

1. 分离应用程序

之前我们U模式下的应用程序是写在`app.c`中的，是和内核一起编译打包的，但是内核和应用程序都需要开启虚拟地址，而这两者的映射状态是不同的，类比于linux windows这种操作系统都是操作系统来将程序加载到内存来运行，因此我们需要将内核和应用程序隔离开来。

在`os`目录下新建`user`文件夹，在此文件夹下的文件组成如下：



`time.c`和`write.c`是两个应用该程序，`Makefile`用于编译，`user.ld`是链接脚本，在开启虚拟地址后，用户程序就可使用同一个链接文件了，因为即使被链接到同一个虚拟地址，也会映射到不同的物理地址。`bin`目录是编译生成的可执行文件，是`elf`格式的。

我把`lib`目录下的`stack.c`移动到了`src`目录下

`write.c`

```

#include <timeros/os.h>
#include <timeros/syscall.h>
#include <timeros/stdio.h>
uint64_t syscall(size_t id, reg_t arg1, reg_t arg2, reg_t
arg3) {

    register uintptr_t a0 asm ("a0") = (uintptr_t)(arg1);
    register uintptr_t a1 asm ("a1") = (uintptr_t)(arg2);
    register uintptr_t a2 asm ("a2") = (uintptr_t)(arg3);
    register uintptr_t a7 asm ("a7") = (uintptr_t)(id);

    asm volatile ("ecall"
                  : "+r" (a0)
                  : "r" (a1), "r" (a2), "r" (a7)
                  : "memory");

    return a0;
}

uint64_t sys_write(size_t fd, const char* buf, size_t len)
{
    return syscall(__NR_write,fd,buf, len);
}

int main()
{

    const char *message = "task1 is running!\n";
    int len = strlen(message);
    while (1)
    {
        printf(message);
    }
    return 0;
}

```

time.c

```

#include <timeros/types.h>
#include <timeros/syscall.h>

uint64_t syscall(size_t id, reg_t arg1, reg_t arg2, reg_t
arg3) {

    register uintptr_t a0 asm ("a0") = (uintptr_t)(arg1);
    register uintptr_t a1 asm ("a1") = (uintptr_t)(arg2);
    register uintptr_t a2 asm ("a2") = (uintptr_t)(arg3);
    register uintptr_t a7 asm ("a7") = (uintptr_t)(id);

    asm volatile ("ecall"
                  : "+r" (a0)
                  : "r" (a1), "r" (a2), "r" (a7)
                  : "memory");

    return a0;
}

uint64_t sys_gettime()
{
    return syscall(__NR_gettimeofday,0,0,0);
}

int main()
{
    uint64_t current_timer = 0;
    while (1)
    {
        current_timer = sys_gettime();
    }
    return 0;
}

```

分离了两个应用程序

Makefile

```
CROSS_COMPILE = riscv64-unknown-elf-
CFLAGS = -nostdlib -fno-builtin

CC = ${CROSS_COMPILE}gcc
OBJCOPY = ${CROSS_COMPILE}objcopy
OBJDUMP = ${CROSS_COMPILE}objdump
INCLUDE:=-I../include

LIB = ../lib

write: write.c $(LIB)/*.c
    ${CC} ${CFLAGS} $(INCLUDE) -T user.ld -Wl,-Map=write.map -o
bin/write $^

time: time.c
    ${CC} ${CFLAGS} $(INCLUDE) -T user.ld -Wl,-Map=time.map -o
bin/time $^
```

`user.ld`: 在链接脚本中将程序的入口地址指明为`main`函数, 链接地址为`0x10000`

```

OUTPUT_ARCH(riscv)
ENTRY(main)
BASE_ADDRESS = 0x10000;

SECTIONS
{
    . = BASE_ADDRESS;
    .text : {
        *(.text.entry)
        *(.text .text.*)
    }
    . = ALIGN(4K);
    .rodata : {
        *(.rodata
.rodata.*)
        *(.srodata
.srodata.*)
    }
    . = ALIGN(4K);
    .data : {
        *(.data .data.*)
        *(.sdata
.sdata.*)
    }
    .bss : {
        *(.bss .bss.*)
        *(.sbss .sbss.*)
    }
    /DISCARD/ : {
        *(.eh_frame)
        *(.debug*)
    }
}

```

2. 装载应用程序

在rCore中，在编译os之前，它是使用了一个build.rs来生成了一段汇编代码，这段汇编代码会嵌入到内核中，用于指示APP的个数和夹杂APP的二进制文件到内存中。我将这个build.rs改成了c语言的实现，在os目录下新建一个build.c

代码逻辑就是遍历user/bin目录下的文件个数，然后记录用户程序数量，生成link_app.S


```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dirent.h>

#define TARGET_PATH "../user/bin/"

int compare_strings(const void* a, const void* b) {
    return strcmp(*(const char**)a, *(const char**)b);
}

void insert_app_data() {

    FILE* f = fopen("src/link_app.S", "w");
    if (f == NULL) {
        perror("Failed to open file");
        exit(EXIT_FAILURE);
    }

    char* apps[100]; // Assuming a maximum of 100 apps
    int app_count = 0;

    // Read the directory and collect app names
    DIR* dir = opendir("../user/bin");
    if (dir == NULL) {
        perror("Failed to open directory");
        exit(EXIT_FAILURE);
    }

    struct dirent* dir_entry;
    while ((dir_entry = readdir(dir)) != NULL) {

        char* name_with_ext = dir_entry->d_name;

        // 排除掉 . 和 .. 条目
        if (name_with_ext[0] == '.' && (name_with_ext[1] == '\\0' || (name_with_ext[1]
== '.' && name_with_ext[2] == '\\0')) {
            continue; // Skip this entry
        }

        int len = strlen(name_with_ext);

        // Remove file extension by replacing the dot with a null terminator
        for (int i = 0; i < len; i++) {
            if (name_with_ext[i] == '.') {
                name_with_ext[i] = '\\0';
                break;
            }
        }

        // strdup 函数用于创建一个字符串的副本，并返回指向新字符串的指针。
        apps[app_count] = strdup(name_with_ext);
        app_count++;
        printf("File name: %s, app_count: %d\\n", name_with_ext, app_count);
    }
}

```

```

    closedir(dir);

    // 对 app name 排序
    qsort(apps, app_count, sizeof(char*), compare_strings);

    fprintf(f, "\n.align 3\n.section .data\n.global _num_app\n_num_app:\n.quad %d",
app_count);

    for (int i = 0; i < app_count; i++) {
        fprintf(f, "\n.quad app_%d_start", i);
    }
    fprintf(f, "\n.quad app_%d_end", app_count - 1);

    for (int i = 0; i < app_count; i++) {
        printf("app_%d: %s\n", i, apps[i]);
        fprintf(f, "\n.section .data\n.global app_%d_start\n.global app_%d_end\n.align
3\napp_%d_start:\n.incbin \"%s%s\"\napp_%d_end:", i, i, i, TARGET_PATH, apps[i], i);
        free(apps[i]);
    }

    fclose(f);
}

int main() {
    insert_app_data();
    return 0;
}

```

在os的Makefile中添加对build.c的编译:

c

```

build_app: build.c
    gcc $< -o
build.out

```

然后在编译内核代码之前需要先编译此代码，然后执行生成link_app.S，link_app.S会被放在src目录下。

然后修改一下build.sh：先编译执行build.c


```

# 编译os
if [ ! -d "$SHELL_FOLDER/output/os" ]; then
mkdir $SHELL_FOLDER/output/os
fi
cd $SHELL_FOLDER/os
# 编译app加载模块
make build_app
./build.out
# 编译os
make
cp $SHELL_FOLDER/os/os.bin
$SHELL_FOLDER/output/os/os.bin
make clean

```

运行`build.sh`，就可以看见在`src`目录下生成了`link_app.S`文件：

plaintext

```

    .align 3
    .section .data
    .global _num_app
_num_app:
    .quad 2
    .quad app_0_start
    .quad app_1_start
    .quad app_1_end

    .section .data
    .global app_0_start
    .global app_0_end
    .align 3
app_0_start:
    .incbin
    "../user/bin/time"
app_0_end:

    .section .data
    .global app_1_start
    .global app_1_end
    .align 3
app_1_start:
    .incbin
    "../user/bin/write"
app_1_end:

```

可以看见总共有两个用户程序，用户程序开始的地方是两个标号：`app_0_start`，`app_1_start`，`.incbin`伪指令的功能是包含可执行文件、文字或其他任意数据。文件的内容将按字节逐一添加到当前 ELF 节中，而不进行任何方式的解释。简单来说就是加载二进

制数据到内存中，之后我们就可以去访问此段内存的数据了，并将其解析出来。`.quad`的功能是声明一个64位的数据。

接下来来读取这些信息，在`include`目录下新建一个`loader.h`的文件，`src`目录下新建`loader.c`

`loader.h`：定义了一个描述app数据的结构体

C

```
#ifndef TOS_LOADER_H__
#define TOS_LOADER_H__

#include <timeros/types.h>
#include <timeros/assert.h>
#include <timeros/stdio.h>

// 假设这个结构用于存储应用程序元数据
typedef struct {
    uint64_t start;
    uint64_t size;
} AppMetadata;

// 获取加载的app数量
size_t get_num_app();
// 获取app的数据
AppMetadata get_app_data(size_t
app_id);

#endif
```

`loader.c`：`_num_app`由于在汇编中进行了声明，在c语言中可以将其看作一个数组，数组大小是4，`_num_app[0]`的值就是2。`app`数据的起始地址也放在了`_num_app`数组中，根据传入的 `app_id`进行索引，由于app的数据是挨着放置的，所以app数据的大小可用下一个app的起始地址减去当前app的起始地址。

```

#include <timeros/loader.h>

extern u64 _num_app[];

// 获取加载的app数量
size_t get_num_app()
{
    return _num_app[0];
}

AppMetadata get_app_data(size_t app_id)
{
    AppMetadata metadata;

    size_t num_app = get_num_app();

    metadata.start = _num_app[app_id]; // 获取app起始地址
    metadata.size = _num_app[app_id+1] - _num_app[app_id]; // 获取app数据长度

    printk("app start:%x , app end: %x\n", metadata.start, metadata.size);

    assert(app_id <= num_app);

    return metadata;
}

```

3. 测试

首先在user目录下编译生成用户程序：

sh

```

make
time
make
write
e

```

在main函数中打印一下app的数量：

```

void os_main()
{
    printk("hello timer os!\n");

    // 内存分配器初始化
    frame_alloctor_init();

    printk("num
app:%d\n",get_num_app());
    //初始化内存
    kvminit();

    //映射内核
    kvminithart();

    //trap初始化
    trap_init();

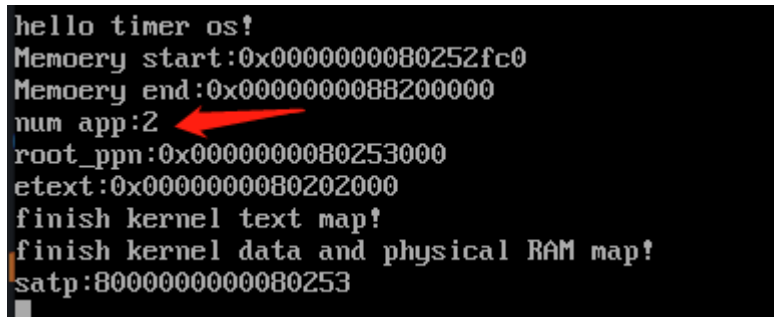
    while (1)
    {
        /* code */
    }

    // task_init();

    // timer_init();

    // run_first_task();
}

```



```

hello timer os!
Memoery start:0x0000000080252fc0
Memoery end:0x0000000088200000
num app:2
root_ppn:0x0000000080253000
etext:0x0000000080202000
finish kernel text map!
finish kernel data and physical RAM map!
satp:800000000080253

```

确保没问题，下一节就来解析用户程序的数据，用户程序都是elf格式的。

参考链接

实现批处理操作系统 - rCore-Tutorial-Book-v3 3.6.0-alpha.1 文档 (rcore-os.cn)