Lab 6：Linux系统调用

重建内核

采用交叉编译的方式编译内核能获得很好的效率，但是需要更多的配置。由于没有安装交叉编译器，又考虑到树莓派 3 的处理器性能还不错，所以选择了在本机进行编译。

树莓派的官网提供了关于内核的参考文档，下面简要叙述一下内核编译的过程。

下载源码

各版本的内核源码提供在 Github 上，可以选定版本下载，将源码下载到本地：

```
git clone --depth=1 https://github.com/raspberrypi/linux
```

参数 `--depth=1` 指定了拷贝深度为 1，即只下载最新版本的内核源码。

配置内核

可以手工地修改 `.config` 文件进行内核的配置，但是更好的方法是借助菜单工具：

```
sudo apt-get install libncurses5-dev
```

然后启动配置菜单：

```
make menuconfig
```

对于每一个配置选项，用户有三种选择，编译进内核、不编译进内核，或将该功能编译为内核模块，在需要时动态插入。

如果想使用原有的内核配置，可以将原有的 config 文件复制过来。或者树莓派提供了默认的内核配置方式，对于 PI 3 执行：

```
KERNEL=kernel7
make bcm2709_defconfig
```

编译内核

配置完成后，准备编译生成内核，首先安装所需的依赖：

```
sudo apt-get install bc
```

开始编译内核：

```
make -j4 zImage modules dtbs
```

为了充分利用 PI 3 的四核 CPU，`-j4` 参数将任务分派到 4 个核心上执行，能显著地提升编译的效率。

编译过程大概持续两个小时，编译完成后看到这样一条信息：

```
Kernel: arch/arm/boot/zImage is ready
```

在 `arch/arm/boot/` 目录下，可以看到 `zImage` 文件就是刚刚编译生成的内核镜像。

安装内核

如果选择了可加载模块，要对选择的模块进行安装:

```
sudo make modules_install
```

备份原内核：

```
sudo cp /boot/$KERNEL.img /boot/$KERNEL-backup.img
```

将相关启动文件拷贝到 `/boot/` 目录中，覆盖原文件：

```
sudo cp arch/arm/boot/dts/*.dtb /boot/
sudo cp arch/arm/boot/dts/overlays/*.dtb* /boot/overlays/
sudo cp arch/arm/boot/dts/overlays/README /boot/overlays/
```

安装内核镜像，用新的内核镜像替换 `/boot/kernal7.img`

```
sudo scripts/mkknlimg arch/arm/boot/zImage /boot/$KERNEL.img
```

重新启动后，利用 `uname -a` 命令查看当前内核版本：

```
Linux raspberrypi 4.4.11-v7+ #1 SMP Fri May 27 16:26:52 UTC 2016 armv7l
GNU/Linux
```

添加系统调用

为了增加一个系统调用，我们在内核中添加一个函数，并选择一个系统调用号指向该函数。

添加内核函数

在 `arch/arn/kernel` 中新建一个 `sys_mysyscall.c` 文件，在运行后输出一条内核日志。

```
void sys_mysyscall(void) {
    printk("Hello mysyscall !!!\n");
}
```

修改系统调用表

系统调用表在 `arch/arm/kernel/calls.S` 中，由于 223 号系统调用没有被分配，所以我们在 223 号处添加一个调用函数：

```
/* 220 */       CALL(sys_madvise)
                CALL(ABI(sys_fcntl64, sys_oabi_fcntl64))
                CALL(sys_ni_syscall) /* TUX */
                CALL(sys_mysyscall) /* add mysyscall */
                CALL(sys_gettid)
```

添加宏定义

在 `include/uapi/asm-generic/unistd.h` 中为新添加的系统调用号 223 指定宏定义，指示该系统调用可用：

```
#define __NR_mysyscall 223
__SYSCALL(__NR_mysyscall, sys_mysyscall)
```

重新编译

在重新编译内核之前，要把我们新添加的源文件包含进去，修改 `arch/arm/kernel/` 目录下的 Makefile 文件，在 obj-y 后面添加 sys_mysyscall.o，即将该函数纳入系统的编译进程。

接下来重新编译内核，由于只做了很少的修改，所以本次编译时间很快。

编译完成后，重新载入内核启动。

系统调用

编写 C 代码，用两种方法做系统调用：

嵌入汇编代码，用 r0 传参数：

```
#include <stdio.h>
#define sys_call() {__asm__ __volatile__ ("swi 0x900000+223\n\t");} while(0)
int main(void) {
    sys_call();
    return 0;
}
```

用系统提供的 syscall() 函数：

```
#include <linux/unistd.h>
#include <sys/syscall.h>
int main() {
    syscall(223);
    return 0;
}
```

分别编译并运行两个程序，`dmesg` 命令查看内核输出：

```
[   94.542796] Hello mysyscall !!!
[  512.558956] Hello mysyscall !!!
```

内核模块

编写内核模块

在模块加载和卸载时能通过内核打印函数输出提示信息：

```
#include <linux/init.h>
#include <linux/module.h>

static int hello_init(void) {
    printk("my kernel module init\n");
    return 0;
}

static void hello_exit(void) {
    printk("my kernel module exit\n");
}

module_init(hello_init);
module_exit(hello_exit);
MODULE_LICENSE("GPL");
```

`module_init()` 和 `module_exit()` 这两个宏设置模块的生命周期回调函数。

Makefile

为了编译内核模块，我们需要编写一个 `Makefile`

```
TARGET = hello
KDIR = /lib/modules/$(shell uname -r)/build
PWD = $(shell pwd)
        obj-m += $(TARGET).o
default:
        make -C $(KDIR) M=$(PWD) modules
```

执行 `make` 命令就可以对模块进行编译，顺利的话会生成 `hello.ko` 文件。

插入模块

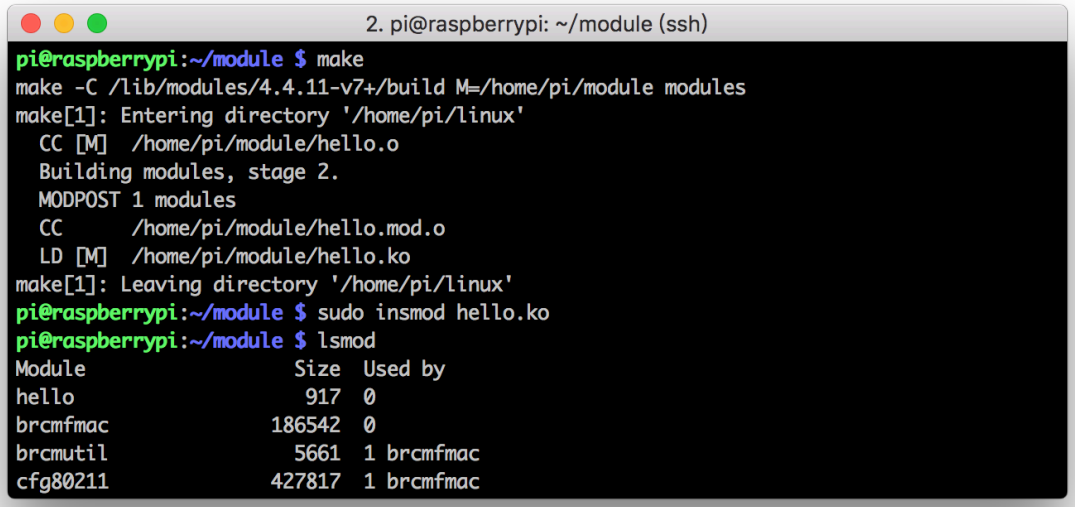
为了使编译好的内核模块能够在内核态运行，要执行插入操作：

```
sudo insmod hello.ko
```

内核模块在插入时会调用之前设置的回调函数，该函数的功能是在内核中输出一条语句。

查看模块

`lsmod` 命令可以查看当前已经插入的内核模块。



移除模块

内核模块在移除时同样会调用生命周期回调函数，移除命令：

```
sudo rmmod hello
```

查看内核输出

`dmesg` 命令显示内核的环形缓冲区信息，我们可以看到内核模块插入和移除时输出的语句。