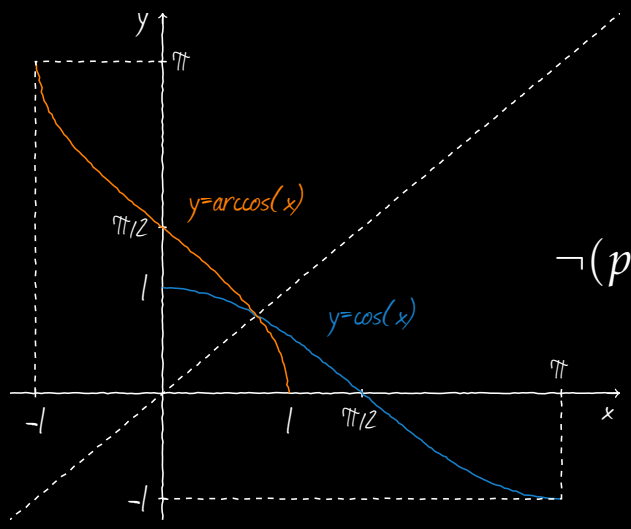
ZJUNIX

实验操作流程

浙江大学

2017.08.20

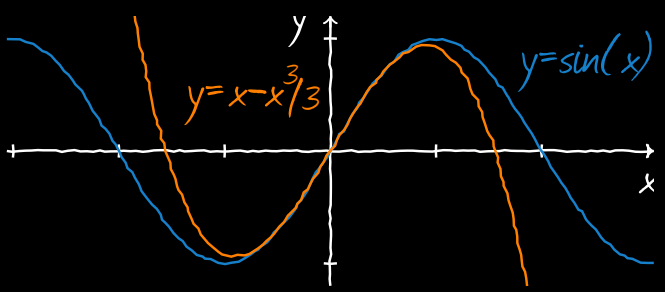
$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k}$$



$$\zeta_k = |a|^{1/n} \mathrm{e}^{i(\arg(a)+2k\pi)/n}$$

$$\mathrm{e}^{i\pi} + 1 = 0$$

$$\neg(p \vee q) \equiv (\neg p) \wedge (\neg q)$$



$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

目录

1.1 实验目的

1. 熟悉工程结构
2. 编译一个完整的内核

1.2 熟悉工程结构

工程结构说明请参考章 2

1.3 编译一个完整的内核

1.3.1 编译内核

1. 获取代码工程，具体请参考《实验 0》
2. 进入代码工程的 exp1 文件夹
3. 在 exp1 文件夹下执行 make 命令

1.3.2 准备 SD 卡

Windows 下

- 将 SD 卡格式化为 FAT32 分区，每簇大小 4096KB(如果内存卡太大无法指定 4096 KB 簇大小，可以格式化为多个分区，将第一个分区设为 4096 KB 簇大小即可)

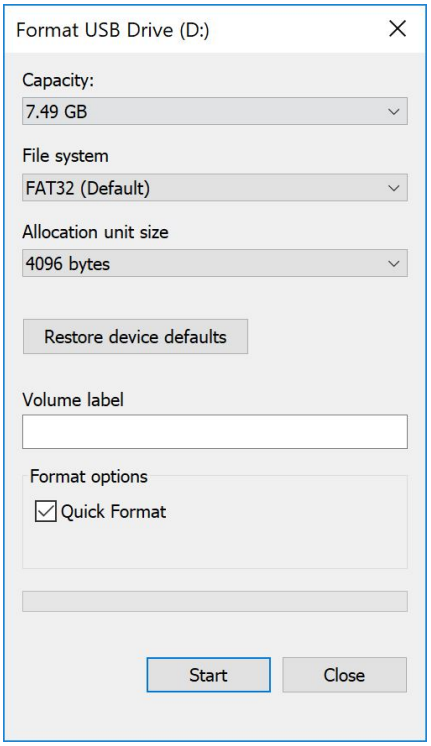


图 1.1: 格式化 SD 卡

Linux 下

代码 1.1: 格式化 sd 卡

```
# FAT32 Sector=512KB Cluster=512*8=4096KB
sudo mkfs.vfat -F 32 -S 512 -s 8 ... # device to format
```

1.3.3 复制内核

将编译得到的 kernel.bin 复制到 SD 卡根目录，弹出 SD 卡

1.3.4 启动操作系统

1. 将.bit 烧入 SWORD 板卡
2. 插入小小节 1.3.2制作好的 SD 卡
3. 使用 CPU Reset 按钮重启板卡，即可载入操作系统内核

2.1 工程结构说明

2.1.1 ZJUNIX 目录与源码构成

代码 2.1: 工程目录结构

```
ZJUNIX/
| arch # 架构相关代码
    | mips32 # 具体架构
        | ... # 架构相关代码实现
        | Makefile
    | Makefile
| config # 编译相关选项
    | debug.h         # 所有编译期debug宏的声明
    | flags.conf      # 各工具的功能选项，包括编译选项等
    | kernel.ld       # 内核镜像链接脚本
    | make.global     # Makefile全局命令
    | submake.global  # 子Makefile全局命令
    | tools.conf      # 工具链位置与名称定义
| include # 公有方法声明
    | driver   # 驱动相关声明
    | zjunix   # 内核相关声明
    | assert.h # 运行期断言
| kernel # 内核源码
    | ...    # 内核源码目录
    | init.c # 内核初始化代码
    | Makefile
| usr # 用户程序源码
    | ...
    | Makefile
| utils # 辅助功能库
    | ...
    | Makefile
| Makefile # 总Makefile
| Readme.md # 工程简介
| LICENSE # 开源协议
```

2.1.2 工程结构分析

这部分内容根据上文列出的目录顺序，分析各目录下的重要文件，对于 Makefile 的工作原理和具体使用方法会在后面的章节进行介绍

arch 目录

架构相关代码的声明与实现，根据编译时 ARCH 不同而选取不同目录加入编译，默认的 ARCH 变量为 mips32，此时进行编译，编译期会将 arch/mips32 下的源文件进行编译并加入链接。

这样做是为了更好地支持多平台，因为不同的硬件平台对于硬件相关代码 (如启动，异常等) 不具有可重用性，即不同硬件平台应当维护各自的硬件相关代码，在 arch 目录下的每一个子目录代表一个硬件体系架构，改体系架构的所有硬件相关代码都实现在这里。但是所有的硬件代码应当提供统一的抽象接口，供操作系统上层的逻辑进行调用。

但是操作系统上层的代码具有比较高的可重用性，所以一般不需要针对特定硬件架构编写代码，大部分情况下只需要调用 arch 提供的抽象的接口即可。

比如现在有两个硬件架构，分别是 armv7 和 mips32，他们都提供了中断机制，其具体实现并不相同，但是对外提供相同的接口 `void init_interrupts()`，这样一来，在 `init.c` 中只需要在正确的位置调用 `init_interrupts()` 方法即可，而不需要关心当前针对的硬件架构是什么，编译构建工具会自动找到合适的具体实现并调用。

config 目录

此目录包含与编译有关的多个选项，我们将其中与 Makefile 有关的文件的介绍推后，在这里主要介绍 debug.h 和 kernel.ld

debug.h 文件中包含内核编译期间所有的 DEBUG 宏，用于打开/关闭一些调试功能。比如如下代码包含了对于一个 DEBUG 宏的判断，来决定是否将调试功能编译到内核

代码 2.2: 源文件 DEBUG 用法

```
void func() {
    printf("Start of func.\n");
#ifdef FUNC_DEBUG
    printf("Debug of func.\n");
#endif // ! FUNC_DEBUG
    printf("End of func.\n");
```

此时如果在 debug.h 文件中编写如下代码，声明了 FUNC_DEBUG 宏，那么编译器在编译到 `func()` 函数时，会将 `printf("Debug of func.\n")` 编译到函数体当中

代码 2.3: debug.h 开启调试宏

```
// func: display debug info
#define FUNC_DEBUG
```

这样在运行时我们可以得到如下结果

代码 2.4: 运行结果

```
Start of func.
Debug of func.
End of func.
```

但如果将 debug.h 中的相关宏注释掉，如下代码所示

代码 2.5: 在 debug.h 注释相关宏

```
// func: display debug info
```

```
// #define FUNC_DEBUG
```

那么编译运行的结果应当是

代码 2.6: 注释后的运行结果

```
Start of func.
End of func.
```

这样，我们可以在编译前，通过注释/解注释一些功能的 DEBUG 宏，来关闭/开启对应功能模块的调试功能。这就像是在 debug.h 中放置了所有调试功能的开关，只需要注释/解注释相关宏，就可关闭/开启调试功能。

这样的功能对于内核开发调试非常有用，在调试阶段我们希望在一些功能模块中嵌入一些调试代码，但是又不希望在发布内核的时候手动删除这些调试代码。那么我们可以将调试代码放到 `#ifdef XXX_DEBUG` 和 `#endif` 之间，然后在 debug.h 中编写对应的宏的声明，即可进行调试，发布时只需将 debug.h 中的宏注释掉然后重新编译内核，这些调试功能就被消去了。

include 和 kernel 目录

在内核开发时，我们经常会遇到这样几个问题

1. 某个模块开发了许多功能，其中一些功能可供模块外部使用 (public)；但另一些只能供模块内部使用 (private)，如果被其他模块调用，则可能产生错误。但是由于编译的需要，这些功能的声明都声明在了对应的 `.h` 文件中，任何希望调用此模块的其他模块都会引用这个 `.h`，导致我们很难保证其他模块不会调用这些**仅供内部使用**的功能。

   - 这样的情况在多人协作开发的时候尤其常见，一位开发者很难完全理解其他开发者所开发的功能，也就难以区分哪些功能是供外部模块使用的，哪些是仅限内部使用的。这种情况下很容易导致功能的误用，给工程开发带来麻烦
   - 或许给变量和函数起一个合适的名字会解决这个问题？比如所有内部使用的函数都以 `private_` 开头，所有供外部调用的函数都以 `public_` 开头？这似乎也是一个解决方案，但是这并不能根绝错误调用 (因为编写代码的开发者并非时时刻刻都是可靠的，敲错几个字母导致函数误用还是会带来很大麻烦)
   - 或许可以使用面向对象的编程语言来做这件事情？比如使用 C++ 来编写这个工程？这也是一个解决方案，但是依然有许多情况下，我们无法使用面向对象的模型来构建我们的操作流程

2. 开发时引用的头文件如果分散在多个目录，没有一个清晰合理的组织结构，那么其他模块的编写时引用头文件是一个让人头大的事情，我们可能会看到大量的 `#include "../../dir1/subdir2/` 的引用。这样做虽然也没有问题，但是当我们移动了一个头文件的时候，会有相当多的头文件引用需要改，这是一件令人苦恼的事情。

针对以上几个问题，我们希望有这样的一种机制

1. 由编译器来保证模块 A 调用模块 B 的功能时，不会调用到模块 B 的 private 的功能
2. 将所有的头文件都合理地组织在一起，引用时就避免了相对路径引用的难题

所以我们提出这样一个组织形式：

代码 2.7: public 与 private 声明分开的目录组织

```
ZJUNIX/
    | include
        | zjunix
            | mod.h # mod模块的public功能声明
    | kernel
        | mod
            | mod.h # mod模块的private功能声明
            | mod.c # mod模块的public功能和private功能的实现
```

同时通过编译器的配置，让引用 include/zjunix/mod.h 的过程简化为 #include <zjunix/mod.h>
这样假设我们在开发新模块 newmod，那么在 newmod.c 中可以这样引用

代码 2.8: newmod.c

```
// 实际引用了include/zjunix/mod.h文件
// 其中仅包含了mod模块的public功能声明
// 当newmod调用mod模块的private功能时，编译器会报出warnning或error
#include <zjunix/mod.h>
```

注意，在这种情况下，每个模块可能有多达两个头文件需要维护，这两个文件的头文件保护需要合理编写，如果都使用了同一个头文件保护宏，那么会导致一些声明的丢失。比如参见如下代码

代码 2.9: public 文件头保护

```
// include/zjunix/mod.h
#ifndef _MOD_H
#define _MOD_H
...
#endif
```

代码 2.10: private 文件头保护

```
// kernel/mod/mod.h
#ifndef _MOD_H
#define _MOD_H
...
#endif
```

两者都使用了 _MOD_H 来保护头文件，在这种情况下，两个 mod.h 中会有一个无法被展开，即在编译时丢失了其中一个 mod.h 的所有声明

所以我们建议在声明 public 功能的头文件的保护中添加一个额外的宏，与 private 声明的头文件做出区分，例如假如一个 ZJUNIX 的前缀，如下

代码 2.11: 修改后的 public 文件头保护

```
// include/zjunix/mod.h
#ifndef _ZJUNIX_MOD_H
#define _ZJUNIX_MOD_H
...
#endif
```

这样就不容易产生编译时的错误了

小结：

- 在 include 目录下放置 public 功能的声明
- 在 kernel 目录中放置 private 功能的声明，以及对所有功能的实现代码
- 其他模块使用 `#include <zjunix/...h>` 而非 `#include <../../...h>` • 的形式来引用头文件，避免误用 private 功能

2.1.3 编译流程

内核的编译与普通应用程序的编译有一定的区别，在 ZJUNIX 的内核编译当中，基本流程如下

编译所有源文件 (.c/.s) 得到中间文件.o → 链接所有中间文件得到.elf 文件 → 从.elf 文件生成.bin 文件

最终得到的.bin 文件就是我们期望得到的 ZJUNIX 内核镜像，将其复制到启动分区并重启，启动时 Bootloader 会寻找这个.bin 并将其加载到内存指定位置，然后从内存的这个位置开始执行内核镜像中的代码即可完成系统启动

在编译过程中我们默认输出了.map 文件，这个文件可以帮助我们查找某个功能模块或者某个数据被映射到了内存的什么位置，在内核调试当中可以快速数据和代码的位置，帮助进行调试

同时由于编译生成了.bin 文件，在我们希望阅读汇编代码的情况下，需要对内核进行反汇编，通过 make disassembly 可以生成反汇编文件，在反汇编文件中可以查看某个内存地址具体放置了哪一条代码

2.2 Makefile 解析

为了简化开发流程，ZJUNIX 构建了一个使用方法相对简单的 Makefile，这一部分会分析 Makefile 的工作原理以及扩展方式。同时此工程使用的 Makefile 模板也开源到Github。

2.2.1 使用方法

Makefile 分为两种

- 一种是根 Makefile，位于根目录，用于处理递归编译，链接，反汇编等
- 另一种是子 Makefile，位于每个含有源文件的目录，用于指定该目录下的中间文件

根 Makefile

在根 Makefile 中，可以指定如下内容

名称	默认值	说明
DIRS	arch kernel utils usr	所有含有源文件的子目录
OBJS		当前目录下的中间文件
ARCH	mips32	体系结构版本
TARGET	kernel.bin	生成的系统镜像名
MAP	kernel.map	生成的映射文件名
ELF	kernel.elf	生成的 elf 文件名
DISASSEMBLY	kernel.txt	生成的反汇编文件名
ENTRANCE	exception	内核入口标签
ALL_OBJS		所有中间文件的列表

修改说明

- DIRS 根据实际情况修改，例如添加了一个含有源文件的子目录，则在 DIRS 中增加这个子目录名称，并在子目录中创建对应的子 Makefile
- OBJS 根据实际情况修改，如果当前根目录没有源文件，则留空；如果有源文件，则将其生成的中间文件列出
- TARGET 为生成的系统镜像名，**建议不修改**，因为在 Bootloader 中默认查找 kernel.bin 文件，修改文件名会导致无法找到系统镜像。如果修改了 Bootloader，则可以将 TARGET 修改为对应的文件名
- MAP,ELF,DISASSEMBLY 可以修改为任意需要的文件名，不会影响内核编译结果
- ENTRANCE 为内核入口标签，**建议不修改**，这个值对应了 start.s 汇编代码中的内核入口标签，如果修改了 start.s 的入口标签，则需要对应地修改这个值
- ALL_OBJS **故意留空，不可修改**，编译过程中会自动将其填写好并执行链接功能

子 Makefile

在每个子 Makefile 中，需要指定如下内容

名称	说明
DIRS	所有含有源文件的子目录
OBJS	当前目录下的中间文件
include $(SUB_MAKE_INCLUDE)	引入一些默认配置

修改说明

- DIRS 根据实际情况修改，例如添加了一个含有源文件的子目录，则在 DIRS 中增加这个子目录名称，并在子目录中创建对应的子 Makefile
- OBJS 根据实际情况修改，如果当前根目录没有源文件，则留空；如果有源文件，则将其生成的中间文件列出
- include $(SUB_MAKE_INCLUDE) 语句放在当前子 Makefile 末尾，用于引入一些默认配置和规则，不引入会导致无法编译的错误

全局规则

一些通用的规则放置在 config/make.global 和 config/submake.global，分别对应根 Makefile 中的全局规则以及子 Makefile 中的全局规则

在使用过程中，Makefile 引入这些文件中的规则，以避免每个 Makefile 都重复编写规则

工具路径

工具的路径指定都在 config/tools.conf 中完成

编译参数

编译参数的指定在 config/flags.conf 中完成