boot", action_add_queue_tail);
    }

        /* run all property triggers based on current state of the properties */
    queue_builtin_action(queue_property_triggers_action, "queue_property_triggers");


#if BOOTCHART
    queue_builtin_action(bootchart_init_action, "bootchart_init");
#endif

    for(;;) {
        int nr, i, timeout = -1;
```

```

        execute_one_command();
        restart_processes();

        if (!property_set_fd_init && get_property_set_fd() > 0) {
            ufds[fd_count].fd = get_property_set_fd();
            ufds[fd_count].events = POLLIN;
            ufds[fd_count].revents = 0;
            fd_count++;
            property_set_fd_init = 1;
        }
        if (!signal_fd_init && get_signal_fd() > 0) {
            ufds[fd_count].fd = get_signal_fd();
            ufds[fd_count].events = POLLIN;
            ufds[fd_count].revents = 0;
            fd_count++;
            signal_fd_init = 1;
        }
        if (!keychord_fd_init && get_keychord_fd() > 0) {
            ufds[fd_count].fd = get_keychord_fd();
            ufds[fd_count].events = POLLIN;
            ufds[fd_count].revents = 0;
            fd_count++;
            keychord_fd_init = 1;
        }

        if (process_needs_restart) {
            timeout = (process_needs_restart - gettime()) * 1000;
            if (timeout < 0)
                timeout = 0;
        }

        if (!action_queue_empty() || cur_action)
            timeout = 0;

#if BOOTCHART
        if (bootchart_count > 0) {
            if (timeout < 0 || timeout > BOOTCHART_POLLING_MS)
                timeout = BOOTCHART_POLLING_MS;
            if (bootchart_step() < 0 || --bootchart_count == 0) {
                bootchart_finish();
                bootchart_count = 0;
            }
        }
#endif
```

```
        nr = poll(ufds, fd_count, timeout);
        if (nr <= 0)
            continue;

        for (i = 0; i < fd_count; i++) {
            if (ufds[i].revents == POLLIN) {
                if (ufds[i].fd == get_property_set_fd())
                    handle_property_set_fd();
                else if (ufds[i].fd == get_keychord_fd())
                    handle_keychord();
                else if (ufds[i].fd == get_signal_fd())
                    handle_signal();
            }
        }
    }

    return 0;
}
```

Share this article

说点什么吧...

(http://duoshuo.com/settings/avatar/)

分享到: 发布

allenlsy正在使用多说 (http://duoshuo.com)

Blog (http://allenlsy.com) · Casts (http://cast.allenlsy.com) ·

(https://twitter.com/feed.xml)m/allenlsy)