

实验案例三：内核子系统—启动流程与系统调用

实验案例三：内核子系统—启动流程与系统调用

- 一、实验简介
- 二、实验内容及要求
 - 1.内核启动
 - 2. 添加系统调用
- 三、实验原理
 - 1. 内核启动流程
 - 2. 系统调用
 - 2.1 基本概念
 - 2.2 源码分析
- 四、实验流程
 - 任务一：内核启动
 - 1. 流程
 - 任务二：添加系统调用
 - 1.流程
- 五、参考资料

一、实验简介

OpenHarmony 采用多内核设计，目前支持 **Linux**、**LiteOS-m** 与 **LiteOS-a** 内核。开发者可根据设备需求和资源情况选择不同内核实现产品。所有内核均使用 **KAL（Kernel Abstraction Layer）** 模块，通过屏蔽内核差异向上提供统一接口，从而降低开发难度。

LiteOS 是华为面向物联网（IoT）领域开发的轻量级操作系统，主要应用于智能家居、个人穿戴、车联网、城市公共服务及制造业等场景。LiteOS 具有高实时性、高稳定性和低功耗特点，基础内核可裁剪至不到 10 KB，适合部署在小型设备中。因此，LiteOS 成为 OpenHarmony 在轻量 and 小型系统中的主要内核选择。

本节实验旨在通过对内核进行基础修改，增加对 OpenHarmony 支持的内核有初步了解。考虑到 Linux 内核相关资料较为完备，本次实验仅聚焦 **LiteOS** 内核。LiteOS-a 与 LiteOS-m 系统结构相似，仅在面向不同应用层级时有少量差异。本实验及后续实验均选用 **LiteOS-a** 内核，其功能更完善，并且支持 MMU 等特性，更贴近现代操作系统功能。

二、实验内容及要求

本次实验需要依次完成下列实验要求，并提交在qemu中运行的结果截图。

1.内核启动

本节实验需要在OpenHarmony的内核子系统的LiteOS-A内核启动框架中注册新模块，并在新模块中打印信息。具体来说，需要分别在 `LOS_INIT_LEVEL_EARLIEST`、`LOS_INIT_LEVEL_KMOD_PREVM`、`LOS_INIT_LEVEL_KMOD_EXTENDED` 中注册新模块，并打印简单的信息。最终运行 `qemu_run` 运行结果如下：

```
Enter to start qemu[y/n]:y
Waiting VNC connection on: 5920 ...(Ctrl-C exit)
初始化：LOS_INIT_LEVEL_EARLIEST

*****welcome*****
```

```
Processor    : Cortex-A7
Run Mode     : UP
GIC Rev      : GICv2
build time   : Feb 24 2024 20:42:22
Kernel       : Huawei LiteOS 2.0.0.37/debug
```

```
main core booting up...
初始化: LOS_INIT_LEVEL_KMOD_PREVM
初始化: LOS_INIT_LEVEL_KMOD_EXTENDED
cpu 0 entering scheduler
mem dev init ...
DevMmzRegister...
Date:Feb 24 2024.
Time:20:42:21.
```

2. 添加系统调用

本实验要求在 OpenHarmony 中开发系统调用，为 **LiteOS-a** 内核添加新的系统调用接口。通过在 **musl** 库中添加用户态函数接口，使后续应用程序能够调用该接口，从而使用新增的系统调用功能。具体要求如下：

1. LiteOS-a内核中新增系统调用接口A,在其中打印信息“Kernel mode:: SYS_new_syscall”
2. musl库中新增函数接口B，在其中打印信息“User mode:: SYS_new_syscall”，并调用系统接口B
3. 在实验二中创建的sysu子系统下新建新的组件和可执行程序模块C，使得运行C时其会调用函数接口B
4. 最终在OHOS系统中运行程序C的结果应当如下所示：

```
OHOS:/$ ./bin/new_syscall
User mode:: SYS_new_syscall
Kernel mode: SYS_new_syscall
```

完成之后，请提供运行截图，并简述如何实现。

三、实验原理

1. 内核启动流程

LiteOS-A的内核启动流程大致包含有4个阶段，分别是**BootLoader阶段**、**汇编启动阶段**、**C语言启动阶段**以及**系统服务启动阶段**。其中，内核的主要工作集中在 **汇编启动阶段** 和 **C语言启动阶段**，本实验任务也主要涉及这两个阶段。

第一阶段——BootLoader阶段，BootLoader 程序负责将内核镜像从闪存复制到内存指定区域，并跳转至地址 `0x80000000`，进入汇编代码引导阶段，即 **汇编启动阶段**，以启动 LiteOS-A 内核。

第二阶段——汇编启动阶段，此阶段的功能在于继续完成CPU初始设置，关闭dcache/icache，使能FPU及neon，设置MMU建立虚实地址映射，设置系统栈，清理bss段，最后调用C语言main函数，进入C语言启动阶段。其入口代码主要定义在编译系统生成系统镜像时所依据的链接脚本之中。具体而言，位于源码 `//kernel/liteos_a/tool/build/liteos_11vm.ld` 中定义的 `ENTRY(reset_vector)` 即为此阶段的入口。

```

/*liteos_llvm.ld*/
ENTRY(reset_vector)
INCLUDE board.ld
INPUT(libuserinit.o)
SECTIONS
{
...
}

```

`reset_vector` 则定义在 `//kernel/liteos_a/arch/arm/arm/src/startup` 目录下，其中包含有文件 `reset_vector_mp.S` 与 `reset_vector_up.S`，`up`(unique processor)与`mp`(multi processor)分别对应着单核与多核处理器平台。两者都主要完成一些初始化工作，并最后跳转至前文提到的 `main()` 函数中，也就是进入内核的C语言启动阶段。

```

reset_vector:
...
    bl    main
_start_hang:
    b     _start_hang

```

第三阶段——C语言启动阶段。该阶段主要包含`OsMain`函数及开始调度等，如图1所示，C语言阶段执行的`main`函数会继续调用`OsMain`完成内核各个模块的初始化，最后运行`OsScheduleStart`函数开始调度任务。同时可以看到图中包含许多子任务需要完成，其中的灰色块任务为LiteOS内核固定需要完成的任务。而橙色方框的任务则可根据需要自行添加与修改。

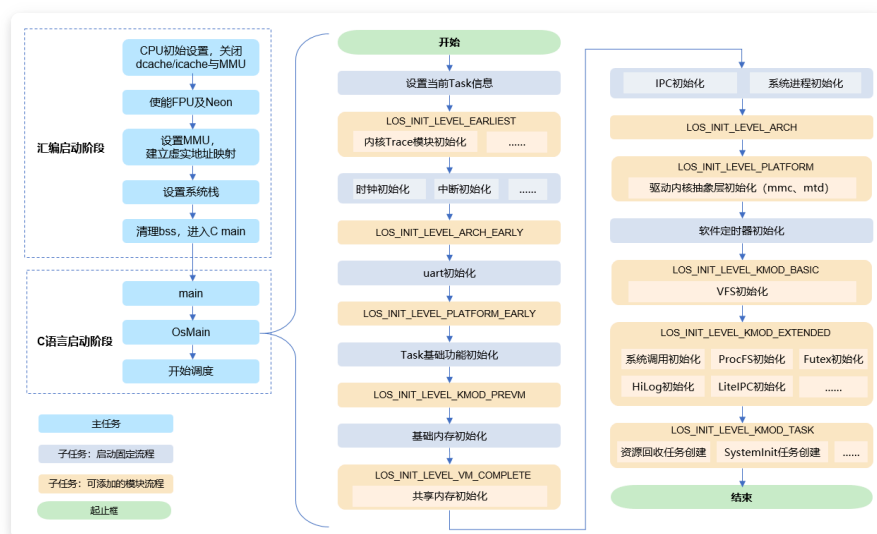


图1.内核态启动阶段示意图[1]

`main`函数代码具体位于 `//kernel/liteos_a/kernel/common/main.c` 之中，其中主要在 `osMain()` 与 `osSchedStart()` 函数之中完成内核阶段各模块的初始化以及内核的调度任务。

```

/*main.c*/
LITE_OS_SEC_TEXT_INIT INT32 main(VOID)
{
    UINT32 ret = OsMain();
    ...
    OsSchedStart();
    ...
}

```

`OsMain()` 主要完成liteos-a系统的各个模块的初始化任务，而其中的 `OsSystemProcessCreate()` 函数完成系统的内核进程的初始化与创建工作。在LiteOS-a系统内核中内核态被视为单独的一个内存空间，整个系统仅存在两个内核态进程，即根进程KProcess(2号进程)与KIdle (0号进程)。其中KIdle进程为KProcess进程的子进程，并且二者共享同一个进程空间。KProcess进程优先级为0，其中包含着子线程ResourceTask负责资源的回收，KIdle进程的优先级则为最低的31，其中包含子线程Idle负责在CPU空闲时执行，这两者进程即在OsSystemProcessCreate()中完成创建工作。

同时在 `OsMain()` 函数中的 `OsInitCall()` 函数则负责内核系统中相应注册模块的启动。而在 `LOS_INIT_LEVEL_KMOD_TASK` 模块之中所注册的任务函数，就包含有 `SystemInit` 任务的创建工作。

```

LITE_OS_SEC_TEXT_INIT UINT32 OsMain(VOID)
{
    ...
    ret = OsSystemProcessCreate();
    ...
    OsInitCall(LOS_INIT_LEVEL_KMOD_TASK);
    ...
}

```

在 `SystemInit` 任务是LiteOS-a系统软硬件初始化的入口，其中会依次调用各个外围软硬件模块的初始化入口函数进行初始化工作，包括有启动内核态驱动框架、挂载根文件系统以及最后的创建用户态根进程**init**。需要注意的是，由于OpenHarmony系统需要安装于不同的硬件开发板之中，因此这一部分的初始化也同样需要根据开发板的不同进行适配。故而 `SystemInit` 任务的执行需要跳出内核态代码，以实验所使用的 `qemu_small_system_demo` 为例，其 `SystemInit` 任务定义位于 `//device/qemu/qem_virt/liteos_a/board/os_adapt/os_adapt.c` 之中。

```

void SystemInit(void)
{
#ifdef LOSCFG_DRIVERS_RANDOM
    dprintf("dev random init ...\n");
    (void)DevRandomRegister();
    ...
    dprintf("OsUserInitProcess start ...\n");
    if (OsUserInitProcess()) {
        PRINT_ERR("Create user init process failed!\n");
        return;
    }
    dprintf("OsUserInitProcess end ...\n");
    return;
}

```

`SystemInit()` 函数之中调用的 `OsUserInitProcess()` 任务则会重新跳回到内核态代码之中，并在其中创建 LiteOS-a 系统的用户态进程 `Init` (1号进程)，并在其中为 `Init` 进程创建以符号 `__user_init_entry` 为入口的子线程。

```
LITE_OS_SEC_TEXT_INIT UINT32 OsUserInitProcess(VOID)
{
    ...
    LosProcessCB *processCB = OsGetUserInitProcess();
    ret = OsSystemProcessInit(processCB, OS_USER_MODE, "Init");           // 新建用户态Init
进程
    ...
    ret = OsLoadUserInit(processCB);                                     // 虚拟地址到物理
地址转换
    ...
    param.pfnTaskEntry = (TSK_ENTRY_FUNC)(CHAR *)&__user_init_entry;    // 线程入口函
数
    ...
}
```

事实上 `__user_init_entry` 所指向的函数，即是 `los_user_init.c` 文件之中的 `OsUserInit()` 函数。在 `OsUserInit()` 中，系统通过 `sys_call3` 切换到用户态，运行 `g_initPath` 指向的 `/bin/init` 程序，从而启动 LiteOS-A 系统的用户态服务框架。不过由于 `/bin/init` 程序所进行的工作由 OpenHarmony 的启动恢复子系统所完成，已经超越内核子系统的范畴，因此在此处不再赘述。

```
// los_user_init.h
#ifndef LITE_USER_SEC_ENTRY
#define LITE_USER_SEC_ENTRY    __attribute__((section(".user.entry"))) //定义
__user_init_entry所指向的入口
#endif

// los_user_init.c
#ifdef LOSCFG_QUICK_START
LITE_USER_SEC_RODATA STATIC CHAR *g_initPath = "/dev/shm/init";
#else
LITE_USER_SEC_RODATA STATIC CHAR *g_initPath = "/bin/init";
#endif

LITE_USER_SEC_ENTRY VOID OsUserInit(VOID *args)
{
#ifdef LOSCFG_KERNEL_DYNLOAD
    sys_call3(__NR_execve, (UINTPTR)g_initPath, 0, 0);
#endif
    while (true) {
    }
}
```

2. 系统调用

2.1 基本概念

LiteOS-A 内核实现了用户态与内核态的隔离，以保护内核资源。用户态程序无法直接访问内核资源，但在某些场景下，用户态程序确实需要使用内核提供的功能。因此，需要提供一种安全机制，使用户态程序能够访问内核资源——这正是系统调用的设计目的。

系统调用为用户态程序提供了一种访问内核资源、与内核进行交互的通道。用户程序通过调用系统调用接口，会触发操作系统的SVC/SWI异常，使得程序由用户态转换为内核态，进而对接到内核中的系统调用统一处理接口中，根据触发的系统调用类型运行相应的处理函数之中，最后完成用户程序对内核资源的访问。但是大多数情况下用户程序很少直接使用系统调用接口，而是调用内核对外提供的POSIX接口。

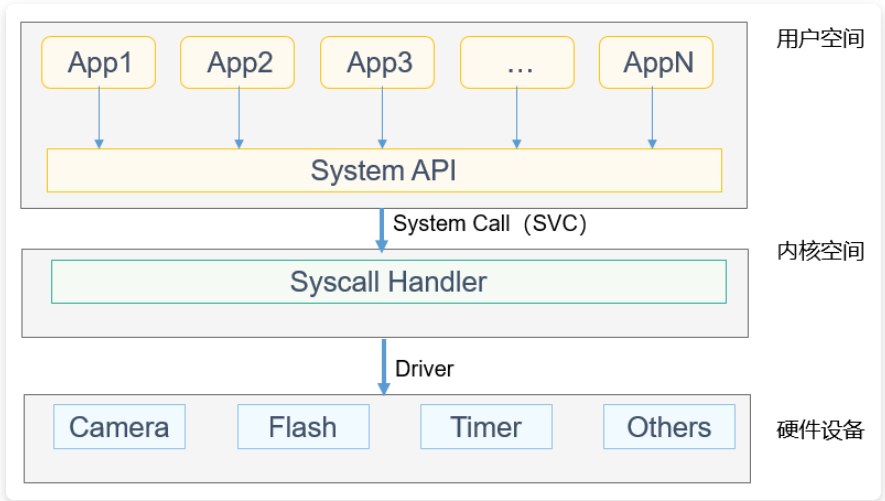


图2.系统调用示意图[2]

musl 是一种 C 标准库，主要用于基于 Linux 内核的操作系统。与广泛使用的 glibc 相比，musl 对库函数的实现更加简洁易读，生成的二进制文件体积也更小，因此在硬件资源受限的系统中具有显著优势，主要面向嵌入式系统和移动设备。在 OpenHarmony 的 LiteOS-A 内核中，musl 被用作操作系统的 C 标准库。实验中编写的程序也基于该库。musl 支持 POSIX 标准，并提供访问内核系统调用的 API 接口，使应用程序能够通过它调用系统调用接口，从而访问内核空间。

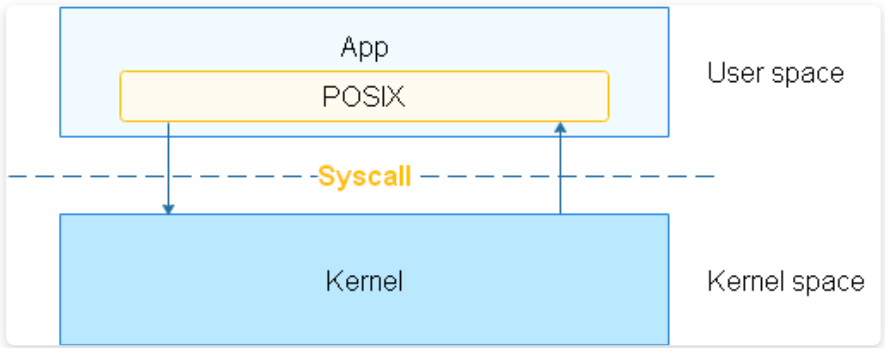


图3.POSIX接口框架[3]

2.2 源码分析

musl 库并非 OpenHarmony 原生组成部分，可在第三方库源码目录 `///third_party/musl` 中找到。其代码目录结构如下所示：

```

musl
├── include                # musl头文件
├── ldso                   # 动态链接器
├── libc-test              # 关于musl的测试用例
├── ndk_musl_include       # 生成ndk版本的头文件
├── ndk_script             # 生成ndk版本的脚本
├── porting                # linux liteos平台对musl适配文件
│   ├── liteos_a           # liteos-a适配文件
│   │   ├── kernel         # liteos-a内核调用文件
│   │   └── user            # 平台适配库函数实现
│   └── ...                # 其它平台适配文件
├── src                   # musl文件源
└── scripts               # 测试用脚本

```

在 LiteOS-a 内核的 OpenHarmony 程序中调用的标准库函数其实现均来自 musl。编译过程中，`hb` 工具会将 musl 头文件、链接脚本及其他必要文件复制到产品输出目录，例如 `//out/arm_virt/qemu_small_system_demo/sysroot`，程序引用的 C 头文件均位于此目录下。

在其中需要注意的是，由于操作系统的不同一些c标准库函数也同样有着不同，因此如Linux与LiteOS就都需要有相应的适配文件，它们都位于musl库目录下的porting中。像LiteOS-a内核的适配文件就位于 `porting/liteos_a` 目录之下，其中的`kernel`目录文件为内核系统所调用。`user`目录文件则将被覆盖到产品输出目录 `//out/arm_virt/qemu_small_system_demo/sysroot` 之中。若 musl 库与平台适配库函数存在重名，系统将优先调用平台适配实现。

以调用 POSIX 标准 API 函数 `popen()` 为例，其执行流程如下：

1. `popen()` 在 `musl/src/stdio/popen.c` 与 `porting/liteos_a/user/stdio/popen.c` 中均有实现。由于目标系统为 LiteOS-a，最终调用的是平台适配文件中的实现。
2. `popen()` 创建管道并调用 `fork` 生成子进程执行指定命令。若管道创建失败，将调用 `__syscall(SYS_close, p[0])` 触发系统调用。系统调用号 `SYS_close` 定义在 `porting/liteos_a/kernel` 下的 `syscall.h` 与 `porting/liteos_a/user/` 下的 `syscall.h.in` 之中。其中 `syscall.h` 文件将被LiteOS_a内核所引用，而 `syscall.h.in` 将生成 `syscall.h` 文件并被复制到产品输出目录下，供程序所调用。
3. 当使用系统调用之后将会产生操作系统的SVC/SWI异常转换到内核态，并在系统调用统一处理接口中处理，也就是 `//kernel/liteos_a/syscall/los_syscall.c` 中定义的 `OsArmA32SyscallHandle()`。它会根据所触发的系统调用号，跳转到相应的函数中进行处理。
4. `//kernel/liteos_a/syscall/los_syscall.c` 中还定义了 `OsSyscallHandleInit()` 函数，它负责为操作系统注册不同的系统调用号，在里面定义了注册系统调用的宏定义 `SYSCALL_HAND_DEF`，并且引用了头文件 `syscall_lookup.h`。`popen()` 中所使用的系统调用号 `SYS_close` 就在其中通过 `SYSCALL_HAND_DEF(__NR_close, Sysclose, int, ARG_NUM_1)` 注册到 `Sysclose()` 函数。
5. 最后通过系统调用统一接口，程序将跳转到 `Sysclose()` 函数执行，完成用户程序对内核资源的访问并在最后退回到用户态之中。由此即完成了一次系统调用。

四、实验流程

任务一：内核启动

1. 流程

本节任务主要需要修改LiteOS-a内核，在内核的启动框架之中进行模块的注册，可自行选择实现方法，一种可行的方法如下：

1. 在LiteOS-a内核目录下创建新的文件夹,并在其中添加模块函数实现，主要使用 `LOS_MODULE_INIT` 宏定义为启动框架注册模块，需要注意的是 `LOS_MODULE_INIT` 宏定义在 `los_init.h` 之中，打印信息函数 `PRINTK` 定义在 `los_printf.h` 之中。无需手动配置头文件路径，GN 构建文件已自动包含所需路径。
2. 修改 `//kernel/liteos_a/BUILD.gn`，在目标 group("modules") 中添加新建模块目录。
3. 参照 `//kernel/liteos_a/syscall/BUILD.gn` 编写新模块的编译构建文件。

任务二：添加系统调用

1. 流程

此次任务需要为LiteOS-a内核添加新的系统调用，要求通过musl库来间接调用系统调用接口。本次的任务需要修改musl库与LiteOS-a内核，大致流程如下：

1. 需要在musl库，即 `third_party/musl/porting/liteos_a/user/arch/arm/bits/syscall.h.in` 下添加新系统调用声明。需要注意，新建的系统调用号需要在 `__NR_syscallend` 之前，因为内核系统调用的注册就以该标志截至。
2. 在 `third_party/musl/porting/liteos_a/user/include` 与 `third_party/musl/porting/liteos_a/user/src` 新建新的头文件添加函数。该函数需要打印"**User mode:: SYS_new_syscall**"并且帮助调用系统调用接口。这一部分不需要编写BUILD.gn编译构建文件。
3. 在 `third_party/musl/porting/liteos_a/kernel/include/bits/syscall.h` 添加系统调用号声明，此文件为LiteOS-a内核引用。
4. 在 `kernel/liteos_a/syscall/los_syscall.h` 添加系统调用处理函数声明的。
5. 在 `kernel/liteos_a/syscall/syscall_lookup.h` 添加系统调用号的注册，将新建的系统调用号与相应的处理函数联系在一起。
6. `kernel/liteos_a/syscall` 下新建文件，其中需要实现系统调用处理函数,函数的实现需要打印"**Kernel mode:: SYS_new_syscall**"。并在 `kernel/liteos_a/syscall/BUILD.gn` 中 `sources` 添加新建的文件,将其加入到文件编译之中。
7. 上一次实验之中为OpenHarmony新建了sysu子系统，此次任务需要在其中再次新建组件，用于调用此次任务中为musl库新增加的接口，进而使用系统调用。
8. 编译系统并进入 OHOS 系统，运行新增组件程序，验证其是否能够正确调用新增的 musl 接口及系统调用，实现预期功能。

五、参考资料

- [1]. 内核启动:<https://docs.openharmony.cn/pages/v4.1/zh-cn/device-dev/kernel/kernel-small-start-kernel.md>
- [2]. OpenHarmony系统调用:<https://docs.openharmony.cn/pages/v4.1/zh-cn/device-dev/kernel/kernel-small-bundles-system.md>

[3]. musl标准库: [https://docs.openharmony.cn/pages/v4.1/zh-cn/device-dev/kernel/kernel-small-apx-library.m
d](https://docs.openharmony.cn/pages/v4.1/zh-cn/device-dev/kernel/kernel-small-apx-library.md)