Magenta操作系统内核

教你研发操作系统

http://elastos.org

同济大学电子与信息工程学院计算机科学与技术系

2016/9

操作系统是有话语权的厂商对本生态系统做事规则实施掌控的一种技术手段。一个体系内掌控话语权的可能是芯片公司、整机厂商等，当这个体系不是超大，且发展速度不足够快，但足够强大得以自立为一个行当，他会排斥开放，于是市场上有了各类操作系统，但这些操作系统都不是所属公司的主业。

现在人们心目中的操作系统概念，是微软教育出来的，适用于PC及服务器，是因为硬件设备互相兼容度高，而不为某一家公司所控造成的。

找到下一个象手机这样的操作系统在业务中占主导地位的市场，很难。但针对半导体技术的迅猛发展，这样的市场随时可能出现，如AR/VR设备、集成工控设备等。

**目录**

[1. 开发环境搭建 1](#_Toc466983074)

[1.1 先试试（Getting Started） 1](#_Toc466983075)

[1.2 在硬件上玩Magenta 1](#_Toc466983076)

[1.2.1 Raspberry Pi 3 1](#_Toc466983077)

[1.2.2 PC x86平台 2](#_Toc466983078)

[1.3 源码树目录结构 2](#_Toc466983079)

[1.4 开源许可证 3](#_Toc466983080)

[2. 基础概念 4](#_Toc466983081)

[2.1 内核对象（Kernel Objects） 4](#_Toc466983082)

[2.1.1 内核对象生命周期 4](#_Toc466983083)

[2.1.2 内核对象实现 5](#_Toc466983084)

[2.1.3 内核对象安全 5](#_Toc466983085)

[2.2 句柄（Capabilities） 5](#_Toc466983086)

[2.2.1 Magenta句柄 6](#_Toc466983087)

[2.2.2 Magenta句柄垃圾回收 9](#_Toc466983088)

[2.2.3 句柄的权限 9](#_Toc466983089)

[2.3 虚拟地址空间（Virtual Address Spaces） 10](#_Toc466983090)

[2.4 运行代码：作业（Job），进程（Processe）与线程（Thread） 10](#_Toc466983091)

[2.5 线程（Threads） 11](#_Toc466983092)

[2.6 进程间通信（Inter-Process Communication，IPC） 11](#_Toc466983093)

[2.7 对象与信号 12](#_Toc466983094)

[2.8 其它IPC机制：事件、事件对和用户信号 13](#_Toc466983095)

[2.9 等待操作：Wait One、Wait Many和端口Port 13](#_Toc466983096)

[2.10 共享内存：虚拟内存对象（VMOs） 14](#_Toc466983097)

[2.11 快速用户区互斥（Futexes） 15](#_Toc466983098)

[2.12 设备驱动与中断（Device Drivers and IRQs） 15](#_Toc466983099)

[2.13 抢占（Preemption） 15](#_Toc466983100)

[3. 加电开始 16](#_Toc466983101)

[3.1 lk\_main() 16](#_Toc466983102)

[3.1.1 lk\_main 源码分析 17](#_Toc466983103)

[3.1.2 arch\_early\_init() 17](#_Toc466983104)

[3.1.3 platform\_early\_init() 18](#_Toc466983105)

[3.1.4 bootstrap2 18](#_Toc466983106)

[3.2 Boot程序中的trampoline(弹簧床) 19](#_Toc466983107)

[3.3 起动过程 20](#_Toc466983108)

[3.4 内核命令行参数 21](#_Toc466983109)

[3.4.1 Magenta内核命令行参数 21](#_Toc466983110)

[3.4.2 如何把这些命令行参数传给内核。 23](#_Toc466983111)

[3.5 系统启动平稳后的进程、线程、Job情况 23](#_Toc466983112)

[4. Magenta内核 25](#_Toc466983113)

[4.1 内核起动地址 25](#_Toc466983114)

[4.2 从内核回用户态 25](#_Toc466983115)

[4.3 内核对象 25](#_Toc466983116)

[4.3.1 Process 26](#_Toc466983117)

[4.3.2 Thread 26](#_Toc466983118)

[4.3.3 Event 27](#_Toc466983119)

[4.3.4 Channel 27](#_Toc466983120)

[4.3.5 Interrupt request 27](#_Toc466983121)

[4.3.6 VMObject 27](#_Toc466983122)

[4.3.7 Data pipe 27](#_Toc466983123)

[4.3.8 Job 28](#_Toc466983124)

[4.3.9 IOPort 28](#_Toc466983125)

[4.4 程序加载器Loader 28](#_Toc466983126)

[4.4.1 内核elf-loader 29](#_Toc466983127)

[4.4.2 应用程序加载器 30](#_Toc466983128)

[5. 系统功能调用 31](#_Toc466983129)

[5.1 svc中断向量 32](#_Toc466983130)

[5.2 内核栈svc stack 32](#_Toc466983131)

[5.3 系统调用在内核侧的实现 34](#_Toc466983132)

[5.4 系统调用在应用程序侧的实现 35](#_Toc466983133)

[5.4.1 系统调用定义 35](#_Toc466983134)

[5.4.2 用户态与内核态调用者 35](#_Toc466983135)

[5.4.3 动态库libmagenta.so 36](#_Toc466983136)

[5.4.4 应用程序系统调用头文件 38](#_Toc466983137)

[5.4.5 导出系统调用符号 38](#_Toc466983138)

[5.5 用户态与内核态间的参数传递 39](#_Toc466983139)

[5.6 向内核传递命令行 40](#_Toc466983140)

[6. 线程调度与时间管理 42](#_Toc466983141)

[6.1 当前线程 42](#_Toc466983142)

[6.2 调度器 42](#_Toc466983143)

[6.3 线程优先级管理 43](#_Toc466983144)

[6.4 context切换 44](#_Toc466983145)

[6.5 定时器 46](#_Toc466983146)

[6.6 线程抢先preempt的实现 46](#_Toc466983147)

[7. 内存管理 48](#_Toc466983148)

[7.1 是否使用虚拟内存VM/NOVM 48](#_Toc466983149)

[7.2 ARM MMU基本原理 48](#_Toc466983150)

[7.3 地址空间布局 49](#_Toc466983151)

[7.3.1 虚拟地址规划 49](#_Toc466983152)

[7.3.2 物理内存管理 50](#_Toc466983153)

[7.4 MMU页表组织 51](#_Toc466983154)

[7.5 物理内存管理 52](#_Toc466983155)

[7.6 虚拟地址管理 52](#_Toc466983156)

[7.7 虚实映射与页表切换 52](#_Toc466983157)

[8. 中断 55](#_Toc466983158)

[8.1 内核态中断编程 56](#_Toc466983159)

[8.2 用户态中断编程 57](#_Toc466983160)

[8.3 中断程序执行过程 57](#_Toc466983161)

[9. 文件系统 61](#_Toc466983162)

[9.1 虚拟文件系统MXIO 61](#_Toc466983163)

[9.2 内置文件系统 64](#_Toc466983164)

[9.3 MinFS 65](#_Toc466983165)

[10. 设备驱动与管理 67](#_Toc466983166)

[10.1 DDK 67](#_Toc466983167)

[10.2 硬件IO 67](#_Toc466983168)

[10.3 驱动程序 68](#_Toc466983169)

[10.4 设备管理 69](#_Toc466983170)

[10.4.1 设备节点 69](#_Toc466983171)

[10.5 设备管理应用devmgr 70](#_Toc466983172)

[10.6 虚拟文件系统vfs 70](#_Toc466983173)

[10.7 设备树DeviceTree 70](#_Toc466983174)

[11. 编程概念 72](#_Toc466983175)

[11.1 Dispatcher 72](#_Toc466983176)

[11.2 信号Signal 74](#_Toc466983177)

[11.3 内核对象唯一标识值（KOID） 75](#_Toc466983178)

[11.4 StateTracker/ StateObserver 76](#_Toc466983179)

[11.5 资源Resource 76](#_Toc466983180)

[11.6 异常处理ExceptionPort 77](#_Toc466983181)

[11.7 事件对（EventPair） 77](#_Toc466983182)

[11.8 状态信息status\_t 78](#_Toc466983183)

[11.8.1 Descriptions 78](#_Toc466983184)

[11.8.2 Categories 78](#_Toc466983185)

[12. 编程基础设施 82](#_Toc466983186)

[12.1 mxtl，c++模板 82](#_Toc466983187)

[12.1.1 智能指针 82](#_Toc466983188)

[12.1.2 数组Array 83](#_Toc466983189)

[12.1.3 数据类型支持type\_support 84](#_Toc466983190)

[12.1.4 资源获取即初始化RAII 84](#_Toc466983191)

[12.1.5 容器、链表等 85](#_Toc466983192)

[12.2 内核编程可用程序资源 86](#_Toc466983193)

[12.3 内核堆 87](#_Toc466983194)

[12.3.1 cmpctmalloc 87](#_Toc466983195)

[12.3.2 miniheap 88](#_Toc466983196)

[12.3.3 物理内存申请 88](#_Toc466983197)

[12.3.4 c++ new/delete支持及AllocChecker 88](#_Toc466983198)

[12.4 用户堆/c库 90](#_Toc466983199)

[12.5 互斥机制 90](#_Toc466983200)

[12.5.1 cond 94](#_Toc466983201)

[12.5.2 mutex 94](#_Toc466983202)

[12.5.3 semaphore 94](#_Toc466983203)

[12.5.4 spinlock 95](#_Toc466983204)

[12.5.5 Futexes 97](#_Toc466983205)

[12.6 看门狗（watchdog） 98](#_Toc466983206)

[12.7 调试debug机制 98](#_Toc466983207)

[12.7.1 LTRACEF 98](#_Toc466983208)

[12.7.2 DEBUG\_ASSERT 99](#_Toc466983209)

[12.7.3 DEBUGLOG 100](#_Toc466983210)

[12.7.4 系统控制台console 100](#_Toc466983211)

[12.7.5 The libunwind project 101](#_Toc466983212)

[12.7.6 libbacktrace 101](#_Toc466983213)

[12.8 调优profile机制 101](#_Toc466983214)

[12.8.1 THREAD\_STATS 101](#_Toc466983215)

[12.8.2 KTRACE 102](#_Toc466983216)

[12.8.3 KLOG（remove facilities） 102](#_Toc466983217)

[12.8.4 内核事件EVLOG（remove facilities） 103](#_Toc466983218)

[12.9 监控运行log机制 104](#_Toc466983219)

[12.9.1 LogDispatcher 104](#_Toc466983220)

[12.9.2 内核诊断信息diagnostics 104](#_Toc466983221)

[13. Magenta应用编程基础设施 105](#_Toc466983222)

[13.1 进程间通信（IPC） 105](#_Toc466983223)

[13.1.1 网络套接字（socket） 105](#_Toc466983224)

[13.1.2 管道（Pipes） 106](#_Toc466983225)

[14. 调试Magenta 107](#_Toc466983226)

[14.1 用QEMU+gdb调试Magenta 107](#_Toc466983227)

[14.1.1 通过QEMU跑起来Magenta。 107](#_Toc466983228)

[14.1.2 起调试器arm-none-eabi-gdb 107](#_Toc466983229)

[14.1.3 QEMU跑操作系统的第一条指令 109](#_Toc466983230)

[14.1.4 经验总结 110](#_Toc466983231)

[14.2 用 Python 拓展 GDB 110](#_Toc466983232)

[14.2.1 如何用Python 拓展 GDB 111](#_Toc466983233)

[14.2.2 Magenta中的python调试角本 111](#_Toc466983234)

[14.3 GCC程序调试信息 112](#_Toc466983235)

[15. Magenta系统功能调用 114](#_Toc466983236)

[15.1 句柄Handles 114](#_Toc466983237)

[15.1.1 handle\_close 114](#_Toc466983238)

[15.1.2 handle\_duplicate 115](#_Toc466983239)

[15.1.3 handle\_replace 116](#_Toc466983240)

[15.1.4 handle\_wait\_many 117](#_Toc466983241)

[15.1.5 handle\_wait\_one 120](#_Toc466983242)

[15.2 Objects 121](#_Toc466983243)

[15.2.1 object\_signal 121](#_Toc466983244)

[15.2.2 object\_signal\_peer 122](#_Toc466983245)

[15.2.3 object\_get\_info 122](#_Toc466983246)

[15.2.4 object\_get\_property 124](#_Toc466983247)

[15.2.5 object\_set\_property 125](#_Toc466983248)

[15.2.6 object\_bind\_exception\_port 125](#_Toc466983249)

[15.2.7 object\_get\_child 125](#_Toc466983250)

[15.3 作业（Job） 125](#_Toc466983251)

[15.3.1 job\_create 125](#_Toc466983252)

[15.4 任务（Task, Process, or Job） 125](#_Toc466983253)

[15.4.1 task\_resume 125](#_Toc466983254)

[15.4.2 task\_kill 126](#_Toc466983255)

[15.5 消息管道（Channel） 126](#_Toc466983256)

[15.5.1 channel\_create 126](#_Toc466983257)

[15.5.2 channel\_read 128](#_Toc466983258)

[15.5.3 channel\_write 129](#_Toc466983259)

[15.6 数据管道（Data Pipe） 130](#_Toc466983260)

[15.6.1 datapipe\_create 131](#_Toc466983261)

[15.6.2 datapipe\_read 132](#_Toc466983262)

[15.6.3 datapipe\_begin\_read 134](#_Toc466983263)

[15.6.4 datapipe\_end\_read 136](#_Toc466983264)

[15.6.5 datapipe\_write 137](#_Toc466983265)

[15.6.6 datapipe\_begin\_write 139](#_Toc466983266)

[15.6.7 datapipe\_end\_write 141](#_Toc466983267)

[15.7 端口（Port） 142](#_Toc466983268)

[15.7.1 port\_create 142](#_Toc466983269)

[15.7.2 port\_queue 143](#_Toc466983270)

[15.7.3 port\_wait 145](#_Toc466983271)

[15.7.4 port\_bind 146](#_Toc466983272)

[15.8 套接字（socket） 147](#_Toc466983273)

[15.8.1 socket\_create 148](#_Toc466983274)

[15.8.2 socket\_write 149](#_Toc466983275)

[15.8.3 socket\_read 150](#_Toc466983276)

[15.9 线程（Threads） 151](#_Toc466983277)

[15.9.1 thread\_arch\_prctl 151](#_Toc466983278)

[15.9.2 thread\_create 151](#_Toc466983279)

[15.9.3 thread\_exit 152](#_Toc466983280)

[15.9.4 thread\_start 153](#_Toc466983281)

[15.9.5 thread\_read\_state 154](#_Toc466983282)

[15.9.6 thread\_write\_state 154](#_Toc466983283)

[15.10 进程（Processes） 154](#_Toc466983284)

[15.10.1 process\_create 155](#_Toc466983285)

[15.10.2 process\_start 156](#_Toc466983286)

[15.10.3 process\_map\_vm 158](#_Toc466983287)

[15.10.4 process\_protect\_vm 159](#_Toc466983288)

[15.10.5 process\_unmap\_vm 160](#_Toc466983289)

[15.10.6 process\_read\_memory 161](#_Toc466983290)

[15.10.7 process\_write\_memory 162](#_Toc466983291)

[15.11 等待集（Wait Sets） 163](#_Toc466983292)

[15.11.1 waitset\_create 163](#_Toc466983293)

[15.11.2 waitset\_add 164](#_Toc466983294)

[15.11.3 waitset\_remove 165](#_Toc466983295)

[15.11.4 waitset\_wait 166](#_Toc466983296)

[15.12 Events and Event Pairs 168](#_Toc466983297)

[15.12.1 event\_create 168](#_Toc466983298)

[15.12.2 eventpair\_create 169](#_Toc466983299)

[15.13 中断Interrupt 171](#_Toc466983300)

[15.13.1 interrupt\_create 171](#_Toc466983301)

[15.13.2 interrupt\_complete 172](#_Toc466983302)

[15.14 硬件IO（Device Memory） 172](#_Toc466983303)

[15.14.1 mmap\_device\_memory 172](#_Toc466983304)

[15.14.2 alloc\_device\_memory 172](#_Toc466983305)

[15.15 快速用户区互斥futex 172](#_Toc466983306)

[15.15.1 futex\_wait 172](#_Toc466983307)

[15.15.2 futex\_wake 173](#_Toc466983308)

[15.15.3 futex\_requeue 174](#_Toc466983309)

[15.16 Virtual Memory Objects (VMOs) 175](#_Toc466983310)

[15.16.1 vmo\_create 175](#_Toc466983311)

[15.16.2 vmo\_read 175](#_Toc466983312)

[15.16.3 vmo\_write 176](#_Toc466983313)

[15.16.4 vmo\_get\_size 176](#_Toc466983314)

[15.16.5 vmo\_set\_size 176](#_Toc466983315)

[15.16.6 vmo\_op\_range 176](#_Toc466983316)

[15.17 Cryptographically Secure RNG 176](#_Toc466983317)

[15.17.1 cprng\_draw 176](#_Toc466983318)

[15.17.2 cprng\_add\_entropy 177](#_Toc466983319)

[15.18 时间Time 178](#_Toc466983320)

[15.18.1 nanosleep 178](#_Toc466983321)

[15.18.2 time\_get 179](#_Toc466983322)

[15.19 系统信息（Information） 180](#_Toc466983323)

[15.19.1 version\_get 180](#_Toc466983324)

[15.19.2 num\_cpus 181](#_Toc466983325)

[15.20 系统运行log 181](#_Toc466983326)

[15.20.1 log\_create 182](#_Toc466983327)

[15.20.2 log\_write 182](#_Toc466983328)

[15.20.3 log\_read 182](#_Toc466983329)

[16. 开发工具链 183](#_Toc466983330)

[16.1 Make文件 183](#_Toc466983331)

[16.2 源代码管理 184](#_Toc466983332)

[17. Magenta深入开发 185](#_Toc466983333)

[17.1 实时系统 185](#_Toc466983334)

[17.1.1 概念 185](#_Toc466983335)

[17.1.2 Linux在实时方面存在的不足 186](#_Toc466983336)

[17.1.3 操作系统实时性设计 186](#_Toc466983337)

[17.2 高精度定时器 187](#_Toc466983338)

[17.3 高可用系统 188](#_Toc466983339)

[17.3.1 概念 188](#_Toc466983340)

[17.3.2 软件容错技术 191](#_Toc466983341)

[17.3.3 借鉴Erlang 194](#_Toc466983342)

[17.4 安全 195](#_Toc466983343)

[17.4.1 故障安全属性 195](#_Toc466983344)

[17.4.2 固有故障安全原则 195](#_Toc466983345)

[17.5 思考题 196](#_Toc466983346)

[18. 问答Q&A 198](#_Toc466983347)

[18.1 程序技术问题 198](#_Toc466983348)

[18.2 计算机体系结构architecture相关 204](#_Toc466983349)

[18.2.1 ARM 205](#_Toc466983350)

[18.2.2 X86 206](#_Toc466983351)

[18.3 高阶逻辑在操作系统开发中的应用 207](#_Toc466983352)

[18.3.1 布尔可满足性问题 207](#_Toc466983353)

[18.4 操作系统领域顶级国际学术会议 208](#_Toc466983354)

[18.5 内核调试技术 209](#_Toc466983355)

[18.5.1 内核探测工具systemtap 209](#_Toc466983356)

[19. 结语 211](#_Toc466983357)

[20. 参考文献 212](#_Toc466983358)

# 开发环境搭建

## 先试试（Getting Started）

文档：<https://github.com/fuchsia-mirror/magenta/blob/master/docs/getting_started.md>

源码位置：

<https://github.com/fuchsia-mirror/magenta.git>

Travis-Ci持续集成：

<https://travis-ci.org/fuchsia-mirror>

Google内部Gerrit：

<https://fuchsia-review.googlesource.com/#/q/status:open>

## 在硬件上玩Magenta

### Raspberry Pi 3

In order to boot on Raspberry Pi3 you will need an sd card with the following:

config.txt file containing the following:

enable\_uart=1 framebuffer\_depth=16

bootcode.bin and start.elf files. Can be obtainied from here

build armstub8.bin from source that can be found here and copy to sd card

copy the magenta.bin file from you build to the sd card as kernel8.img

serial console is available on the rpi3 header.

Pin 6 - GND

Pin 8 - TXD (output from Pi)

Pin 10 - RXD (input to pi)

Baudrate = 115200

### PC x86平台

## 源码树目录结构

源码：$Magenta/magenta

* kernel
  + app
  + arch：arm、arm64、x86
  + dev：平台内置设备：gpio、timer、intel\_rng

Only bus devices (like pci) and a few very special devices (like null)

should be visible under /dev. Shuffle the others out of the way under

/dev/misc. Devices, in general, should be located via their class

under /dev/class/<classname>/...

* + include

├── arch

├── dev

├── hw

├── kernel

│   └── vm

├── lib

├── lk

├── platform

└── sys

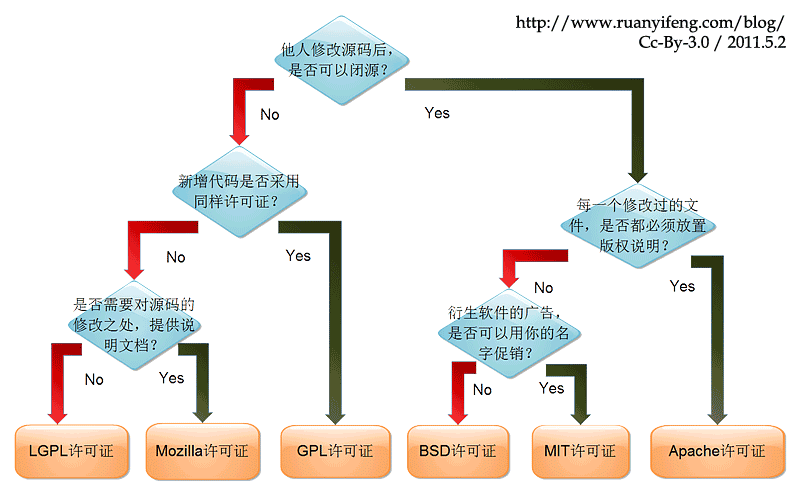
* + kernel：小内核，处理物理内存、线程、定时器，提供seamphore、mutex等。
  + lib：库
  + make：build makefile文件
  + platform：平台：bcm28xx、pc、qemu-virt等
  + project：针对目标板的makefile文件
  + target：目标板
  + top：lk\_init\_level、lk\_main，第0个进程init0程序入口。

总的makefile，包括：lk\_inc.mk、engine.mk

* prebuilt预编译内容
* scripts工作角本
* system
* third\_party第三方库

## 开源许可证

参考：<http://www.ruanyifeng.com/blog/2011/05/how_to_choose_free_software_licenses.html>



# Use of this source code is governed by a MIT-style

# license that can be found in the LICENSE file or at

# https://opensource.org/licenses/MIT

# 基础概念

## 内核对象（Kernel Objects）

操作系统内核提供的功能，很多并不是内核本身提供的，而是跑在内核态的程序提供的，象Linux这样的操作系统的内核，这类内核程序可能是驱动，或以伪驱动形式存在的内核程序。

Magenta中的内核对象是内核向用户程序提供的一组功能集，应用程序眼中的这些内核对象，与内核眼中的这些对象，在接口上是一样的，就是说，服务态程序中看到的这些对象，并没被单独包装过，只是需要通过句柄才能标识这些对象。

Magenta是一个基于面向对象思想设计的内核，用户态的代码几乎都是只通过映射到内核对象的句柄与内核资源打交道。

内核管理着许许多多类型的实体（entity），这些都是对象，实现代码中，它们都是从c++类 Dispatcher (a name which will likely change in the future -- think of it as meaning "Kernel Object")继承出来的c++对象。

Magenta内核对象有如下几种：

* Process
* Thread
* Event
* Message pipe
* Interrupt request
* VMObject
* Data pipe
* Job
* IOPort

### 内核对象生命周期

通过引用计数（ref-counted）方式管理内核对象

Kernel objects are ref-counted. Most kernel objects are born during a syscall and are held alive at refcount = 1 by the handle which binds the handle value given as the output of the syscall. The handle object is held alive as long it is attached to a handle table. Handles are detached from the handle table closing them (for example via sys\_close()) which decrements the refcount of the kernel object. Usually, when the last handle is closed the kernel object refcount will reach 0 which causes the destructor to be run.

The refcount increases both when new handles (referring to the object) are created and when a direct pointer reference (by some kernel code) is acquired; therefore a kernel object lifetime might be longer than the lifetime of the process that created it.

### 内核对象实现

内核对象都继承自分发器Dispatcher，每个具体的对象在基类的基础上，扩充实现自己的功能。

A kernel object is implemented as a C++ class that derives from Dispatcher and that overrides the methods it implements. Thus, for example, the code of the Thread object is found in ThreadDispatcher. There is plenty of code that only cares about kernel objects in the generic sense, in that case the name you'll see is mxtl::RefPtr<Dispatcher>.

### 内核对象安全

Kernel Object security

In principle, kernel objects do not have an intrinsic notion of security and do not do authorization checks; security rights are held by each handle. A single process can have two different handles to the same object with different rights.

## 句柄（Capabilities）

Magenta has a capability-based security model.

Capability概念上类似于句柄（Handle），在操作系统这样的系统软件设计中，有一个可靠性的设计原则，就是下层软件不信任上层软件，而上层软件要无条件信任下层软件。那么，下层软件如何把自己的实现细节隐藏，不让上层软件通过“简单”方法就可以直接操作下层软件的资源呢，句柄是一个常用的设计。

一个句柄指向一个内核对象，内核对象也只能通过与其绑定的句柄来操作它，无法直接访问。大多数的内核对象都有方法，用来操作这个对象的数据结构，就象是c++中的封装中的protect数据结构，方法自身可以访问数据结构中的数据，但是，使用这个对象的其它程序，只能通过API与这个对象打交道。

举例来说，用户来以通过调用一个原始内存untyped-memory 句柄的invoke方法，用来在这个原始内存上创建内核对象。

有些内核对象只有方法，但自身没有什么状态（state）信息，这类对象的目的是为了让应用程序与内核打交道，这类对象没有自身的存储实例。

下面泛义地讨论一下什么是句柄？为什么会有句柄？

句柄是一个概念，Handler是一个句柄，Magenta中的Capability也是一个句柄，打开文件，fopen得到的那个FILE \*，open得到的那个int都是句柄。

从广义上，能够从一个数值（或数据结构）拎起一大堆数据的东西都可以叫做句柄。句柄的英文是“Handle”，本义就是“柄”，只是在计算机科学中，被特别地翻译成“句柄”，其实还是个“柄”。从一个小东西拎起一大堆东西，这难道不像是个“柄”吗？

然后，指针其实也是一种“句柄”，只是由于指针同时拥有更特殊的含义——实实在在地对应内存里的一个地址——所以，通常不把指针说成是“句柄”。但指针也有着能从一个32位的值（在32位硬件平台上）引用到一大堆数据的作用，这不是句柄又是什么？

一个操作系统中，可能有许多内核对象（这里的对象不完全等价于“面向对象程序设计”一词中的“对象”，虽然实质上还真差不多），比如打开的文件，创建的线程，程序的窗口，等等。这些重要的对象肯定不是4个字节或者8个字节足以完全描述的，他们拥有大量的属性。为了保存这样一个“对象”的状态，往往需要上百甚至上千字节的内存空间，那么怎么在程序间或程序内部的子过程（函数）之间传递这些数据呢？拖着这成百上千的字节拷贝来拷贝去吗？显然会浪费效率。那么怎么办？当然传递这些对象的首地址是一个办法，但这至少有两个缺点：

* 暴露了内核对象本身，使得程序（而不是操作系统内核）也可以任意地修改对象的内部状态（首地址都知道了，还有什么不能改的？），这显然是操作系统内核所不允许的。上面不是已经介绍过操作系统的可靠性设计原则了吗？
* 操作系统有定期整理内存的责任，如果一些内存整理过一次后，对象被搬走了怎么办？

Magenta提供了基于句柄的访问控制（Capability-based Access Control）机制，访问控制掌控着所有对内核的访问。用户态程序，必须有足够的权限，通过访问它自己拥有的句柄来发出请求（invoke）。通过这个机制，系统使得软件构件间得以隔离，并且允许审计、控制构件间的通信，手段就是有选择地授权特定的通信句柄。这个机制使得软件构件在更高层面上得以检查，因为只有被明确授权过的操作才能以被调用。

一个句柄是一个不可伪造的令牌，它唯一地标识了一个内核对象（如线程控制块TCB），句柄的管理机构中，有这个句柄的访问权限信息，这些信息中包含哪些方法可以被访问。概念上，句柄存储于应用程序的特定存储区（capability space），这个存储区的存储单元你为slot，一个slot可以包含一个句柄，也可以啥也不包含。应用程序通过句柄（即存有这个句柄的那个slot的地址）来请求内核中的服务。这意味着Magenta的capability模型是一个被内核管理的隔离的（segregated）或分区的（partitioned) capability模型，也就是说这些句柄的内存虽然是存储在用户态中的，但是被内核管理。

句柄上的权限也可以被撤销，递归地撤销所有从原句柄中继承来的权限。句柄的繁殖控制模型被称为take-grant-based模型。

### Magenta句柄

句柄（Handle）由内核创建，供用户态的应用程序使用，应用程序通过句柄引用内核对象。

多个句柄可以对应一个内核对象，但是一个句柄只能对应一个对象。当一个句柄对应一个内核对象时，我们称它为在运输途中（in-transit）。

In user-mode a handle is simply a specific number returned by some syscall. Only handles that are not in-transit are visible to user-mode.

在用户程序中，句柄是一个整数，句柄是与进程相关的数据结构，在这个进程中的句柄，到另外一个进程中是无效的，同样的句柄值（就是那个整数值）完全可能对应完全不同的内核对象。

内核中，句柄是一个至少包含了下列成员的c++对象：

* A reference to a kernel object
* The rights to the kernel object
* The process it is bound to (or if its bound to kernel)

操作权限用来标识这个对象允许的操作，同一个内核对象，通过多个句柄引用它，这多个句柄，完全可能自己有自己的权限，也就是虽然指向的是同一个对象，但你通过这个句柄能实现的操作，通过另外一个句柄却不可以。

Objects may have multiple Handles (in one or more Processes) that refer to them.

For almost all Objects, when the last open Handle that refers to an Object is closed, the Object is either destroyed, or put into a final state that may not be undone.

Handles may be moved from one Process to another by writing them into a Channel (using mx\_channel\_write), or by using mx\_process\_start to pass a Handle as the argument of the first thread in a new Process.

The actions which may be taken on a Handle or the Object it refers to are governed by the Rights associated with that Handle. Two Handles that refer to the same Object may have different Rights.

The mx\_handle\_duplicate and mx\_handle\_replace system calls may be used to obtain additional Handles referring to the same Object as the Handle passed in, optionally with reduced Rights. The mx\_handle\_close() system call closes a Handle, releasing the Object it refers to, if that Handle is the last one for that Object.

#### 句柄的使用

很多系统调用创建出句柄并且返回结果为句柄，如：

* mx\_event\_create
* mx\_process\_create
* mx\_thread\_create

These calls create both the kernel object and the first handle pointing to it. The handle is bound to the process that issued the syscall and the rights are the default rights for that type of kernel object.

#### 句柄的复制

只有一个系统调用实现句柄复制，它创建一个指向同一内核对象的句柄，并且与同一进程绑定：

mx\_handle\_duplicate

#### 句柄的替换

There is one syscall that creates an equivalent handle (possibly with fewer rights), invalidating the original handle:

mx\_handle\_replace

#### 句柄的销毁

There is one syscall that just destroys a handle:

mx\_handle\_close

#### 句柄的写

There is only one syscall that takes a handle bound to calling process and binds it into kernel (puts the handle in-transit):

mx\_message\_write

#### 句柄的写

There is only one syscall that takes an in-transit handle and binds it to the calling process:

mx\_message\_read

The pair of 'message' syscalls above are used to effectively transfer a handle from one process to another. The gist is that it is possible to connect two processes with a 'message pipe'. To transfer a handle the source process calls mx\_message\_write and then the destination process calls mx\_message\_read on the same pipe.

Finally, there is a single syscall that gives a new process its boostrapping handle, that is, the handle that it can use to request other handles:

mx\_process\_start

It is natural that the bootstrapping handle points to one end of a message pipe.

### Magenta句柄垃圾回收

当一个内核对象是有效的，则指向它的句柄都是有效的，因为内核对象是通过引用计数管理内存的，每个句柄持有内核对象的的一个引用计数。

The opposite does not hold. When a handle is destroyed it does not mean its object is destroyed. There could be other handles pointing to the object or the kernel itself could be holding a reference to the kernel object.

When there is but one handle referencing a kernel object and the handle is being closed, the kernel object is either destroyed then or the kernel marks the object for garbage collection; the object will be destroyed when the current set of operations on it are completed.

特殊情况

When a handle is in-transit and the message pipe it was written to is destroyed, the handle is closed.

Debugging sessions (and debuggers) might have special syscalls to get access to handles.

### 句柄的权限

Rights are associated with handles and convey privileges to perform actions on either the associated handle or the object associated with the handle.

|  |  |
| --- | --- |
| **Right** | **Conferred Privileges** |
| **MX\_RIGHT\_DUPLICATE** | Allows handle duplication via [mx\_handle\_duplicate](https://github.com/fuchsia-mirror/magenta/blob/master/docs/syscalls/handle_duplicate.md) |
| **MX\_RIGHT\_TRANSFER** | Allows handle transfer via [mx\_message\_write](https://github.com/fuchsia-mirror/magenta/blob/master/docs/syscalls/message_write.md) |
| **MX\_RIGHT\_READ** | Allows inspection of object state |
|  | Allows reading of data from containers (pipes, VM objects, etc) |
| **MX\_RIGHT\_WRITE** | Allows modification of object state |
|  | Allows writing of data to containers (pipes, VM objects, etc) |
| **MX\_RIGHT\_EXECUTE** |  |
| **MX\_RIGHT\_DEBUG** | Placeholder for debugger use, pending audit of all rights usage |

## 虚拟地址空间（Virtual Address Spaces）

传统上，象Linux这样的操作系统，虚拟地址空间是针对进程（Process）来说的，每个进程工作地独立的地址空间中，进程为其中的计算提供隔离的计算容器。Java虚拟机中的class-loader也提供了一个计算的隔离能力，Linux的cgroup也提供了一种计算隔离能力。因为不同的计算隔离能力，产生了很多的内存管理措施。

Magenta内核没有虚拟地址空间管理的能力，它也没有进程这样的概念，换句话说，你是自己想玩Windows CE slot那样的进程，还是Linux那样的完整的概念的进程，那是在Magenta上搭建的计算模型支持层的事。Magenta+Elastos中的Elastos就是样的一个计算模型支持层。

Windows CE只能管理512MB的物理内存和4GB大小的虚拟地址空间。在Windows CE 4GB的虚拟地址空间中，从0x00000000到Ox41FFFFFF由所有应用程序使用。这块地址空间分成33个槽(Slot)，每个槽占有32MB的地址空间。槽0由当前占有CPU的进程使用。槽1由XIP DLL使用。其它槽用于进程使用，每个进程占用一个槽，进程彼此不能够随意访问。因为槽0只有32MB，因此每个进程运行时只有32MB的虚拟地址空间。

Magenta没有对虚拟地址空间的定义，虚拟地址与物理地址的影射（map）是被用户态的程序自己管理的。例如，一个page fault中断触发的异常是被送到定义在用户态空间的处理线程用户态页机制（user-level pager），这个页处理机制决定这个缺页中断如何处理，你是映射到一块物理内存，还是做什么处理，Magenta是不管的。

用户态的桢（User-level frame）是唯一可以在用户态被直接访问的内存，它们通过原始内存（untyped memory）创建而来。在这些帧被访问之前，它们需要被映射到虚拟地址空间，地址空间通过创建、链接页表对象得到，地个地址空间是一个指向最上层页表的句柄可能（a capability to its top-level page table）标识的。

虚拟地址空间对象 Virtual Address Space Objects

用来创建虚拟地址空间（或VSpace），这些虚拟地址空间可以给一个线程或多个线程使用。这些对象管理着物理的存储设备，例如，页字典（page directory）管理着页表（page directory），页表就是让你的虚拟地址与物理地址对应起来的那个物理器件MMU，内核还包括ASID Pool和ASID Control对象，用来跟踪地址空间状态。

## 运行代码：作业（Job），进程（Processe）与线程（Thread）

线程（thread）是Magenta能够进行运算调度的最小单位。它被包含在进程（process）之中，是进程中的实际运作单位。一条线程指的是进程中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务。

线程（Thread）代表一个执行单位，是一些计算资源的集合，这些计算资源包括CPU寄存器、栈等等。

同一进程中的多条线程将共享该进程中的全部系统资源，如虚拟地址空间，文件描述符和信号处理等等。但同一进程中的多个线程有各自的调用栈（call stack），自己的寄存器环境（register context），自己的线程本地存储（thread-local storage）。

一个进程可以有很多线程，每条线程并行执行不同的任务。

进程（Process）归作业（Job）所有，Jobs定义了各种资源的限制，Job间有父子关系，Magenta的Job有一个共同的根，根Job是在系统起动时，由内核创建，传递给userboot（第一个用户态进程，the first userspace Process）

任何一个新的Job，必须继承自已有Job，这意味着，一个线程想创建一个Job，必须有一个现成的Job句柄。想想Magenta的Job有一个共同的根，你不从一个现在Job入手，如何加入这个Job组成的Job树？

参见：

* process\_create
* process\_map\_vm
* process\_protect\_vm
* process\_start
* process\_unmap\_vm
* thread\_create
* thread\_start.

## 线程（Threads）

线程（Thread）是被内核调度的执行单位。线程被创建后，需要为其指定内存地址空间，它在运行的过程中，可以改变这个指定的内存地址空间。一段内存地址空间，可以被0个、一个或多个线程指定，这些线程没有主线程、原始线程之类的概念。

Magenta中，一个象Linux这样的传统的操作系统进程的概念被分解为：地址空间、虚实映射机制、程序及原始数据（装到内存里的那些用户自己写的逻辑）、执行线索等等。

Magenta的每个线程对应内存中的一个线程控制块（TCB，thread control block）对象，没有内核线程这样的说法，也就是说，没有一个线程是只工作在内核态中的，除非它是一个哑（idle）线程，所有系统调用都是工作在调用它的那个线程的用户态空间中的。

## 进程间通信（Inter-Process Communication，IPC）

IPC发生在端点（endpoint）间。一个端点是一个独立的内核对象，拥有端点的线程通过调用端点的句柄（endpoint capability）发送或接收IPC消息。当一个端点被调用（发送消息Send或接收消息Receive），IPC消息传输立即发生在这两个线程（发送者与接收者）之间。IPC只能发生在处理同一个端点的两个线程之间。

IPC在语义上并不要求发生请求与应答的双方在一个地址空间或不在一个地址空间。

IPC的端点有同步与异步的说法，当访问同步端点，当发出请求或接收请求，线程将被阻塞，直到消息发出（此时接收者需要要接收），或有消息传过来（当然是请求者把消息传过来，它发了请求）。

消息（Message）最大480字节（120个消息字，每个消息字32比特）。

发送一个异步消息从来不被阻塞，异步消息只能是一个消息字，即32比特，4个字节。这个消息被存储在这个端点，端点上不能存储多个消息，如果多个消息被发送，这些消息将被通过位或（bitwase OR operation）运算为新的异步消息。

#### Message Passing: Sockets and Channels

Both Sockets and Channels are IPC Objects which are bi-directional and two-ended. Creating a Socket or a Channel will return two Handles, one referring to each endpoint of the Object.

Sockets are stream-oriented and data may be written into or read out of them in units of one or more bytes. Short writes (if the Socket's buffers are full) and short reads (if more data is requested than in the buffers) are possible.

Channels are datagram-oriented and have a maximum message size of 64K (subject to change, likely to be smaller) and may also have up to 1024 Handles attached to a message (also subject to change, also likely to be smaller). They do not support short reads or writes -- either a message fits or it does not.

When Handles are written into a Channel, they are removed from the sending Process. When a message with Handles is read from a Channel, the Handles are added to the receiving Process. Between these two events, the Handles continue to exist (ensuring the Objects they refer to continue to exist), unless the end of the Channel which they have been written towards is closed -- at which point messages in flight to that endpoint are discarded and any Handles they contained are closed.

See: channel\_create, channel\_read, and channel\_write.

## 对象与信号

在计算机科学中，信号（Signal）是Unix、类Unix以及其他POSIX兼容的操作系统中进程间通讯的一种有限制的方式。它是一种异步的通知机制，用来提醒进程一个事件已经发生。

Linux系统中当一个信号发送给一个进程，操作系统中断了进程正常的控制流程，此时，任何非原子操作都将被中断。如果进程定义了信号的处理函数，那么它将被执行，否则就执行默认的处理函数。

Magenta的信号是依附于对象的，每个对象（Object）有多达32个信号（具体定义参考： mx\_signals\_t数据类型 和MX\_SIGNAL宏) ，用来表达这个对象当前状态的一些信息。这些状态信息如：Channel和Socket中的READABLE或WRITABLE，进程或线程中的TERMINATED。

一个线程可以同时阻塞等待在多个对象上的信号。

## 其它IPC机制：事件、事件对和用户信号

事件（Event）、事件对（Event Pair）和用户信号（User Signal）。

事件（Event）是最简单的对象，它没有其它状态，只是用来传递信号（Signal）。

事件对（Event Pair）是由两个事件（Event）组成的事件对，用来互相通报信号。一个应用场景是：当一个事件离开了（所有与它相关的句柄被关闭了），一个PEER\_CLOSED信号被发送到另一端。试想如果没有这样的机制，你咋让操作系统通知你某个对象被析构了？

任何一个对象上都有8个用户信号（从MX\_USER\_SIGNAL\_0 到MX\_USER\_SIGNAL\_7）被设为活跃（active）或不活跃（inactive），通过系统调用mx\_object\_signal()来完成这个工作。

参见：

* event\_create
* eventpair\_create

## 等待操作：Wait One、Wait Many和端口Port

#### 端口（Port）

端口是指接口电路中的一些寄存器，这些寄存器分别用来存放数据信息、控制信息和状态信息，相应的端口分别称为数据端口、控制端口和状态端口。

电脑运行的系统程序，其实就像一个闭合的圆圈，但是电脑是为人服务的，他需要接受一些指令，并且要按照指令调整系统功能来工作，于是系统程序设计者，就把这个圆圈截成好多段，这些线段接口就叫端口（通俗讲是断口，就是中断），系统运行到这些端口时，一看端口是否打开或关闭，如果关闭，就是绳子接通了，系统往下运行，如果端口是打开的，系统就得到命令，有外部数据输入，接受外部数据并执行。

#### Wait One、Wait Many和端口Port

一个线程，可以通过mx\_wait\_one()等待一个信号变活跃（active），也可以通过mx\_wait\_many()等待一些句柄上的信号，这两种等待都有超时机制。

如果一个线程要同时等待很多的句柄上的信号，更有效的办法是用Port。Port也是一个对象，其它对象可以绑定到Port对象上，当这些绑定到Port上的对象产生了信号，即某个信号变Active了，Port接收到一个包，这个包里包含挂起的信号的信息。

参见：

* port\_create
* port\_queue
* port\_wait
* port\_bind.

源码：$Magenta/magenta/kernel/lib/magenta/include/magenta/port\_dispatcher.h

// Port job is to deliver packets to threads waiting in Wait(). There

// are two types of packets:

//

// 1- Manually posted via Queue(), they are allocated in the

// heap by the caller and freed in the syscall layer at the bottom

// of mx\_port\_wait(). These Packets are of type IOP\_Packet and only

// live in the |packets| list.

//

// 2- Posted by bound dispatchers via Signal(), they are allocated once

// during their first Signal() call and only freed when the IO port

// reaches 'zero handles' state. These Packets are of type IOP\_Signal

// and bounce between the |packets\_| list and the |at\_zero\_| list

// depending on the |count| value:

//

// +----------+

// | |

// v |

// dispatcher-->observer->Signal()-->packets\_-->Wait()-->port\_wait()

// | ^ |

// | | |

// +----->at\_zero\_ <----+

//

## 共享内存：虚拟内存对象（VMOs）

虚拟内存对象(（Virtual Memory Objects，VMO）代表一个物理内存的集合，或者可能的页（pages），这些页将按需、延后创建（lazily, on-demand）。

它们可以通过mx\_process\_map\_vm()和mx\_process\_unmap\_vm()与进程地址空间建立映射关系或解除映射关系。内存操作权限可以通过mx\_process\_protect\_vm调整。

VMOs may also be read from and written to directly with vmo\_read and vmo\_write. Thus the cost of mapping them into an address space may be avoided for one-shot operations like "create a VMO, write a dataset into it, and hand it to another Process to use."

## 快速用户区互斥（Futexes）

Futexes are kernel primitives used with userspace atomic operations to implement efficient synchronization primitives -- for example, Mutexes which only need to make a syscall in the contended case. Usually they are only of interest to implementers of standard libraries. Magenta's libc and libc++ provide C11, C++, and pthread APIs for mutexes, condition variables, etc, implemented in terms of Futexes.

参见：

* futex\_wait
* futex\_wake
* futex\_requeue.

## 设备驱动与中断（Device Drivers and IRQs）

微内核操作系统的驱动通常不在内核中实现，为了让用户态中的程序可以访问设备寄存器，内核向用户态程序提供了设备桢（device frame），这些设备桢覆盖了设备的寄存器端口地址空间（memory-mapped registers）。

中断被异步递送到侦听这个中断的驱动程序端点。Magenta内核允许设置哪些端点侦听哪些中断，这些端点也被用来设置中断的使能与告知（enable、disable、acknowledge）。

## 抢占（Preemption）

在处理一个事件时，Magenta不能直接被打断，在内核执行时，不允许发生中断。为了防止高优先级的中心被延迟，Magenta在需要长时间操作的事务（long-running operation）中，设置是抢占点（preemption point）。在内核执行到抢占点时，将检查被挂起的中断，如果检查到有挂起的中断，触发一个抢点异常（preemption exception），这个异常被传给内核入口函数（the main kernel entry function），在那里，中断被触发，然后我们再次进入内核处理这个中断。处理完这个中断后，内核返回到用户态，原来的系统调用被重新执行，然后这个长事务被继续执行。

# 加电开始

对于ARM系统：系统复位或开机时马上进入到SVC模式，即管理模式（Supervisor，SVC）。

万物之始。加电后CPU执行的第一条外部指令（在CPU执行FLASH中的指令前，SOC内部可能有一些初始化程序的）。

源码：$Magenta/magenta/kernel/arch/arm/arm/start.S

.section ".text.boot"

.globl \_start

\_start:

b platform\_reset

b arm\_undefined

b arm\_syscall

b arm\_prefetch\_abort

b arm\_data\_abort

b arm\_reserved

b arm\_irq

b arm\_fiq

#if WITH\_SMP

b arm\_reset

#endif

.weak platform\_reset

platform\_reset:

/\* Fall through for the weak symbol \*/

## lk\_main()

lk\_main 主要做两件事：

1、本身 lk 这个系统模块的初始化；

2、boot 的启动初始化动作。

### lk\_main 源码分析

源码：Magenta/magenta/kernel/top/main.c

/\* called from arch code \*/

void lk\_main(ulong arg0, ulong arg1, ulong arg2, ulong arg3)

void lk\_main（）

{

1. 初始化进程（lk 中的简单进程）相关结构体。

thread\_init\_early();

1. 做一些如 关闭 cache，使能 mmu 的 arm 相关工作。

arch\_early\_init();

1. 相关平台的早期初始化

platform\_early\_init();

1. 现在就一个函数跳转，初始化UART（板子相关）

target\_early\_init();

1. 构造函数相关初始化

call\_constructors();

1. lk系统相关的堆栈初始化

heap\_init();

}

以上与boot启动初始化相关函数是arch\_early\_init、platform\_early\_init、bootstrap2。

### arch\_early\_init()

体系架构相关的初始化我们一般用的 ARM 体系

1. 关闭cache

arch\_disable\_cache(UCACHE);

1. 设置向量基地址（中断相关）

set\_vector\_base(MEMBASE);

1. 初始化MMU

arm\_mmu\_init();

1. 初始化MMU映射\_\_平台相关

platform\_init\_mmu\_mappings();

1. 开启cache

arch\_enable\_cache(UCACHE)

1. 使能 cp10 和 cp11

\_\_asm\_\_ volatile("mrc p15, 0, %0, c1, c0, 2" : "=r" (val));

val |= (3<<22)|(3<<20);

\_\_asm\_\_ volatile("mcr p15, 0, %0, c1, c0, 2" :: "r" (val));

1. 设置使能 fpexc 位 （中断相关）

\_\_asm\_\_ volatile("mrc p10, 7, %0, c8, c0, 0" : "=r" (val));

val |= (1<<30);

\_\_asm\_\_ volatile("mcr p10, 7, %0, c8, c0, 0" :: "r" (val));

1. 使能循环计数寄存器

\_\_asm\_\_ volatile("mrc p15, 0, %0, c9, c12, 0" : "=r" (en));

en &= ~(1<<3); /\*循环计算每个周期\*/

en |= 1;

\_\_asm\_\_ volatile("mcr p15, 0, %0, c9, c12, 0" :: "r" (en));

1. 使能循环计数器

en = (1<<31);

\_\_asm\_\_ volatile("mcr p15, 0, %0, c9, c12, 1" :: "r" (en));

### platform\_early\_init()

平台相关初始化不同平台不同的初始化。

1. 初始化中断

platform\_init\_interrupts();

1. 初始化定时器

platform\_init\_timer();

### bootstrap2

bootstrap2 在lk\_main的末尾以线程方式开启。主要分三步：platform\_init、target\_init、apps\_init。

1. platform\_init

platform\_init 中主要是函数 acpu\_clock\_init。

在 acpu\_clock\_init 对 arm11 进行系统时钟设置，超频

1. target\_init

针对硬件平台进行设置。主要对 arm9 和 arm11 的分区表进行整合，初始化flash和读取FLASH信息

1. apps\_init

apps\_init 是关键，对 LK 中所谓 app 初始化并运行起来。

源码：$Magenta/magenta/kernel/app/app.c

/\* one time setup \*/

void apps\_init(void)

{

const struct app\_descriptor \*app;

/\* call all the init routines \*/

for (app = \_\_start\_apps; app != \_\_stop\_apps; app++) {

if (app->init)

app->init(app);

}

/\* start any that want to start on boot \*/

for (app = \_\_start\_apps; app != \_\_stop\_apps; app++) {

if (app->entry && (app->flags & APP\_FLAG\_DONT\_START\_ON\_BOOT) == 0) {

start\_app(app);

}

}

}

## Boot程序中的trampoline(弹簧床)

Boot Loader 的 stage2 <https://www.ibm.com/developerworks/cn/linux/l-btloader/>

stage2 的代码通常用 C 语言来实现，以便于实现更复杂的功能和取得更好的代码可读性和可移植性。但是与普通 C 语言应用程序不同的是，在编译和链接 boot loader 这样的程序时，我们不能使用 glibc 库中的任何支持函数。其原因是显而易见的。这就给我们带来一个问题，那就是从那里跳转进 main() 函数呢？直接把 main() 函数的起始地址作为整个 stage2 执行映像的入口点或许是最直接的想法。但是这样做有两个缺点：1)无法通过main() 函数传递函数参数；2)无法处理 main() 函数返回的情况。一种更为巧妙的方法是利用 trampoline(弹簧床)的概念。也即，用汇编语言写一段trampoline 小程序，并将这段 trampoline 小程序来作为 stage2 可执行映象的执行入口点。然后我们可以在 trampoline 汇编小程序中用 CPU 跳转指令跳入 main() 函数中去执行；而当 main() 函数返回时，CPU 执行路径显然再次回到我们的 trampoline 程序。简而言之，这种方法的思想就是：用这段 trampoline 小程序来作为 main() 函数的外部包裹(external wrapper)。

下面给出一个简单的 trampoline 程序示例(来自blob)：

.text

.globl \_trampoline

\_trampoline:

bl main

/\* if main ever returns we just call it again \*/

b \_trampoline

可以看出，当 main() 函数返回后，我们又用一条跳转指令重新执行 trampoline 程序――当然也就重新执行 main() 函数，这也就是 trampoline(弹簧床)一词的意思所在。

如果你的main()函数不是call进去的，从main()返回时，就不知道返回到哪里去，通过trampoline(弹簧床)这个机制，相当于是造了一个栈，好让main()函数能够正确进入与退出。

## 起动过程

起动过程中的这些信息，是哪个程序出来的？

[00000.070] K initializing platform

[00000.070] K INIT: cpu 0, calling hook 0xffff00000001f008 (pcie\_early\_init) at level 0x8ffff, flags 0x1

[00000.083] K initializing target

[00000.083] K calling apps\_init()

[00000.083] K INIT: cpu 0, calling hook 0xffff000000026368 (ktrace) at level 0xaffff, flags 0x1

[00000.328] K ktrace: buffer at 0xffff0007811c0000 (33554432 bytes)

[00000.328] K INIT: cpu 0, calling hook 0xffff000000035140 (userboot) at level 0xaffff, flags 0x1

[00000.328] K userboot: console init

[00000.330] K userboot: bootfs 3268608 @ 0xffff00000007c000

[00000.346] K userboot: userboot-rodata 0 @ [0x1000000,0x1001000)

[00000.347] K userboot: userboot-code 0x1000 @ [0x1001000,0x1003000)

[00000.348] K userboot: vdso-rodata 0 @ [0x1003000,0x1007000)

[00000.349] K userboot: vdso-code 0x4000 @ [0x1007000,0x1008000)

[00000.357] K userboot: entry point @ 0x1001840

[00000.364] U userboot: searching bootfs for "bin/devmgr"

[00000.369] U userboot: bin/devmgr has PT\_INTERP "lib/ld.so.1"

[00000.370] U userboot: searching bootfs for "lib/ld.so.1"

[00000.384] U userboot: bin/devmgr started. userboot exiting.

[00000.394] U Loaded at [0x10de000,0x1111000): <application>

[00000.394] U Loaded at [0x1099000,0x109e000): <vDSO> (libmagenta.so)

[00000.395] U Loaded at [0x1000000,0x1099000): libc.so

[00000.397] U devmgr: main()

[00000.401] U devmgr: vfs init

[00000.405] U devmgr: bootfs #0 contains 84 files

[00000.418] U devmgr: launch /boot/bin/crashlogger (crashlogger)

[00000.507] U devmgr: launch acpisvc

[00000.581] U crashlogger service ready

[00000.688] U devmgr: load drivers

[00000.689] U console\_init()

[00000.695] U debug\_reader()

[00000.700] U console\_init()

[00000.702] U devmgr: shell startup

[00000.741] U devmgr: launch /boot/bin/mxsh (mxsh:console)

[00000.783] U devmgr: launch /boot/bin/netsvc (netsvc)

[00000.878] U devmgr: launch /boot/bin/mxsh (mxsh:autorun)

源码：$Magenta/magenta/kernel/lib/debuglog/debuglog.c

static int debuglog\_reader(void\* arg) {

n = snprintf(tmp, sizeof(tmp), "[%05d.%03d] %c %s\n",

(int) (rec->timestamp / 1000000000ULL),

(int) ((rec->timestamp / 1000000ULL) % 1000ULL),

(rec->flags & DLOG\_FLAG\_KERNEL) ? 'K' : 'U',

rec->data);

## 内核命令行参数

Magenta内核从处bootloader接收命令行参数，参数的格式是：参数，或“关键字=值（key=value）”。

对于型参数option=0, option=false，或option=off将禁止相应option。其它形式的（option, option=true, option=wheee, etc）将使能这个参数。

The kernel commandline is passed from the kernel to the userboot process and the device manager, so some of the options described below apply to those userspace processes, not the kernel itself.

### Magenta内核命令行参数

#### crashlogger.disable

If this option is set, the crashlogger is not started. You should leave this option off unless you suspect the crashlogger is causing problems.

#### driver..disable

Disables the driver with the given name. The driver name comes from the magenta\_driver\_info, and can be found as the second argument to the MAGENTA\_DRIVER\_BEGIN macro.

Example: driver.usb-audio.disable

#### kernel.watchdog=

If this option is set (disabled by default), the system will attempt to detect hangs/crashes and reboot upon detection.

#### gfxconsole.early=

This option (disabled by default) requests that the kernel start a graphics console during early boot (if possible), to display kernel debug print messages while the system is starting. When userspace starts up, a usermode graphics console driver takes over.

The early kernel console can be slow on some platforms, so if it is not needed for debugging it may speed up boot to disable it.

#### gfxconsole.font=

This option asks the graphics console to use a specific font. Currently only "9x16" (the default) and "18x32" (a double-size font) are supported.

#### smp.maxcpus=

This option caps the number of CPUs to initialize. It cannot be greater than SMP\_MAX\_CPUS for a specific architecture.

#### smp.ht=

This option can be used to disable the initialization of hyperthread logical CPUs. Defaults to true.

#### userboot=

This option instructs the userboot process (the first userspace process) to execute the specified binary within the bootfs, instead of following the normal userspace startup process (launching the device manager, etc).

It is useful for alternate boot modes (like a factory test or system unit tests).

#### userboot.shutdown

If this option is set, userboot will attempt to power off the machine when the process it launches exits.

### 如何把这些命令行参数传给内核。

#### in Qemu, using scripts/run-magenta\*

Pass each option using -c, for example:

./scripts/run-magenta-x86-64 -c gfxconsole.font=18x32 -c gfxconsole.early=false

#### in GigaBoot20x6, when netbooting

Pass the kernel commandline at the end, after a -- separator, for example:

bootserver magenta.bin bootfs.bin -- gfxconsole.font=18x32 gfxconsole.early=false

#### in GigaBoot20x6, when booting from USB flash

Create a text file named "cmdline" in the root of the USB flash drive's filesystem containing the command line.

## 系统启动平稳后的进程、线程、Job情况

>`mx ps

[kernel][magenta] Job objects (part 2)

In this installment we 'thread' the root job which is held in a

global into the userboot process, which has a handle to it.

The userboot sends the job object to the device manager which

then creates a child job object, which is not used at the moment.

The change includes a syscall to create a job object which only

devmanager can call since its the only one that gets passed a

job object.

the `mx ps shoud show at total of two job object now, like this

[00017.656] K id-s #t #h: #jb #pr #th #vm #mp #ev #ip #dp [name]

[00017.656] K 270-R 6 104: 2 1 6 4 84 0 3 0 [bin/devmgr]

# Magenta内核

## 内核起动地址

ARM平台

源码：$Magenta/magenta/kernel/arch/arm/arm/start.S

中断向量

.section ".text.boot"

.globl \_start

\_start:

b platform\_reset

b arm\_undefined

b arm\_syscall

b arm\_prefetch\_abort

b arm\_data\_abort

b arm\_reserved

b arm\_irq

b arm\_fiq

#if WITH\_SMP

b arm\_reset

#endif

## 从内核回用户态

源码：$Magenta/magenta/kernel/arch/arm/arm/arch.c

/\* switch to user mode, set the user stack pointer to user\_stack\_top, put the svc stack pointer to the top of the kernel stack \*/

void arch\_enter\_uspace(uintptr\_t pc, uintptr\_t sp,

uintptr\_t arg1, uintptr\_t arg2)

## 内核对象

内核对象，如VmObject，如果不只是被内核编程使用，用户态程序还会与这个对象直接打交道，则它会被包装为分发器Dispatcher。

|  |  |
| --- | --- |
| 内核对象 | 分发器Dispatcher |
|  | ProcessDispatcher |
| VmObject | VmObjectDispatcher |
| UserThread | ThreadDispatcher |
|  | EventDispatcher  EventPairDispatcher |
|  | DataPipeConsumerDispatcher  DataPipeProducerDispatcher |
|  | InterruptDispatcher  InterruptEventDispatcher |
|  | ResourceDispatcher |
|  | WaitSetDispatcher |
|  | IOPortDispatcher  IoMappingDispatcher |

### Process

#### 取当前进程

ProcessDispatcher::GetCurrent

auto up = ProcessDispatcher::GetCurrent();

### Thread

The thread object is the construct that represents a time-shared CPU execution context. Thread objects live associated to a particular Process Object which provides the memory and the handles to other objects necessary for I/O and computation.

Lifetime

A thread can be created implicitly by calling sys\_process\_start(), in which case the new thread is the "main thread" and the thread entrypoint is defined by the previously loaded binary. Or it can be created by calling sys\_thread\_create() which takes the entrypoint as a parameter.

A thread terminates when it returns from executing the routine specified as the entrypoint or by calling sys\_thread\_exit().

### Event

### Channel

进程间双向通信。

### Interrupt request

### VMObject

A Virtual Memory Object (VMO) represents a contiguous region of virtual memory that may be mapped into multiple address spaces.

源码：$Magenta/magenta/kernel/include/kernel/vm/vm\_object.h

源码：$Magenta/magenta/kernel/kernel/vm/vm\_object.cpp

// The base vm object that holds a range of bytes of data

//

// Can be created without mapping and used as a container of data, or mappable

// into an address space via VmAspace::MapObject

class VmObject : public mxtl::RefCounted<VmObject> {

源码：$Magenta/magenta/kernel/lib/magenta/include/magenta/vm\_object\_dispatcher.h

class VmObjectDispatcher : public Dispatcher {

### Data pipe

### Job

JobDispatcher

*Erlang最初为电信产品的开发，这种目的，决定了其对错误处理的严格要求。Erlang中提供一般语言所提供的exception，catch，try…catch等语法，同一时候Erlang支持Link和Monitor两种机制，我们能够将Process连接起来，让他们组成一个总体，某个Process出错，或推出时，其它Process都具有得知其推出的能力。而Monitor顾名思义，能够用来监控某个Process，推断其是否退出或出错。全部的这些Erlang都提供内在支持，我们高速的开发牢固的产品，不再是奢望。*

#### Linux中的工作队列

工作队列(work queue)是Linux kernel中将工作推后执行的一种机制。这种机制和BH或Tasklets不同之处在于工作队列是把推后的工作交由一个内核线程去执行，因此工作队列的优势就在于它允许重新调度甚至睡眠。

对于中断处理而言，Linux将其分成了两个部分，一个叫做中断handler（top half），是全程关闭中断的，另外一部分是deferable task（bottom half），属于不那么紧急需要处理的事情。在执行bottom half的时候，是开中断的。有多种bottom half的机制，例如：softirq、tasklet、workqueue或是直接创建一个kernel thread来执行bottom half。

### IOPort

## 程序加载器Loader

链接角本

源码：$Magenta/magenta/scripts/tmpl-rodso.ld

/\*

\* This is a linker script for producing a DSO (shared library) image

\* that is entirely read-only and trivial to map in without using a

\* proper ELF loader. It has two segments: read-only starting at the

\* beginning of the file, and executable code page-aligned and marked

\* by the (hidden) symbols CODE\_START and CODE\_END.

\*

\* Ideally this could be accomplished without an explicit linker

\* script. The linker would need an option to make the .dynamic

\* section (aka PT\_DYNAMIC segment) read-only rather than read-write;

\* in fact that could be the default for Magenta/Fuchsia or for

\* anything using a dynamic linker like musl's that doesn't try to

\* write into the .dynamic section at runtime (for -shared that is;

\* for -pie and dynamically-linked executables there is the DT\_DEBUG

\* question). The linker would need a second option to entirely

\* segregate code from rodata (and from non-loaded parts of the file),

\* and page-align the code segment (and pad the end to a page

\* boundary); in fact that could be the default for any system that

\* wants to minimize what can go into pages mapped with execute

\* permission, which is a worthwhile trade-off of security mitigation

\* over tiny amounts of wasted space in the ELF file. Beyond that,

\* the linker should not generate the .got\* or .plt\* sections at all

\* when there are no relocs being generated, but today's linkers still

\* do; since some of those sections are writable, they cause the

\* creation of a writable PT\_LOAD segment by normal linker logic.

\*/

### 内核elf-loader

Magenta所有程序都是ELF格式的。

bootstrap()是系统起动过程中的加载应用程序的函数。

load\_child\_process()会调用elf\_load\_bootfs(),elf\_load\_bootfs()会调用elf\_load\_prepare()

源码：$Magenta/magenta/system/uapp/userboot/start.c

// This is the main logic:

// 1. Read the kernel's bootstrap message.

// 2. Load up the child process from ELF file(s) on the bootfs.

// 3. Create the initial thread and allocate a stack for it.

// 4. Load up a message pipe with the mx\_proc\_args\_t message for the child.

// 5. Start the child process running.

// 6. Optionally, wait for it to exit and then shut down.

static noreturn void bootstrap(mx\_handle\_t log, mx\_handle\_t bootstrap\_pipe) {

load\_child\_process(log,proc\_self,&o,bootfs\_vmo, vdso\_vmo, proc, to\_child,

&entry, &vdso\_base, &stack\_size);

源码：$Magenta/magenta/system/uapp/userboot/userboot-elf.c

mx\_vaddr\_t elf\_load\_bootfs(mx\_handle\_t log, mx\_handle\_t proc\_self,

struct bootfs \*fs, mx\_handle\_t proc,

const char\* filename, mx\_handle\_t to\_child,

size\_t\* stack\_size) {

源码：$Magenta/magenta/system/ulib/elfload/elf-load.c

mx\_status\_t elf\_load\_prepare(mx\_handle\_t vmo, elf\_load\_header\_t\* header,

uintptr\_t\* phoff) {

### 应用程序加载器

源码：$Magenta/magenta/system/ulib/launchpad/launchpad.c

struct launchpad {

uint32\_t argc;

uint32\_t envc;

char\* args;

size\_t args\_len;

char\* env;

size\_t env\_len;

mx\_handle\_t\* handles;

uint32\_t\* handles\_info;

size\_t handle\_count;

size\_t handle\_alloc;

mx\_vaddr\_t entry;

mx\_vaddr\_t vdso\_base;

size\_t stack\_size;

mx\_handle\_t special\_handles[HND\_SPECIAL\_COUNT];

bool loader\_message;

};

# 系统功能调用

操作系统的系统，分内核侧和应用程序侧，工作在用户态的程序（UserLand）通过svc指令调用内核侧的代码arm\_syscall。

应用程序侧的用户空间代码通过系统调用与内核对象打交道，内核对象在用户空间只表现为句柄。用户空间的程序无法通过句柄推测句柄在内核中的内存使用情况。句柄是一个32位整数（类型：mx\_handle\_t）。

当系统调用执行时，内核首先检查句柄确实存在于进程句柄表中，并且句柄指向的资源是的一个合适的类型的资源，比如当前系统调用需要的是一个线程的句柄，而你传入的是一个事件Event的句柄，就出错了，然后，内核检查你对这个句柄有没有完成这个系统调用需要的合适的操作权限。

typedef uint32\_t mx\_rights\_t;

#define MX\_RIGHT\_NONE ((mx\_rights\_t)0u)

#define MX\_RIGHT\_DUPLICATE ((mx\_rights\_t)1u << 0)

#define MX\_RIGHT\_TRANSFER ((mx\_rights\_t)1u << 1)

#define MX\_RIGHT\_READ ((mx\_rights\_t)1u << 2)

#define MX\_RIGHT\_WRITE ((mx\_rights\_t)1u << 3)

#define MX\_RIGHT\_EXECUTE ((mx\_rights\_t)1u << 4)

#define MX\_RIGHT\_MAP ((mx\_rights\_t)1u << 5)

#define MX\_RIGHT\_GET\_PROPERTY ((mx\_rights\_t)1u << 6)

#define MX\_RIGHT\_SET\_PROPERTY ((mx\_rights\_t)1u << 7)

#define MX\_RIGHT\_DEBUG ((mx\_rights\_t)1u << 8)

#define MX\_RIGHT\_SAME\_RIGHTS ((mx\_rights\_t)1u << 31)

从调用者的角度看，系统调用分三大类：

1. 没有限定，或者限定很少。例如：mx\_time\_get()和mx\_nano\_sleep()可以被任何线程调用。
2. 第一个参数是一个句柄，引用这个句柄指向的内核对象的功能，也就是说，你的系统引用的功能是由这个内核对象提供的。这受这个句柄或内核对象的各种约定限制，例如：mx\_channel\_write()和mx\_port\_bind()。
3. 调用内核功能，但不传入任何句柄，而是要求内核在完成该系统功能调用后，返回句柄，例如：mx\_event\_create()和mx\_channel\_create()，这种情况下得到的句柄，从属于某个作业（Job），因为这个句柄是从属于这个进程的，而这个进程是从属于这个Job的。

系统调用由虚拟动态链接库libmagenta.so提供。虚拟动态链接库（virtual shared library，VDSO）是由内核提供给用户态程序，函数访问标准为：C ELF ABI，API的名字是mx\_noun\_verb()或者mx\_noun\_verb\_direct-object()的形式。

## svc中断向量

#ifndef WITH\_LIB\_SYSCALL

FUNCTION(arm\_syscall)

saveall

/\* r0 now holds pointer to iframe \*/

bl arm\_syscall\_handler

restoreall

#endif

## 内核栈svc stack

源码：$Magenta/magenta/kernel/arch/arm/arm/arch.c

/\* initial and abort stacks \*/

uint8\_t abort\_stack[ARCH\_DEFAULT\_STACK\_SIZE \*SMP\_MAX\_CPUS] \_\_CPU\_ALIGN;

在ARM平台上，通常ARCH\_DEFAULT\_STACK\_SIZE=4096。系统起动时，初始化CPU时会设置内核栈。

源码：$Magenta/magenta/kernel/arch/arm/arm/start.S

/\* set up the stack for irq, fiq, abort, undefined, system/user, and lastly supervisor mode \*/

mov r1, #0

cpsid i,#0x12 /\* irq \*/

mov sp, r1

cpsid i,#0x11 /\* fiq \*/

mov sp, r1

cpsid i,#0x17 /\* abort \*/

mov sp, r1

cpsid i,#0x1b /\* undefined \*/

mov sp, r1

cpsid i,#0x1f /\* system \*/

mov sp, r1

cpsid i,#0x13 /\* supervisor \*/

ldr r1, =abort\_stack

mov r2, #ARCH\_DEFAULT\_STACK\_SIZE

add r0, #1

mul r2, r2, r0

add r1, r2

mov sp, r1

通过下面两段源码：system-onesegment.ld、qemu-virt/rules.mk，能看出内核在内存中的布局，在QEMU-VIRT仿真器上，内核留了64KB内存，就是为了摆这些stack的。

源码：$Magenta/magenta/kernel/arch/arm/system-onesegment.ld

ENTRY(\_start)

SECTIONS

{

. = %KERNEL\_BASE% + %KERNEL\_LOAD\_OFFSET%;

\_start = .;

源码：$Magenta/magenta/kernel/platform/qemu-virt/rules.mk

MEMBASE := 0x40000000

MEMSIZE ?= 0x08000000 # 512MB

KERNEL\_LOAD\_OFFSET := 0x10000 # 64k

有些操作系统内核，当程序从用户态跑到内核里来，就把请求的栈及当前的线程等信息保存起来，归内核的调度体使用，有些内核则直接利用用户态的栈，直接干活。Magenta属于后者，而且一上来不管什么情况，把通用寄存器都保存了起来。

源码：$Magenta/magenta/kernel/arch/arm/arm/exceptions.S

FUNCTION(arm\_syscall)

saveall

/\* r0 now holds pointer to iframe \*/

bl arm\_syscall\_handler

restoreall

/\* Save all registers.

\* At exit r0 contains a pointer to the register frame.

\*/

.macro saveall

/\* save spsr and r14 onto the svc stack \*/

srsdb #0x13!

/\* switch to svc mode, interrupts disabled \*/

cpsid i,#0x13

/\* save all regs \*/

push { r0-r12, lr }

/\* save user space sp/lr \*/

sub sp, #8

stmia sp, { r13, r14 }^

#if ARM\_WITH\_VFP

/\* save and disable the vfp unit \*/

vfp\_save r0

#endif

/\* make sure the stack is 8 byte aligned \*/

stack\_align r0

/\* r0 now holds the pointer to the original iframe (before alignment) \*/

.endm

## 系统调用在内核侧的实现

源码：$Magenta/magenta/kernel/lib/syscalls/syscalls.cpp

/\* build a function pointer to call the routine.

\* the args are jammed into the function independent of if the function

\* uses them or not, which is safe for simple arg passing.

\*/

syscall\_func sfunc;

switch (syscall\_num) {

#define MAGENTA\_SYSCALL\_DEF(nargs64, nargs32, n, ret, name, args...) \

case n: \

sfunc = reinterpret\_cast<syscall\_func>(sys\_##name); \

break;

#include <magenta/syscalls.inc>

default:

sfunc = reinterpret\_cast<syscall\_func>(sys\_invalid\_syscall);

}

真正的系统调用都是通过宏MAGENTA\_SYSCALL\_DEF定义的。

源码：$Magenta/magenta/system/ulib/magenta/include/magenta/syscalls.inc

// Threads

MAGENTA\_SYSCALL\_DEF(4, 4, 50, mx\_handle\_t, thread\_create, int (\*entry)(void\*), USER\_PTR(void) arg,

USER\_PTR(const char) name, uint32\_t name\_len)

MAGENTA\_SYSCALL\_DEF\_WITH\_ATTRS(0, 0, 51, void, thread\_exit, (noreturn), void)

MAGENTA\_SYSCALL\_DEF(3, 3, 52, mx\_status\_t, thread\_arch\_prctl, mx\_handle\_t handle, uint32\_t op,

USER\_PTR(uintptr\_t) value\_ptr)

## 系统调用在应用程序侧的实现

### 系统调用定义

源码：$Magenta/magenta/system/ulib/magenta/include/magenta/syscalls.h

// define all of the syscalls from the syscall list header.

// user space syscall vaneer routines are all prefixed with mx\_

#define MAGENTA\_DDKCALL\_DEF(a...)

#define MAGENTA\_SYSCALL\_DEF(nargs64, nargs32, n, ret, name, args...) extern ret mx\_##name(args);

#define MAGENTA\_SYSCALL\_DEF\_WITH\_ATTRS(nargs64, nargs32, n, ret, name, attrs, args...) extern ret mx\_##name(args) \_\_attribute\_\_(attrs);

#include <magenta/syscalls.inc>

In userspace the syscall dispatch layer (libmagenta) exposes the result values as variables of type mx\_status\_t that currently use the same spelling and values as the kernel for errors, but which will transition to using 0 for success and positive values for errors.

### 用户态与内核态调用者

工作在用户态的应用程序通过这些程序进入内核态，实现对系统功能的调用。用户态程序中关于系统调用的符号（如：mx\_handle\_duplicate），也就是通过这个程序提供的链接符号。

源码：

$Magenta/magenta/system/ulib/magenta/syscalls-arm32.S

$Magenta/magenta/system/ulib/magenta/syscalls-arm64.S

$Magenta/magenta/system/ulib/magenta/syscalls-x86-64.S

如ARM64的源码：

/\* define and implement the magenta syscall wrappers for arm64 \*/

.text

.cfi\_sections .eh\_frame, .debug\_frame

.macro syscall name, n

.globl \name

.type \name,STT\_FUNC

\name:

.cfi\_startproc

mov x16, #\n

svc #0xf0f /\* magic value to differentiate magenta syscalls \*/

ret

.cfi\_endproc

.size \name, . - \name

.endm

#if LIBDDK

#define MAGENTA\_SYSCALL\_DEF(a...)

#define MAGENTA\_DDKCALL\_DEF(nargs64, nargs32, n, ret, name, args...) syscall mx\_##name, n

#else

#define MAGENTA\_DDKCALL\_DEF(a...)

#define MAGENTA\_SYSCALL\_DEF(nargs64, nargs32, n, ret, name, args...) syscall mx\_##name, n

#endif

#include <magenta/syscalls.inc>

### 动态库libmagenta.so

源码：$Magenta/magenta/kernel/lib/userboot/vdso.S

// The whole thing can't be just an assembler macro because a macro

// operand cannot be a string like .incbin needs for the filename.

#define FILE\_IMAGE(name, NAME) \

image\_section name, NAME##\_CODE\_END; \

DATA(name##\_image) \

.incbin NAME##\_FILENAME, 0, NAME##\_CODE\_END; \

END(name##\_image)

// The vDSO image, aka libmagenta.so.

FILE\_IMAGE(vdso, VDSO)

// The earliest user-mode process runs this image, aka userboot.

FILE\_IMAGE(userboot, USERBOOT)

**make文件相关：**

源码：$Magenta/magenta/system/ulib/magenta/rules.mk

MODULE\_TYPE := userlib

# This library should not depend on libc.

MODULE\_COMPILEFLAGS := -ffreestanding

# Both kernel and userspace code needs magenta headers.

GLOBAL\_INCLUDES += $(LOCAL\_DIR)/include

ifeq ($(ARCH),arm)

MODULE\_SRCS += $(LOCAL\_DIR)/syscalls-arm32.S

else ifeq ($(ARCH),arm64)

MODULE\_SRCS += $(LOCAL\_DIR)/syscalls-arm64.S

else ifeq ($(ARCH),x86)

ifeq ($(SUBARCH),x86-64)

MODULE\_SRCS += $(LOCAL\_DIR)/syscalls-x86-64.S

else

MODULE\_SRCS += $(LOCAL\_DIR)/syscalls-x86.S

endif

endif

# This gets an ABI stub installed in sysroots, but the DSO never gets

# installed on disk because it's delivered magically by the kernel.

MODULE\_SO\_NAME := magenta

MODULE\_SO\_INSTALL\_NAME := -

# All the code this DSO is pure read-only/reentrant code that

# does not need any writable data (except its caller's stack).

# Make it use a simplified, hardened memory layout.

MODULE\_LDFLAGS := -T $(BUILDDIR)/rodso.ld

MODULE\_EXTRA\_OBJS := $(BUILDDIR)/rodso-stamp

### 应用程序系统调用头文件

源码：$Magenta/magenta/system/ulib/magenta/include/magenta/syscalls.h

// define all of the syscalls from the syscall list header.

// user space syscall vaneer routines are all prefixed with mx\_

#define MAGENTA\_DDKCALL\_DEF(a...)

#define MAGENTA\_SYSCALL\_DEF(nargs64, nargs32, n, ret, name, args...) extern ret mx\_##name(args);

#define MAGENTA\_SYSCALL\_DEF\_WITH\_ATTRS(nargs64, nargs32, n, ret, name, attrs, args...) extern ret mx\_##name(args) \_\_attribute\_\_(attrs);

#include <magenta/syscalls.inc>

### 导出系统调用符号

Mangeta设有userboot机制，就是一个用户态应用起来时，你要加载的有些符号是系统向你提供的，就象是Linux动态链接库(.so)一样，只是这些库是预先装入的。

我再科普一下什么是系统调用。操作系统的系统调用大部分是实现在内核里的，应用程序通过一个类似ARM中的svc指令，调用内核的功能。但这些系统调用，需要完善、包装，才能被应用程序使用，比如堆heap，是c/c++库提供的功能，内核提供的内存管理API很简单，不易用。这个包装后的程序，被通俗称为系统调用。

#### make角本

# This generated linker script defines symbols for each vDSO entry point

# giving the relative address where it will be found at runtime. With

# this hack, the userboot code doesn't need to do any special work to

# find the vDSO and its entry points, keeping the code far simpler.

$(BUILDDIR)/$(LOCAL\_DIR)/vdso-syms.ld: \

$(LOCAL\_DIR)/vdso-syms.ld.h $(BUILDDIR)/$(LOCAL\_DIR)/vdso-syms.h

@$(MKDIR)

@echo generating $@

$(NOECHO)$(CC) -E -P -include $^ > $@

GENERATED += $(BUILDDIR)/$(LOCAL\_DIR)/vdso-syms.ld.h

源码：$Magenta/magenta/system/uapp/userboot/vdso-syms.ld.h

// For each function in the vDSO ABI, define a symbol in the linker script

// pointing to its address. The vDSO is loaded immediately after the

// userboot DSO image's last page, which is marked by the CODE\_END symbol.

// So these symbols tell the linker where each vDSO function will be found

// at runtime. The userboot code uses normal calls to these, declared as

// have hidden visibility so they won't generate PLT entries. This results

// in the userboot binary having simple PC-relative calls to addresses

// outside its own image, to where the vDSO will be found at runtime.

#define FUNCTION(name, address, size) \

PROVIDE\_HIDDEN(name = CODE\_END + address);

#### GCC链接参数

-E

Stop after the preprocessing stage; do not run the compiler proper. The output is in the form of preprocessed source code, which is sent to the standard output.

Input files which don't require preprocessing are ignored.

-P

Inhibit generation of linemarkers in the output from the preprocessor. This might be useful when running the preprocessor on something that is not C code, and will be sent to a program which might be confused by the linemarkers.

## 用户态与内核态间的参数传递

源码：$Magenta/magenta/kernel/arch/arm/arm/user\_copy.c

status\_t arch\_copy\_from\_user(void \*dst, const void \*src, size\_t len)

{

thread\_t \*thr = get\_current\_thread();

return \_arm\_copy\_from\_user(dst,src,len, &thr->arch.data\_fault\_resume);

}

status\_t arch\_copy\_to\_user(void \*dst, const void \*src, size\_t len)

{

thread\_t \*thr = get\_current\_thread();

return \_arm\_copy\_to\_user(dst,src, len, &thr->arch.data\_fault\_resume);

}

源码：$Magenta/magenta/kernel/lib/user\_copy/include/lib/user\_copy.h

inline status\_t copy\_to\_user\_unsafe(void\* dst, const void\* src, size\_t len) {

return arch\_copy\_to\_user(dst, src, len);

}

inline status\_t copy\_from\_user\_unsafe(void\* dst, const void\* src, size\_t len) {

return arch\_copy\_from\_user(dst, src, len);

}

源码：$Magenta/magenta/kernel/lib/user\_copy/user\_copy.c

// Default implementations of arch\_copy\* functions, almost certainly

// want to override with arch-specific versions that check access permissions

\_\_WEAK status\_t arch\_copy\_from\_user(void \*dst, const void \*src, size\_t len) {

memcpy(dst, src, len);

return NO\_ERROR;

}

\_\_WEAK status\_t arch\_copy\_to\_user(void \*dst, const void \*src, size\_t len) {

memcpy(dst, src, len);

return NO\_ERROR;

}

## 向内核传递命令行

从mxsh向内核传递命令行。

源码：$Magenta/magenta/system/uapp/mxsh/mxsh.c

static void send\_debug\_command(const char\* cmd) {

const char\* prefix = "kerneldebug ";

int fd = open("/dev/class/misc/dmctl", O\_WRONLY);

源码：$Magenta/magenta/system/core/devmgr/devhost-main.c

mx\_status\_t devmgr\_control(const char\* cmd) {

…………

const char\* prefix = "kerneldebug ";

if (!strncmp(cmd, prefix, strlen(prefix))) {

const char\* arg = cmd + strlen(prefix);

return mx\_debug\_send\_command(get\_root\_resource(), arg, strlen(arg));

源码：$Magenta/magenta/kernel/lib/syscalls/syscalls\_debug.cpp

int sys\_debug\_send\_command(mx\_handle\_t handle, const void\* ptr, uint32\_t len) {

return console\_run\_script(buf);

#### 手动向内核传递命令行

通过mxsh，手动向内核传递命令行。

> dm help

[00004.637] U dump - dump device tree

[00004.637] U lsof - list open remoteio files and devices

[00004.637] U crash - crash the device manager

[00004.637] U poweroff - poweroff the system

[00004.638] U reboot - reboot the system

[00004.638] U kerneldebug - send a command to the kernel

[00004.638] U ktraceoff - stop kernel tracing

[00004.638] U ktraceon - start kernel tracing

[00004.639] U acpi-ps0 - invoke the \_PS0 method on an acpi object

>

# 线程调度与时间管理

用户态时间单位：mx\_time\_t 单位是纳秒。

内核态时间单位：lk\_time\_t 是毫秒，lk\_bigtime\_t单位是纳秒。

1秒=1000毫秒

1毫秒=1000微秒

1微秒=1000纳秒

1纳秒=1000皮秒

1皮秒=1000飞秒

## 当前线程

## 调度器

调度器是一个操作系统整体调度的策略，不只是某一块具体的程序。

源码：$Magenta/magenta/kernel/kernel/thread.c

void thread\_resched(void) {

timer\_set\_periodic(&preempt\_timer[cpu],10,(timer\_callback)thread\_timer\_tick, NULL);

Magenta/magenta/kernel/kernel/timer.c

/\* let the scheduler have a shot to do quantum expiration, etc \*/

/\* in case of dynamic timer, the scheduler will set up a periodic timer \*/

if (thread\_timer\_tick() == INT\_RESCHEDULE)

ret = INT\_RESCHEDULE;

系统时钟初始化

源码：$Magenta/magenta/kernel/kernel/timer.c

void timer\_init(void)

{

#if !PLATFORM\_HAS\_DYNAMIC\_TIMER

/\* register for a periodic timer tick \*/

platform\_set\_periodic\_timer(timer\_tick, NULL, 10); /\* 10ms \*/

#endif

系统时钟函数体

源码：$Magenta/magenta/kernel/kernel/thread.c

enum handler\_return thread\_timer\_tick(void)

{

thread\_t \*current\_thread = get\_current\_thread();

if (thread\_is\_real\_time\_or\_idle(current\_thread))

return INT\_NO\_RESCHEDULE;

current\_thread->remaining\_quantum--;

if (current\_thread->remaining\_quantum <= 0) {

return INT\_RESCHEDULE;

} else {

return INT\_NO\_RESCHEDULE;

}

}

## 线程优先级管理

线程优先级表现在当需要进行调度时，从哪个数据结构里，如何拿到就续的线程。get\_top\_thread()函数就是干这个的。

源码：$Magenta/magenta/kernel/kernel/thread.c

static thread\_t \*get\_top\_thread(int cpu)

{

thread\_t \*newthread;

uint32\_t local\_run\_queue\_bitmap = run\_queue\_bitmap;

while (local\_run\_queue\_bitmap) {

/\* find the first (remaining) queue with a thread in it \*/

uint next\_queue = HIGHEST\_PRIORITY - \_\_builtin\_clz(local\_run\_queue\_bitmap)

- (sizeof(run\_queue\_bitmap) \* 8 - NUM\_PRIORITIES);

list\_for\_every\_entry(&run\_queue[next\_queue], newthread, thread\_t, queue\_node) {

#if WITH\_SMP

if (newthread->pinned\_cpu < 0 || newthread->pinned\_cpu == cpu)

#endif

{

list\_delete(&newthread->queue\_node);

if (list\_is\_empty(&run\_queue[next\_queue]))

run\_queue\_bitmap &= ~(1<<next\_queue);

return newthread;

}

}

local\_run\_queue\_bitmap &= ~(1<<next\_queue);

}

/\* no threads to run, select the idle thread for this cpu \*/

return idle\_thread(cpu);

}

## context切换

在中断返回的之前，把中断记录下来的上下文环境改变一下，就实现了线程切换。

CPU执行时，是没有线程的概念的，当内核处理中断返回时，你做了工作，本来是要执行进入中断时的那条指令的下一条指令，而在中断处理时，你把记录中断进入上下文环境context的那个数据结构frame给改了，于是，中断就如你所愿，按照你的指引，跑你预计的程序（另一个线程）中去了。

源码：$Magenta/magenta/kernel/kernel/thread.c

void thread\_resched(void) {

/\* do the low level context switch \*/

arch\_context\_switch(oldthread, newthread);

arch\_context\_switch()对应到具体平台上就是诸如arm\_context\_switch()这样的函数。

void arch\_context\_switch(thread\_t \*oldthread, thread\_t \*newthread)

{

// TRACEF("arch\_context\_switch: cpu %u old %p (%s), new %p (%s)\n", arch\_curr\_cpu\_num(), oldthread, oldthread->name, newthread, newthread->name);

#if ARM\_WITH\_VFP

arm\_fpu\_thread\_swap(oldthread, newthread);

#endif

arm\_context\_switch(&oldthread->arch.sp, newthread->arch.sp);

}

硬件中表示一个线程并不是用的thread\_t这样的复杂的数据结构，是一个与硬件相关的结构体arch\_thread，它的第一个参数必须是vaddr\_t sp，用来存储栈地址。

源码：$Magenta/magenta/kernel/arch/arm/arm/include/arch/arch\_thread.h

struct arch\_thread {

vaddr\_t sp;

/\* if non-NULL, address to return to on data fault \*/

void \*data\_fault\_resume;

#if ARM\_WITH\_VFP

/\* has this thread ever used the floating point state? \*/

bool fpused;

uint32\_t fpscr;

uint32\_t fpexc;

double fpregs[32];

#endif

};

源码：$Magenta/magenta/kernel/arch/arm/arm/asm.S

/\* context switch frame is as follows:

\* lr

\* r11

\* r10

\* r9

\* r8

\* r7

\* r6

\* r5

\* r4

\* cp15\_ro

\* cp15\_rw

\*/

/\* arm\_context\_switch(addr\_t \*old\_sp, addr\_t new\_sp) \*/

FUNCTION(arm\_context\_switch)

/\* save non callee trashed supervisor registers \*/

/\* spsr and user mode registers are saved and restored in the iframe by exceptions.S \*/

mrc p15, 0, r2, c13, c0, 2

mrc p15, 0, r3, c, c0, 3

push { r2-r11, lr }

/\* save old sp \*/

str sp, [r0]

/\* clear any exlusive locks that the old thread holds \*/

#if ARM\_ARCH\_LEVEL >= 7

/\* can clear it directly \*/

clrex

#elif ARM\_ARCH\_LEVEL == 6

/\* have to do a fake strex to clear it \*/

ldr r0, =strex\_spot

strex r3, r2, [r0]

#endif

/\* load new regs \*/

mov sp, r1

pop { r2-r11, lr }

mcr p15, 0, r2, c13, c0, 2

mcr p15, 0, r3, c13, c0, 3

bx lr

## 定时器

计时器属于操作系统中的基础组件，不管是用户空间的程序开发，还是内核空间的程序开发，很多时候都需要有定时器作为基础组件的支持。使用定时器的目的无非是为了周期性的执行某一任务，或者是到了一个指定时间去执行某一个任务。

源码：$Magenta/magenta/kernel/kernel/timer.c

void timer\_init(void)

{

timer\_lock = SPIN\_LOCK\_INITIAL\_VALUE;

for (uint i = 0; i < SMP\_MAX\_CPUS; i++) {

list\_initialize(&timers[i].timer\_queue);

}

#if !PLATFORM\_HAS\_DYNAMIC\_TIMER

/\* register for a periodic timer tick \*/

platform\_set\_periodic\_timer(timer\_tick, NULL, 10); /\* 10ms \*/

#endif

}

每个CPU核拥有一个定时器队列。不是设在硬件中断上的定时器，只有在系统tick时。才被检查是否到时间，可见其时间精度受系统时钟的精度影响。

## 线程抢先preempt的实现

一个中断向量程序执行完后，判断是否需要线程抢先，不需要抢，接着干原来线程的活。

源码：$Magenta/magenta/kernel/arch/arm/arm/exceptions.S

FUNCTION(arm\_irq)

#if TIMESTAMP\_IRQ

/\* read the cycle count \*/

mrc p15, 0, sp, c9, c13, 0

str sp, [pc, #\_\_irq\_cycle\_count - . - 8]

#endif

save\_offset #4

/\* r0 now holds pointer to iframe \*/

/\* track that we're inside an irq handler \*/

LOADCONST(r2, \_\_arm\_in\_handler)

mov r1, #1

str r1, [r2]

/\* call into higher level code \*/

bl platform\_irq

/\* clear the irq handler status \*/

LOADCONST(r1, \_\_arm\_in\_handler)

mov r2, #0

str r2, [r1]

/\* reschedule if the handler returns nonzero \*/

cmp r0, #0

blne thread\_preempt

restore

# 内存管理

如果使用虚拟内存，内存管理可以分为页表机制和内存分配机制两大块。页表机制简单说来就是管理着设备真实物理地址与虚拟地址的一个动态或静态的映射，是基于硬件的MMU进行的。

* vmm：virtual memory manager，虚拟内存管理
* vm：vm commands，虚拟内存与物理内存map相关
* pmm：physical memory manager，物理内存管理

## 是否使用虚拟内存VM/NOVM

编译选项WITH\_KERNEL\_VM

源码：$Magenta/magenta/kernel/arch/arm/rules.mk

# we have a mmu and want the vmm/pmm

WITH\_KERNEL\_VM ?= 1

* 没有虚拟内存则编译目录：$Magenta/magenta/kernel/kernel/novm/rules.mk
* 有虚拟内存，则编译目录：$Magenta/magenta/kernel/kernel/vm/rules.mk

源码：$Magenta/magenta/kernel/kernel/rules.mk

ifeq ($(WITH\_KERNEL\_VM),1)

MODULE\_DEPS += kernel/vm

else

MODULE\_DEPS += kernel/novm

endif

## ARM MMU基本原理

CPU在访问VA（虚拟地址）时，TLB硬件完成VA到PA（物理地址）的转换，但是如果没有该VA的TLB entry，MMU的硬件单元translation table walk hardware（页表索引单元）会索引CP15寄存器c0提供的内存页表，进行地址转换，获取PA进行访问。并且会将该页表信息更新到TLB中，页表跟TLB可不是一个概念，TLB是针对于内存页表的一个缓存硬件！

也就是说ARM的MMU不仅使用TLB进行地址转换，还能够对内存中提供的页表进行解析并地址转换，而TLB中存储的是CPU最常用的一些地址。TLB速度快，这样可以加快地址转换效率。

如果都找不到该VA的页表信息，MMU会向CPU发出异常（根据data还是instruct不同，发出data abort或者instruct abort），异常处理函数中进行页表填充。

ARM的MMU使用内存页表如何完成地址转换，手册一张图将MMU操作页表的几种方式列了出来，如图 7‑1所示：

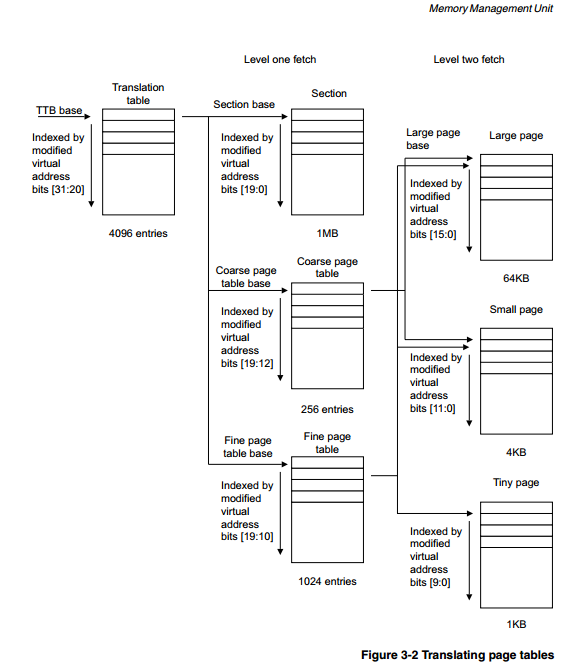


图 ‑1 ARM页表结构示意图

## 地址空间布局

### 虚拟地址规划

源码：$Magenta/magenta/kernel/include/kernel/vm.h

/\* kernel address space \*/

#define KERNEL\_ASPACE\_BASE ((vaddr\_t)0x80000000UL)

#define KERNEL\_ASPACE\_SIZE ((vaddr\_t)0x80000000UL)

/\* user address space, defaults to below kernel space with a 16MB guard gap on either side \*/

#define USER\_ASPACE\_BASE ((vaddr\_t)0x01000000UL)

#define USER\_ASPACE\_SIZE ((vaddr\_t)KERNEL\_ASPACE\_BASE - USER\_ASPACE\_BASE - 0x01000000UL)

/\*

\* user thread virtual memory layout:

\*

\* +-----------------------+ 0xFFFFFFFF

\* | |

\* | 内核空间 |

\* | |

\* +-----------------------+ KERNEL\_ASPACE\_BASE: 0x80000000UL

\* | | 警卫缝隙(Guard Gap)：16MB

\* +-----------------------+ USER\_ASPACE\_BASE：0x01000000

\* | |

\* | |

\* | 用户空间 |

\* | |

\* | |

\* +-----------------------+ USER\_ASPACE\_BASE：0x01000000

\* | | 警卫缝隙(Guard Gap)：16MB

### 物理内存管理

在系统Boot时，Boot程序本身会占用物理内存，当内核起动后，有些物理内存被新操作系统管理起来，但通常情况下，物理内存会被分割成几片。

物理内存管理的数据结构是pmm\_arena\_t。

源码：$Magenta/magenta/kernel/include/kernel/vm.h

/\* physical allocator \*/

typedef struct pmm\_arena {

struct list\_node node;

const char\* name;

uint flags;

uint priority;

paddr\_t base;

size\_t size;

size\_t free\_count;

struct vm\_page\* page\_array;

struct list\_node free\_list;

} pmm\_arena\_t;

系统初始化阶段，通过platform\_early\_init()函数，得到物理内存信息。

源码：$Magenta/magenta/kernel/kernel/vm/pmm.cpp

status\_t pmm\_add\_arena(pmm\_arena\_t\* arena)

源码：$Magenta/magenta/kernel/platform/qemu-virt/platform.c

static pmm\_arena\_t arena = {

.name = "ram",

.base = MEMORY\_BASE\_PHYS,

.size = DEFAULT\_MEMORY\_SIZE,

.flags = PMM\_ARENA\_FLAG\_KMAP,

};

void platform\_early\_init(void) {

/\* add the main memory arena \*/

pmm\_add\_arena(&arena);

## MMU页表组织

ARM中有两个协处理器寄存器用来存放一级页表基地址，页表基址寄存器TTBR0和TTBR1。操作系统把虚拟内存划分为内核空间和用户空间，TTBR0存放用户空间的一级页表基址，TTBR1存放内核空间的一级页表基址。

页表控制寄存器TTBCR。

ARM CPU中，TTBR0， TTBR1， TTBCR寄存器的操作指令mcr p15, 0, r10, c2, c0, 2。只是最后一个参数分别是0、1、2。

Magenta ARM平台彩用短描述符。

源码：$Magenta/magenta/kernel/arch/arm/arm/mmu.c

#if KERNEL\_ASPACE\_BASE != 0

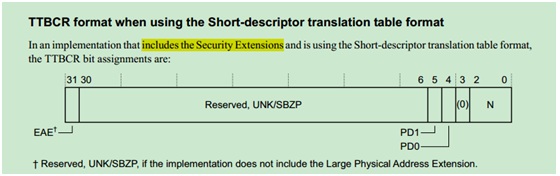
/\* bounce the ttbr over to ttbr1 and leave 0 unmapped \*/

uint32\_t n = \_\_builtin\_clz(KERNEL\_ASPACE\_BASE) + 1;

DEBUG\_ASSERT(n <= 7);

uint32\_t ttbcr = (1<<4) | n; /\* disable TTBCR0 and set the split between TTBR0 and TTBR1 \*/

arm\_write\_ttbr1(arm\_read\_ttbr0());



## 物理内存管理

源码：$Magenta/magenta/kernel/kernel/vm/pmm.cpp

STATIC\_COMMAND("pmm", "physical memory manager", &cmd\_pmm)

## 虚拟地址管理

源码：$Magenta/magenta/kernel/kernel/vm/vmm.cpp

STATIC\_COMMAND("vmm", "virtual memory manager", &cmd\_vmm)

## 虚实映射与页表切换

源码：Magenta/magenta/kernel/arch/arm/arm/mmu.c

int arch\_mmu\_map(arch\_aspace\_t \*aspace, addr\_t vaddr, paddr\_t paddr, uint count, uint flags)

参数说明：

* aspace：页表地址

struct arch\_aspace {

/\* magic value for use-after-free detection \*/

uint32\_t magic;

/\* pointer to the translation table \*/

paddr\_t tt\_phys;

uint32\_t \*tt\_virt;

/\* range of address space \*/

vaddr\_t base;

size\_t size;

/\* list of pages allocated for these page tables \*/

struct list\_node pt\_page\_list;

};

* vaddr：虚拟地址空间
* paddr：物理地址
* count：页数
* flags：属性

不同的平台，ARM、ARM64、X86等等，切换页表的方式是不一样的，因为内核页表只有一份，所以不存在切换的问题，所谓切换页表，是指用户空间的页表。

以ARM32为例，它的页表切换程序是：arch\_mmu\_context\_switch()。

源码：$Magenta/magenta/kernel/arch/arm/arm/mmu.c

void arch\_mmu\_context\_switch(arch\_aspace\_t \*old\_aspace, arch\_aspace\_t \*aspace)

{

if (LOCAL\_TRACE && TRACE\_CONTEXT\_SWITCH)

LTRACEF("aspace %p\n", aspace);

uint32\_t ttbr;

uint32\_t ttbcr = arm\_read\_ttbcr();

if (aspace) {

DEBUG\_ASSERT(aspace->magic == ARCH\_ASPACE\_MAGIC);

ttbr = MMU\_TTBRx\_FLAGS | (aspace->tt\_phys);

ttbcr &= ~(1<<4); // enable TTBR0

} else {

ttbr = 0;

ttbcr |= (1<<4); // disable TTBR0

}

if (LOCAL\_TRACE && TRACE\_CONTEXT\_SWITCH)

LTRACEF("ttbr 0x%x, ttbcr 0x%x\n", ttbr, ttbcr);

arm\_write\_ttbr0(ttbr);

arm\_write\_ttbcr(ttbcr);

}

在地址管理时，遇到需要切换页表时，可以调用vmm\_set\_active\_aspace(vmm\_aspace\_t\* aspace)来将当前页表切换为新的页表

源码：$Magenta/magenta/kernel/kernel/vm/vmm.cpp

static inline void vmm\_context\_switch(VmAspace\* oldspace, VmAspace\* newaspace) {

DEBUG\_ASSERT(thread\_lock\_held());

arch\_mmu\_context\_switch(oldspace ? &oldspace->arch\_aspace() : nullptr,

newaspace ? &newaspace->arch\_aspace() : nullptr);

}

# 中断

现代操作系统是中断驱动的。如果没有进程可执行，没有I/O设备可服务，没有用户可响应，那么操作系统就会安静的等待事件的发生。

事件的发生几乎总是通过中断或者陷阱来表征的。

陷阱（trap）或异常（exception）是因错误（如除0、非法访问内存）或者用户程序（执行操作系统服务）的特定请求所引起的软件生成中断。

对于中断，硬件可以随时通过系统总线向cpu发出信号，触发中断。

软件通过执行系统调用的特别操作触发中断。

对于每种类型的中断，操作系统都有一段独立的代码决定采取动作，中断服务子程序负责处理中断。

中断系统中的中断发生后，系统停止响应，并执行中断响应，这个过程就叫系统中断。

所谓中断是指CPU对系统发生的某个事件做出的一种反应，CPU暂停正在执行的程序，保留现场后自动地转去执行相应的处理程序，处理完该事件后再返回断点继续执行被“打断”的程序。

**中断可分为三类：**

第一类是由CPU外部引起的，称作中断，如I/O中断、时钟中断、控制台中断等。

第二类是来自CPU的内部事件或程序执行中的事件引起的过程，称作异常，如由于CPU本身故障（电源电压低于105V或频率在47～63Hz之外）、程序故障（非法操作码、地址越界、浮点溢出等）等引起的过程。

第三类由于在程序中使用了请求系统服务的系统调用而引发的过程，称作“陷入”(trap,或者陷阱)。

前两类通常都称作中断，它们的产生往往是无意、被动的，而陷入是有意和主动的。

**中断处理**

中断处理一般分为中断响应和中断处理两个步骤。

中断响应由硬件实施，中断处理主要由软件实施。

（1）中断响应

对中断请求的整个处理过程是由硬件和软件结合起来而形成的一套中断机构实施的。发生中断时，CPU暂停执行当前的程序，而转去处理中断。这个由硬件对中断请求作出反应的过程，称为中断响应。一般说来，中断响应顺序执行下述三步动作：

* 中止当前程序的执行；
* 保存原程序的断点信息（主要是程序计数器PC和程序状态寄存器PS的内容）；
* 从中断控制器取出中断向量，转到相应的处理程序。

通常CPU在执行完一条指令后，立即检查有无中断请求，如果有，则立即做出响应。

当发生中断时，系统作出响应，不管它们是来自硬件（如来自时钟或者外部设备）、程序性中断（执行指令导致“软件中断”—Software Interrupts），或者来自意外事件（如访问页面不在内存）。

如果当前CPU的执行优先级低于中断的优先级，那么它就中止对当前程序下条指令的执行，接受该中断，并提升处理机的执行级别（一般与中断优先级相同），以便在CPU处理当前中断时，能屏蔽其它同级的或低级的中断，然后保存断点现场信息，通过取得的中断向量转到相应的中断处理程序的入口。

（2）中断处理

CPU从中断控制器取得中断向量，然后根据具体的中断向量从中断向量表IDT中找到相应的表项，该表项应是一个中断门。于是，CPU就根据中断门的设置而到达了该通道的总服务程序的入口。

核心对中断处理的顺序主要由以下动作完成：

* 保存正在运行进程的各寄存器的内容，把它们放入核心栈的新帧面中。
* 确定“中断源”或核查中断发生，识别中断的类型（如时钟中断或盘中断）和中断的设备号（如哪个磁盘引起的中断）。系统接到中断后，就从机器那里得到一个中断号，它是检索中断向量表的位移。中断向量因机器而异，但通常都包括相应中断处理程序入口地址和中断处理时处理机的状态字。
* 核心调用中断处理程序，对中断进行处理。
* 中断处理完成并返回。中断处理程序执行完以后，核心便执行与机器相关的特定指令序列，恢复中断时寄存器内容和执行核心栈退栈，进程回到用户态。如果设置了重调度标志，则在本进程返回到用户态时做进程调度。

中断向量

中断必须将控制转移到合适的中断服务程序，处理这种转移最简单的方法是调用一个通用子程序以检查中断信息，接着，该子程序调用相应的中断处理程序。

处理中断必须要快，通过使用中断处理子程序的指针表可间接调用中断处理子程序，而不需要其他中间子程序。通常指针表位于低内存地址（前一百左右的位置）这些位置包含各种设备中断处理子程序的地址（即中断向量）．中断向量对于给定的中断请求通过唯一设备号来索引，以提供设备的中断处理子程序的地址。

中断体系结构也保持被中断指令的地址（系统堆栈中存储返回地址），处理中断后，保存的返回地址会装入程序计数器，被中断的计算可以重新开始，就好像中断没有发生过一样。

## 内核态中断编程

如果是多核CPU，下面这些函数都是针对当前核来说的。

源码：$Magenta/magenta/kernel/dev/interrupt/include/dev/interrupt.h

enum interrupt\_trigger\_mode {

IRQ\_TRIGGER\_MODE\_EDGE = 0,

IRQ\_TRIGGER\_MODE\_LEVEL = 1,

};

enum interrupt\_polarity {

IRQ\_POLARITY\_ACTIVE\_HIGH = 0,

IRQ\_POLARITY\_ACTIVE\_LOW = 1,

};

status\_t mask\_interrupt(unsigned int vector);

status\_t unmask\_interrupt(unsigned int vector);

// Configure the specified interrupt vector. If it is invoked, it muust be

// invoked prior to interrupt registration

status\_t configure\_interrupt(unsigned int vector,

enum interrupt\_trigger\_mode tm,

enum interrupt\_polarity pol);

//中断处理函数原型

typedef enum handler\_return (\*int\_handler)(void\* arg);

//注册中断向量

void register\_int\_handler(unsigned int vector, int\_handler handler, void\* arg);

bool is\_valid\_interrupt(unsigned int vector, uint32\_t flags);

unsigned int remap\_interrupt(unsigned int vector);

## 用户态中断编程

Magenta提供的用户态编程的API：

* mx\_handle\_t mx\_interrupt\_create(mx\_handle\_t handle, uint32\_t vector, uint32\_t flags)
* mx\_status\_t mx\_interrupt\_complete(mx\_handle\_t handle)

## 中断程序执行过程

* 硬件封装层

它包含了体系架构相关的所有代码，包括中断控制器的抽象封装，arch相关的中断初始化，以及各个IRQ的相关数据结构的初始化工作，cpu的中断入口也会在arch相关的代码中实现。中断通用逻辑层通过标准的封装接口（实际上就是struct irq\_chip定义的接口）访问并控制中断控制器的行为，体系相关的中断入口函数在获取IRQ编号后，通过中断通用逻辑层提供的标准函数，把中断调用传递到中断流控层中。

* 中断流控层

所谓中断流控是指合理并正确地处理连续发生的中断，比如一个中断在处理中，同一个中断再次到达时如何处理，何时应该屏蔽中断，何时打开中断，何时回应中断控制器等一系列的操作。该层实现了与体系和硬件无关的中断流控处理操作，它针对不同的中断电气类型（level，edge......），实现了对应的标准中断流控处理函数，在这些处理函数中，最终会把中断控制权传递到驱动程序注册中断时传入的处理函数或者是中断线程中。

* 中断通用逻辑层

该层实现了对中断系统几个重要数据的管理，并提供了一系列的辅助管理函数。同时，该层还实现了中断线程的实现和管理，共享中断和嵌套中断的实现和管理，另外它还提供了一些接口函数，它们将作为硬件封装层和中断流控层以及驱动程序API层之间的桥梁，例如以下API：

* 驱动程序API

该部分向驱动程序提供了一系列的API，用于向系统申请/释放中断，打开/关闭中断，设置中断类型和中断唤醒系统的特性等操作。驱动程序的开发者通常只会使用到这一层提供的这些API即可完成驱动程序的开发工作，其他的细节都由另外几个软件层较好地“隐藏”起来了。

ARM平台，中断IRQ都是从同一个函数platform\_irq开始工作的。

源码：$Magenta/magenta/kernel/arch/arm/arm/exceptions.S

FUNCTION(arm\_irq)

#if TIMESTAMP\_IRQ

/\* read the cycle count \*/

mrc p15, 0, sp, c9, c13, 0

str sp, [pc, #\_\_irq\_cycle\_count - . - 8]

#endif

save\_offset #4

/\* r0 now holds pointer to iframe \*/

/\* track that we're inside an irq handler \*/

LOADCONST(r2, \_\_arm\_in\_handler)

mov r1, #1

str r1, [r2]

/\* call into higher level code \*/

bl platform\_irq

/\* clear the irq handler status \*/

LOADCONST(r1, \_\_arm\_in\_handler)

mov r2, #0

str r2, [r1]

/\* reschedule if the handler returns nonzero \*/

cmp r0, #0

blne thread\_preempt

restore

源码：$Magenta/magenta/kernel/dev/interrupt/arm\_gic/arm\_gic.c

enum handler\_return platform\_irq(struct iframe \*frame)

{

#if WITH\_LIB\_SM

uint32\_t ahppir = GICREG(0, GICC\_AHPPIR);

uint32\_t pending\_irq = ahppir & 0x3ff;

struct int\_handler\_struct \*h;

uint cpu = arch\_curr\_cpu\_num();

LTRACEF("ahppir %d\n", ahppir);

if (pending\_irq < MAX\_INT && get\_int\_handler(pending\_irq, cpu)->handler) {

enum handler\_return ret = 0;

uint32\_t irq;

uint8\_t old\_priority;

spin\_lock\_saved\_state\_t state;

spin\_lock\_save(&gicd\_lock, &state, GICD\_LOCK\_FLAGS);

/\* Temporarily raise the priority of the interrupt we want to

\* handle so another interrupt does not take its place before

\* we can acknowledge it.

\*/

old\_priority = arm\_gic\_get\_priority(pending\_irq);

arm\_gic\_set\_priority\_locked(pending\_irq, 0);

DSB;

irq = GICREG(0, GICC\_AIAR) & 0x3ff;

arm\_gic\_set\_priority\_locked(pending\_irq, old\_priority);

spin\_unlock\_restore(&gicd\_lock, state, GICD\_LOCK\_FLAGS);

LTRACEF("irq %d\n", irq);

if (irq < MAX\_INT && (h = get\_int\_handler(pending\_irq, cpu))->handler)

ret = h->handler(h->arg);

else

TRACEF("unexpected irq %d != %d may get lost\n", irq, pending\_irq);

GICREG(0, GICC\_AEOIR) = irq;

return ret;

}

return sm\_handle\_irq();

#else

return \_\_platform\_irq(frame);

#endif

}

# 文件系统

总体上说 Magenta 下的文件系统主要可分为三大块：一是上层的文件系统的系统调用，二是虚拟文件系统 VFS(Virtual Filesystem System)，三是挂载到 VFS 中的各实际文件系统，例如 FAT16、FAT32 等。

VFS 是一种软件机制，也许称它为 Magenta 的文件系统管理者更确切点，与它相关的数据结构只存在于物理内存当中。所以在每次系统初始化期间，Linux 都首先要在内存当中构造一棵 VFS 的目录树(在 Linux 的源代码里称之为 namespace)，实际上便是在内存中建立相应的数据结构。VFS 目录树在 Linux 的文件系统模块中是个很重要的概念，希望读者不要将其与实际文件系统目录树混淆，在笔者看来，VFS 中的各目录其主要用途是用来提供实际文件系统的挂载点，当然在 VFS 中也会涉及到文件级的操作，本文不阐述这种情况。下文提到目录树或目录，如果不特别说明，均指 VFS 的目录树或目录。图 9‑1是一种可能的目录树在内存中的影像。

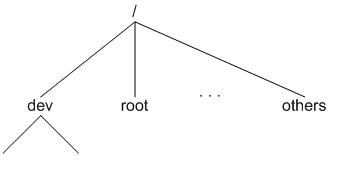


图 ‑1 Magenta文件系统结构

## 虚拟文件系统MXIO

mxio是一个接口约定，只要实现了这个接口函数，到系统里登记一下，就成为文件系统中的一部分了。

源码：$Magenta/magenta/system/ulib/mxio/private.h

// MXIO provides open/close/read/write io over various transports

// via the mxio\_t interface abstraction.

//

// The PIPE protocol uses message ports as simple, no-flow-control

// io pipes with a maximum message size of MX\_PIPE\_SIZE.

//

// The REMOTEIO protocol uses message ports to implement simple

// synchronous remoting of read/write/close operations.

//

// The NULL protocol absorbs writes and is never readable.

typedef struct mxio\_ops {

ssize\_t (\*read)(mxio\_t\* io, void\* data, size\_t len);

ssize\_t (\*read\_at)(mxio\_t\* io, void\* data, size\_t len, off\_t offset);

ssize\_t (\*write)(mxio\_t\* io, const void\* data, size\_t len);

ssize\_t (\*write\_at)(mxio\_t\* io, const void\* data, size\_t len, off\_t offset);

off\_t (\*seek)(mxio\_t\* io, off\_t offset, int whence);

mx\_status\_t (\*misc)(mxio\_t\* io, uint32\_t op, uint32\_t maxreply, void\* data, size\_t len);

mx\_status\_t (\*close)(mxio\_t\* io);

mx\_status\_t (\*open)(mxio\_t\* io, const char\* path, int32\_t flags, uint32\_t mode, mxio\_t\*\* out);

mx\_status\_t (\*clone)(mxio\_t\* io, mx\_handle\_t\* out\_handles, uint32\_t\* out\_types);

mx\_status\_t (\*wait)(mxio\_t\* io, uint32\_t events, uint32\_t\* pending, mx\_time\_t timeout);

ssize\_t (\*ioctl)(mxio\_t\* io, uint32\_t op, const void\* in\_buf, size\_t in\_len, void\* out\_buf, size\_t out\_len);

} mxio\_ops\_t;

程序调用关系：

typedef struct mxrio mxrio\_t;

struct mxrio {

// base mxio io object

mxio\_t io;

// message pipe handle for rpc

mx\_handle\_t h;

// event handle for device state signals

mx\_handle\_t e;

uint32\_t flags;

// TODO: replace with reply-pipes to allow

// true multithreaded io

mtx\_t lock;

};

创建一个文件系统。

mxio\_t\* mxio\_remote\_create(mx\_handle\_t h, mx\_handle\_t e) {

mxrio\_t\* rio = calloc(1, sizeof(\*rio));

if (rio == NULL)

return NULL;

rio->io.ops = &mx\_remote\_ops;

rio->io.magic = MXIO\_MAGIC;

rio->io.refcount = 1;

rio->h = h;

rio->e = e;

mtx\_init(&rio->lock, mtx\_plain);

return &rio->io;

}

源码：$Magenta/magenta/system/ulib/mxio/remoteio.c

static mx\_status\_t mxrio\_open(mxio\_t\* io, const char\* path, int32\_t flags, uint32\_t mode, mxio\_t\*\* out)

----打开，就是要打开句柄，于是这个函数调用mxio\_from\_handles ----🡪

源码：$Magenta/magenta/system/ulib/mxio/remoteio.c

mx\_status\_t mxio\_from\_handles(uint32\_t type, mx\_handle\_t\* handles, int hcount,

void\* extra, uint32\_t esize, mxio\_t\*\* out) {

case MXIO\_PROTOCOL\_REMOTE:

case MXIO\_PROTOCOL\_PIPE:

case MXIO\_PROTOCOL\_VMOFILE: {

源码：$Magenta/magenta/system/ulib/mxio/vmofile.c

typedef struct vmofile {

mxio\_t io;

mx\_handle\_t vmo;

mx\_off\_t off;

mx\_off\_t end;

mx\_off\_t ptr;

mtx\_t lock;

} vmofile\_t;

## 内置文件系统

系统一起动，就注册起来的几个文件系统，这几个文件系统都是内存文件系统。

源码：$Magenta/magenta/system/uapp/devmgr/vfs-memory.c

vnode\_t\* vfs\_get\_root(void) {

if (vfs\_root\_dn.vnode == NULL) {

vfs\_root\_dn.vnode = &vfs\_root.vn;

//TODO implement fs mount mechanism

dn\_add\_child(&vfs\_root\_dn, devfs\_get\_root()->dnode);

dn\_add\_child(&vfs\_root\_dn, bootfs\_get\_root()->dnode);

dn\_add\_child(&vfs\_root\_dn, memfs\_get\_root()->dnode);

\_mem\_create(&vfs\_root, &vn\_data, "data", 4, true);

}

return &vfs\_root.vn;

}

#### boot

Magenta/magenta/system/uapp/devmgr/vfs-boot.c

static dnode\_t vnb\_root\_dn = {

.name = "boot",

.flags = 4,

.children = LIST\_INITIAL\_VALUE(vnb\_root\_dn.children),

};

加载程序：

源码：$Magenta/magenta/system/uapp/userboot/userboot-elf.c

mx\_vaddr\_t elf\_load\_bootfs(mx\_handle\_t log, mx\_handle\_t proc\_self,

struct bootfs \*fs, mx\_handle\_t proc,

const char\* filename, mx\_handle\_t to\_child,

size\_t\* stack\_size) {

文件系统实现：

源码：$Magenta/magenta/system/uapp/userboot/bootfs.c

#### dev

源码：$Magenta/magenta/system/uapp/devmgr/vfs-device.c

static dnode\_t vnd\_root\_dn = {

.name = "dev",

.flags = 3,

.children = LIST\_INITIAL\_VALUE(vnd\_root\_dn.children),

};

#### data

data是通过这个函数创建的文件系统：

static mx\_status\_t \_mem\_create(mnode\_t\* parent, mnode\_t\*\* out,

const char\* name, size\_t namelen,

bool isdir);

#### tmp

源码：$Magenta/magenta/system/uapp/devmgr/vfs-memory.c

static dnode\_t mem\_root\_dn = {

.name = "tmp",

.flags = 3,

.children = LIST\_INITIAL\_VALUE(mem\_root\_dn.children),

};

#### system

[devmgr] mount additional bootfs images at /system, not /boot

Look for Fuchsia components (application\_manager, thinfs) on /system

## MinFS

MinFS是一个简单的，UNIX风格的Magenta内置文件系统。

当前支持的最大尺寸为512MB。

#define MINFS\_BLOCK\_SIZE 8192

// blocksize 8K 16K 32K

// 16 dir = 128K 256K 512K

// 32 ind = 512M 1024M 2048M

MinFS有两种使用情形：宿主设备或目标设备。

#### 宿主设备（Host Device）

1. 创建一个存有MinFS的磁盘镜像文件。

$ dd if=/dev/zero of=blk.bin bs=1M count=512

1. 执行Magenta时，通过’-d’参数，通知QEMU加载镜象文件。约定磁盘镜像文件文件名为：blk.bin。

$ ./scripts/run-magenta-x86-64 -d

#### 目标设备（Target Device）

**注意：**

* 在具体硬件中，"/dev/class/block/..." 指向的是具体的物理存储设备，如USB、SSD等等。
* 别格式化错了设备，如果不确定，可以通过Magenta的命令lsblk查看设备情况。
* Within Magenta, create a MinFS volume

> minfs /dev/class/block/000 mkfs

* Within Magenta, mount the MinFS volume

> minfs /dev/class/block/000 mount /data &

On any future boots, MinFS will be mounted automatically. Any files written to "/data" (the MinFS mount point) will persist across boots.

# 设备驱动与管理

## DDK

DDK是Device Development Kit，设备开发包的意思。

源码目录：$Magenta/magenta/system/ulib/ddk

## 硬件IO

几乎对每一种外设的控制都是通过读写设备上的寄存器来进行的，通常包括控制寄存器、状态寄存器和数据寄存器三大类，外设的寄存器通常被连续地编址。根据CPU体系结构的不同，CPU对IO端口的编址方式有两种：

（1）I/O映射方式（I/O-mapped）

典型地，如X86处理器为外设专门实现了一个单独的地址空间，称为”I/O地址空间”或者”I/O端口空间”，CPU通过专门的I/O指令（如X86的IN和OUT指令）来访问这一空间中的地址单元。

（2）内存映射方式（Memory-mapped）

RISC指令系统的CPU（如ARM、PowerPC等）通常只实现一个物理地址空间，外设I/O端口成为内存的一部分。此时，CPU可以象访问一个内存单元那样访问外设I/O端口，而不需要设立专门的外设I/O指令。　但是，这两者在硬件实现上的差异对于软件来说是完全透明的，驱动程序开发人员可以将内存映射方式的I/O端口和外设内存统一看作是”I/O内存”资源。

一般来说，在系统运行时，外设的I/O内存资源的物理地址是已知的，由硬件的设计决定。但是CPU通常并没有为这些已知的外设I/O内存资源的物理地址预定义虚拟地址范围，驱动程序并不能直接通过物理地址访问I/O内存资源，而必须将它们映射到核心虚地址空间内（通过页表），然后才能根据映射所得到的核心虚地址范围，通过访内指令访问这些I/O内存资源。

Magenta提供的硬件IO API：

* mx\_status\_t sys\_mmap\_device\_memory(mx\_handle\_t hrsrc, uintptr\_t paddr, uint32\_t len,

mx\_cache\_policy\_t cache\_policy,

user\_ptr<void\*> out\_vaddr)

* mx\_status\_t sys\_alloc\_device\_memory(mx\_handle\_t hrsrc, uint32\_t len,

mx\_paddr\_t\* out\_paddr, void\*\* out\_vaddr)

## 驱动程序

驱动程序是安装在driver/$(MODULE\_NAME).so里的动态链接库，编译时模块类型为driver（MODULE\_TYPE := driver），驱动程序不是sysroot的一部分，不产生调用ABI残根、静态链接库等。

涉及到的make文件：

源码：Magenta/magenta/kernel/make/module-driver.mk

# all drivers have a so name so they get installed

ifeq ($(MODULE\_SO\_NAME),)

ifeq ($(MODULE\_NAME),)

MODULE\_NAME := $(basename $(notdir $(MODULE)))

endif

MODULE\_SO\_NAME := $(MODULE\_NAME)

endif

# by default drivers live in lib/driver/...

ifeq ($(MODULE\_SO\_INSTALL\_NAME),)

MODULE\_SO\_INSTALL\_NAME := lib/driver/$(MODULE\_SO\_NAME).so

endif

include make/module-userlib.mk

驱动的make文件示例：

源码：$Magenta/magenta/system/ulib/driver/rules.mk

LOCAL\_DIR := system/uapp/devmgr

MODULE := system/ulib/driver

MODULE\_NAME := driver

MODULE\_TYPE := userlib

MODULE\_SO\_NAME := driver

MODULE\_DEFINES := LIBDRIVER=1

MODULE\_COMPILEFLAGS := -fvisibility=hidden

MODULE\_SRCS := \

$(LOCAL\_DIR)/devhost.c \

$(LOCAL\_DIR)/devhost-api.c \

$(LOCAL\_DIR)/devhost-binding.c \

$(LOCAL\_DIR)/devhost-core.c \

$(LOCAL\_DIR)/devhost-rpc-server.c \

system/udev/kpci/kpci.c \

MODULE\_STATIC\_LIBS := ulib/acpisvc-client ulib/ddk

MODULE\_LIBS := ulib/mxio ulib/launchpad ulib/magenta ulib/musl

include make/module.mk

## 设备管理

随着计算机的周边外设越来越丰富，设备管理已经成为现代操作系统的一项重要任务，在物理上，外设之间是有一种层次关系的，比如把一个U盘插到笔记本上，实际上这个U盘是接在一个USB Hub上，USB Hub又是接在USB 2.0 Host Controller (EHCI)上，最终EHCI又是一个挂在PCI Bus上的设备。这里的一个层次关系是：PCI->EHCI->USB Hub->USB Disk。如果操作系统要进入休眠状态，首先要逐层通知所有的外设进入休眠模式，然后整个系统才可以休眠。因此，需要有一个树状的结构可以把所有的外设组织起来。这就是最初建立Linux设备模型的目的。

当然，Magenta设备模型给我们带来的便利远不止如此。既然已经建立了一个组织所有设备和驱动的树状结构，用户就可以通过这棵树去遍历所有的设备，建立设备和驱动程序之间的联系，根据类型不同也可以对设备进行归类，这样就可以更清晰的去“看”这颗枝繁叶茂的大树。另外，Linux驱动模型把很多设备共有的一些操作抽象出来，大大减少了重复造轮子的可能。同时Magenta设备模型提供了一些辅助的机制，比如引用计数，让开发者可以安全高效的开发驱动程序。达成了以上这些好处之后，我们还得到了一个非常方便的副产品，这就是/dev----一个虚拟的文件系统。/dev给用户提供了一个从用户空间去访问内核设备的方法，它在Magenta里的路径是/dev。这个目录并不是存储在硬盘上的真实的文件系统，只有在系统启动之后才会建起来。

### 设备节点

源码：$Magenta/magenta/system/uapp/devmgr/dnode.h

typedef struct dnode dnode\_t;

#define DN\_NAME\_MAX 255

#define DN\_NAME\_LEN(flags) ((flags) & 0xFF)

#define DN\_TYPE\_MASK 0xF00

#define DN\_TYPE\_DIR 0x100

#define DN\_TYPE\_FILE 0x200

#define DN\_TYPE\_DEVICE 0x300

#define DN\_TYPE\_SYMLINK 0x400

#define DN\_TYPE(flags) ((flags) & DN\_TYPE\_MASK)

struct dnode {

dnode\_t\* parent;

vnode\_t\* vnode;

list\_node\_t children;

list\_node\_t dn\_entry; // entry in parent's list

list\_node\_t vn\_entry; // entry in vnode's list

uint32\_t flags;

char name[];

};

## 设备管理应用devmgr

源码：$Magenta/magenta/system/uapp/devmgr/devmgr.c

## 虚拟文件系统vfs

虚拟文件系统（VFS）是Magenbta内核和具体I/O设备之间的封装的一层共通访问接口，通过这层接口，Magenta内核可以以同一的方式访问各种I/O设备。

虚拟文件系统本身是Magenta内核的一部分，是纯软件的东西，并不需要任何硬件的支持。

源码：$Magenta/magenta/system/uapp/devmgr/vfs.c

## 设备树DeviceTree

*这部分工作还没有纳入最终系统，可是内部设备管理有了，基于描述数据的设备管理，那就应该是肯定的。*

源码：$Magenta/magenta/kernel/lib/devicetree/include/lib/devicetree.h

typedef struct dt\_slice {

u8 \*data;

u32 size;

} dt\_slice\_t;

struct devicetree\_header {

u32 magic;

u32 size;

u32 off\_struct; // offset from start to DT 'structure'

u32 off\_strings; // offset from start to stringdata

u32 off\_reserve; // offset from start to reserve memory map

u32 version;

u32 version\_compat; // last compatible version

u32 boot\_cpuid;

u32 sz\_strings; // size of stringdata

u32 sz\_struct; // size of DT 'structure'

};

typedef struct devicetree {

dt\_slice\_t top;

dt\_slice\_t dt;

dt\_slice\_t ds;

struct devicetree\_header hdr;

void (\*error)(const char \*msg);

} devicetree\_t;

# 编程概念

## Dispatcher

操作系统中scheduler与dispatcher是一个问题的两个角度，scheduler是在计算机资源（计算的执行机构，CPU、IO等）的角度看问题，当我有资源了，要怎么调度？dispatcher是从计算，被调度体的角度看问题，你有CPU资源了，我这里有N个可被执行的（进程、线程、事件、锁等概念的综合体）对象；你有IO了，我这里有N个IO的对象。

Dispatcher是一个靠引用计数管理的用来管理句柄的一个c++类。

源码：$Magenta/magenta/kernel/lib/magenta/include/magenta/dispatcher.h

// The Kernel Objects. Keep this list sorted.

class DataPipeConsumerDispatcher;

class DataPipeProducerDispatcher;

class InterruptDispatcher;

class IoMappingDispatcher;

class IOPortDispatcher;

class LogDispatcher;

class MessagePipeDispatcher;

class PciDeviceDispatcher;

class PciInterruptDispatcher;

class ProcessDispatcher;

class SocketDispatcher;

class ThreadDispatcher;

class VmObjectDispatcher;

class WaitSetDispatcher;

class ResourceDispatcher;

class Dispatcher : public mxtl::RefCounted<Dispatcher> {

public:

Dispatcher();

virtual ~Dispatcher() {}

mx\_koid\_t get\_koid() const { return koid\_; }

void add\_handle();

void remove\_handle();

通过下面这段程序，了解一下句柄与Dispatcher的关系。

源码：$Magenta/magenta/kernel/lib/syscalls/syscalls\_magenta.cpp

mx\_handle\_t sys\_port\_create(uint32\_t options) {

LTRACEF("options %u\n", options);

mxtl::RefPtr<Dispatcher> dispatcher;

mx\_rights\_t rights;

mx\_status\_t result = IOPortDispatcher::Create(options, &dispatcher, &rights);

if (result != NO\_ERROR)

return result;

//$Magenta/magenta/system/ulib/mxtl/include/mxtl/type\_support.h

//template <typename T>

//constexpr typename remove\_reference<T>::type&& move(T&& t) {

/ return static\_cast<typename remove\_reference<T>::type&&>(t);

//}

HandleUniquePtr handle(MakeHandle(mxtl::move(dispatcher), rights));

if (!handle)

return ERR\_NO\_MEMORY;

auto up = ProcessDispatcher::GetCurrent();

//把handle变为外面可访问的值，隐藏实现信息

mx\_handle\_t hv = up->MapHandleToValue(handle.get());

up->AddHandle(mxtl::move(handle));

return hv;

}

源码：$Magenta/magenta/kernel/lib/magenta/io\_port\_dispatcher.cpp

mx\_status\_t IOPortDispatcher::Create(uint32\_t options,

mxtl::RefPtr<Dispatcher>\* dispatcher,

mx\_rights\_t\* rights) {

AllocChecker ac;

auto disp = new (&ac) IOPortDispatcher(options);

if (!ac.check())

return ERR\_NO\_MEMORY;

\*rights = kDefaultIOPortRights;

//持有一个引用计数

\*dispatcher = mxtl::AdoptRef<Dispatcher>(disp);

return NO\_ERROR;

}

## 信号Signal

信号（Signal）是Magenta对象间通讯的一种方式。它是一种异步的通知机制，用来提醒内核对象一个事件已经发生。

源码：$Magenta/magenta/system/public/magenta/types.h

typedef uint32\_t mx\_signals\_t;

#define MX\_SIGNAL\_NONE ((mx\_signals\_t)0u)

#define MX\_SIGNAL\_READABLE ((mx\_signals\_t)1u << 0)

#define MX\_SIGNAL\_WRITABLE ((mx\_signals\_t)1u << 1)

#define MX\_SIGNAL\_PEER\_CLOSED ((mx\_signals\_t)1u << 2)

#define MX\_SIGNAL\_SIGNALED MX\_SIGNAL\_SIGNAL0

#define MX\_SIGNAL\_SIGNAL0 ((mx\_signals\_t)1u << 3)

#define MX\_SIGNAL\_SIGNAL1 ((mx\_signals\_t)1u << 4)

#define MX\_SIGNAL\_SIGNAL2 ((mx\_signals\_t)1u << 5)

#define MX\_SIGNAL\_SIGNAL3 ((mx\_signals\_t)1u << 6)

#define MX\_SIGNAL\_SIGNAL4 ((mx\_signals\_t)1u << 7)

#define MX\_SIGNAL\_SIGNAL\_ALL ((mx\_signals\_t)31u << 3)

typedef struct {

mx\_signals\_t satisfied;

mx\_signals\_t satisfiable;

} mx\_signals\_state\_t;

数据结构mx\_signals\_state\_t用来存储信号状态。

数据结构mx\_signals\_state\_t中，satisfied是一个已经发生的信号的比特图bitmap，satisfiable是可能发生的信号的比特图bitmap。

#### 适定性(Satisfiability)问题

关于数据结构mx\_signals\_state\_t为什么要设两个域，satisfied、satisfiable的问题。

适定性(Satisfiability)问题，简称 SAT。k-适定性问题，简称 k-SAT。

所谓2-SAT就是2元可满足性问题。首先，作为众所周知的，任何布尔表达式，都可以化为合取范式的形式，即化为： () and () and () ...and () 其中括号里面的是用析取符号or 连接的变量或者变量的非的形式。我们一般称，变量或者变量的非为“文字”,而括号里的叫做子句。可满足性问题是要给所有的变量一个赋值（真或假）使得表达式值为真。而2元可满足性问题，就是化为合取范式后，每个子句最多有两个文字的可满足性问题。

<https://en.wikipedia.org/wiki/2-satisfiability>

In computer science, 2-satisfiability (abbreviated as 2-SAT or just 2SAT) is the problem of determining whether a collection of two-valued (Boolean or binary) variables with constraints on pairs of variables can be assigned values satisfying all the constraints. It is a special case of the general Boolean satisfiability problem, which can involve constraints on more than two variables, and of constraint satisfaction problems, which can allow more than two choices for the value of each variable. But in contrast to those more general problems, which are NP-complete, 2-satisfiability can be solved in polynomial time.

<https://zh.wikipedia.org/wiki/%E5%B8%83%E5%B0%94%E5%8F%AF%E6%BB%A1%E8%B6%B3%E6%80%A7%E9%97%AE%E9%A2%98>

可满足性（英语：Satisfiability）是用来解决给定的布尔方程式，是否存在一组变量赋值，使问题为可满足。布尔可满足性问题（Boolean satisfiability problem；SAT)）属于决定性问题，是第一个被证明NP完全问题。为计算机科学上许多领域的重要问题，包括计算机科学基础理论、算法、人工智能、硬件设计等等。

布尔可满足性问题是第一个被证明的NPC问题。

**直观描述**

对于一个确定的逻辑电路，是否存在一种输入使得输出为真。

## 内核对象唯一标识值（KOID）

每个内核对象都有一个唯一标识值，它是一个64位的整数。0~2014的标识值是保留的。

源码：$Magenta/magenta/kernel/lib/magenta/dispatcher.cpp

// The first 1K koids are reserved.

static mx\_koid\_t global\_koid = 1024ULL;

mx\_koid\_t Dispatcher::GenerateKernelObjectId() {

return atomic\_add\_u64(&global\_koid, 1ULL);

}

## StateTracker/ StateObserver

一个事务，如套接字（socket）、消息管道（message pipe）都有需要调度者了解的状态信息，这些事务在编程者看来，是通过一个句柄表示的。于是有了状态跟踪与发现，比如非中断状态跟踪NonIrqStateTrackerTraits、中断状态跟踪IrqStateTrackerTraits。

using NonIrqStateTracker = internal::StateTrackerImpl<internal::NonIrqStateTrackerTraits>;

class IrqStateTracker : public internal::StateTrackerImpl<internal::IrqStateTrackerTraits>

源码：$Magenta/magenta/kernel/lib/magenta/include/magenta/state\_tracker.h

class StateTracker

源码：$Magenta/magenta/kernel/lib/magenta/include/magenta/state\_observer.h

// Observer base class for state maintained by StateTracker.

class StateObserver

#### Design Pattern Observer观察者模式

观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象。这个主题对象在状态上发生变化时，会通知所有观察者对象，使它们能够自动更新自己。

观察者模式是使用频率最高的设计模式之一，它用于建立一种对象与对象之间的依赖关系，一个对象发生改变时将自动通知其他对象，其他对象将相应作出反应。在观察者模式中，发生改变的对象称为观察目标，而被通知的对象称为观察者，一个观察目标可以对应多个观察者，而且这些观察者之间可以没有任何相互联系，可以根据需要增加和删除观察者，使得系统更易于扩展。

观察者模式定义如下： 观察者模式(Observer Pattern)：定义对象之间的一种一对多依赖关系，使得每当一个对象状态发生改变时，其相关依赖对象皆得到通知并被自动更新。观察者模式的别名包括发布-订阅（Publish/Subscribe）模式、模型-视图（Model/View）模式、源-监听器（Source/Listener）模式或从属者（Dependents）模式。观察者模式是一种对象行为型模式。

## 资源Resource

源码：$Magenta/magenta/kernel/lib/magenta/include/magenta/resource\_dispatcher.h

class ResourceDispatcher : public Dispatcher

Magenta把每个附属设备Device叫Resouce。

#### 获取根Resource

源码：$Magenta/magenta/system/ulib/ddk/include/ddk/driver.h

// temporary accessor for root resource handle

mx\_handle\_t get\_root\_resource(void);

## 异常处理ExceptionPort

CPU的异常发生后，把这个异常交给谁呢？哪些系统行为会引起异常？

* 线程

status\_t UserThread::SetExceptionPort(ThreadDispatcher\* td, mxtl::RefPtr<ExceptionPort> eport)

status\_t ThreadDispatcher::SetExceptionPort(mxtl::RefPtr<ExceptionPort> eport)

* 进程

status\_t ProcessDispatcher::SetExceptionPort(mxtl::RefPtr<ExceptionPort> eport)

## 事件对（EventPair）

EventPair，事件对，一个事件句柄有了啥signal，立马通知另外一个，操作系统内核帮你干的，对于应用程序来说，它们是“同时”有了 signal的。

“事件对”思想的由来

在Windows 98和Windows NT 3.X年代，Windows中的环境子系统(csrss)进程还负责着Windows的图像图形系统核心的任务，其他任何程序需要在屏幕上绘图，系统都需要通过IPC手段通知csrss让其根据相关数据去绘制，所以其他Windows程序进程跟csrss的通讯效率就至关重要，这里微软的程序员就为了csrss的高效通讯加入了一种内核对象，就是现在一直都存在并且再也不会更新的“EventPair”对象。

而后Windows NT将绘图的任务移入了内核模块win32k.sys，csrss肩负的东西一下就少了很多，到现在在绘图上面也仅仅是承担着对控制台窗口的图像绘制任务，但当年那个为了csrss而设置EventPair对象则一直存在于Windows内，并且直到今天的Win7、Win8。

而又由于这个对象本是为了csrss而设置的，而csrss很快又“残疾”了，所以现在微软依然没有把这个对象的相关接口公开进Win32 API里面，不过那么多年过来了，虽然这个对象没有更新功能接口，不过看样子微软也不准把它删掉，所以大家也可以放心的在自己的程序里使用。（注：商业程序请慎重考虑）

而为什么又叫做Pair呢？因为一个EventPair对象就是2个Event对象，有“高位”和“低位”之分，分别给Server\Client使用。

## 状态信息status\_t

This describes the set of userspace-exposed errors used in Magenta. The first section provides the canonical names and description of each error code. The second section provides the concrete values.

Within the kernel, errors are typically resulted as variables of type status\_t and errors are defined by macros of the form ERR\_CANONICAL\_NAME e.g. ERR\_INTERNAL. All error cases are negative values and success is represented by a non-negative value.

In userspace the syscall dispatch layer (libmagenta) exposes the result values as variables of type mx\_status\_t that currently use the same spelling and values as the kernel for errors, but which will transition to using 0 for success and positive values for errors.

See Kernel internal errors for a list of kernel-internal values.

### Descriptions

Each category represents a class of errors. The first error code in each category is the generic code for that category and is used when no more specific code applies. Further error codes (if any) within a category represent particular types of errors within the class. In general, more specific error codes are preferred where possible.

Errors are described in terms of an operation, arguments, a subject, and identifiers. An operation is typically a function or system call. Arguments are typically the parameters to the call. The subject of an operation is the primary object the operation acts on, typically a handle and typically passed as the first argument. Identifiers are typically numbers or strings intended to unambiguously identify a resource used in the operation.

### Categories

所有错误信息在这里定义。

源码：$Magenta/magenta/system/public/magenta/fuchsia-types.h

#### Success

NO\_ERROR Operation succeeded.

#### General errors

These indicate that the system hit a general error while attempting the operation.

INTERNAL The system encountered an otherwise unspecified error while performing the operation.

NOT\_SUPPORTED The operation is not supported, implemented, or enabled.

NO\_RESOURCES The system was not able to allocate some resource needed for the operation.

NO\_MEMORY The system was not able to allocate memory needed for the operation.

#### Parameter errors

These indicate that the caller specified a parameter that does not specify a valid operation or that is invalid for the specified operation.

INVALID\_ARGUMENT An argument is invalid.

WRONG\_TYPE The subject of the operation is the wrong type to perform the operation.

Example: Attempting a message\_read on a thread handle.

BAD\_SYSCALL The specified syscall number is invalid.

BAD\_HANDLE A specified handle value does not refer to a handle.

OUT\_OF\_RANGE An argument is outside the valid range for this operation.

Note: This is used when the argument may be valid if the system changes state, unlike INVALID\_ARGUMENT which is used when the argument will never be valid.

BUFFER\_TOO\_SMALL A caller provided buffer is too small for this operation.

#### Precondition or state errors

These indicate that the operation could not succeed because the preconditions for the operation are not satisfied or the system is unable to complete the operation in its current state.

BAD\_STATE The system is unable to perform the operation in its current state.

Note: FAILED\_PRECONDITION is a reserved alias for this error

NOT\_FOUND The requested entity was not found.

TIMED\_OUT The time limit for the operation elapsed before the operation completed.

ALREADY\_EXISTS An object with the specified identifier already exists.

Example: Attempting to create a file when a file already exists with that name.

ALREADY\_BOUND The operation failed because the named entity is already owned or controlled by another entity. The operation could succeed later if the current owner releases the entity.

HANDLE\_CLOSED A handle being waited on was closed.

REMOTE\_CLOSED The operation failed because the remote end of the subject of the operation was closed.

UNAVAILABLE The subject of the operation is currently unable to perform the operation.

Note: This is used when there's no direct way for the caller to observe when the subject will be able to perform the operation and should thus retry.

SHOULD\_WAIT The operation cannot be performed currently but potentially could succeed if the caller waits for a prerequisite to be satisfied, for example waiting for a handle to be readable or writable.

Example: Attempting to read from a message pipe that has no messages waiting but has an open remote will return SHOULD\_WAIT. Attempting to read from a message pipe that has no messages waiting and has a closed remote end will return REMOTE\_CLOSED.

#### Permission errors

ACCESS\_DENIED The caller did not have permission to perform the specified operation.

Input/output errors

IO Otherwise unspecified error occurred during I/O.

IO\_REFUSED The entity the I/O operation is being performed on rejected the operation.

Example: an I2C device NAK'ing a transaction or a disk controller rejecting an invalid command.

IO\_DATA\_INTEGRITY The data in the operation failed an integrity check and is possibly corrupted.

Example: CRC or Parity error.

IO\_DATA\_LOSS The data in the operation is currently unavailable and may be permanently lost.

Example: A disk block is irrecoverably damaged.

# 编程基础设施

## mxtl，c++模板

源码：$Magenta/magenta/system/ulib/mxtl/include/mxtl/ref\_ptr.h

namespace mxtl {

### 智能指针

由于 C++ 语言没有自动内存回收机制，程序员每次 new 出来的内存都要手动 delete。程序员忘记 delete，流程太复杂，最终导致没有 delete，异常导致程序过早退出，没有执行 delete 的情况并不罕见。用智能指针便可以有效缓解这类问题。

对于编译器来说，智能指针实际上是一个栈对象，并非指针类型，在栈对象生命期即将结束时，智能指针通过析构函数释放有它管理的堆内存。所有智能指针都重载了“operator->”操作符，直接返回对象的引用，用以操作对象。访问智能指针原来的方法则使用“.”操作符。

#### ref\_ptr

智能指针可以自动管理用户在堆上创建的对象的生命周期，让用户不用负责内存回收，避免内存泄漏。一般的智能指针都定义为一个模板类，它的类型由被管理的对象类型初始化，内部包含了指向该对象的指针以及指向辅助生命周期管理的管理对象的指针。

源码：$Magenta/magenta/system/ulib/mxtl/include/mxtl/ref\_ptr.h

// RefPtr<T> holds a reference to an intrusively-refcounted object of type T.

//

// T should be a subclass of mxtl::RefCounted<>, or something that adheres to

// the

// same contract for AddRef() and Release().

//

// Except for initial construction (see below), this generally adheres to a

// subset of the interface for std::shared\_ptr<>. Unlike std::shared\_ptr<> this

// type does not support conversions between different pointer types, vending

// weak pointers, introspecting the reference count, or any operations that

// would result in allocating memory (unless T::AddRef or T::Release allocate

// memory).

//

// Construction: To create a RefPtr around a freshly created object, use the

// AdoptRef free function at the bottom of this header. To construct a RefPtr to

// hold a reference to an object that already exists use the copy or move

// constructor or assignment operator.

#### unique\_ptr

源码：$Magenta/magenta/system/ulib/mxtl/include/mxtl/unique\_ptr.h

auto\_ptr通过复制构造或者通过=赋值后,原来的auto\_ptr对象就报废了.所有权转移到新的对象中去了.而通过shared\_ptr可以让多个智能指针对象同时拥有某一块内存的访问权.但假如我们不希望多个内存块被多个智能指针对象共享,同时又不会像auto\_ptr那样不知不觉的就让原来的auto\_ptr对象失效,可咋整呢?

这个时候就要使用unique\_ptr了,顾名思义嘛,unique是唯一的意思.说明它跟auto\_ptr有点类似,就是同时只能有一个智能指针对象指向某块内存.但它还有些其他特性.

// This is a simplified version of std::unique\_ptr that supports custom

// stateless deleters but doesn't support type conversions between different

// pointer types.

### 数组Array

#### InlineArray

在栈上分配空间的数组。

源码：$Magenta/magenta/system/ulib/mxtl/include/mxtl/inline\_array.h

// Runtime-determined, fixed size arrays that are "inlined" (e.g., on the stack) if the size at most

// |max\_inline\_count| or heap-allocated otherwise. This is typically used like:

//

// AllocChecker ac;

// mxtl::InlineArray<mx\_handle\_t, 4u> handle\_values(&ac, num\_handles);

// if (!ac.check())

// return ERR\_NO\_MEMORY;

//

// Note: Currently, |max\_inline\_count| must be at least 1.

template <typename T, size\_t max\_inline\_count>

class InlineArray {

程序示例：

AllocChecker ac;

//如果元素数小于等于8个，就在栈上分配空间，否则在堆上分配空间

mxtl::InlineArray<mx\_handle\_t, 8> handle\_values(&ac, count);

if (!ac.check())

return ERR\_NO\_MEMORY;

#### Array

源码：$Magenta/magenta/system/ulib/mxtl/include/mxtl/array.h

template <typename T, typename Deleter = default\_delete<T[]>>

class Array {

### 数据类型支持type\_support

源码：$Magenta/magenta/system/ulib/mxtl/include/mxtl/type\_support.h

template <typename T>

constexpr typename remove\_reference<T>::type&& move(T&& t) {

return static\_cast<typename remove\_reference<T>::type&&>(t);

}

### 资源获取即初始化RAII

RAII是resource acquisition is initialization的缩写，意为“资源获取即初始化”。它是C++之父Bjarne Stroustrup提出的设计理念，其核心是把资源和对象的生命周期绑定，对象创建获取资源，对象销毁释放资源。

软件开发中，会用到各种各样的资源。狭义的资源指内存，而广义的资源包括文件、网络连接、数据库连接、信号量、事件、线程、内存等，甚至可以是状态。资源获取后由于种种原因导致永久不能释放的资源称为资源泄漏。针对资源泄漏，提出了各种各样的软件机制和程序设计惯用法,如垃圾收集、RRID、RAII、确定性资源清理等。

RAII是C++语言的一种管理资源、避免泄漏的惯用法。C++标准保证任何情况下，已构造的对象最终会销毁，即它的析构函数最终会被调用。简单的说，RAII 的做法是使用一个对象，在其构造时获取资源，在对象生命期控制对资源的访问使之始终保持有效，最后在对象析构的时候释放资源。

RAII的理念固然简单，不过在具体实现的时候仍有需要小心的地方。比如对于STL的auto\_ptr，可以视为资源的代理对象，auto\_ptr对象间的赋值是一个需要特别注意的地方。简单说来资源代理对象间赋值的语义不满足“赋值相等”，其语义是资源管理权的转移。

什么是“赋值相等”呢？比如：

int a;

int b = 10;

a = b; //这句话执行后 a == b

但对于资源代理对象，这是不满足的，比如：

auto\_ptr<int> a(null);

auto\_ptr<int> b(new int(123));

a = b; //这句话执行后a != b，赋值的语义是b把资源的管理权交给了a。

源码：$Magenta/magenta/system/ulib/mxtl/include/mxtl/auto\_call.h

template <typename T>

class AutoCall {

### 容器、链表等

* intrusive\_container\_utils.h
* intrusive\_double\_list.h：mxtl::DoublyLinkedList<>
* intrusive\_hash\_table.h：mxtl::HashTable<>
* intrusive\_pointer\_traits.h
* intrusive\_single\_list.h：mxtl::SinglyLinkedList<>
* intrusive\_wavl\_tree.h

// Implementation Notes:

//

// WAVLTree<> is an implementation of a "Weak AVL" tree; a self

// balancing binary search tree whose rebalancing algorithm was

// originally described in

//

// Bernhard Haeupler, Siddhartha Sen, and Robert E. Tarjan. 2015.

// Rank-Balanced Trees. ACM Trans. Algorithms 11, 4, Article 30 (June 2015), 26 pages.

// DOI=http://dx.doi.org/10.1145/2689412

//

// See also

// https://en.wikipedia.org/wiki/WAVL\_tree

// http://www.cs.princeton.edu/~sssix/papers/rb-trees-talg.pdf

//

// WAVLTree<>s, like HashTables, are associative containers and support all of

// the same key-centric operations (such as find() and insert\_or\_find()) that

// HashTables support.

//

// Additionally, WAVLTree's are internally ordered by key (unlike HashTables

// which are un-ordered). Iteration from begin() to end() runs in amortized

// constant time and will enumerate the elements in monotonically increasing

// order (as defined by the KeyTraits::LessThan operation).

//

// Two additional operations are supported because of the ordered nature of a

// WAVLTree:

// upper\_bound(key) : Finds the element (E) in the tree such that E.key > key

// lower\_bound(key) : Finds the element (E) in the tree such that E.key >= key

//

// The worst depth of a WAVL tree depends on whether or not the tree has ever

// been subject to erase operations.

// ++ If the tree has seen only insert operations, the worst case depth of the

// tree is log\_phi(N), where phi is the golden ratio. This is the same bound

// as that of an AVL tree.

// ++ If the tree has seen erase operations in addition to insert operations,

// the worst case depth of the tree is 2\*log\_2(N). This is the same bound as

// a Red-Black tree.

//

// Insertion runs in O(log) time; finding the location takes O(log) time while

// post-insert rebalancing runs in amortized constant time.

//

// Erase-by-key runs in O(log) time; finding the node to erase takes O(log) time

// while post-erase rebalancing runs in amortized constant time.

//

// Because of the intrusive nature of the container, direct-erase operations

// (AKA, erase operations where the reference to the element to be erased is

// already known) run in amortized constant time.

//

* intrusive\_wavl\_tree\_internal.h
* string\_piece.h：mxtl::StringPiece

// A string-like object that points to a sized piece of memory.

// The string data may or may not be null terminated.

// The string piece does not own the data it points to.

## 内核编程可用程序资源

内核编程头文件：Magenta/magenta/kernel/include/kernel ，这里提供了下列程序资源：

* AutoLock
* AutoSpinLock
* cond
* debug
* event：event\_t
* mp，多CPU支持
* mutex：mutex\_t
* novm：物理内存
* semaphore ：semaphore\_t
* spin\_lock
* thread：thread\_t
* timer：定时器
* vm：虚拟内存
* wait：等待队列

## 内核堆

内核堆在lk\_main中初始化。

源码：$Magenta/magenta/kernel/top/main.c

/\* called from arch code \*/

void lk\_main(ulong arg0, ulong arg1, ulong arg2, ulong arg3)

{

…… …… ……

dprintf(SPEW, "initializing heap\n");

heap\_init();

源码中的两个堆算法，cmpctmallc和miniheap，最后选择哪一个，是在编译阶段确定的。

源码：$Magenta/magenta/kernel/lib/heap/rules.mk

# pick a heap implementation

ifndef LK\_HEAP\_IMPLEMENTATION

LK\_HEAP\_IMPLEMENTATION=miniheap

endif

ifeq ($(LK\_HEAP\_IMPLEMENTATION),miniheap)

MODULE\_DEPS := lib/heap/miniheap

endif

ifeq ($(LK\_HEAP\_IMPLEMENTATION),cmpctmalloc)

MODULE\_DEPS := lib/heap/cmpctmalloc

endif

### cmpctmalloc

这个算法最早是lk工程里，chromium中也用到它。

源码：$Magenta/magenta/kernel/lib/heap/cmpctmalloc/cmpctmalloc.c

下面一段讨论的话来源于：<https://groups.google.com/forum/#!topic/dartino-discuss/rf-Q1MIfEDQ>

Q: Does it extensively use heap? Or does it use mostly Data and BSS sections. There is a notion in embedded industry (atleast in safety related applications) to stay away from using heap (malloc, free, etc). Since most requirements are already known at compile time and to avoid failures due to memory fragmentation (which are very hard to reproduce during tests) or lack of memory.

A: We try to avoid allocating on the malloc heap, but we haven't been through the system with a fine-toothed comb and there may still be silly places where we use it. Again, the debugger "doesn' count" since it won't be active on a production system. There's a malloc implementation that we normally use, cmpctmalloc, which is very eager to coalesce adjacent free areas, but of course you can still get fragmentation.

A: We provide full access to allocation on the C-heap from Dart code running in the VM through our FFI (foreign function interface) API, in order to integrate well with existing C libraries. However, that part is something that the programmer is in full control of. An alternative pattern to using BSS could be to allocate what is needed outside Dartino's own heap through malloc during startup.

### miniheap

源码：$Magenta/magenta/kernel/lib/heap/miniheap/miniheap.c

### 物理内存申请

源码：Magenta/magenta/kernel/lib/heap/page\_alloc.c

/\* A simple page-aligned wrapper around the pmm or novm implementation of

\* the underlying physical page allocator. Used by system heaps or any

\* other user that wants pages of memory but doesn't want to use LK

\* specific apis.

\*/

### c++ new/delete支持及AllocChecker

源码：$Magenta/magenta/kernel/lib/libc/include/new.h

class AllocChecker {

public:

AllocChecker();

~AllocChecker();

void arm(size\_t sz, bool result);

bool check();

private:

unsigned state\_;

};

源码：$Magenta/magenta/kernel/lib/heap/new.cpp

enum : unsigned {

alloc\_armed = 1,

alloc\_ok = 2,

};

void panic\_if\_armed(unsigned state) {

#if LK\_DEBUGLEVEL > 1

if (state & alloc\_armed)

panic("AllocChecker::check() needs to be called\n");

#endif

}

AllocChecker::AllocChecker() : state\_(0U) {

}

AllocChecker::~AllocChecker() {

panic\_if\_armed(state\_);

}

void AllocChecker::arm(size\_t sz, bool result) {

panic\_if\_armed(state\_);

state\_ = alloc\_armed |

((sz == 0u) ? alloc\_ok : (result ? alloc\_ok : 0u));

}

bool AllocChecker::check() {

state\_ &= ~alloc\_armed;

return (state\_ & alloc\_ok) == alloc\_ok;

}

void \*operator new(size\_t s, AllocChecker\* ac) {

auto mem = malloc(s);

ac->arm(s, mem != nullptr);

return mem;

}

void \*operator new[](size\_t s, AllocChecker\* ac) {

auto mem = malloc(s);

ac->arm(s, mem != nullptr);

return mem;

}

void \*operator new(size\_t , void \*p) {

return p;

}

void operator delete(void \*p) {

return free(p);

}

void operator delete[](void \*p) {

return free(p);

}

## 用户堆/c库

Magenta采用musl作为c库。musl里面有堆的实现。

musl的官网地址是：<https://www.musl-libc.org/> musl is lightweight, fast, simple, free, and strives to be correct in the sense of standards-conformance and safety.

## 互斥机制

#### 互斥锁与条件变量为什么要配合着使用

下面这节中所讲的pthread\_cond......的情况与pthread\_mutex......的情况相同。

参考：再谈互斥锁与条件变量！ <http://blog.chinaunix.net/uid-27164517-id-3282242.html>

pthread\_cond\_wait总和一个互斥锁结合使用。在调用pthread\_cond\_wait前要先获取锁。pthread\_cond\_wait函数执行时先自动释放指定的锁，然后等待条件变量的变化。在函数调用返回之前，自动将指定的互斥量重新锁住。

int pthread\_cond\_signal(pthread\_cond\_t \* cond);

pthread\_cond\_signal通过条件变量cond发送消息，若多个消息在等待，它只唤醒一个。pthread\_cond\_broadcast可以唤醒所有。调用pthread\_cond\_signal后要立刻释放互斥锁，因为pthread\_cond\_wait的最后一步是要将指定的互斥量重新锁住，如果pthread\_cond\_signal之后没有释放互斥锁，pthread\_cond\_wait仍然要阻塞。

无论哪种等待方式，都必须和一个互斥锁配合，以防止多个线程同时请求pthread\_cond\_wait()（或pthread\_cond\_timedwait()，下同）的竞争条件（Race Condition）。mutex互斥锁必须是普通锁（PTHREAD\_MUTEX\_TIMED\_NP）或者适应锁 （PTHREAD\_MUTEX\_ADAPTIVE\_NP），且在调用pthread\_cond\_wait()前必须由本线程加锁 （pthread\_mutex\_lock()），而在更新条件等待队列以前，mutex保持锁定状态，并在线程挂起进入等待前解锁。在条件满足从而离开 pthread\_cond\_wait()之前，mutex将被重新加锁，以与进入pthread\_cond\_wait()前的加锁动作对应。

激发条件有两种形式，pthread\_cond\_signal()激活一个等待该条件的线程，存在多个等待线程时按入队顺序激活其中一个；而pthread\_cond\_broadcast()则激活所有等待线程。

下面是另一处说明：给出了函数运行全过程。 为什么在唤醒线程后要重新mutex加锁？

了解 pthread\_cond\_wait() 的作用非常重要 -- 它是 POSIX 线程信号发送系统的核心，也是最难以理解的部分。

首先，让我们考虑以下情况：线程为查看已链接列表而锁定了互斥对象，然而该列表恰巧是空的。这一特定线程什么也干不了 -- 其设计意图是从列表中除去节点，但是现在却没有节点。因此，它只能：

锁定互斥对象时，线程将调用 pthread\_cond\_wait(&mycond,&mymutex)。pthread\_cond\_wait() 调用相当复杂，因此我们每次只执行它的一个操作。

pthread\_cond\_wait() 所做的第一件事就是同时对互斥对象解锁（于是其它线程可以修改已链接列表），并等待条件 mycond 发生（这样当 pthread\_cond\_wait() 接收到另一个线程的“信号”时，它将苏醒）。现在互斥对象已被解锁，其它线程可以访问和修改已链接列表，可能还会添加项。 【要求解锁并阻塞是一个原子操作】

此时，pthread\_cond\_wait() 调用还未返回。对互斥对象解锁会立即发生，但等待条件 mycond 通常是一个阻塞操作，这意味着线程将睡眠，在它苏醒之前不会消耗 CPU 周期。这正是我们期待发生的情况。线程将一直睡眠，直到特定条件发生，在这期间不会发生任何浪费 CPU 时间的繁忙查询。从线程的角度来看，它只是在等待 pthread\_cond\_wait() 调用返回。

现在继续说明，假设另一个线程（称作“2 号线程”）锁定了 mymutex 并对已链接列表添加了一项。在对互斥对象解锁之后，2 号线程会立即调用函数 pthread\_cond\_broadcast(&mycond)。此操作之后，2 号线程将使所有等待 mycond 条件变量的线程立即苏醒。这意味着第一个线程（仍处于 pthread\_cond\_wait() 调用中）现在将苏醒。

现在，看一下第一个线程发生了什么。您可能会认为在 2 号线程调用 pthread\_cond\_broadcast(&mymutex) 之后，1 号线程的 pthread\_cond\_wait() 会立即返回。不是那样！实际上，pthread\_cond\_wait() 将执行最后一个操作：重新锁定 mymutex。一旦 pthread\_cond\_wait() 锁定了互斥对象，那么它将返回并允许 1 号线程继续执行。那时，它可以马上检查列表，查看它所感兴趣的更改。

来看一个例子（你是否能理解呢?）：

In Thread1:

pthread\_mutex\_lock(&m\_mutex);

pthread\_cond\_wait(&m\_cond,&m\_mutex);

pthread\_mutex\_unlock(&m\_mutex);

In Thread2:

pthread\_mutex\_lock(&m\_mutex);

pthread\_cond\_signal(&m\_cond);

pthread\_mutex\_unlock(&m\_mutex);

为什么要与pthread\_mutex 一起使用呢？ 这是为了应对 线程1在调用pthread\_cond\_wait()但线程1还没有进入wait cond的状态的时候，此时线程2调用了 cond\_singal 的情况。 如果不用mutex锁的话，这个cond\_singal就丢失了。加了锁的情况是，线程2必须等到 mutex 被释放（也就是 pthread\_cod\_wait() 释放锁并进入wait\_cond状态 ，此时线程2上锁） 的时候才能调用cond\_singal.

pthread\_cond\_signal即可以放在pthread\_mutex\_lock和pthread\_mutex\_unlock之间，也可以放在pthread\_mutex\_lock和pthread\_mutex\_unlock之后，但是各有有缺点。

之间：

pthread\_mutex\_lock

xxxxxxx

pthread\_cond\_signal

pthread\_mutex\_unlock

缺点：在某下线程的实现中，会造成等待线程从内核中唤醒（由于cond\_signal)然后又回到内核空间（因为cond\_wait返回后会有原子加锁的 行为），所以一来一回会有性能的问题。但是在LinuxThreads或者NPTL里面，就不会有这个问题，因为在Linux 线程中，有两个队列，分别是cond\_wait队列和mutex\_lock队列， cond\_signal只是让线程从cond\_wait队列移到mutex\_lock队列，而不用返回到用户空间，不会有性能的损耗。

所以在Linux中推荐使用这种模式。

之后：

pthread\_mutex\_lock

xxxxxxx

pthread\_mutex\_unlock

pthread\_cond\_signal

优点：不会出现之前说的那个潜在的性能损耗，因为在signal之前就已经释放锁了

缺点：如果unlock和signal之前，有个低优先级的线程正在mutex上等待的话，那么这个低优先级的线程就会抢占高优先级的线程（cond\_wait的线程)，而这在上面的放中间的模式下是不会出现的。

**示例程序**

源码：$Magenta/magenta/kernel/app/tests/thread\_tests.c

#### 自旋锁和互斥锁区别

POSIX threads（简称Pthreads）是在多核平台上进行并行编程的一套常用的API。线程同步（Thread Synchronization）是并行编程中非常重要的通讯手段，其中最典型的应用就是用Pthreads提供的锁机制（lock）来对多个线程之间共享的临界区（Critical Section）进行保护（另一种常用的同步机制是barrier）。

Pthreads提供了多种锁机制：

1. Mutex（互斥量）：pthread\_mutex\_\*\*\*
2. Spin lock（自旋锁）：pthread\_spin\_\*\*\*
3. Condition Variable（条件变量）：pthread\_con\_\*\*\*
4. Read/Write lock（读写锁）：pthread\_rwlock\_\*\*\*

Pthreads提供的Mutex锁操作相关的API主要有：

1. pthread\_mutex\_lock (pthread\_mutex\_t \*mutex);
2. pthread\_mutex\_trylock (pthread\_mutex\_t \*mutex);
3. pthread\_mutex\_unlock (pthread\_mutex\_t \*mutex);

Pthreads提供的与Spin Lock锁操作相关的API主要有：

1. pthread\_spin\_lock (pthread\_spinlock\_t \*lock);
2. pthread\_spin\_trylock (pthread\_spinlock\_t \*lock);
3. pthread\_spin\_unlock (pthread\_spinlock\_t \*lock);

从实现原理上来讲，Mutex属于sleep-waiting类型的锁。例如在一个双核的机器上有两个线程(线程A和线程B)，它们分别运行在Core0和 Core1上。假设线程A想要通过pthread\_mutex\_lock操作去得到一个临界区的锁，而此时这个锁正被线程B所持有，那么线程A就会被阻塞 (blocking)，Core0 会在此时进行上下文切换(Context Switch)将线程A置于等待队列中，此时Core0就可以运行其他的任务(例如另一个线程C)而不必进行忙等待。而Spin lock则不然，它属于busy-waiting类型的锁，如果线程A是使用pthread\_spin\_lock操作去请求锁，那么线程A就会一直在 Core0上进行忙等待并不停的进行锁请求，直到得到这个锁为止。

**自旋锁(Spin lock)**

自旋锁与互斥锁有点类似，只是自旋锁不会引起调用者睡眠，如果自旋锁已经被别的执行单元保持，调用者就一直循环在那里看是 否该自旋锁的保持者已经释放了锁，“自旋”一词就是因此而得名。其作用是为了解决某项资源的互斥使用。因为自旋锁不会引起调用者睡眠，所以自旋锁的效率远 高于互斥锁。虽然它的效率比互斥锁高，但是它也有些不足之处：

自旋锁一直占用CPU，他在未获得锁的情况下，一直运行——自旋，所以占用着CPU，如果不能在很短的时间内获得锁，这无疑会使CPU效率降低。

在用自旋锁时有可能造成死锁，当递归调用时有可能造成死锁，调用有些其他函数也可能造成死锁，如 copy\_to\_user()、copy\_from\_user()、kmalloc()等。

因此我们要慎重使用自旋锁，自旋锁只有在内核可抢占式或SMP的情况下才真正需要，在单CPU且不可抢占式的内核下，自旋锁的操作为空操作。自旋锁适用于锁使用者保持锁时间比较短的情况下。

### cond

条件变量是条件相关的数据结构。它允许线程在某些条件变为真之前被阻塞。例如，thread\_push可能希望检查队列是否已满，如果是这样，就在队列未满之前阻塞。所以我们感兴趣的“条件”就是“队列未满”。与之相似，thread\_pop希望等待“队列非空”的条件。

源码：$Magenta/magenta/kernel/include/kernel/cond.h

void cond\_init(cond\_t \*cond);

void cond\_destroy(cond\_t \*cond);

status\_t cond\_wait\_timeout(cond\_t \*cond, mutex\_t \*mutex, lk\_time\_t timeout);

void cond\_signal(cond\_t \*cond);

void cond\_broadcast(cond\_t \*cond);

### mutex

源码：$Magenta/magenta/kernel/include/kernel/mutex.h

/\* Rules for Mutexes:

\* - Mutexes are only safe to use from thread context.

\* - Mutexes are non-recursive.

\*/

void mutex\_init(mutex\_t \*);

void mutex\_destroy(mutex\_t \*);

status\_t mutex\_acquire\_timeout(mutex\_t \*, lk\_time\_t); /\* try to acquire the mutex with a timeout value \*/

void mutex\_release(mutex\_t \*);

### semaphore

信号量（semaphore）又称为信号灯，它是用来协调不同进程间的数据对象的，而最主要的应用是共享内存方式的进程间通信。本质上，信号量是一个计数器，它用来记录对某个资源（如共享内存）的存取状况。一般说来，为了获得共享资源，进程需要执行下列操作：

1. 测试控制该资源的信号量。
2. 若此信号量的值为正，则允许进行使用该资源。进程将信号量减1。
3. 若此信号量为0，则该资源目前不可用，进程进入睡眠状态，直至信号量值大于0，进程被唤醒，转入步骤（1）。
4. 当进程不再使用一个信号量控制的资源时，信号量值加1。如果此时有进程正在睡眠等待此信号量，则唤醒此进程。

源码：$Magenta/magenta/kernel/include/kernel/semaphore.h

void sem\_init(semaphore\_t \*, unsigned int);

void sem\_destroy(semaphore\_t \*);

int sem\_post(semaphore\_t \*, bool resched);

status\_t sem\_wait(semaphore\_t \*);

status\_t sem\_trywait(semaphore\_t \*);

status\_t sem\_timedwait(semaphore\_t \*, lk\_time\_t);

### spinlock

自旋锁（spinlock）它是为为实现保护共享资源而提出一种锁机制。其实，自旋锁与互斥锁比较类似，它们都是为了解决对某项资源的互斥使用。无论是互斥锁，还是自旋锁，在任何时刻，最多只能有一个保持者，也就说，在任何时刻最多只能有一个执行单元获得锁。但是两者在调度机制上略有不同。对于互斥锁，如果资源已经被占用，资源申请者只能进入睡眠状态。但是自旋锁不会引起调用者睡眠，如果自旋锁已经被别的执行单元保持，调用者就一直循环在那里看是否该自旋锁的保持者已经释放了锁，“自旋”一词就是因此而得名。

#### 自旋锁一般原理

跟互斥锁一样，一个执行单元要想访问被自旋锁保护的共享资源，必须先得到锁，在访问完共享资源后，必须释放锁。如果在获取自旋锁时，没有任何执行单元保持该锁，那么将立即得到锁；如果在获取自旋锁时锁已经有保持者，那么获取锁操作将自旋在那里，直到该自旋锁的保持者释放了锁。由此我们可以看出，自旋锁是一种比较低级的保护数据结构或代码片段的原始方式，这种锁可能存在两个问题：死锁和过多占用CPU资源。

#### 自旋锁适用情况

自旋锁比较适用于锁使用者保持锁时间比较短的情况。正是由于自旋锁使用者一般保持锁时间非常短，因此选择自旋而不是睡眠是非常必要的，自旋锁的效率远高于互斥锁。信号量和读写信号量适合于保持时间较长的情况，它们会导致调用者睡眠，因此只能在进程上下文使用，而自旋锁适合于保持时间非常短的情况，它可以在任何上下文使用。如果被保护的共享资源只在进程上下文访问，使用信号量保护该共享资源非常合适，如果对共享资源的访问时间非常短，自旋锁也可以。但是如果被保护的共享资源需要在中断上下文访问（包括底半部即中断处理句柄和顶半部即软中断），就必须使用自旋锁。自旋锁保持期间是抢占失效的，而信号量和读写信号量保持期间是可以被抢占的。自旋锁只有在内核可抢占或SMP（多处理器）的情况下才真正需要，在单CPU且不可抢占的内核下，自旋锁的所有操作都是空操作。另外格外注意一点：自旋锁不能递归使用。

源码：$Magenta/magenta/kernel/include/kernel/spinlock.h

/\* interrupts should already be disabled \*/

static inline void spin\_lock(spin\_lock\_t \*lock)

{

arch\_spin\_lock(lock);

}

/\* Returns 0 on success, non-0 on failure \*/

static inline int spin\_trylock(spin\_lock\_t \*lock)

{

return arch\_spin\_trylock(lock);

}

/\* interrupts should already be disabled \*/

static inline void spin\_unlock(spin\_lock\_t \*lock)

{

arch\_spin\_unlock(lock);

}

static inline void spin\_lock\_init(spin\_lock\_t \*lock)

{

arch\_spin\_lock\_init(lock);

}

static inline bool spin\_lock\_held(spin\_lock\_t \*lock)

{

return arch\_spin\_lock\_held(lock);

}

/\* spin lock irq save flags: \*/

/\* Possible future flags:

\* SPIN\_LOCK\_FLAG\_PMR\_MASK = 0x000000ff

\* SPIN\_LOCK\_FLAG\_PREEMPTION = 0x00000100

\* SPIN\_LOCK\_FLAG\_SET\_PMR = 0x00000200

\*/

/\* Generic flags \*/

#define SPIN\_LOCK\_FLAG\_INTERRUPTS ARCH\_DEFAULT\_SPIN\_LOCK\_FLAG\_INTERRUPTS

/\* same as spin lock, but save disable and save interrupt state first \*/

static inline void spin\_lock\_save(

spin\_lock\_t \*lock,

spin\_lock\_saved\_state\_t \*statep,

spin\_lock\_save\_flags\_t flags)

{

arch\_interrupt\_save(statep, flags);

spin\_lock(lock);

}

/\* restore interrupt state before unlocking \*/

static inline void spin\_unlock\_restore(

spin\_lock\_t \*lock,

spin\_lock\_saved\_state\_t old\_state,

spin\_lock\_save\_flags\_t flags)

{

spin\_unlock(lock);

arch\_interrupt\_restore(old\_state, flags);

}

/\* hand(ier) routines \*/

#define spin\_lock\_irqsave(lock, statep) spin\_lock\_save(lock, &(statep), SPIN\_LOCK\_FLAG\_INTERRUPTS)

#define spin\_unlock\_irqrestore(lock, statep) spin\_unlock\_restore(lock, statep, SPIN\_LOCK\_FLAG\_INTERRUPTS)

### Futexes

Futex，Fast Userspace muTEXes，作为linux下的一种快速同步（互斥）机制，已经存在了很长一段时间了（since linux 2.5.7）。

Magenta中，futex是一个底层同步设施，在它上面建设有pthread\_mutex\_t和pthread\_cond\_t.这样的方便编程的程序库。

A futex is a Fast Userspace muTEX. It is a low level synchronization primitive which is a building block for higher level APIs such as pthread\_mutex\_t and pthread\_cond\_t.

Futexes are designed to not enter the kernel or allocate kernel resources in the uncontested case.

#### 参考文献

* Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux <https://www.kernel.org/doc/ols/2002/ols2002-pages-479-495.pdf> , Hubertus Franke and Rusty Russell. This is the original white paper describing the Linux futex. It documents the history and design of the original implementation, prior (failed) attempts at creating a fast userspace synchronization primitive, and performance measurements.
* Futexes Are Tricky <https://www.akkadia.org/drepper/futex.pdf> , Ulrich Drepper This paper describes some gotchas and implementation details of futexes in Linux. It discusses the kernel implementation, and goes into more detail about correct and efficient userspace implementations of mutexes, condition variables, and so on.
* Mutexes and Condition Variables using Futexes <http://locklessinc.com/articles/mutex_cv_futex/> Further commentary on "Futexes are tricky", outlining a simple implementation that avoids the need for FUTEX\_CMP\_REQUEUE
* Locking in WebKit <https://webkit.org/blog/6161/locking-in-webkit/> , Filip Pizlo An in-depth tour of the locking primitives in WebKit, complete with benchmarks and analysis. Contains a detailed explanation of the "parking lot" concept, which allows very compact representation of user-space mutexes.

#### 与Linux的差别

源码：$Magenta/magenta/kernel/lib/magenta/include/magenta/futex\_context.h

源码：$Magenta/magenta/kernel/lib/magenta/futex\_context.cpp

与Linux上的futex的实现有差异，class FutexContext没有记录指向用户空间的虚拟地址的字段。Linux中靠这个字段区分进程内私有futex操作，还是跨进程的futex操作。

As noted above, all of our futex operations leave the value of the futex unmodified from the kernel. Other potential operations, such as Linux's FUTEX\_WAKE\_OP, requires atomic manipulation of the value from the kernel, which our current implementation does not require.

## 看门狗（watchdog）

源码：$Magenta/magenta/kernel/lib/watchdog/watchdog.c

## 调试debug机制

### LTRACEF

头文件：#include <trace.h>

//#define \_\_FUNC\_\_ \_\_PRETTY\_FUNCTION\_\_

#define \_\_FUNC\_\_ \_\_func\_\_

/\* trace routines \*/

#define TRACE\_ENTRY printf("%s: entry\n", \_\_FUNC\_\_)

#define TRACE\_EXIT printf("%s: exit\n", \_\_FUNC\_\_)

#define TRACE\_ENTRY\_OBJ printf("%s: entry obj %p\n", \_\_FUNC\_\_, this)

#define TRACE\_EXIT\_OBJ printf("%s: exit obj %p\n", \_\_FUNC\_\_, this)

#define TRACE printf("%s:%d\n", \_\_FUNC\_\_, \_\_LINE\_\_)

#define TRACEF(str, x...) do { printf("%s:%d: " str, \_\_FUNC\_\_, \_\_LINE\_\_, ## x); } while (0)

/\* trace routines that work if LOCAL\_TRACE is set \*/

#define LTRACE\_ENTRY do { if (LOCAL\_TRACE) { TRACE\_ENTRY; } } while (0)

#define LTRACE\_EXIT do { if (LOCAL\_TRACE) { TRACE\_EXIT; } } while (0)

#define LTRACE\_ENTRY\_OBJ do { if (LOCAL\_TRACE) { TRACE\_ENTRY\_OBJ; } } while (0)

#define LTRACE\_EXIT\_OBJ do { if (LOCAL\_TRACE) { TRACE\_EXIT\_OBJ; } } while (0)

#define LTRACE do { if (LOCAL\_TRACE) { TRACE; } } while (0)

#define LTRACEF(x...) do { if (LOCAL\_TRACE) { TRACEF(x); } } while (0)

#define LTRACEF\_LEVEL(level, x...) do { if (LOCAL\_TRACE >= (level)) { TRACEF(x); } } while (0)

### DEBUG\_ASSERT

头文件：#include <assert.h>

#if (LK\_DEBUGLEVEL > 1)

#define DEBUG\_ASSERT(x) \

do { \

if (unlikely(!(x))) { \

PANIC("DEBUG ASSERT FAILED at (%s:%d): %s\n", \_\_FILE\_\_, \_\_LINE\_\_, #x); \

} \

} while (0)

#define DEBUG\_ASSERT\_MSG(x, msg, msgargs...) \

do { \

if (unlikely(!(x))) { \

PANIC("DEBUG ASSERT FAILED at (%s:%d): %s\n" msg "\n", \

\_\_FILE\_\_, \_\_LINE\_\_, #x, ## msgargs); \

} \

} while (0)

### DEBUGLOG

把系统工作（如：起动）时的输出信息管理。

源码：$Magenta/magenta/kernel/lib/debuglog/include/lib/debuglog.h

源码：$Magenta/magenta/kernel/lib/debuglog/debuglog.c

#define DLOG\_SIZE (64 \* 1024)

static uint8\_t DLOG\_DATA[DLOG\_SIZE];

static dlog\_t DLOG = {

.lock = MUTEX\_INITIAL\_VALUE(DLOG.lock),

.size = DLOG\_SIZE,

.data = DLOG\_DATA,

.readers = LIST\_INITIAL\_VALUE(DLOG.readers),

};

### 系统控制台console

系统控制台。

源码：$Magenta/magenta/kernel/lib/io/console.c

/\* enable this to cause the kernel-originated messages to splat messages out of the platform

\* putc mechanism immediately instead of going through the debug log

\*/

#ifndef ENABLE\_KERNEL\_LL\_DEBUG

#define ENABLE\_KERNEL\_LL\_DEBUG 0

#endif

### The libunwind project

<http://www.nongnu.org/libunwind/index.html>

The primary goal of this project is to define a portable and efficient C programming interface (API) to determine the call-chain of a program. The API additionally provides the means to manipulate the preserved (callee-saved) state of each call-frame and to resume execution at any point in the call-chain (non-local goto). The API supports both local (same-process) and remote (across-process) operation. As such, the API is useful in a number of applications. Some examples include:

* exception handling

The libunwind API makes it trivial to implement the stack-manipulation aspects of exception handling.

* debuggers

The libunwind API makes it trivial for debuggers to generate the call-chain (backtrace) of the threads in a running program.

* introspection

It is often useful for a running thread to determine its call-chain. For example, this is useful to display error messages (to show how the error came about) and for performance monitoring/analysis.

* efficient setjmp()

With libunwind, it is possible to implement an extremely efficient version of setjmp(). Effectively, the only context that needs to be saved consists of the stack-pointer(s).

### libbacktrace

<https://github.com/ianlancetaylor/libbacktrace>

A C library that may be linked into a C/C++ program to produce symbolic backtraces

## 调优profile机制

### THREAD\_STATS

如果打开宏THREAD\_STATS，则系统运行过程中，收集线程运行中的统计信息。

源码：$Magenta/magenta/kernel/include/kernel/thread.h

struct thread\_stats {

lk\_bigtime\_t idle\_time;

lk\_bigtime\_t last\_idle\_timestamp;

ulong reschedules;

ulong context\_switches;

ulong preempts;

ulong yields;

ulong interrupts; /\* platform code increment this \*/

ulong timer\_ints; /\* timer code increment this \*/

ulong timers; /\* timer code increment this \*/

#if WITH\_SMP

ulong reschedule\_ipis;

#endif

};

### KTRACE

源码：$Magenta/magenta/kernel/lib/ktrace/ktrace.cpp

LK\_INIT\_HOOK(ktrace, ktrace\_init, LK\_INIT\_LEVEL\_APPS - 1);

源码：Magenta/magenta/kernel/lib/syscalls/syscalls\_debug.cpp

mx\_ssize\_t sys\_ktrace\_read(mx\_handle\_t handle, void\* ptr, uint32\_t off, uint32\_t len)

mx\_status\_t sys\_ktrace\_control(mx\_handle\_t handle, uint32\_t action, uint32\_t options)

程序中，写KTRACE示例，注意宏WITH\_LIB\_KTRACE的使用：

#if WITH\_LIB\_KTRACE

ktrace(TAG\_CONTEXT\_SWITCH, (uint32\_t)newthread->user\_tid, cpu | (oldthread->state << 16),

(uint32\_t)(uintptr\_t)oldthread, (uint32\_t)(uintptr\_t)newthread);

#endif

### KLOG（remove facilities）

*[kernel] remove evlog and klog facilities*

*Change-Id: Ifb3141c574b7ec451c53a13f4c83c509a60300e1*

源码：$Magenta/magenta/kernel/lib/klog/klog.c

STATIC\_COMMAND("klog", "commands for manipulating klog", &cmd\_klog)

### 内核事件EVLOG（remove facilities）

*[kernel] remove evlog and klog facilities*

*Change-Id: Ifb3141c574b7ec451c53a13f4c83c509a60300e1*

宏WITH\_KERNEL\_EVLOG需要打开，否则系统不记录这个log。

EVLOG包括如下种类的事件，程序通过这里定义的宏记录evlog：

源码：$Magenta/magenta/kernel/include/kernel/debug.h

enum {

KERNEL\_EVLOG\_NULL = 0,

KERNEL\_EVLOG\_CONTEXT\_SWITCH, //上下文切换

KERNEL\_EVLOG\_PREEMPT, //线程抢占

KERNEL\_EVLOG\_TIMER\_TICK, //定时器滴答声

KERNEL\_EVLOG\_TIMER\_CALL, //定时器回调函数执行

KERNEL\_EVLOG\_IRQ\_ENTER, //中断进入

KERNEL\_EVLOG\_IRQ\_EXIT, //中断退出

};

#define KEVLOG\_THREAD\_SWITCH(from, to) kernel\_evlog\_add(KERNEL\_EVLOG\_CONTEXT\_SWITCH, (uintptr\_t)from, (uintptr\_t)to)

#define KEVLOG\_THREAD\_PREEMPT(thread) kernel\_evlog\_add(KERNEL\_EVLOG\_PREEMPT, (uintptr\_t)thread, 0)

#define KEVLOG\_TIMER\_TICK() kernel\_evlog\_add(KERNEL\_EVLOG\_TIMER\_TICK, 0, 0)

#define KEVLOG\_TIMER\_CALL(ptr, arg) kernel\_evlog\_add(KERNEL\_EVLOG\_TIMER\_CALL, (uintptr\_t)ptr, (uintptr\_t)arg)

#define KEVLOG\_IRQ\_ENTER(irqn) kernel\_evlog\_add(KERNEL\_EVLOG\_IRQ\_ENTER, (uintptr\_t)irqn, 0)

#define KEVLOG\_IRQ\_EXIT(irqn) kernel\_evlog\_add(KERNEL\_EVLOG\_IRQ\_EXIT, (uintptr\_t)irqn, 0)

源码：$Magenta/magenta/kernel/kernel/debug.c

static int cmd\_kevlog(int argc, const cmd\_args \*argv)

源码：$Magenta/magenta/kernel/lib/evlog/evlog.c

status\_t evlog\_init(evlog\_t \*e, uint len, uint unitsize)

## 监控运行log机制

### LogDispatcher

内核调试信息记录与管理。

### 内核诊断信息diagnostics

源码：$Magenta/magenta/kernel/lib/magenta/diagnostics.cpp

STATIC\_COMMAND("mx", "magenta diagnostics", &cmd\_diagnostics)

# Magenta应用编程基础设施

## 进程间通信（IPC）

### 网络套接字（socket）

socket就是在不同计算机之间进行通信的一个抽象。他工作于TCP/IP协议中应用层和传输层之间的一个抽象。如图 13‑1所示。

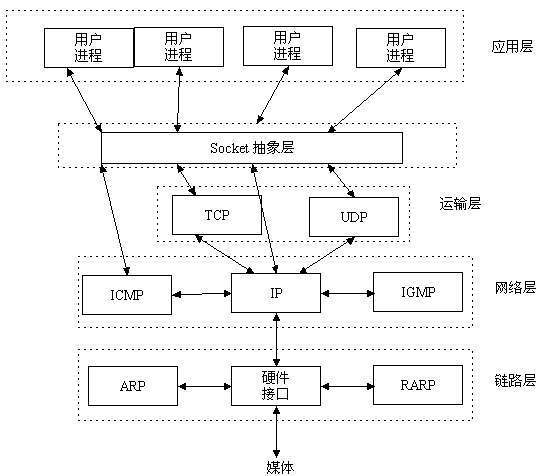


图 ‑1 socket示意图

#### OOB（out of band，带外数据）

Magenta的套接字机制支持低层协议发送和接受带外数据（out of band）。

传输层协议使用带外数据（out-of-band，OOB）来发送一些重要的数据，如果通信一方有重要的数据需要通知对方时，协议能够将这些数据快速地发送到对方。为了发送这些数据，协议一般不使用与普通数据相同的通道，而是使用另外的通道。

#### 半关闭（half-close）

终止一个连接要经过4次握手。这由TCP的半关闭（half-close）造成的。既然一个TCP连接是全双工（即数据在两个方向上能同时传递，可理解为两个方向相反的独立通道），因此每个方向必须单独地进行关闭。这原则就是当一方完成它的数据发送任务后就能发送一个FIN来终止这个方向连接。当一端收到一个FIN，内核让read返回0来通知应用层另一端已经终止了向本端的数据传送。发送FIN通常是应用层对socket进行关闭的结果。例如：TCP客户端发送一个FIN，用来关闭从客户到服务器的数据传送。

### 管道（Pipes）

管道通信（Communication Pipeline）即发送进程以[字符](http://baike.baidu.com/view/263416.htm)流形式将大量数据送入管道，接收进程可从管道接收数据，二者利用管道进行通信。

#### 消息管道（Message Pipes）

#### 数据管道（Data Pipes）

#### 等待集（Wait Sets）

# 调试Magenta

## 用QEMU+gdb调试Magenta

### 通过QEMU跑起来Magenta。

参考：

<http://jingxuanzhang.elastos.org/2015/02/02/qemu%E4%BB%BF%E7%9C%9F%E4%B8%8Egdb%E8%BF%9C%E7%A8%8B%E8%B0%83%E8%AF%95/>

xilong@xilong-OptiPlex-7010:~/swap/Magenta/magenta$ ./scripts/run-magenta-arm32 -- -s -S

[... some Qemu start up text ...]

坑1：

Note that this library provides an implementation for the weak symbol

"platform\_reset". It also assumes that r0 contains the size of memory reserved

for the secure OS.

### 起调试器arm-none-eabi-gdb

xilong@xilong-OptiPlex-7010:~/swap/Magenta/magenta$ arm-none-eabi-gdb

GNU gdb (Sourcery CodeBench Lite 2013.11-24) 7.6.50.20130726-cvs

Copyright (C) 2013 Free Software Foundation, Inc.

License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>

This is free software: you are free to change and redistribute it.

There is NO WARRANTY, to the extent permitted by law. Type "show copying"

and "show warranty" for details.

This GDB was configured as "--host=i686-pc-linux-gnu --target=arm-none-eabi".

Type "show configuration" for configuration details.

For bug reporting instructions, please see:

<https://sourcery.mentor.com/GNUToolchain/>.

Find the GDB manual and other documentation resources online at:

<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".

Type "apropos word" to search for commands related to "word".

(gdb) target remote:1234

Remote debugging using :1234

0x40010000 in ?? ()

(gdb) file /home/xilong/swap/Magenta/magenta/build-magenta-qemu-arm32/magenta.elf

A program is being debugged already.

Are you sure you want to change the file? (y or n) y

Reading symbols from /home/xilong/swap/Magenta/magenta/build-magenta-qemu-arm32/magenta.elf...done.

(gdb) ni

0x40010024 in ?? ()

(gdb) x/10i $pc

=> 0x40010024: mrc 15, 0, r12, cr1, cr0, {0}

0x40010028: bic r12, r12, #4096 ; 0x1000

0x4001002c: bic r12, r12, #5

0x40010030: orr r12, r12, #4096 ; 0x1000

0x40010034: orr r12, r12, #4

0x40010038: mcr 15, 0, r12, cr1, cr0, {0}

0x4001003c: ldr r4, [pc, #672] ; 0x400102e4

0x40010040: sub r11, pc, #12

0x40010044: sub r11, r11, r4

0x40010048: mrc 15, 0, r12, cr0, cr0, {5}

(gdb)

shell2$ gdb build-magenta-pc-x86-64/magenta.elf

(gdb) target extended-remote :1234

Remote debugging using :1234

0x000000000000fff0 in ?? ()

(gdb) # Don't try to do too much at this point.

(gdb) # GDB can't handle architecture switching in one session,

(gdb) # and at this point the architecture is 16-bit x86.

(gdb) break lk\_main

Breakpoint 1 at 0xffffffff8010cb58: file kernel/top/main.c, line 59.

(gdb) continue

Continuing.

Breakpoint 1, lk\_main (arg0=1, arg1=18446744071568293116, arg2=0, arg3=0)

at kernel/top/main.c:59

59 {

(gdb) continue

#### The magenta.elf-gdb.py script

The scripts/magenta.elf-gdb.py script is automagically loaded by gdb. It provides several things:

Pretty-printers for magenta objects (alas none at the moment).

Several magenta specific commands, all with a "magenta" prefix. To see them:

(gdb) help info magenta

(gdb) help set magenta

(gdb) help show magenta

Enhanced unwinder support for automagic unwinding through kernel faults.

Heads up: This script isn't always updated as magenta changes.

#### Terminating the session

To terminate Qemu you can send commands to Qemu from GDB:

(gdb) monitor quit

(gdb) quit

Interacting with Qemu from Gdb

#### To see the list of Qemu commands you can execute from GDB:

(gdb) monitor help

### QEMU跑操作系统的第一条指令

QEMU doesn’t provide NOR flash emulation on the goldfish platform. To make things simple, we will use RAM to create a boot-up process that is similar to the boot process from NOR flash. This approach builds a binary image that includes U-Boot, the Linux kernel, and the RAMDISK image and passes this image to QEMU through the –kernel option.

Before we start, let’s look at how QEMU boots a Linux kernel. To boot up a Linux kernel, the bootloader prepares the following environment:

* The processor is in SVC (Supervisor) mode and IRQ and FIQ are disabled.
* MMU is disabled.
* Register r0 is set to 0.
* Register r1 contains the ARM Linux machine type.
* Register r2 contains the address of the kernel parameter list.

After power-up, QEMU starts to run from address 0x00000000. Before it loads a kernel image, QEMU prepares the environment described previously; it then jumps to address 0x00010000. [Figure 10.4](javascript:popUp('/content/images/chap10_9780134030005/elementLinks/10fig04_alt.jpg'))shows a memory dump before the point at which QEMU launches a kernel image. Notice the five lines of assembly code before control is transferred to the kernel image—these lines are hard-coded by QEMU when the system starts. The first line (0x00000000) sets register r0 to 0. The second line (0x00000004) and third line (0x00000008) set register r1 to 0x5a1, which is the machine type of the goldfish platform. The fourth line (0x0000000c) sets the value of register r2 to 0x100, which is the start address of the kernel parameter list. The fifth line (0x00000010) sets the register pc to 0x10000, so the execution jumps to address 0x10000. QEMU assumes the kernel image is loaded at address 0x10000.

### 经验总结

用gdb远程调试操作系统Magenta，gdb这端的地址表示可能与remote的不一致。因为当MMU还没有工作时，被调试的操作系统还是跑的物理地址，程序都是基于物理地址进行寻址的，而你的gdb这边，它不知道，它按照ELF里面的信息提示给你。这设了的断点无法工作。怎么办才能把断点设到合适的位置上？

#define MEMBASE 0X40000000

#define MEMSIZE 0X08000000

#define KERNEL\_BASE 0X80000000

#define KERNEL\_LOAD\_OFFSET 0X10000

## 用 Python 拓展 GDB

### 如何用Python 拓展 GDB

参考：<http://python.jobbole.com/85415/>

用 Python 拓展 GDB（一） <https://segmentfault.com/a/1190000005718889>

python脚本完成了，该怎么运行呢？在gdb里使用python脚本，需要用source命令：

(gdb) so ~/move.py

(gdb) mv 1 binary\_search.cpp:18

我们最后把自定义命令的实现放到~/.gdbinit里面。这样gdb每次启动时就会运行它，而无需手动source。直接把python代码放进~/.gdbinit当然是不行的。需要变通一下，在~/.gdbinit加入source ~/move.py。这样gdb每次启动时都会替我们source一下。

有两点需要注意的是：

* gdb会用python 3来解释你的python脚本，除非你用的gdb还处于版本感人的上古时代。
* 跟一般情况不同，gdb环境中的sys.path是不包括当前目录的。这意味着，如果你的脚本依赖于当前目录下的其他模块，你需要手工修改sys.path。比如(gdb) python import sys; sys.path.append('')

### Magenta中的python调试角本

ELinOS is a cross development platform for either Linux or Windows host allowing you to even use your Windows desktop for the development of sophisticated embedded Linux applications.

ELinOS is a commercial development environment for embedded Linux. It consists of a Linux distribution for the target embedded system and development tools for a development host computer. The development host computer usually is a standard desktop computer running Linux or Windows. The Linux system and the application software for the target device are both created on the development host.

ELinOS focuses on industrial applications. For that purpose optional extensions for the Linux kernel are provided. For example, real-time patches from OSADL can be used to improve timing behavior. The kernel will always be specifically compiled for the embedded system. Userspace applications are provided as precompiled binaries in order to save time for compilation. Most of the ELinOS software packages are open source and licensed under the GPL.

ELinOS provides embedded Linux as a standalone operating system or it can be integrated into the PikeOS virtualization platform if safety and security demands cannot be met by Linux alone. The latter might be the case if a certification according to a strict industry standard is required.

Eclipse is the technical basis for the functionality provided by the integrated development environment (IDE) on the development host. ELinOS was first published in 1999.

角本：Magenta/magenta/prebuilt/downloads/arm-eabi-5.3.0-Linux-x86\_64/share/gdb/system-gdbinit/elinos.py

"""Configure GDB using the ELinOS environment."""

WRS/Linux是美国 Wind River System 公司（WRS 公司）出的Linux版本。Wind River Linux provides the exceptional security and flexibility necessary for meeting the dynamic performance, reliability, and scaling needs of the Internet of Things (IoT), helping safely and securely tap into new sources of data for operational improvements and business transformation.

网址：<http://www.windriver.com/products/linux/>

角本：Magenta/magenta/prebuilt/downloads/arm-eabi-5.3.0-Linux-x86\_64/share/gdb/system-gdbinit/wrs-linux.py

"""Configure GDB using the WRS/Linux environment."""

## GCC程序调试信息

#### GCC -g选项控制GDB调试级别

如果不打开-g或者-ggdb(GDB专用)调试开关，GCC编译时不会加入调试信息，因为这会增大生成代码的体积。GCC采用了分级调试，通过在-g选项后附加数字1、2或3来指定在代码中加入调试信息量。默认的级别是2（-g2），此时调试信息包括扩展的符号表、行号、局部或外部变量信息。级别3（- g3）包含级别2中的调试信息和源代码中定义的宏。级别1（-g1）不包含局部变量和与行号有关的调试信息，只能用于回溯跟踪和堆栈转储之用。

回溯跟踪指的是监视程序在运行过程中的函数调用历史，堆栈转储则是一种以原始的十六进制格式保存程序执行环境的方法，两者都是经常用到的调试手段。

#### dwarf调试信息格式入门

参考：<http://www.cnblogs.com/zqingnn/archive/2011/01/05/1926384.html>

那么调试器使用的调试信息是从哪里来的呢？答案简单的很，是从编译后的文件中来的(注意这里编译的时候要使用特定的编译选项，如VC使用debug模式，GCC使用”-g”)。在编译的时候，编译器会从源文件中收集大量的信息，例如变量名、变量类型、变量所在行号、函数名、函数参数、函数的地址范围、行号和地址的对应关系等等，然后按照一种特定的格式写入到编译后的文件中。调试的时候，调试器便从文件中读取并解析这些信息，以产生人们可读性比较强的信息。简单的说，调试信息就是在机器码和对应的源代码之间建立一座桥梁，大大方便和提高了调试程序的能力。

调试信息一般都是按照什么样的格式存放的呢？主要有下面几种：stabs，COFF，PE-COFF，OMF，IEEE-695和DWARF。其中DWARF在Linux中被普遍使用，我们主要分析它。

DWARF的全称是"Debugging With Attributed Record Formats"，遵从GNU FDL授权。现在已经有dwarf1，dwarf2，dwarf3三个版本。

Dwarf最初被贝尔实验室设计用来供Unix System V的sdb调试器使用，并且在1989年被Unix国际化部门的PLSIG (Programming Languages Special Interest Group)标准化成为dwarf1.0。但是dwarf1有着很多明显的缺点，于是PLSIG继续开发，改正了缺点，并加入了对C++等语言的支持，并在1990年正式公布了dwarf2的标准草案。但是稍后由于一些原因，PLSIG被解散，dwarf的开发陷入到多个并不合作的组织中间，造成dwarf2的一些实现细节要取决于特定的编译器。这种情况一直持续到1999年，开发工作受到了来自实现对HP/Inter IA-64架构提供较好支持的推动，成立了dwarf委员会，dwarf的原作者担任负责人，

开始了dwarf3的开发，并于2006年1月份推出dwarf3.0，同时为了解决分歧，dwarf委员会加入了自由标准组织，在自由标准组织与来自Linux基金会的OSDL(Open Source Development Labs)合并后，dwarf重返独立状态并创建了自己的网站：dwarfstd.org。

# Magenta系统功能调用

## 句柄Handles

### handle\_close

#### 简述

关闭一个句柄。

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t mx\_handle\_close(mx\_handle\_t handle);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| handle | 句柄 |

#### 功能

handle\_close()关闭一个句柄，这将引起它对应的句柄资源的释放——引用计算减一，如果引用计数减至0，则对象被释放。

#### 返回值

成功，handle\_close()返回NO\_ERROR；失败，一个代表错误值的负数。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_INVALID\_ARGS(-10) | 无效句柄 |

### handle\_duplicate

#### 简述

复制一个句柄。

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t mx\_handle\_duplicate(mx\_handle\_t handle, mx\_rights\_t rights, mx\_handle\_t\* out);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| handle | 句柄 |
| rights | * 无权 * MX\_RIGHT\_SAME\_RIGHTS(1u<<31)，与原句柄权限相同 * 其它值，必须比原句柄权限小 |
| out | 输出参数，返回的句柄 |

#### 功能

handle\_duplicate()复制一个句柄，这个句柄指向原句柄handle对应的同一个对象，但是权限是新的rights表示的权限。

#### 返回值

成功，handle\_duplicate()返回NO\_ERROR；失败，一个代表错误值的负数。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_BAD\_HANDLE(-12) | 源句柄无效 |
| ERR\_INVALID\_ARGS(-10) | 无效句柄 |
| ERR\_ACCESS\_DENIED | 源句柄没有MX\_RIGHT\_DUPLICATE权限 |
| ERR\_NO\_MEMORY (-4) | 没内存 |

### handle\_replace

#### 简述

替换一个句柄。

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t mx\_handle\_replace(mx\_handle\_t handle, mx\_rights\_t rights, mx\_handle\_t\* out);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| handle | 输入句柄 |
| rights | * 无权 * MX\_RIGHT\_SAME\_RIGHTS(1u<<31)，与原句柄权限相同 * 其它值，必须比原句柄权限小 |
| out | 返回句柄 |

#### 功能

handle\_duplicate()替换一个句柄，这个句柄指向原句柄handle对应的同一个对象，但是权限是新的rights表示的权限。

返回句柄的value\_在函数执行过程中，临时赋值为MX\_HANDLE\_INVALID，这使得out参数所指空间与handle参数所指空间相同时，能够被区分。

#### 返回值

成功，handle\_duplicate()返回NO\_ERROR；失败，一个代表错误值的负数。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_BAD\_HANDLE(-12) | 源句柄无效 |
| ERR\_INVALID\_ARGS(-10) | 无效句柄 |
| ERR\_NO\_MEMORY (-4) | 没内存 |

### handle\_wait\_many

#### 简述

等待多个句柄上的信号（signal）。

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t mx\_handle\_wait\_many(mx\_wait\_item\_t\* items, uint32\_t count, mx\_time\_t timeout);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| items | 需要等待的项目描述 |
| count | 句柄数，items中的项数。最大值：kMaxWaitHandleCount(256u) |
| timeout | 超时值，从当前开始时的纳秒nanosecond，0表示立即返回，MX\_TIME\_INFINITE(UINT64\_MAX)则表示无限期等待。 |

typedef struct {

mx\_signals\_t satisfied;

mx\_signals\_t satisfiable;

} mx\_signals\_state\_t;

typedef uint32\_t mx\_signals\_t;

typedef struct {

mx\_handle\_t handle;

mx\_signals\_t waitfor;

mx\_signals\_t pending;

} mx\_wait\_item\_t;

#### 功能

handle\_wait\_many()等待句柄上的信号直到超时，这些句柄上至少有一个产生了信号，则返回。

如果在本函数返回过程中，句柄的状态发生改变，例如被其它线程修改状态，pending字段中的状态可能与句柄当前的状态不一致。

#### 返回值

成功，handle\_wait\_one()返回NO\_ERROR。

当超时（返回值：ERR\_TIMED\_OUT）的情况下返回时，items中可能存有超时后，但函数返回前的状态。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_INVALID\_ARGS(-10) | 无效句柄 |
| ERR\_BAD\_HANDLE(-12) | 源句柄无效 |
| ERR\_ACCESS\_DENIED | 有的句柄没有读（MX\_RIGHT\_READ）权限 |
| ERR\_HANDLE\_CLOSED | 有的句柄无效，比如被关闭了 |
| ERR\_NO\_MEMORY (-4) | 没内存 |

源码：$Magenta/magenta/system/public/magenta/types.h

typedef uint32\_t mx\_signals\_t;

#define MX\_SIGNAL\_NONE ((mx\_signals\_t)0u)

#define MX\_OBJECT\_SIGNAL\_ALL ((mx\_signals\_t)0x00ffffff)

#define MX\_USER\_SIGNAL\_ALL ((mx\_signals\_t)0xff000000)

#define MX\_OBJECT\_SIGNAL\_0 ((mx\_signals\_t)1u << 0)

#define MX\_OBJECT\_SIGNAL\_1 ((mx\_signals\_t)1u << 1)

#define MX\_OBJECT\_SIGNAL\_2 ((mx\_signals\_t)1u << 2)

#define MX\_OBJECT\_SIGNAL\_3 ((mx\_signals\_t)1u << 3)

#define MX\_OBJECT\_SIGNAL\_4 ((mx\_signals\_t)1u << 4)

#define MX\_OBJECT\_SIGNAL\_5 ((mx\_signals\_t)1u << 5)

#define MX\_OBJECT\_SIGNAL\_6 ((mx\_signals\_t)1u << 6)

#define MX\_OBJECT\_SIGNAL\_7 ((mx\_signals\_t)1u << 7)

#define MX\_OBJECT\_SIGNAL\_8 ((mx\_signals\_t)1u << 8)

#define MX\_OBJECT\_SIGNAL\_9 ((mx\_signals\_t)1u << 9)

#define MX\_OBJECT\_SIGNAL\_10 ((mx\_signals\_t)1u << 10)

#define MX\_OBJECT\_SIGNAL\_11 ((mx\_signals\_t)1u << 11)

#define MX\_OBJECT\_SIGNAL\_12 ((mx\_signals\_t)1u << 12)

#define MX\_OBJECT\_SIGNAL\_13 ((mx\_signals\_t)1u << 13)

#define MX\_OBJECT\_SIGNAL\_14 ((mx\_signals\_t)1u << 14)

#define MX\_OBJECT\_SIGNAL\_15 ((mx\_signals\_t)1u << 15)

#define MX\_OBJECT\_SIGNAL\_16 ((mx\_signals\_t)1u << 16)

#define MX\_OBJECT\_SIGNAL\_17 ((mx\_signals\_t)1u << 17)

#define MX\_OBJECT\_SIGNAL\_18 ((mx\_signals\_t)1u << 18)

#define MX\_OBJECT\_SIGNAL\_19 ((mx\_signals\_t)1u << 19)

#define MX\_OBJECT\_SIGNAL\_20 ((mx\_signals\_t)1u << 20)

#define MX\_OBJECT\_SIGNAL\_21 ((mx\_signals\_t)1u << 21)

#define MX\_OBJECT\_SIGNAL\_22 ((mx\_signals\_t)1u << 22)

#define MX\_OBJECT\_SIGNAL\_23 ((mx\_signals\_t)1u << 23)

#define MX\_USER\_SIGNAL\_0 ((mx\_signals\_t)1u << 24)

#define MX\_USER\_SIGNAL\_1 ((mx\_signals\_t)1u << 25)

#define MX\_USER\_SIGNAL\_2 ((mx\_signals\_t)1u << 26)

#define MX\_USER\_SIGNAL\_3 ((mx\_signals\_t)1u << 27)

#define MX\_USER\_SIGNAL\_4 ((mx\_signals\_t)1u << 28)

#define MX\_USER\_SIGNAL\_5 ((mx\_signals\_t)1u << 29)

#define MX\_USER\_SIGNAL\_6 ((mx\_signals\_t)1u << 30)

#define MX\_USER\_SIGNAL\_7 ((mx\_signals\_t)1u << 31)

// Event

#define MX\_EVENT\_SIGNAL\_SIGNALED MX\_OBJECT\_SIGNAL\_3

#define MX\_EVENT\_SIGNAL\_MASK (MX\_USER\_SIGNAL\_ALL | MX\_OBJECT\_SIGNAL\_3)

// EventPair

#define MX\_EPAIR\_SIGNAL\_SIGNALED MX\_OBJECT\_SIGNAL\_3

#define MX\_EPAIR\_SIGNAL\_CLOSED MX\_OBJECT\_SIGNAL\_2

#define MX\_EPAIR\_SIGNAL\_MASK (MX\_USER\_SIGNAL\_ALL | MX\_OBJECT\_SIGNAL\_2 | MX\_OBJECT\_SIGNAL\_3)

// Task signals

#define MX\_TASK\_SIGNAL\_TERMINATED MX\_OBJECT\_SIGNAL\_3

#define MX\_TASK\_SIGNAL\_MASK MX\_OBJECT\_SIGNAL\_3

// Legacy signal names, to be removed.

#define MX\_SIGNAL\_READABLE MX\_OBJECT\_SIGNAL\_0

#define MX\_SIGNAL\_WRITABLE MX\_OBJECT\_SIGNAL\_1

#define MX\_SIGNAL\_PEER\_CLOSED MX\_OBJECT\_SIGNAL\_2

#define MX\_SIGNAL\_SIGNALED MX\_OBJECT\_SIGNAL\_3

#define MX\_SIGNAL\_READ\_THRESHOLD MX\_OBJECT\_SIGNAL\_4

#define MX\_SIGNAL\_WRITE\_THRESHOLD MX\_OBJECT\_SIGNAL\_5

#define MX\_SIGNAL\_SIGNAL0 MX\_USER\_SIGNAL\_0

#define MX\_SIGNAL\_SIGNAL1 MX\_USER\_SIGNAL\_1

#define MX\_SIGNAL\_SIGNAL2 MX\_USER\_SIGNAL\_2

#define MX\_SIGNAL\_SIGNAL3 MX\_USER\_SIGNAL\_3

#define MX\_SIGNAL\_SIGNAL4 MX\_USER\_SIGNAL\_4

### handle\_wait\_one

#### 简述

等待一个句柄上的信号。

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t mx\_handle\_wait\_one(mx\_handle\_t handle,

mx\_signals\_t signals,

mx\_time timeout,

mx\_signals\_t\* pending);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| handle | 句柄 |
| signals | 信号 |
| timeout | 超时，从当前开始的纳秒数，0表示立即返回，MX\_TIME\_INFINITE表示无限期等待。 |
| pending | 返回参数。如果函数返回时非NULL，则是一个比特图（bitmap），存有挂起在指定句柄上的信号状态。 |

#### 功能

handle\_wait\_one()调用将引起调用者阻塞，直到至少一个指定的信号被挂起（pending）在指定的句柄，或者超时。

如果在本函数返回过程中，句柄的状态发生改变，例如被其它线程修改状态，pending中的状态可能与句柄当前的状态不一致。

#### 返回值

成功，handle\_wait\_one()返回NO\_ERROR。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_BAD\_HANDLE(-12) | 源句柄无效 |
| ERR\_INVALID\_ARGS(-10) | 无效句柄 |
| ERR\_ACCESS\_DENIED(-30) | 句柄没有读权限MX\_RIGHT\_READ |
| ERR\_HANDLE\_CLOSED (-24) | 句柄在等待过程中失效，比如句柄被关闭。 |

## Objects

### object\_signal

#### 简述

向一个句柄发信号

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t mx\_object\_signal(mx\_handle\_t handle, uint32\_t clear\_mask, uint32\_t set\_mask);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| handle |  |
| clear\_mask |  |
| set\_mask |  |

#### 功能

mx\_object\_signal\_peer() 向一个内核对象发信号。

### object\_signal\_peer

#### 简述

向一个句柄的对等对象发信号

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t mx\_object\_signal\_peer(mx\_handle\_t handle, uint32\_t clear\_mask, uint32\_t set\_mask);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| handle |  |
| clear\_mask |  |
| set\_mask |  |

#### 功能

mx\_object\_signal() 向一个内核对象发信号。

### object\_get\_info

#### 简述

取一个句柄的信息

#### 函数原型

mx\_ssize\_t sys\_object\_get\_info(mx\_handle\_t handle, uint32\_t topic, uint16\_t topic\_size,

user\_ptr<void> \_buffer, mx\_size\_t buffer\_size)

// Valid topics for mx\_object\_get\_info.

typedef enum {

MX\_INFO\_HANDLE\_VALID = 1,

MX\_INFO\_HANDLE\_BASIC,

MX\_INFO\_PROCESS,

MX\_INFO\_PROCESS\_THREADS,

} mx\_object\_info\_topic\_t;

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| handle |  |
| topic | * MX\_INFO\_HANDLE\_VALID：句柄是否有效。有效，返回NO\_ERROR; 无效，返回ERR\_BAD\_HANDLE。 * MX\_INFO\_HANDLE\_BASIC：句柄基本信息 * MX\_INFO\_PROCESS * MX\_INFO\_PROCESS\_THREADS |
| topic\_size | 要请求句柄的信息的记录大小   * MX\_INFO\_HANDLE\_BASIC：sizeof(mx\_record\_handle\_basic\_t) * MX\_INFO\_PROCESS：sizeof(mx\_record\_process\_t) * MX\_INFO\_PROCESS\_THREADS：sizeof(mx\_record\_process\_thread\_t) |
| buffer |  |
| buffer\_size |  |

typedef struct mx\_record\_handle\_basic {

mx\_koid\_t koid;

mx\_rights\_t rights;

uint32\_t type; // mx\_obj\_type\_t;

uint32\_t props; // mx\_obj\_props\_t;

} mx\_record\_handle\_basic\_t;

// Returned for topic MX\_INFO\_HANDLE\_BASIC

typedef struct mx\_info\_handle\_basic {

mx\_info\_header\_t hdr;

mx\_record\_handle\_basic\_t rec;

} mx\_info\_handle\_basic\_t;

#### 功能

object\_get\_info() 取一个句柄的信息。

### object\_get\_property

#### 简述

取一个句柄的属性

#### 函数原型

mx\_status\_t sys\_object\_get\_property(mx\_handle\_t handle\_value, uint32\_t property,

user\_ptr<void> \_value, mx\_size\_t size)

// Object properties.

// Argument is MX\_POLICY\_BAD\_HANDLE\_... (below, uint32\_t).

#define MX\_PROP\_BAD\_HANDLE\_POLICY 1u

// Argument is a uint32\_t.

#define MX\_PROP\_NUM\_STATE\_KINDS 2u

// Argument is an mx\_size\_t.

#define MX\_PROP\_DATAPIPE\_READ\_THRESHOLD 3u

// Argument is an mx\_size\_t.

#define MX\_PROP\_DATAPIPE\_WRITE\_THRESHOLD 4u

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| handle |  |
| property |  |

#### 功能

object\_get\_property() 取一个句柄的信息。

### object\_set\_property

mx\_status\_t sys\_object\_set\_property(mx\_handle\_t handle\_value, uint32\_t property,

user\_ptr<const void> \_value, mx\_size\_t size)

### object\_bind\_exception\_port

- attach an exception port to a task

### object\_get\_child

- find the child of an object by its koid

## 作业（Job）

### job\_create

- create a new job within a job

## 任务（Task, Process, or Job）

### task\_resume

- cause a suspended task to continue running

### task\_kill

- cause a task to stop running

## 消息管道（Channel）

管道通信是消息传递的一种特殊方式。所谓“管道”，是指用于连接一个“读”和一个“写”以实现它们之间通信的一个机制。向管道提供输入的发送者，以字符流形式将大量的数据送入（写）管道；而接收管道输出的接收者，则从管道中接收（读）数据。

为了协调双方的通信，管道机制必须提供以下三方面的协调能力：互斥、同步和确定对方的存在。

### channel\_create

#### 简述

创建一个消息管道。

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t mx\_channel\_create(uint32\_t flags,

mx\_handle\_t\* out0, mx\_handle\_t\* out1);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| flags | 0 or MX\_FLAG\_REPLY\_PIPE((1u << 0)) |
| out0 | 句柄 |
| out1 | 句柄 |

#### 功能

channel\_create()创建一个消息管道，一个双向数据报（bi-directional datagram-style）风格的数据传输机制。

句柄将要有下列权限：

|  |  |
| --- | --- |
| 权限 | 能力 |
| MX\_RIGHT\_TRANSFER | 允许这个句柄被通过消息管道发送到其它进程 |
| MX\_RIGHT\_WRITE | 允许数据包被队列化 |
| MX\_RIGHT\_READ | 允许读数据包 |

constexpr mx\_rights\_t kDefaultIOPortRights =

MX\_RIGHT\_DUPLICATE | MX\_RIGHT\_TRANSFER | MX\_RIGHT\_READ | MX\_RIGHT\_WRITE;

A reply pipe behaves like a regular message pipe except for mx\_channel\_write() which must include itself as the last handle being transfered.

当标志flags=MX\_FLAG\_REPLY\_PIPE时，只有out1 是应答管道，out0是正常管道。

#### 返回值

成功，channel\_create() 返回NO\_ERROR；失败，一个代表错误值的负数。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_INVALID\_ARGS(-10) | 句柄数组是NULL |
| ERR\_NO\_MEMORY (-4) | 没内存 |

#### 程序示例

mx\_handle\_t h0, h1;

mx\_status\_t r;

if ((r = mx\_channel\_create(h, &h0, &h1)) < 0) {

…… ……

}

h[0], h[1]中有可供后续操作的句柄

### channel\_read

#### 简述

从消息管道中读消息。

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t mx\_channel\_read(mx\_handle\_t handle, uint32\_t flags,

void\* bytes,

uint32\_t num\_bytes, uint32\_t\* actual\_bytes

mx\_handle\_t\* handles,

uint32\_t num\_handles, uint32\_t\* actual\_handles);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| flags | （好象没用） |
| handle | 句柄 |
| bytes | 要读出的字节，out参数 |
| num\_bytes | 输入时：bytes空间大小，字节为单位。NULL表示不想读bytes。  输出时：读出的字节的数量 |
| handles | 读出的句柄，out参数 |
| num\_handles | 输入时：handles空间大小，句柄数。注意：不是字节数。NULL表示不想读handles。  输出时：读出的句柄的数量 |

#### 功能

channel\_read()试图从消息队列中读出它的第一个消息到bytes或者handles中。

一次必须把一个消息报读完，不可以读消息报的一部分。

#### 返回值

成功，channel\_read() 返回NO\_ERROR；失败，一个代表错误值的负数。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_INVALID\_ARGS(-10) | 参数错。   * handle不是有效的句柄 * num\_bytes如果不是NULL，不是有效地址；或者num\_bytes是NULL，而bytes不是NULL * num\_handles如果不是NULL，不是有效地址；或者num\_handles是NULL，而handles不是NULL |
| ERR\_ACCESS\_DENIED | 句柄没有读权限（MX\_RIGHT\_READ） |
| ERR\_BAD\_STATE | 管道内没消息 |
| ERR\_REMOTE\_CLOSED |  |
| ERR\_NO\_MEMORY (-4) | 没内存 |
| ERR\_NOT\_ENOUGH\_BUFFER | 缓冲区太小。就是你给的bytes或handles的空间太小，不足以容纳消息中的内容 |

### channel\_write

#### 简述

向消息管道中写消息。

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t mx\_channel\_write(mx\_handle\_t handle, uint32\_t flags,

void\* bytes, uint32\_t num\_bytes,

mx\_handle\_t\* handles, uint32\_t num\_handles);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| handle | 句柄 |
| flags | （好象没用） |
| bytes | 要写出的字节 |
| num\_bytes | bytes空间大小，字节为单位 |
| handles | 要写的句柄 |
| num\_handles | handles空间大小，句柄数。注意：不是字节数。 |

#### 功能

channel\_write()试图向消息队列写一个消息，消息的内容在bytes和/或者handles中。

#### 返回值

成功，channel\_write() 返回NO\_ERROR；失败，一个代表错误值的负数。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_INVALID\_ARGS(-10) | 参数错。   * handle不是有效的句柄 * num\_bytes如果不是NULL，不是有效地址；或者num\_bytes是NULL，而bytes不是NULL * num\_handles如果不是NULL，不是有效地址；或者num\_handles是NULL，而handles不是NULL |
| ERR\_NOT\_SUPPORTED |  |
| ERR\_ACCESS\_DENIED | 句柄没有写权限（MX\_RIGHT\_WRITE） |
| ERR\_BAD\_STATE | 管道的另一端已经关闭或者管道是一个应答管道（reply pipe） |
| ERR\_REMOTE\_CLOSED | 消息中有句柄，但是这个管道不是应答管道（reply pipe） |
| ERR\_NO\_MEMORY (-4) | 没内存 |
| ERR\_TOO\_BIG | 要传的消息太大。  kMaxMessageSize(65536u)  kMaxMessageHandles (1024u) |

## 数据管道（Data Pipe）

示例程序：$Magenta/magenta/system/utest/core/data-pipe/data-pipe.c

### datapipe\_create

#### 简述

创建一个数据管道。

#### 函数原型

#include <magenta/syscalls.h>

mx\_handle\_t mx\_datapipe\_create(uint32\_t options,

mx\_size\_t element\_size,

mx\_size\_t capacity,

mx\_handle\_t\* consumer\_handle);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| options |  |
| element\_size | >=1，字节为单位 |
| capacity | capacity % element\_size != 0u，字节为单位  0表示使用系统缺省管道大小  capacity=kDefaultDataPipeCapacity-(kDefaultDataPipeCapacity % element\_size);  if (capacity==0)  capacity = element\_size; |
| consumer\_handle | out参数，靠它返回消费者句柄 |

constexpr mx\_size\_t kDefaultDataPipeCapacity = 32 \* 1024u;

#### 功能

datapipe\_create()创建一个适用于无格式数据传输的单向数据管道。

成功则返回句柄，这个句柄将要有下列权限：

|  |  |
| --- | --- |
| 权限 | 能力 |
| MX\_RIGHT\_TRANSFER | 允许这个句柄被通过管道发送到其它进程 |
| MX\_RIGHT\_WRITE | 允许数据包被队列化 |
| MX\_RIGHT\_READ | 允许读数据包 |

通过参数返回的consumer\_handle将有下列权限：

|  |  |
| --- | --- |
| 权限 | 能力 |
| MX\_RIGHT\_TRANSFER | 允许这个句柄被通过消息管道发送到其它进程 |
| MX\_RIGHT\_READ | 允许读数据包 |

#### 返回值

成功，datapipe\_create() 返回一个有效的数据管道句柄（一个正整数）；失败，一个代表错误值的负数。不可能返回代表无效句柄的0（Zero，the "invalid handle"）。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_INVALID\_ARGS(-10) | 参数错 |
| ERR\_NO\_MEMORY (-4) | 没内存 |

### datapipe\_read

#### 简述

从数据管道中读数据。

#### 函数原型

#include <magenta/syscalls.h>

mx\_ssize\_t mx\_datapipe\_read(mx\_handle\_t consumer\_handle,

uint32\_t flags,

mx\_size\_t requested,

void\* buffer);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| consumer\_handle | 句柄 |
| flags | 函数行为 |
| requested |  |
| buffer |  |

// DISCARD, QUERY, and PEEK are mutually exclusive.

#define MX\_DATAPIPE\_READ\_FLAG\_ALL\_OR\_NONE 1u

#define MX\_DATAPIPE\_READ\_FLAG\_DISCARD 2u

#define MX\_DATAPIPE\_READ\_FLAG\_QUERY 4u

#define MX\_DATAPIPE\_READ\_FLAG\_PEEK 8u

// Mask for all the valid MX\_DATAPIPE\_READ\_FLAG\_... flags:

#define MX\_DATAPIPE\_READ\_FLAG\_MASK 15u

#### 功能

datapipe\_read()从数据管道中读出数据，具体行为（read/discarded/available/peeked）依赖于参数flags。

There are three mutually-exclusive mode flags that may be set in flags: MX\_DATAPIPE\_READ\_FLAG\_DISCARD, MX\_DATAPIPE\_READ\_FLAG\_QUERY, and MX\_DATAPIPE\_READ\_FLAG\_PEEK, which indicate the desired mode (none of these flags being set means "read"). If MX\_DATAPIPE\_READ\_FLAG\_ALL\_OR\_NONE is set, then either the entire requested operation is performed, or not at all.

In read mode, requested indicates the number of bytes (which must be a multiple of the element size) to be read (to buffer, which should have room for the requested number of bytes). On success, the number of bytes actually read is returned. Unless MX\_DATAPIPE\_READ\_FLAG\_ALL\_OR\_NONE is set in flags, this number may be less than requested (but will always be a multiple of the element size).

In discard mode, requested indicates the number of bytes to discard and the return value indicates the number of bytes actually discarded (with MX\_DATAPIPE\_READ\_FLAG\_ALL\_OR\_NONE operating as above). buffer is ignored in this case.

In query mode, both requested and buffer are ignored, as is the MX\_DATAPIPE\_READ\_FLAG\_ALL\_OR\_NONE flag. The return value is the number of bytes available to be read.

Peek模式类似于读（read）模式，只是它不象读模式那样，它在读完后不把数据从数据管道中移除。

#### 返回值

成功返回read/discarded/available/peeked的字节数；失败，一个代表错误值的负数。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_INVALID\_ARGS(-10) | 参数错。  flags has an invalid combination of flags set, requested is not a multiple of the data pipe's element size (and query mode is not specified), or buffer is an invalid pointer (or NULL). |
| ERR\_BAD\_HANDLE | consumer\_handle不是一个有效句柄 |
| ERR\_WRONG\_TYPE | consumer\_handle不是一个与DataPipe生产者有关联的句柄 |
| ERR\_ACCESS\_DENIED | 句柄没有读权限（MX\_RIGHT\_READ） |
| ERR\_NOT\_SUPPORTED | flag参数有非预期的标记 |
| ERR\_ALREADY\_BOUND | consumer\_handle is currently in a two-phase read. |
| ERR\_REMOTE\_CLOSED | (In read/discard/peek mode) The requested operation could not be performed (due to a lack of data) and the remote data pipe producer handle is closed. |
| ERR\_SHOULD\_WAIT | (In read/discard/peek mode) requested is nonzero and MX\_DATAPIPE\_READ\_FLAG\_ALL\_OR\_NONE is not set, but the data pipe is empty (and the producer is still open). |
| ERR\_OUT\_OF\_RANGE | (In read/discard/peek mode) requested is nonzero and MX\_DATAPIPE\_READ\_FLAG\_ALL\_OR\_NONE is set, but the data pipe does not have the requested amount of data available (and the producer is still open).  BUGS  The ERR\_OUT\_OF\_RANGE will be changed to ERR\_SHOULD\_WAIT once read thresholds are implemented (and a corresponding Mojo change is made). |

### datapipe\_begin\_read

#### 简述

开始从数据管道中两阶段（two-phase）读数据。

#### 函数原型

#include <magenta/syscalls.h>

mx\_ssize\_t mx\_datapipe\_begin\_read(mx\_handle\_t consumer\_handle,

uint32\_t flags,

uintptr\_t\* buffer);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| consumer\_handle | 句柄 |
| flags | 固定为0 |
| buffer | out参数，缓冲区指针，这个缓冲区指向的是管道里面的数据 |

#### 功能

datapipe\_begin\_read()开始对一个数据管道的两阶段读（two-phase read）。首先，它要确认是不是已经有一个ConsumerReadBegin()挂在这个数据管道上了，其次，它要确定这个管道是不是到了容量限制、不是处于活跃（active）状态，管道数据生产者在远端没有关闭这个管道，内存可以在当前进程访问。

如果管道地址不是连续的，则从管道中读出的数据可能少于管道中拥有的数据。

from a data pipe. That is, it first ensures that (some of) the data available to be read is accessible in the caller's address space, e.g., possibly by mapping memory, and then it provides that data's address in buffer. The amount of data available to be read starting from that address is returned (on success). (This address may not be written to; doing so may cause a fault.) Note that this amount may be strictly less than the total amount available to be read from the data pipe (e.g., if the data is not contiguous).

调用完datapipe\_begin\_read()之后，需要调用datapipe\_end\_read()来终止这个两阶段读，datapipe\_end\_read()将告诉数据管道数据被消费了。

#### 返回值

成功返回可以从管道中读出的内容的字节数，这些内容的起始地址通过参数buffer返回；失败，一个代表错误值的负数。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_BAD\_HANDLE | consumer\_handle不是一个有效句柄 |
| ERR\_WRONG\_TYPE | consumer\_handle不是一个与DataPipe生产者有关联的句柄 |
| ERR\_ACCESS\_DENIED | 句柄没有读权限（MX\_RIGHT\_READ） |
| ERR\_NOT\_SUPPORTED | flag参数有非预期的标记 |
| ERR\_ALREADY\_BOUND | consumer\_handle is currently in a two-phase read. |
| ERR\_INVALID\_ARGS(-10) | 参数错。  flags has an invalid combination of flags set, requested is not a multiple of the data pipe's element size (and query mode is not specified), or buffer is an invalid pointer (or NULL). |
| ERR\_REMOTE\_CLOSED | (In read/discard/peek mode) The requested operation could not be performed (due to a lack of data) and the remote data pipe producer handle is closed. |
| ERR\_SHOULD\_WAIT | (In read/discard/peek mode) requested is nonzero and MX\_DATAPIPE\_READ\_FLAG\_ALL\_OR\_NONE is not set, but the data pipe is empty (and the producer is still open). |
| ERR\_NO\_MEMORY | 内存不足 |

### datapipe\_end\_read

#### 简述

结束从数据管道中两阶段（two-phase）读数据。

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t mx\_datapipe\_end\_read(mx\_handle\_t consumer\_handle, mx\_size\_t read);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| consumer\_handle | 句柄 |
| read | 具体读了多少 |

#### 功能

datapipe\_end\_read()结束从数据管道中读数据。其实就是通知管道管理机构，哪些内容我已经读了，你可以越过这些内容了。

#### 返回值

成功，返回NO\_ERROR；失败，一个代表错误值的负数。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_BAD\_HANDLE | consumer\_handle不是一个有效句柄 |
| ERR\_WRONG\_TYPE | consumer\_handle不是一个与DataPipe生产者有关联的句柄 |
| ERR\_ACCESS\_DENIED | 句柄没有读权限（MX\_RIGHT\_READ） |
| ERR\_BAD\_STATE | consumer\_handle is currently not in a two-phase read (i.e., there is no preceding call to datapipe\_begin\_read() not "terminated" by a corresponding call to datapipe\_end\_read()). |
| ERR\_INVALID\_ARGS(-10) | 参数错。参数read的值比调用datapipe\_begin\_read()返回的那个值大，或者不是管道元素大小的整数倍。  read is larger than the value returned by the preceding call to datapipe\_begin\_read() or is not a multiple of the data pipe's element size.. |

### datapipe\_write

#### 简述

向数据管道中写数据。

#### 函数原型

#include <magenta/syscalls.h>

mx\_ssize\_t mx\_datapipe\_write(mx\_handle\_t producer\_handle,

uint32\_t flags,

mx\_size\_t requested,

const void\* buffer);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| producer\_handle | 句柄 |
| flags |  |
| requested |  |
| buffer |  |

#### 功能

requested is the amount of data that the caller would like to write; it must be a multiple of the data pipe's element size. buffer is the source of the data (and should have requested bytes available to read).

If MX\_DATAPIPE\_WRITE\_FLAG\_ALL\_OR\_NONE is set in flags, this either writes the requested amount of data or none at all. Otherwise this may do a partial write (if the data pipe has insufficient available capacity to accomodate the entire requested write), with the amount of data written returned.

#### 返回值

成功返回写出去的字节数， this may be less than requested, but will always be a multiple of the data pipe's element size. ；失败，一个代表错误值的负数。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_INVALID\_ARGS(-10) | 参数错。  requested is not a multiple of the data pipe's element size, or buffer is an invalid pointer (or NULL). |
| ERR\_BAD\_HANDLE | consumer\_handle不是一个有效句柄 |
| ERR\_WRONG\_TYPE | consumer\_handle不是一个与DataPipe生产者有关联的句柄 |
| ERR\_ACCESS\_DENIED | 句柄没有读权限（MX\_RIGHT\_READ） |
| ERR\_NOT\_SUPPORTED | flag参数有非预期的标记 |
| ERR\_ALREADY\_BOUND | consumer\_handle is currently in a two-phase read. |
| ERR\_REMOTE\_CLOSED | (In read/discard/peek mode) The requested operation could not be performed (due to a lack of data) and the remote data pipe producer handle is closed. |
| ERR\_SHOULD\_WAIT | (In read/discard/peek mode) requested is nonzero and MX\_DATAPIPE\_READ\_FLAG\_ALL\_OR\_NONE is not set, but the data pipe is empty (and the producer is still open). |
| ERR\_OUT\_OF\_RANGE | (In read/discard/peek mode) requested is nonzero and MX\_DATAPIPE\_READ\_FLAG\_ALL\_OR\_NONE is set, but the data pipe does not have the requested amount of data available (and the producer is still open).  BUGS  The ERR\_OUT\_OF\_RANGE will be changed to ERR\_SHOULD\_WAIT once read thresholds are implemented (and a corresponding Mojo change is made). |

### datapipe\_begin\_write

#### 简述

开始一个向数据管道中两阶段（two-phase）写数据。

#### 函数原型

#include <magenta/syscalls.h>

mx\_ssize\_t mx\_datapipe\_begin\_write(mx\_handle\_t producer\_handle,

uint32\_t flags,

uintptr\_t\* buffer);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| producer\_handle | 句柄 |
| flags | （好象没用） |
| buffer | out参数，用来返回空间指针 |

#### 功能

datepipe\_begin\_write()试图开始一个向数据管道中两阶段（two-phase）写数据。所谓两阶段写就是先向管道服务机制申请一段空间，然后向这个空间里写数据。

#### 返回值

成功，datapipe\_begin\_write() 返回空间的大小；失败，一个代表错误值的负数。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_INVALID\_ARGS(-10) | 参数错。   * handle不是有效的句柄 * num\_bytes如果不是NULL，不是有效地址；或者num\_bytes是NULL，而bytes不是NULL * num\_handles如果不是NULL，不是有效地址；或者num\_handles是NULL，而handles不是NULL |
| ERR\_NOT\_SUPPORTED |  |
| ERR\_ACCESS\_DENIED | 句柄没有写权限（MX\_RIGHT\_WRITE） |
| ERR\_BAD\_STATE | 管道的另一端已经关闭或者管道是一个应答管道（reply pipe） |
| ERR\_REMOTE\_CLOSED | 消息中有句柄，但是这个管道不是应答管道（reply pipe） |
| ERR\_NO\_MEMORY (-4) | 没内存 |
| ERR\_TOO\_BIG | 要传的消息太大。  kMaxMessageSize(65536u)  kMaxMessageHandles (1024u) |

#### 程序示例

源码：$Magenta/magenta/system/utest/core/data-pipe/data-pipe.c

static bool loop\_write\_full(void) {

BEGIN\_TEST;

mx\_handle\_t producer;

mx\_handle\_t consumer;

mx\_status\_t status;

producer = mx\_datapipe\_create(0u, 1u, KB\_(32), &consumer);

ASSERT\_GT(producer, 0, "could not create producer data pipe");

ASSERT\_GT(consumer, 0, "could not create consumer data pipe");

for (int ix = 0; ; ++ix) {

uintptr\_t buffer = 0;

mx\_ssize\_t avail = mx\_datapipe\_begin\_write(producer, 0u, &buffer);

if (avail < 0) {

ASSERT\_EQ(avail, ERR\_SHOULD\_WAIT, "wrong error");

ASSERT\_EQ(ix, 8, "wrong capacity");

break;

}

memset((void\*)buffer, ix, KB\_(4));

status = mx\_datapipe\_end\_write(producer, KB\_(4));

ASSERT\_EQ(status, NO\_ERROR, "failed to end write");

}

ASSERT\_EQ(get\_satisfied\_signals(producer), 0u, "");

ASSERT\_EQ(get\_satisfiable\_signals(producer), MX\_SIGNAL\_WRITABLE | MX\_SIGNAL\_PEER\_CLOSED, "");

status = mx\_handle\_close(consumer);

ASSERT\_GE(status, NO\_ERROR, "failed to close data pipe");

ASSERT\_EQ(get\_satisfied\_signals(producer), MX\_SIGNAL\_PEER\_CLOSED, "");

ASSERT\_EQ(get\_satisfiable\_signals(producer), MX\_SIGNAL\_PEER\_CLOSED, "");

status = mx\_handle\_close(producer);

ASSERT\_GE(status, NO\_ERROR, "failed to close data pipe");

END\_TEST;

}

### datapipe\_end\_write

#### 简述

结束一个向数据管道中两阶段（two-phase）写数据。

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t mx\_datapipe\_end\_write(mx\_handle\_t producer\_handle,

mx\_size\_t written);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| producer\_handle | 句柄 |
| written | 要写出内容的大小，字节为单位   * >=1 * <=datapipe\_begin\_write()中的缓存区大小 * written % element\_size\_ == 0u |

#### 功能

datepipe\_end\_write()真正一个向数据管道中写数据。程序示例参考datapipe\_begin\_write()中的程序示例。

#### 返回值

成功，datapipe\_end\_write()返回NO\_ERROR（0）；失败返回错误值。

## 端口（Port）

### port\_create

#### 简述

创建一个IO端口

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t mx\_port\_create(uint32\_t options, mx\_handle\_t\* out);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| options | 0（好象没啥用） |
| out | 句柄 |

#### 功能

port\_create() 创建一个IO端口，一个可以对其实施等待操作的对象，用户态程序或内核可以通过它读IO数据包。

返回句柄将有权限：

|  |  |
| --- | --- |
| 权限 | 能力 |
| MX\_RIGHT\_TRANSFER | 允许这个句柄被通过消息管道发送到其它进程 |
| MX\_RIGHT\_WRITE | 允许数据包被队列化 |
| MX\_RIGHT\_READ | 允许读数据包 |
| MX\_RIGHT\_DUPLICATE | 这个句柄可被复制 |

constexpr mx\_rights\_t kDefaultIOPortRights =

MX\_RIGHT\_DUPLICATE | MX\_RIGHT\_TRANSFER | MX\_RIGHT\_READ | MX\_RIGHT\_WRITE;

#### 返回值

成功，port\_create() 返回NO\_ERROR，并通过参数out返回一个有效的IO端口句柄；失败，一个代表错误值的负数。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_INVALID\_ARGS(-10) | options中的值无效 |
| ERR\_NO\_MEMORY (-4) | 没内存 |

### port\_queue

#### 简述

把要发送内容送入IO端口发送队列

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t mx\_port\_queue(mx\_handle\_t handle, const void\* packet, mx\_size\_t size);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| packet | 数据包 |
| size | 数据包的大小 |

#### 功能

port\_queue() 把一个数据包送入IO端口发送队列这个数据包的开始部分必须是数据结构mx\_packet\_header\_t，并且比MX\_PORT\_MAX\_PKT\_SIZE.（128u）小。

typedef struct mx\_packet\_header {

uint64\_t key;

uint32\_t type;

uint32\_t extra;

} mx\_packet\_header\_t;

key和exta的值暂保留，type的值必须是MX\_PORT\_PKT\_TYPE\_USER（2u），这个值是告诉系统这个数据包来源于port\_queue()，而不是来源于绑定的句柄，如果来源于绑定的句柄，这个type值是MX\_PORT\_PKT\_TYPE\_IOSN（1u）。

The queue is drained by calling port\_wait().

#### 返回值

成功，port\_queue() 返回NO\_ERROR（0）。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_INVALID\_ARGS(-10) | handle不是一个有效的IO端口，或者packet不是一个有效的指针，或者size比数据结构mx\_packet\_header\_t的尺寸还小。 |
| ERR\_ACCESS\_DENIED(-30) | 句柄没写（MX\_RIGHT\_WRITE.）权限 |
| ERR\_NOT\_ENOUGH\_BUFFER(-14) | 包太大 |

### port\_wait

#### 简述

在一个IO端口上等待一个包

#### 函数原型

#include <magenta/syscalls.h>

#include <magenta/syscalls/port.h>

mx\_status\_t mx\_port\_wait(mx\_handle\_t handle, mx\_time\_t timeout, void\* packet, mx\_size\_t size);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| handle | 句柄 |
| timeout | 超时时间，0表示不等待，INFINITE\_TIME表示无限期等待 |
| packet | 数据包 |
| size | 数据包的大小 |

#### 功能

port\_wait() 调用将引起调用者等待，直到该IO端口上至少有一个包，而且包是有效的。包队列是先到先出FIFO序的。

不象mx\_wait\_one() 和mx\_wait\_many()，一个数据包只唤醒一个线程，这使得IO端口可被线程池服务。

如果使用mx\_port\_queue()把数据包送入IO端口数据队列，这个数据包的长度不是固定的，但这个数据包的开始部分一定是数据结构mx\_packet\_header\_t，并且成果type的值是MX\_PORT\_PKT\_TYPE\_USER（2u）。

typedef struct mx\_packet\_header {

uint64\_t key;

uint32\_t type;

uint32\_t extra;

} mx\_packet\_header\_t;

If using port\_bind() the dequeued packet is of type mx\_io\_packet\_t with hdr.type set to MX\_PORT\_PKT\_TYPE\_IOSN（1u）.

typedef struct mx\_io\_packet {

mx\_packet\_header\_t hdr;

mx\_time\_t timestamp;

mx\_size\_t bytes;

mx\_signals\_t signals;

uint32\_t reserved;

} mx\_io\_packet\_t;

The key field in the packet header is the key that was in the packet as send via mx\_port\_queue(), or the key that was provided to mx\_port\_bind() when the binding was made.

#### 返回值

成功，port\_queue() 返回NO\_ERROR（0）。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_INVALID\_ARGS(-10) | handle不是一个有效的IO端口，或者packet不是一个有效的指针，或者size比数据结构mx\_packet\_header\_t的尺寸还小。 |
| ERR\_ACCESS\_DENIED(-30) | 句柄没写（MX\_RIGHT\_WRITE.）权限 |
| ERR\_NOT\_ENOUGH\_BUFFER(-14) | 包太大 |

### port\_bind

#### 简述

把IO端口与一个内核对象绑定

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t mx\_port\_bind(mx\_handle\_t handle, uint64\_t key,

mx\_handle\_t source, mx\_signals\_t signals);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| handle | 内核对象句柄 |
| key | 数据包的大小 |
| source | IO句柄 |
| signals | 信号：MX\_SIGNAL\_READABLE | MX\_SIGNAL\_PEER\_CLOSED |

#### 功能

io\_port\_bind() binds the waitable kernel object source to the IO port identified by handle. Whenever source signals match any of signals, the magenta kernel queues a packet of type mx\_io\_packet\_t to the IO port with the key key and type equal to MX\_PORT\_PKT\_TYPE\_IOSN.

To unbind a source from an IO port, simply close the source handle.

#### 返回值

成功，port\_bind() 返回NO\_ERROR（0）。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_INVALID\_ARGS(-10) | * handle不是一个有效的IO端口， * source不是一个有效的句柄，或者它不是一个可等待的句柄。 * signals是0 |
| ERR\_ACCESS\_DENIED(-30) | * 句柄没写（MX\_RIGHT\_WRITE.）权限 * source没有读（MX\_RIGHT\_READ）权限 |

## 套接字（socket）

// Socket flags and limits.

#define MX\_SOCKET\_CONTROL 1u

#define MX\_SOCKET\_HALF\_CLOSE 2u

#define MX\_SOCKET\_CONTROL\_MAX\_LEN 1024u

### socket\_create

#### 简述

创建一个IO端口。

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t mx\_socket\_create(mx\_handle\_t out\_handle[2], uint32\_t flags);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| out\_handle | 句柄数组，用来返回生成的句柄 |
| flags | 0 |

#### 功能

socket\_create() 创建一对套接字。

返回句柄将有权限：

|  |  |
| --- | --- |
| 权限 | 能力 |
| MX\_RIGHT\_TRANSFER | 允许这个句柄被通过消息管道发送到其它进程 |
| MX\_RIGHT\_WRITE | 允许数据包被队列化 |
| MX\_RIGHT\_READ | 允许读数据包 |
| MX\_RIGHT\_DUPLICATE | 这个句柄可被复制 |

constexpr mx\_rights\_t kDefaultSocketRights =

MX\_RIGHT\_TRANSFER | MX\_RIGHT\_DUPLICATE | MX\_RIGHT\_READ | MX\_RIGHT\_WRITE;

#### 返回值

成功，返回NO\_ERROR；出错，返回一个表示错误信息的整数值。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_INVALID\_ARGS(-10) | options中的值无效 |
| ERR\_NO\_MEMORY (-4) | 没内存 |

### socket\_write

#### 简述

向套接字中写数据。

#### 函数原型

#include <magenta/syscalls.h>

mx\_ssize\_t socket\_write(mx\_handle\_t handle, uint32\_t flags,

mx\_size\_t size, user\_ptr<const void> \_buffer)

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| handle | 句柄 |
| flags | * 0，普通数据 * MX\_SOCKET\_CONTROL，OOB（带外数据） * MX\_SOCKET\_HALF\_CLOSE，此时size不为0，则ERR\_INVALID\_ARGS，半关闭写，实际上这时不写任何数据 |
| size | 缓冲区大小 |
| buffer |  |

#### 功能

#### 返回值

成功返回写出去的字节数；失败，一个代表错误值的负数。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_INVALID\_ARGS(-10) | 参数错。  requested is not a multiple of the data pipe's element size, or buffer is an invalid pointer (or NULL). |

### socket\_read

#### 简述

从套接字中读数据。

#### 函数原型

#include <magenta/syscalls.h>

mx\_ssize\_t socket\_read(mx\_handle\_t handle, uint32\_t flags,

mx\_size\_t size, user\_ptr<void> \_buffer)

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| handle | 句柄 |
| flags | * MX\_SOCKET\_CONTROL，读取带外数据 * 其它，普通数据读 |
| size |  |
| buffer |  |

#### 功能

#### 返回值

成功返回读出的字节数；失败，一个代表错误值的负数。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_INVALID\_ARGS(-10) | 参数错。  requested is not a multiple of the data pipe's element size, or buffer is an invalid pointer (or NULL). |

## 线程（Threads）

### thread\_arch\_prctl

### thread\_create

#### 简述

创建一个线程。

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t mx\_thread\_create(mx\_handle\_t process, const char\* name,

uint32\_t name\_len, uint32\_t flags, mx\_handle\_t\* out);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| process | 进程句柄 |
| name | 线程名称 |
| name\_len | 线程名称长度，包括尾'\0'。如果线程名称（含'\0'）长度为MX\_MAX\_NAME\_LEN，则最后一个字母被忽略。 |
| flags | 标志 |
| out | 返回句柄 |

// Maximum string length for kernel names (process name, thread name, etc)

#define MX\_MAX\_NAME\_LEN (32)

#### 功能

thread\_create()指定的进程中创建一个线程，这个线程直到thread\_start()后才开始执行。

当指向一个线程的最后一个句柄关闭后，线程被销毁。

线程句柄可以被等待，这个等待直到收到信号（signal）MX\_SIGNAL\_SIGNALED，这个信号当线程停止时发出，终止一个线程的API是thread\_exit\*()。

#### 返回值

成功，thread\_create() 返回NO\_ERROR，并通过参数out返回一个有效的IO端口句柄；失败，一个代表错误值的负数。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_INVALID\_ARGS(-10) | * 进程句柄无效 * 名字太长 * flags中的值非0 |
| ERR\_NO\_MEMORY (-4) | 没内存 |

### thread\_exit

#### 简述

终止当前线程。

#### 函数原型

#include <magenta/syscalls.h>

void mx\_thread\_exit(void);

#### 功能

mx\_thread\_exist()终止当前线程，没有参数，也没有返回值，这个函数执行完，当前线程就终止了，就以它不返回了。

这个函数执行完后，线程对象将得到信号（signal）MX\_SIGNAL\_SIGNALED，这个信号通过函数handle\_wait\_one()或handle\_wait\_many()监听线程句柄得到。

#define MX\_SIGNAL\_SIGNALED ((mx\_signals\_t)1u << 3)

### thread\_start

#### 简述

开始执行一个线程。

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t mx\_thread\_start(mx\_handle\_t thread, uintptr\_t entry, uintptr\_t stack,

uintptr\_t arg1, uintptr\_t arg2);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| thread | 线程句柄，需要有写（MX\_RIGHT\_WRITE）权限。 |
| entry | 线程入口 |
| stack | 栈 |
| arg1 | 参数1 |
| arg2 | 参数2 |

typedef unsigned long uintptr\_t;

#### 功能

thread\_start()使得一个线程从程序入口entry处开始执行，执行的栈来源于参数stack，参数arg1和arg2，根据不同的平台，被组织到不同的寄存器中去，所有其它寄存器是0。

#### 返回值

成功，thread\_start() 返回NO\_ERROR；失败，一个代表错误值的负数。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_BAD\_HANDLE | 句柄无效 |
| ERR\_WRONG\_TYPE | 句柄thread不是一个线程句柄 |
| ERR\_ACCESS\_DENIED | 句柄没有写（MX\_RIGHT\_WRITE）权限 |
| ERR\_BAD\_STATE | 线程没有为运行准备好，thread is not ready to run or the process thread is part of is no longer alive. |

### thread\_read\_state

- read register state from a thread

### thread\_write\_state

- modify register state of a thread

## 进程（Processes）

Magenta操作系统有Process的概念，但与传统操作系统的进程的概念有很大差异，它更强调它是一个计算容器。从代码上来看，这个概念还没有实现完，如系统调用mx\_process\_start()代码还不完整，从已经实现的代码上看，Google对这时的Process具体怎么定义，也是在探索中。

### process\_create

#### 简述

创建一个进程。

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t mx\_process\_create(mx\_handle\_t job,

const char\* name, uint32\_t name\_len,

uint32\_t flags, mx\_handle\_t\* out);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| job | 作业句柄 |
| name | 线程名称 |
| name\_len | 线程名称长度，包括尾'\0'。如果线程名称（含'\0'）长度为MX\_MAX\_NAME\_LEN，则最后一个字母被忽略。 |
| flags | 标志 |
| out | 返回句柄 |

// Maximum string length for kernel names (process name, thread name, etc)

#define MX\_MAX\_NAME\_LEN (32)

#### 功能

process\_create()创建一个进程，这个线程直到process\_start()后才开始执行。

当指向一个进程的最后一个句柄关闭后，进程被销毁。

进程句柄可以被等待，这个等待直到收到信号（signal）MX\_SIGNAL\_SIGNALED，这个信号当进程停止时发出。

#### 返回值

成功，process\_create() 返回NO\_ERROR，并通过参数out返回一个有效的IO端口句柄；失败，一个代表错误值的负数。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_INVALID\_ARGS(-10) | * 进程句柄无效 * 名字太长 * flags中的值非0 |
| ERR\_NO\_MEMORY (-4) | 没内存 |

### process\_start

#### 简述

开始执行一个进程。

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t mx\_process\_start(mx\_handle\_t process, mx\_handle\_t thread,

uintptr\_t entry, uintptr\_t stack,

mx\_handle\_t arg1, uintptr\_t arg2);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| process | 线程句柄，需要有写（MX\_RIGHT\_WRITE）权限。 |
| thread | 线程句柄，需要有写（MX\_RIGHT\_WRITE）权限。 |
| entry | 线程入口 |
| stack | 栈 |
| arg1 | 参数1 ??????为什么需要是句柄？ |
| arg2 | 参数2 |

typedef unsigned long uintptr\_t;

#### 功能

process\_start()类似于thread\_start()，它的目的是启动进程所属的第一个线程。

使得进程的第一个线程从程序入口entry处开始执行，执行的栈来源于参数stack，参数arg1和arg2，根据不同的平台，被组织到不同的寄存器中去，所有其它寄存器是0。

每一个参数arg1是一个句柄，它将被从调用进程传输到被起动的进程里去。

The first argument (arg1) is a handle, which will be transferred from the process of the caller to the process which is being started, and an appropriate handle value will be placed in arg1 for the newly started thread.

#### 返回值

成功，process\_start() 返回NO\_ERROR；失败，一个代表错误值的负数。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_BAD\_HANDLE | 句柄无效 |
| ERR\_WRONG\_TYPE | 句柄thread不是一个进程句柄，或者thread不是一个线程句柄 |
| ERR\_ACCESS\_DENIED | 句柄没有写（MX\_RIGHT\_WRITE）权限，或者线程thread不从属于进程process |
| ERR\_INVALID\_ARGS | 参数arg1不是一个有传输权限（MX\_RIGHT\_TRANSFER）的有效句柄。 |
| ERR\_BAD\_STATE | 进程已经处于运行状态或者已经退出。 |

### process\_map\_vm

#### 简述

为进程保留一块内存。

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t sys\_process\_map\_vm(mx\_handle\_t proc\_handle, mx\_handle\_t vmo\_handle,

uint64\_t offset, mx\_size\_t len, mxtl::user\_ptr<uintptr\_t> user\_ptr,

uint32\_t flags);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| proc\_handle | 进程句柄 |
| vmo\_handle | 内存对象句柄 |
| offset |  |
| len |  |
| user\_ptr |  |
| flags | 标志 |

#### 功能

The mapping retains a reference to the underlying virtual memory object, which means closing the VMO handle does not remove the mapping added by this function.

A virtual memory object can be larger than the address space, which means you should check for overflow before converting the uint64\_t size of the VMO to mx\_process\_map\_vm's mx\_size\_t len parameter.

#### 返回值

成功，process\_map\_vm() 返回NO\_ERROR；失败，一个代表错误值的负数。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_INVALID\_ARGS(-10) |  |
|  |  |

### process\_protect\_vm

#### 简述

设置一块内存的访问权限属性。

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t mx\_process\_protect\_vm(mx\_handle\_t proc\_handle,

uintptr\_t address, mx\_size\_t len,

uint32\_t prot);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| proc\_handle | 进程句柄 |
| address | 内存地址，这个地址不一定是process\_vm\_map()内存时的首地址，只要是在process\_vm\_map()时得到的内存区域内就行。 |
| len |  |
| prot | PROT\_NONE(0)与0个或多个PROT\_READ(1)、PROT\_WRITE(2)和PROT\_EXEC(4).的或（OR）值。 |

#define PROT\_NONE 0 //无权限

#define PROT\_READ 1 //读

#define PROT\_WRITE 2 //写

#define PROT\_EXEC 4 //执行

#### 功能

process\_protect\_vm()调整地址address所在内存区域的访问权限（读、写、执行）属性，地址address所在内存区域通过查找内存映射表获得。如果address没有通过process\_vm\_map()进行过内存映射，则本函数行为不确定。

#### 返回值

成功，process\_protect\_vm() 返回NO\_ERROR；失败，一个代表错误值的负数。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_INVALID\_ARGS(-10) | proc\_handle isn't a valid process handle, or address is not from a valid mapped region, or prot is an unsupported combination of flags (e.g., PROT\_WRITE but not PROT\_READ). |
| ERR\_ACCESS\_DENIED | proc\_handle句柄没有写权限（MX\_RIGHT\_WRITE） |

### process\_unmap\_vm

#### 简述

解除进程内存与物理内存的映射关系。

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t sys\_process\_unmap\_vm(mx\_handle\_t proc\_handle, uintptr\_t address, mx\_size\_t len)

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| proc\_handle | 进程句柄 |
| address | 内存地址，这个地址不一定是process\_vm\_map()内存时的首地址，只要是在process\_vm\_map()时得到的内存区域内就行。 |
| len |  |

#### 功能

释放为进程保留的内存。

#### 返回值

成功，process\_map\_vm() 返回NO\_ERROR；失败，一个代表错误值的负数。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_INVALID\_ARGS(-10) |  |
|  |  |

### process\_read\_memory

#### 简述

从进程中读一段内存到缓冲区。

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t sys\_process\_read\_memory(mx\_handle\_t proc, uintptr\_t vaddr,

user\_ptr<void> buffer,

mx\_size\_t len, user\_ptr<mx\_size\_t> actual)

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| proc\_handle | 进程句柄 |
| vaddr |  |
| buffer |  |
| len |  |
| actual |  |

#### 功能

主要为了调试程序用，从进程中读一段内存到缓冲区。

#### 返回值

成功，process\_read\_memory() 返回NO\_ERROR；失败，一个代表错误值的负数。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_INVALID\_ARGS(-10) |  |
|  |  |

### process\_write\_memory

#### 简述

往进程内存中写。

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t sys\_process\_write\_memory(mx\_handle\_t proc, uintptr\_t vaddr,

user\_ptr<const void> buffer,

mx\_size\_t len, user\_ptr<mx\_size\_t> actual)

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| proc\_handle | 进程句柄 |
| vaddr |  |
| buffer |  |
| len |  |
| actual |  |

#### 功能

主要为了调试程序用，往进程内存中写。

#### 返回值

成功，process\_write\_memory () 返回NO\_ERROR；失败，一个代表错误值的负数。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_INVALID\_ARGS(-10) |  |
|  |  |

## 等待集（Wait Sets）

### waitset\_create

#### 简述

创建一个等待集

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t mx\_waitset\_create(uint32\_t options, mx\_handle\_t\* out);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| out | out参数，等待集句柄 |

#### 功能

mx\_waitset\_create() 创建一个等待集（waitset），一个可以对其实施等待操作的对象，可以向这个对象中加入多个其它种类的句柄。

返回句柄将有权限：

|  |  |
| --- | --- |
| 权限 | 能力 |
| MX\_RIGHT\_WRITE | 允许写 |
| MX\_RIGHT\_READ | 允许读 |

#### 返回值

成功，port\_create() 返回一个有效的句柄（一个正整数）；失败，一个代表错误值的负数。不可能返回代表无效句柄的0（Zero，the "invalid handle"）。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_NO\_MEMORY (-4) | 没内存 |

### waitset\_add

#### 简述

向等待集中增加一个入口。

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t mx\_waitset\_add(mx\_handle\_t waitset\_handle,

uint64\_t cookie,

mx\_handle\_t handle,

mx\_signals\_t signals);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| waitset\_handle | 等待集句柄，需要有写（MX\_RIGHT\_WRITE）权限。 |
| cookie |  |
| handle | 要加入等待集中的对象，需要有读（MX\_RIGHT\_READ）权限。 |
| signals |  |

#### 功能

waitset\_add() adds an entry to a wait set; an entry consists of a handle, a set of signals that the wait set will "watch", and a cookie to uniquely identify the entry. Note that there may be multiple entries with the same handle (with the same or different set of signals to watch), but that each entry must have a distinct cookie to identify it.

#### 返回值

成功，waitset\_add() 返回NO\_ERROR；失败，一个代表错误值的负数。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_NO\_MEMORY | 没内存 |
| ERR\_BAD\_HANDLE | waitset\_handle不是一个有效的句柄。 |
| ERR\_INVALID\_ARGS | waitset\_handle指向的不是等待集WaitSet或者handle不是一个有效句柄。 |
| ERR\_ACCESS\_DENIED | waitset\_handle句柄没有写（MX\_RIGHT\_WRITE）权限，或者handle句柄没有读（MX\_RIGHT\_READ）权限 |
| RR\_NOT\_SUPPORTED | handle指向的不是一个可等待对象 |
| ERR\_ALREADY\_EXISTS | 等待集中已经有同样的入口并且同样的cookie的入口handle |

### waitset\_remove

#### 简述

移除等待集中的一个入口。

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t mx\_waitset\_remove(mx\_handle\_t waitset\_handle, uint64\_t cookie);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| waitset\_handle | 等待集句柄，需要有写（MX\_RIGHT\_WRITE）权限。 |
| cookie |  |

#### 功能

waitset\_remove()移除等待集中的一个入口。

#### 返回值

成功，waitset\_remove() 返回NO\_ERROR；失败，一个代表错误值的负数。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_BAD\_HANDLE | waitset\_handle不是一个有效的句柄。 |
| ERR\_INVALID\_ARGS | waitset\_handle指向的不是等待集WaitSet。 |
| ERR\_ACCESS\_DENIED | waitset\_handle句柄没有写（MX\_RIGHT\_WRITE）权限 |
| ERR\_NOT\_FOUND | 等待集中没有cookie指定的入口 |

### waitset\_wait

#### 简述

等待一个WaitSet。

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t mx\_waitset\_wait(mx\_handle\_t waitset\_handle,

mx\_time\_t timeout,

mx\_waitset\_result\_t\* results,

uint32\_t\* count);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| waitset\_handle | 等待集句柄，需要有读（MX\_RIGHT\_READ）权限。 |
| timeout | 超时时间，0表示不等待，INFINITE\_TIME表示无限期等待 |
| results | 等待结果集 |
| count | in-out参数。  输入是results参数的空间大小  输出是实际写到results中的结果集大小  它可能是NULL，因为count可以是0 |

// time in nanoseconds

typedef uint64\_t mx\_time\_t;

#define MX\_TIME\_INFINITE UINT64\_MAX

#define INFINITE\_TIME UINT32\_MAX

#### 功能

waitset\_wait()等待直到至少一个等待项有结果或者超时，结果包括：

* + 等待比特（waited for bits）是挂起（pending）
  + 句柄是关闭（closed）

每个等待项是一个mx\_waitset\_result\_t数据结构，

// Structure for mx\_waitset\_\*():

typedef struct mx\_waitset\_result {

uint64\_t cookie;

mx\_status\_t status;

mx\_signals\_t observed;

} mx\_waitset\_result\_t;

这个数据结构中，

* + 字段cookie是在waitset\_add()时提供的cookie
  + status的值：
    - NO\_ERROR if the watched signals provided to waitset\_add() were satisfied
    - ERR\_BAD\_STATE if the watched signals became unsatisfiable
    - ERR\_HANDLE\_CLOSED if the entry's handle was closed
  + observed 该项对应句柄在waitset\_wait()返回前一刻的observed signals。

#### 返回值

成功，waitset\_wait() 返回NO\_ERROR，此时，各输出参数中填入了预期的值；失败，一个代表错误的负数，超时也是一种失败，返回ERR\_TIMED\_OUT。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_NO\_MEMORY | 内存不足 |
| ERR\_BAD\_HANDLE | waitset\_handle不是一个有效的句柄。 |
| ERR\_HANDLE\_CLOSED | waitset\_handle在waitset\_wait()期间被关闭。 |
| ERR\_WRONG\_TYPE | waitset\_handle不是一个指向等待集WaitSet的句柄。 |
| ERR\_INVALID\_ARGS | waitset\_handle指向的不是等待集WaitSet。  num\_results无效  results无效  max\_results无效 |
| ERR\_ACCESS\_DENIED | waitset\_handle句柄没有读（MX\_RIGHT\_READ）权限 |
| ERR\_TIMED\_OUT | 超时 |

## Events and Event Pairs

### event\_create

#### 简述

创建一个事件。

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t mx\_event\_create(uint32\_t options, mx\_handle\_t\* out);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| options | 未用 |
| out | 返回句柄 |

#### 功能

event\_create()创建一个事件，也就是一个可以有信号的对象。这个对象可以有4个信号，被命名为：MX\_SIGNAL\_SIGNALn (0<=n<=4)。对象信号可以通过object\_signal()操作。

#define MX\_SIGNAL\_SIGNALED MX\_SIGNAL\_SIGNAL0

#define MX\_SIGNAL\_SIGNAL0 ((mx\_signals\_t)1u << 3)

#define MX\_SIGNAL\_SIGNAL1 ((mx\_signals\_t)1u << 4)

#define MX\_SIGNAL\_SIGNAL2 ((mx\_signals\_t)1u << 5)

#define MX\_SIGNAL\_SIGNAL3 ((mx\_signals\_t)1u << 6)

#define MX\_SIGNAL\_SIGNAL4 ((mx\_signals\_t)1u << 7)

#define MX\_SIGNAL\_SIGNAL\_ALL ((mx\_signals\_t)31u << 3)

返回句柄将有权限：

|  |  |
| --- | --- |
| 权限 | 能力 |
| MX\_RIGHT\_TRANSFER | 允许这个句柄被通过消息管道等机制发送到其它进程 |
| MX\_RIGHT\_WRITE | 允许数据包被队列化 |
| MX\_RIGHT\_READ | 允许读数据包 |
| MX\_RIGHT\_DUPLICATE | 这个句柄可被复制 |

#### 返回值

成功，event\_create() 返回NO\_ERROR，并通过参数out返回一个有效的IO端口句柄；失败，一个代表错误值的负数。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_NO\_MEMORY (-4) | 没内存 |

### eventpair\_create

#### 简述

创建一个事件对（pair）。

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t mx\_eventpair\_create(mx\_handle\_t handles[2], uint32\_t flags);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| handles | out参数，创建的事件句柄，两个事件对象组成一对 |
| flags |  |

#### 功能

eventpair\_create()创建一个事件对（eventpair），也就是两个可以有信号的对象。这两个对象可以有4个信号，被命名为：MX\_SIGNAL\_SIGNALn (0<=n<=4)。对象信号可以通过object\_signal()操作。

#define MX\_SIGNAL\_SIGNALED MX\_SIGNAL\_SIGNAL0

#define MX\_SIGNAL\_SIGNAL0 ((mx\_signals\_t)1u << 3)

#define MX\_SIGNAL\_SIGNAL1 ((mx\_signals\_t)1u << 4)

#define MX\_SIGNAL\_SIGNAL2 ((mx\_signals\_t)1u << 5)

#define MX\_SIGNAL\_SIGNAL3 ((mx\_signals\_t)1u << 6)

#define MX\_SIGNAL\_SIGNAL4 ((mx\_signals\_t)1u << 7)

#define MX\_SIGNAL\_SIGNAL\_ALL ((mx\_signals\_t)31u << 3)

返回句柄将有权限：

|  |  |
| --- | --- |
| 权限 | 能力 |
| MX\_RIGHT\_TRANSFER | 允许这个句柄被通过消息管道等机制发送到其它进程 |
| MX\_RIGHT\_WRITE | 允许数据包被队列化 |
| MX\_RIGHT\_READ | 允许读数据包 |
| MX\_RIGHT\_DUPLICATE | 这个句柄可被复制 |

示例程序：$Magenta/magenta/system/utest/core/event-pair/event-pair.c

#### 返回值

成功，event\_create() 返回一个有效的句柄（一个正整数）；失败，一个代表错误值的负数。不可能返回代表无效句柄的0（Zero，the "invalid handle"）。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_NO\_MEMORY (-4) | 没内存 |
| ERR\_INVALID\_ARGS | handles指针错 |
| ERR\_NOT\_SUPPORTED | flags错，应该是0 |

## 中断Interrupt

### interrupt\_create

#### 简述

创建一个中断事件

#### 函数原型

#include <magenta/syscalls.h>

#### 功能

mx

返回句柄将有权限：

|  |  |
| --- | --- |
| 权限 | 能力 |
| MX\_RIGHT\_WRITE | 允许写 |
| MX\_RIGHT\_READ | 允许读 |

#### 返回值

成功，port\_create() 返回一个有效的句柄（一个正整数）；失败，一个代表错误值的负数。不可能返回代表无效句柄的0（Zero，the "invalid handle"）。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_NO\_MEMORY (-4) | 没内存 |

### interrupt\_complete

## 硬件IO（Device Memory）

### mmap\_device\_memory

### alloc\_device\_memory

## 快速用户区互斥futex

### futex\_wait

#### 简述

等待一个futex。

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t mx\_futex\_wait(int\* value\_ptr, int current\_value,

mx\_time\_t timeout);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| value\_ptr |  |
| current\_value |  |
| timeout | 超时 |

// time in nanoseconds

typedef uint64\_t mx\_time\_t;

#define MX\_TIME\_INFINITE UINT64\_MAX

#### 功能

等待或占有一个futex将引起线程睡眠，直到mx\_futex\_wake()，或者超时。

#### 返回值

成功，返回NO\_ERROR；出错，返回一个表示错误信息的整数值。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_INVALID\_ARGS | value\_ptr不是一个合法指针 |
| ERR\_ALREADY\_BOUND | current\_value与value\_ptr中的当前值不相等 |
| ERR\_TIMED\_OUT | 线程在超时之前未唤醒 |

### futex\_wake

#### 简述

唤醒等待在futex上的多个线程。

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t mx\_futex\_wake(int\* value\_ptr, uint32\_t wake\_count);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| value\_ptr |  |
| wake\_count | 要唤醒的线程数量。0表示不做任何动作。 |

#### 功能

唤醒等待在futex上的多个线程。

#### 返回值

成功，返回NO\_ERROR。futex\_wake()永远成功，唤醒0个线程也不出错。

### futex\_requeue

#### 简述

唤醒一些等待在futex上的线程，把更多的等待线程移到另外一个等待队列中去。

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t mx\_futex\_requeue(int\* value\_ptr, uint32\_t wake\_count,

int current\_value, int\* requeue\_ptr,

uint32\_t requeue\_count);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| value\_ptr | 等待队列 |
| wake\_count | 唤醒的线程数量 |
| current\_value | 当前值 |
| requeue\_ptr | 新的等待队列 |
| requeue\_count | 移动的等待线程数 |

#### 功能

Requeuing is a generalization of waking. First, the kernel verifies that the value in wake\_count matches the value of the futex at value\_ptr, and if not reports ERR\_ALREADY\_BOUND. After waking wake\_count threads, requeue\_count threads are moved from the original futex's wait queue to the wait queue corresponding to requeue\_ptr, another futex.

This requeueing behavior may be used to avoid thundering herds on wake.

#### 返回值

成功，返回NO\_ERROR；出错，返回一个表示错误信息的整数值。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_INVALID\_ARGS | value\_ptr不是一个合法指针 |
| ERR\_ALREADY\_BOUND | current\_value与value\_ptr中的当前值不相等 |

## Virtual Memory Objects (VMOs)

### vmo\_create

- create a new vmo

### vmo\_read

- read from a vmo

### vmo\_write

- write to a vmo

### vmo\_get\_size

- obtain the size of a vmo

### vmo\_set\_size

- adjust the size of a vmo

### vmo\_op\_range

## Cryptographically Secure RNG

### cprng\_draw

#### 简述

利用内核的CPRNG机制在一个缓冲区内生成随机内容。

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t mx\_cprng\_draw(void\* buffer, mx\_size\_t len, mx\_size\_t\* actual);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| buffers | 缓冲区 |
| len | 缓冲区大小 |
| actual | out参数，实际生成的随机内容的长度 |

// Buffer size limits on the cprng syscalls

#define MX\_CPRNG\_DRAW\_MAX\_LEN 256

#### 功能

mx\_cprng\_draw()利用内核的CPRNG机制在一个绘冲区内生成随机内容。

#### 返回值

成功，返回NO\_ERROR ，参数actual中存有写到缓冲区里的字节数。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_INVALID\_ARGS | 参数错   * len太大 * buffer不是用户空间中的一个有效指针 |

### cprng\_add\_entropy

#### 简述

增加熵（entropy）到内核CPRNG。

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t mx\_cprng\_add\_entropy(const void\* buffer, mx\_size\_t len);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| buffers | 缓冲区 |
| len | 缓冲区大小 |

// Buffer size limits on the cprng syscalls

#define MX\_CPRNG\_ADD\_ENTROPY\_MAX\_LEN 256

#### 功能

mx\_cprng\_add\_entropy增加熵给内核的CPRNG机制。

#### 返回值

状态信息，成功返回NO\_ERROR，失败则返回一个表示出错信息的负数。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_INVALID\_ARGS | 参数错 |

## 时间Time

### nanosleep

#### 简述

高精度睡眠

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t mx\_nanosleep(mx\_time\_t nanoseconds);

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| nanoseconds | 纳秒。0表示放弃当前线程的后续时间片，MX\_TIME\_INFINITE表示无限期等待。  飞秒（fs）：10-15秒  皮秒（ps）：10-12秒  纳秒（ns）：10-9秒  微秒（µs）：10-6秒  毫秒（ms）：10-3秒 |

// time in nanoseconds

typedef uint64\_t mx\_time\_t;

#### 功能

挂起当前线程至少nanoseconds纳秒。

#### 返回值

成功，mx\_nanosleep () 返回NO\_ERROR。

### time\_get

#### 简述

取当前时间。

#### 函数原型

#include <magenta/syscalls.h>

mx\_time\_t mx\_time\_get(uint32\_t clock\_id)

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| clock\_id | 时钟编号，系统全局缺省时钟为MX\_CLOCK\_MONOTONIC (0u)，该时钟记录了自系统起动后经过的纳秒数。 |

// clock ids

#define MX\_CLOCK\_MONOTONIC (0u)

#### 功能

mx\_current\_time()取系统自加电起到当前的时间，纳秒为单位。

#### 返回值

返回纳秒数，不能返回错误信息。

## 系统信息（Information）

### version\_get

#### 简述

取内核版本字符串。

#### 函数原型

#include <magenta/syscalls.h>

mx\_status\_t mx\_version\_get(char\* version, uint32\_t version\_len)

#### 参数说明

|  |  |
| --- | --- |
| 参数 | 说明 |
| version | 返回参数，字符串缓冲区，必须容纳得下40个以上字符。  MAGENTA\_GIT\_REV "28b32d155dce6bd29fc947d72136bd81805c59d3" |
| version\_len | 字符串缓冲区长度。 |

#### 功能

mx\_version\_get()取内核版本信息，这个版本是最后一次源码checkin的git仓库的版本。

#### 返回值

成功返回NO\_ERROR，version中存放内核版本信息。

#### 出错信息

|  |  |
| --- | --- |
| 错误值 | 说明 |
| ERR\_BUFFER\_TOO\_SMALL (-14) | 缓冲区太小 |

### num\_cpus

#### 简述

取硬件CPU核（core）数。

#### 函数原型

#include <magenta/syscalls.h>

unsigned int mx\_num\_cpus(void)

#### 返回值

硬件CPU核（core）数。

## 系统运行log

// Defines and structures for mx\_log\_\*()

typedef struct mx\_log\_record {

uint32\_t reserved;

uint16\_t datalen;

uint16\_t flags;

mx\_time\_t timestamp;

char data[0];

} mx\_log\_record\_t;

#define MX\_LOG\_RECORD\_MAX 256

#define MX\_LOG\_FLAG\_KERNEL 0x0100

#define MX\_LOG\_FLAG\_DEVMGR 0x0200

#define MX\_LOG\_FLAG\_CONSOLE 0x0400

#define MX\_LOG\_FLAG\_DEVICE 0x0800

#define MX\_LOG\_FLAG\_MASK 0x0F00

#define MX\_LOG\_FLAG\_WAIT 0x80000000

#define MX\_LOG\_FLAG\_READABLE 0x40000000

### log\_create

int mx\_log\_create(uint32\_t flags);

### log\_write

int mx\_log\_write(mx\_handle\_t log\_handle, uint32\_t len, mxtl::user\_ptr<const void> ptr, uint32\_t flags)

### log\_read

int mx\_log\_read(mx\_handle\_t log\_handle, uint32\_t len, mxtl::user\_ptr<void> ptr, uint32\_t flags) {

# 开发工具链

## Make文件

源码：$Magenta/magenta/makefile

LKMAKEROOT := .

LKROOT := kernel

LKINC := system third\_party

BUILDROOT ?= .

DEFAULT\_PROJECT ?= magenta-pc-x86-64

TOOLCHAIN\_PREFIX ?=

ENABLE\_BUILD\_SYSROOT ?= false

# if true, $BUILDDIR/sysroot/{lib,include,...} will be populated with

# public libraries, headers, and other "build artifacts" necessary

# for a toolchain to compile binaries for Magenta.

ENABLE\_BUILD\_LISTFILES ?= false

# If true, various verbose listings (\*.lst, \*.sym, \*,dump, etc) will

# be generated for the kernel and userspace binaries. These can be

# useful for debugging, but are large and can slow the build some.

# check if LKROOT is already a part of LKINC list and add it only if it is not

ifneq ($(findstring $(LKROOT),$(LKINC)), $(LKROOT))

LKINC := $(LKROOT) $(LKINC)

endif

export LKMAKEROOT

export LKROOT

export LKINC

export BUILDROOT

export DEFAULT\_PROJECT

export TOOLCHAIN\_PREFIX

# vaneer makefile that calls into the engine with lk as the build root

# if we're the top level invocation, call ourselves with additional args

$(MAKECMDGOALS) \_top:

@$(MAKE) -C $(LKMAKEROOT) --no-print-directory -rR -f $(LKROOT)/engine.mk $(addprefix -I,$(LKINC)) $(MAKECMDGOALS)

.PHONY: \_top

## 源代码管理

参考：<https://github.com/fuchsia-mirror/manifest>。这里用到的工具jiri，可以通过这个链接<https://github.com/fuchsia-mirror/jiri> 了解更详细信息。

xilong@xilong-OptiPlex-7010:~$ curl -s https://raw.githubusercontent.com/fuchsia-mirror/jiri/master/scripts/bootstrap\_jiri | bash -s fuchsia

Please add /home/xilong/fuchsia/.jiri\_root/scripts to your PATH

xilong@xilong-OptiPlex-7010:~$ cd fuchsia/

xilong@xilong-OptiPlex-7010:~/fuchsia$ export PATH=`pwd`/.jiri\_root/scripts:$PATH

xilong@xilong-OptiPlex-7010:~/fuchsia$ jiri import fuchsia https://fuchsia.googlesource.com/manifest

xilong@xilong-OptiPlex-7010:~/fuchsia$ jiri update

[10:11:58.96] >> move project "manifest" located in "/tmp/jiri-load278940617/manifest\_ad81d9e784efd6cc" to "/home/xilong/fuchsia/manifest" and advance it to "578ca153"

[10:12:01.15] >> OK

[10:12:01.15] >> create project "apps/compositor" in "/home/xilong/fuchsia/apps/compositor" and advance it to "HEAD"

[10:12:07.06] >> OK

[10:12:07.06] >> create project "apps/dart\_content\_handler" in "/home/xilong/fuchsia/apps/dart\_content\_handler" and advance it to "HEAD"

[10:12:11.93] >> OK

[10:12:11.93] >> create project "apps/ledger" in "/home/xilong/fuchsia/apps/ledger" and advance it to "HEAD"

[10:12:17.87] >> OK

# Magenta深入开发

稳定性测试

连续运行能力

MTBF,即平均故障间隔时间，英文全称是“Mean Time Between Failure”。

内存泄漏

其它资源泄漏

平台移植

MIPS、PPC平台支持

可靠性、安全性策略支持

safety策略

security策略

容错计算

恢复块技术recovery支持

## 实时系统

### 概念

实时系统最重要的特点就是实时性，即系统的正确性不仅仅依赖于计算的逻辑结果的正确性，还取决于输出结果时间的及时性。从这个角度看，实时系统是“一个能够在指定或者确定的时间内完成系统功能和对外部环境做出响应的系统”。按对实时性能要求的程度，实时系统可分为两类：

1. 硬实时系统：要求可确定性强，具有明确的实时约束，在某个限定的时刻之前不能完成任务将造成灾难性的后果。
2. 软实时系统：也对时间敏感，但偶尔发生不能满足严格实时要求的情况也是允许的。

#### WCET

事先获知系统中程序最差情况的执行时间（Worst-Case Execution Time，WCET），是设计和验证实时系统调度及可调度性分析的前提，也是确定周期性任务是否满足其性能目标，从而发现系统性能瓶颈的基础。

#### 线程管理数据结构thread\_t中需要记录的数据

实时任务定义了4个调度属性：

* priority（优先级：限制该任务比相关联的其他任务的优先权）
* starttime（起始时间：任务开始执行时间）
* finishtime（截止时间：任务停止时间）
* budget（预设值：任务允许执行时间）。

### Linux在实时方面存在的不足

Linux虽然符合POSIX1003.1b关于实时扩展部分的标准，例如：支持SCHED\_FIFO和SCHED\_RR实时调度策略，锁内存机制(memory locking)，实时信号等功能，但是由于其最初的设计目标为通用分时操作系统，因此作为一个实时操作系统，Linux仍然存在如下缺陷：

1. Linux的内核本身是非抢占的。Linux下分用户态和核心态两种模式，当进程运行在用户态时，可被优先级更高的进程抢占，但当它进入核心态时，其他用户态进程优先级再高也不能抢占它。
2. Linux虽然给实时进程提供了较高的优先级，但是没有加入时间限制。例如：完成的最后期限、应在多长时间内完成、执行周期等等。同时，其他大量的非实时进程也可能对实时进程造成阻塞，无法确保实时进程的响应时间。
3. 时钟粒度粗糙。时钟管理是操作系统的脉搏，任务的执行和中止在很多情况下都是由时钟直接或间接唤起的，它还是进程调度的重要依据。Linux的周期模式定时器频率仅为100Hz，远不能满足实时应用的要求。

### 操作系统实时性设计

#### 响应时间的分析及解决方法

任务的响应时间被定义为一个事件的发生和任务响应这一事件开始执行之间的间隔时间，通常有以下几个因素影响任务的响应时间。

1. 中断分配时间IDT（interrupt dispatch time）：当一个中断产生时，在调用中断处理程序占用CPU以前，操作系统用来保存所有的寄存器中的内容和系统中其他的关于这一任务状态的时间。
2. 中断服务时间：IST（interrupt service time）：中断服务程序用来从硬件设备读取信息或从操作系统收集信息所用的时间。
3. 内核抢占时间KPT（kernel preemption time）：在操作系统意欲抢占当前进程与抢占实际上发生之间的时间间隔。
4. 调度延迟SD（schedule delay）：调度程序用来调度另一个线程投入运行的时间。
5. 进程切换时间CST（contest switching time）：当前线程用来保存寄存器和系统状态的时间与将要运行的线程恢复寄存器中的内容和系统状态的时间总和。
6. 系统调用返回时间RST（return from systemcall）：处于内核态的线程在它返回用户态之前检查一些状态所用的时间。

以上这些时间中，SD、CST和RST总是固定不变的，如果内核设计得当的话，IDT、IST和KPT可以有效的减少。在实时应用的环境中，若干个中断同时发生的情况是完全可能存在的。这时任务的响应时间最多将包含N(IDT+IST)，N为中断数。

#### 实时调度的算法

常用的实时调度算法有：基于优先级的调度算法（priority-drivenscheduling，PD）；基于时间驱动的调度算法（time-drivenscheduling，TD）；基于比例共享的调度算法（share-drivensched2uling，SD）。

**基于优先级的调度算法**

调度器以优先级作为寻求下一个任务执行的依据。可分为如下两种类型：

1. 静态优先级调度算法：该算法给系统中所有进程都静态的分配一个优先级。静态优先级的分配可以根据应用的属性来进行，例如任务的周期、用户优先级或者其他预先确定的策略。RM（RateMonotonic）是一种典型的静态优先级调度算法，它根据任务执行周期的长短来决定调度优先级，执行周期小的任务具有较高的优先级。
2. 动态优先级调度算法：这种算法根据任务的资源需求来动态的分配任务的优先级。EDF（earliestdeadlinefirst）算法是一种典型的动态优先级调度算法，该算法根据就绪队列中各个任务的截止期限来分配优先级，具有最近截止期限的任的优先级最高。

**基于时间驱动的调度算法**

该算法本质上是一种设计时就确定下来的离线的静态调度方法。在系统的设计阶段，在明确系统中所有处理的情况下，对于各个任务的开始、切换以及结束时间等事先组出明确的安排和设计。

**基于比例共享的调度算法**

这是一种越来越受到关注的实时调度模式，基于GPS（general processor scheduling）的算法，其基本思想就是按照一定的权重（CPU使用的比例）对一组需要调度的任务进行调度，使其执行时间与权重完全成正比。可以通过两种方法来实现比例共享调度算法：

1. 调节各个就绪进程出现在当前调度队列队首的频率，并调度队首的进程执行；
2. 逐次调度就绪队列中的各个进程投入运行，但根据分配的权重调节分配给每个进程的运行时间片。

比例共享算法包括轮转法、公平共享法、公平队列法和彩票调度法等几类。

## 高精度定时器

hrtimer，是High-resolution kernel timers的缩写，从字面意思就知道，这是一个高精度内核timer。hrtimer根据系统的配置和硬件提供更加精确的定时器解决方案。

当hrtimer处于高精度模式时，操作系统把hrtimer\_interrupt()绑定到高精度的时间设备，这时就可以提供纳秒级的定时器服务了。 但是频繁的调用检查hrtimer函数会非常消耗机器性能。 为了避免这个缺陷，hrtimer\_interrupt()的调用采用了one-shot的模式。每一次调用都会设定下一次调用的时间。

在实时系统中，如果逻辑和时序出现偏差会引起严重后果。实时系统广泛应用于工厂生产过程控制、办公自动化、通信、航空航天、民用消费等领域。对于不太复杂的实时系统，一般可设计成前后台式或者超循环式。对于大多数实时系统来说，因为有多任务处理和共享资源的要求，则必须有操作系统支持，这时的操作系统就被称为实时操作系统。隐藏在内部的计算机系统称为嵌入式系统，而实时操作系统又是嵌入式的，则被称为嵌入式实时操作系统。

嵌入式实时操作系统一般由任务调度管理、时间管理、任务同步和通信、内存管理等几部分组成。对于通用操作系统，设置定时在50 ms以下的定时器没有意义的，而实时操作系统由于对时限的要求非常严格，经常设置定时在50 ms以下。因为实时操作系统对时序的偏差依赖性很强，因此作为操作系统时间管理的核心部件??定时器就显得很重要。

在通信领域的产品(如大型程控交换机)中，一次业务呼叫需要几个不同的定时。当同时处理的业务呼叫多达上万个时，则需要大规模(如20 000个)的定时器群。如果定时器的计时方法效率较差，则会因定时器个数太多而造成定时处理耗用系统CPU资源过大，影响定时精度，影响呼叫业务处理的及时性，使定时成为影响整体系统性能的“瓶颈”，因此定时器算法的优劣是影响实时系统性能的重要环节。

## 高可用系统

The difference between hardware and software is that the hardware’s fault is isolated, one can detect the hardware’s fault with instrument.

If we want to build a fault-tolerant system with software, the detection functions should be implemented in kernel, super-world.

We cann’t achieve fault-tolerant system without kernel support.

### 概念

#### MTBF

MTBF,即平均故障间隔时间，英文全称是“Mean Time Between Failure”。

#### 可信性

作为计算机科学的一个学科领域，可信性一般以IEEE国际容错计算会议(FTCS)为标志。由IFIP（国际信息处理联合会）从1979年开始举办的“可靠计算与容错RCFT（Reliable Computing and Fault Tolerance）”会议特此改名为“关键应用可信计算DCCA（Dependable Computing for CriticalApplications）”会议；由IEEE（国际电气与电子工程师学会）主办、IFIP支持和协办的IEEE太平洋沿岸容错系统会议于1999年改名为IEEE可信计算会议，而从1971年起连续举行了29届的IEEE国际容错计算年会（FTCS）到2000年也与IFIP10.4工作组主办的关键应用可信计算DCCA会议合并。从此改名为IEEE“可信系统与网络DSN（Dependable Systems and Networks）”国际学术会议。简言之，可信性指系统在规定时间与环境内可交付可信服务的能力。可信性是一个复杂的综合性概念，它所包含的特征有：可用性、可靠性、防危性、安全性（安全性可细分为保密性与完整性）、可维护性。其中可用性表示系统在给定的时间内可运行的概率，它通常用来度量可延迟或短暂停止提供服务而不会导致系统发生严重后果的品质。可靠性表示系统在给定的时间内连续提供期望服务的能力。防危性表示系统在给定的时间内不发生灾难性事故的概率。保密性表示未经访问许可禁止访问系统敏感数据的能力。完整性指保持数据一致性的能力。可维护性指系统具有可修复和升级的能力。

目前对于在可信性领域的研究主要有：以Dionysius Lardner博士为代表的可信计算（Dependable Computing）概念；源于美国军方可信赖计算（Trusted Computing）；由微软提出的高信度计算（Trustworthy Computing）的概念。

#### 单比特错误/比特反转

“单比特错误”真的会造成整个程序乃至计算机系统的崩溃么？现代计算机有哪些检错、纠错机制？

如果没有误码校验或检纠错机制，单比特翻转是完全有可能造成程序崩溃的。

举个例子：在存在变长编码的系统中，码流中的单比特错误在译码端可能导致连续的码字译码错误，导致错误大规模扩散，这种现象被称为误码扩散。

检纠错机制主要有两大方面：

* + 系统级或硬件级的抗误码，即硬件冗余；
  + 抗误码编解码机制，即编码冗余；

1. 常见的系统级或者硬件级的抗误码机制，一般被称为“三模冗余(Triple Modular Redundancy-TMR)”——核心计算模块由三个完全相同的模块共同完成，三个模块共用一个输入源，各自的输出经过逐比特“表决”后，得到的比特流的误码率会大大降低；只有超过两个模块的输入比特流在同一比特位置均发生比特翻转时，表决码流才会出错；三模冗余可以将线性的误码概率降低至平方级的误码概率。

2. 纠错编码（Error Correcting Code-ECC），纠错编码的算法过于庞杂，但是核心思想大多是通过引入“编码冗余”，增加编码后码字之间的汉明距离（Hamming Distance表征某个编码的抗误码性能），从而达到更强的抗误码性能；

此外，需要补充的一点是：只检错不纠错的检错编码算法，依然具有抗误码能力。因为检错往往即可将误码扩散限制在系统允许的范围内，从而避免程序崩溃。

**怎么解决DRAM硬差错的问题？**

硬差错是永久故障，如果差错在同一个地址发生，只要把该地址列入黑名单，不许使用，就可以防止宕机。如果该单元有许多位需要ECC校正。那么，在较少位差错时就要给出警告。较少位的差错还可以用ECC校正。

操作系统把计算机主存分成页，一般是4K字节为一页。操作系统认为差错来源于很少的几页。事实上，85%的差错来源于0.0001%的页，去掉这些页就可以保持正常运行，而对存储能力影响很小。这被称为页退役。但很少有人用它，因为不知道如何检测和诊断到这种故障。我们的研究组使Facebook在其数据中心用了页退役。我们研究了5个策略。一个是发现第一个差错就退役这一页。其他则是退役一行或一列。我们发现，一般一个计算机退役1M字节就可以防止90%的存储器差错。去掉1M字节对于一个计算机系统来说是无足轻重的。当然，技术人员可以更换整个存储模块。但是，它其实只有几个单元是坏的。页退役不但能防止宕机，而且节省了存储器。

同样的策略也可用于智能手机、平板电脑、和可穿戴电子等设备。DRAM在广泛使用，DRAM一个单元差错就换整个设备，显然是过于浪费。

#### ALARP准则

风险接受准则的确定是一个决策过程。在风险分析中，ALARP准则是最常用的风险可接受准则，如图 17‑1所示。ALARP准则最早是由英国健康、安全和环境部门(Health Safety and environment,HSE)提出的进行风险管理和决策的准则，现已成为可接受风险标准确立的基本框架。ALARP准则适用于对个人死亡风险、环境风险和财产风险的评估。

ALARP准则的含义是：任何工业活动都具有风险，不可能通过预防措施来彻底消除风险，必须在风险水平与利益之间做出平衡。 如图 17‑1所示，ALARP准则包括两条风险分界线（容许上限和容许下限），分别称为可接受风险上限、可接受风险下限。两条线将风险分为3个区域：风险不可接受区、合理可行的最低限度区（ALARP区）、风险可接受区。若风险评价所得的风险等级落在风险不可接受区，除特殊情况外，该风险无论如何不能被接受。对处于设计阶段的装置，该设计方案不能被通过；对现有装置，必须立即停止生产，采取强制性措施降低风险水平。若风险等级处在风险可接受区，由于风险水平很低，无需采取安全改进措施。若风险等级处在ALARP区，则需要考察实施各种降低风险水平措施后的后果，并进行成本效益分析，据此确定该风险是否可以接受。如果增加危险防范措后，对降低系统风险水平无显著影响，则可以认为该风险不可接受。

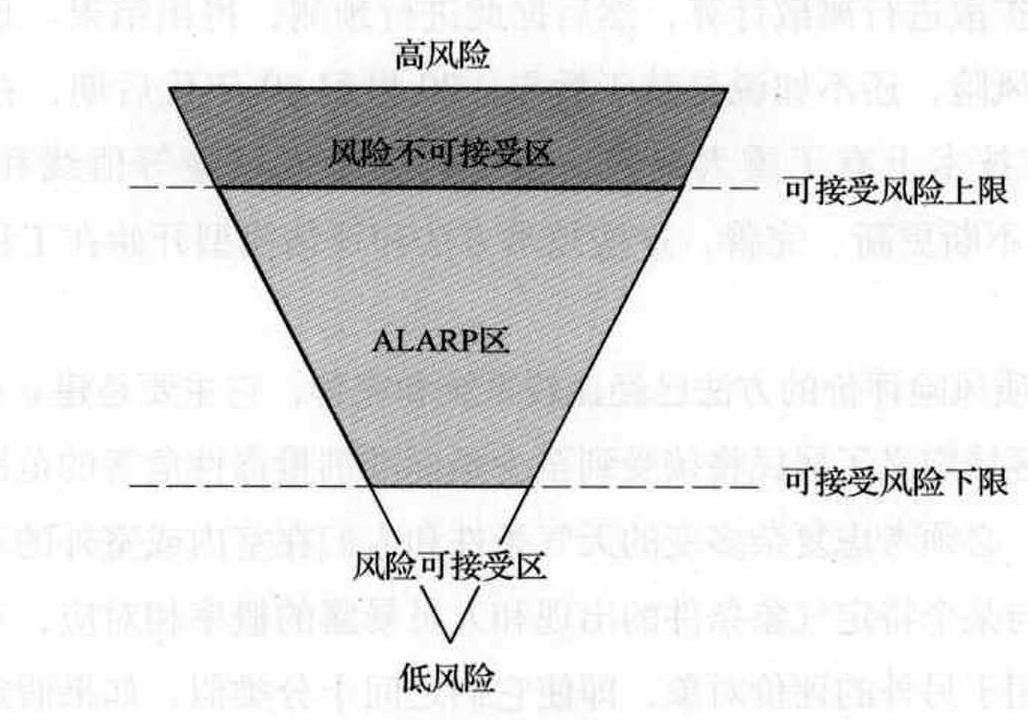


图 ‑1 ALARP风险可接受准则

### 软件容错技术

#### 恢复块技术（Recovery Blocks）

代码热替换

热代码升级 - 一些系统不能够由于软件维护而停止运行。Erlang允许程序代码在运行系统中被修改。旧代码能被逐步淘汰而后被新代码替换。在此过渡期间，新旧代码是共存的。这也使得安装bug补丁、在运行系统上升级而不干扰系统操作成为了可能。

递增式代码装载 - 用户能够控制代码如何被装载的细节。在嵌入式系统中，所有代码通常是在启动时就被完全装载。而在开发系统中，代码是按需装载的，甚至在系统运行时被装载的。如果测试到了未覆盖的bug，那么只有具有bug的代码需要被替换。

The recovery block method is a simple method developed by Randell from what was observed as somewhat current practice at the time. [Lyu95] The recovery block operates with an adjudicator which confirms the results of various implementations of the same algorithm. In a system with recovery blocks, the system view is broken down into fault recoverable blocks. The entire system is constructed of these fault tolerant blocks. Each block contains at least a primary, secondary, and exceptional case code along with an adjudicator. (It is important to note that this definition can be recursive, and that any component may be composed of another fault tolerant block composed of primary, secondary, exceptional case, and adjudicator components.) The adjudicator is the component which determines the correctness of the various blocks to try. The adjudicator should be kept somewhat simple in order to maintain execution speed and aide in correctness. Upon first entering a unit, the adjudicator first executes the primary alternate. (There may be N alternates in a unit which the adjudicator may try.) If the adjudicator determines that the primary block failed, it then tries to roll back the state of the system and tries the secondary alternate. If the adjudicator does not accept the results of any of the alternates, it then invokes the exception handler, which then indicates the fact that the software could not perform the requested operation.

Recovery block operation still has the same dependency which most software fault tolerance systems have: design diversity. The recovery block method increases the pressure on the specification to be specific enough to create different multiple alternatives that are functionally the same. This issue is further discussed in the context of the N-version method.

The recovery block system is also complicated by the fact that it requires the ability to roll back the state of the system from trying an alternate. This may be accomplished in a variety of ways, including hardware support for these operations. This try and rollback ability has the effect of making the software to appear extremely transactional, in which only after a transaction is accepted is it committed to the system. There are advantages to a system built with a transactional nature, the largest of which is the difficult nature of getting such a system into an incorrect or unstable state. This property, in combination with checkpointing and recovery may aide in constructing a distributed hardware fault tolerant system.

#### N版本技术（N-version Software）

The N-version software concept attempts to parallel the traditional hardware fault tolerance concept of N-way redundant hardware. In an N-version software system, each module is made with up to N different implementations. Each variant accomplishes the same task, but hopefully in a different way. Each version then submits its answer to voter or decider which determines the correct answer, (hopefully, all versions were the same and correct,) and returns that as the result of the module. This system can hopefully overcome the design faults present in most software by relying upon the design diversity concept. An important distinction in N-version software is the fact that the system could include multiple types of hardware using multiple versions of software. The goal is to increase the diversity in order to avoid common mode failures. Using N-version software, it is encouraged that each different version be implemented in as diverse a manner as possible, including different tool sets, different programming languages, and possibly different environments. The various development groups must have as little interaction related to the programming between them as possible. [Avizienis85] N-version software can only be successful and successfully tolerate faults if the required design diversity is met.

The dependence on appropriate specifications in N-version software, (and recovery blocks,) can not be stressed enough. The delicate balance required by the N-version software method requires that a specification be specific enough so that the various versions are completely inter-operable, so that a software decider may choose equally between them, but cannot be so limiting that the software programmers do not have enough freedom to create diverse designs. The flexibility in the specification to encourage design diversity, yet maintain the compatibility between versions is a difficult task, however, most current software fault tolerance methods rely on this delicate balance in the specification.

The N-version method presents the possibility of various faults being generated, but successfully masked and ignored within the system. It is important, however, to detect and correct these faults before they become errors. First, the classification of faults applied to N-version software method: if only a single version in an N-version system, the error is classified as a simplex fault. If M versions within an N-version system have faults, the the fault is declared to be an M-plex fault. M-plex faults are further classified into two classes of faults of related and independent types. Detecting, classifying, and correcting faults is an important task in any fault tolerant system for long term correct operation.

The differences between the recovery block method and the N-version method are not too numerous, but they are important. In traditional recovery blocks, each alternative would be executed serially until an acceptable solution is found as determined by the adjudicator. The recovery block method has been extended to include concurrent execution of the various alternatives. The N-version method has always been designed to be implemented using N-way hardware concurrently. In a serial retry system, the cost in time of trying multiple alternatives may be too expensive, especially for a real-time system. Conversely, concurrent systems require the expense of N-way hardware and a communications network to connect them. Another important difference in the two methods is the difference between an adjudicator and the decider. The recovery block method requires that each module build a specific adjudicator; in the N-version method, a single decider may be used. The recovery block method, assuming that the programmer can create a sufficiently simple adjudicator, will create a system which is difficult to enter into an incorrect state. The engineering tradeoffs, especially monetary costs, involved with developing either type of system have their advantages and disadvantages, and it is important for the engineer to explore the space to decide on what the best solution for his project is.

#### 自检查技术（Self-Checking Software）

Self checking software is not a rigorously described method in the literature, but rather a more ad hoc method used in some important systems. [Lyu95] Self-checking software has been implemented in some extremely reliable and safety-critical systems already deployed in our society, including the Lucent ESS-5 phone switch and the Airbus A-340 airplanes. [Lyu95]

Self-checking software are the extra checks, often including some amount checkpointing and rollback recovery methods added into fault-tolerant or safety critical systems. Other methods including separate tasks that "walk" the heap finding and correcting data defects and the options of using degraded performance algorithms. While self-checking may not be a rigorous methodology, it has shown to be surprisingly effective.

The obvious problem with self-checking software is its lack of rigor. Code coverage for a fault tolerant system is unknown. Furthermore, just how reliable a system made with self-checking software? Without the proper rigor and experiments comparing and improving self-checking software cannot effectively be done.

### 借鉴Erlang

《面对软件错误构建可靠的分布系统》 写道

Erlang 的世界观可以归纳为如下的一些观念：

* 一切皆进程。
* 进程强隔离。
* 进程的生成和销毁都是轻量的操作。
* 消息传递是进程交互的唯一方式。
* 每个进程有其独有的名字。
* 你若知道进程的名字，你就可以给它发消息。
* 进程之间不共享资源。
* 错误处理非本地化。
* 进程要么好好跑，要么死翘翘

Erlang创始人Joe Armstrong对Erlang世界观的总结，其中也包含了编写大型高并发可容错的软件系统应遵循的最基本原则：

1. 隔离，破除数据共享；
2. 消息传递实现并行协作；
3. 速错且异地纠错。

#### erlang进程监控：link和monitor

Erlang最开始是为了电信产品而发展起来的语言，因为这样的目的，决定了她对错误处理的严格要求。Erlang除了提供exception，try catch等语法，还支持Link和Monitor两种监控进程的机制，使得所有进程可以连接起来，组成一个整体。当某个进程出错退出时，其他进程都会收到该进程退出的消息通知。有了这些特点，使用erlang建立一个简单，并且健壮的系统就不是什么难事。

*一个事务没有按约定时间完成，也可以被monitor。*

* 进程双向监控-Link

link方式可以建立进程之间的双向链接关系，当其中一个进程退出时，另一个进程会收到该进程退出的消息。

* 进程单向监控-Monitor

Monitor方式则实现进程的单向监控，当被监控进程退出时，监控进程会收到该进程退出的消息。

## 安全

#### 安全向左，可靠向右

平衡好安全和其它（RAM）的问题，典型的是平衡好安全和可靠，既不能过度设计，也不能欠设计。

经验表明，安全就是个无底洞。为了保证安全，为了避免各种隐患，我们总是给安全功能叠加各种各样的措施，以至于设计越来越复杂。但安全性和RAM之间，却是一对矛盾，为了安全所增加的硬件，导致了可靠性的下降，维护复杂性的提升，毕竟多了一个物件，就多了一个故障点。所以，需要平衡好安全和RAM。那平衡的依据是什么呢？复杂度要控制到什么程度才算合理呢？可以从很多个角度回答这个问题，避开技术自身，追根溯源，我们捋一捋这个问题的本末，看看在设计始末到底发生了什么。讨论的逻辑从以人为本出发，将人的需求解构为功能，和对应的非功能性支撑和一些商业因素。

### 故障安全属性

如果一个器件的工作状态是可知的，就说明该器件具备固有故障安全属性。

安全设计原则是欧标体系下重要的设计和开发技术，常用到的安全设计原则包括功能安全原则和技术安全原则。固有故障安全原则属于技术安全原则中的故障安全系列。常见的故障安全原则包括了：固有故障安全原则、组合式安全原则和反应式故障安全原则。

### 固有故障安全原则

固有故障安全原则是指：允许安全功能由单一部件来完成，且该部件自身所有的故障模式没有危险性，都是可以预见的，可信的，并需要在指定的应用环境下，结合该部件的故障模式，分析该部件的所有特性。固有故障安全的部件除了能够单独实现安全功能外，也可以用在组合式或反应式故障安全系统的某些功能上，如危险事件发生时切断系统输出。安全设计原则通常应用在产品技术层面，典型的应用是表决器设计、I/O设计、防护设备设计等。常用的措施包括FMEA分析、保护性编码技术等。

#### 失效模式与影响分析（FMEA）

失效模式与影响分析（英文：Failure mode and effects analysis，FMEA），又称为失效模式与后果分析、失效模式与效应分析、故障模式与后果分析或故障模式与效应分析等，是一种操作规程，旨在对系统范围内潜在的失效模式加以分析，以便按照严重程度加以分类，或者确定失效对于该系统的影响。FMEA广泛应用于制造业产品生命周期的各个阶段；而且，FMEA在服务行业的应用也在日益增多。失效原因是指加工处理、设计过程中或项目/物品（英文：item）本身存在的任何错误或缺陷，尤其是那些将会对消费者造成影响的错误或缺陷；失效原因可分为潜在的和实际的。影响分析指的是对于这些失效之处的调查研究。

**基本术语**

* 失效模式（又称为故障模式）

观察失效时所采取的方式；一般指的是失效的发生方式。

* 失效影响（又称为失效后果、故障后果）

失效对于某物品/项目（英文：item）之操作、功能或功能性，或者状态所造成的直接后果。

* 约定级别（又称为约定级）
* 代表物品/项目复杂性的一种标识符。复杂性随级数接近于1而增加。
* 局部影响：仅仅累及所分析物品/项目的失效影响。
* 上阶影响：累及上一约定级别的失效影响。
* 终末影响：累及最高约定级别或整个系统的失效影响。
* 失效原因（又称为故障原因）

作为失效之根本原因的，或者启动导致失效的某一过程的，设计、加工处理、质量或零部件应用方面所存在的缺陷

* 严重程度（又称为严重度）

失效的后果。严重程度考虑的是最终可能出现的损伤程度、财产损失或系统损坏所决定的，失效最为糟糕的潜在后果。

## 思考题

#### Magenta+lua 🡺 Erlang

假如我做一个操作系统内核，有巨快、巨轻量的process分配能力，每个process里我跑一个轻型的角本语言引擎，如lua，每个process之间，有Erlang那样的link、monitor能力。是不是能够造出象Erlang一样的容错计算模型？

参考：<http://blog.codingnow.com/2012/09/the_design_of_skynet.html>

充分利用CPU多核优势，将不同的业务放在独立的执行环境中处理，协同工作。这个执行环境，最早的时候，我期望是利用 OS 的进程，后来发现，如果我们必定采用嵌入式语言，比如 Lua 的话，独立 OS 进程的意义不太大。Lua State 已经提供了良好的沙盒，隔离不同执行环境。而多线程模式，可以使得状态共享、数据交换更加高效。而多线程模型的诸多弊端，比如复杂的线程锁、线程调度问题等，都可以通过减小底层的规模，精简设计，最终把危害限制在很小的范围内。

#### Actor模式

Actor模式是一种并发模型，与另一种模型共享内存完全相反，Actor模型share nothing。所有的线程（或进程）通过消息传递的方式进行合作，这些线程（或进程）称为Actor。共享内存更适合单机多核的并发编程，而且共享带来的问题很多，编程也困难。随着多核时代和分布式系统的到来，共享模型已经不太适合并发编程，因此几十年前就已经出现的Actor模型又重新受到了人们的重视。MapReduce就是一种典型的Actor模式，而在语言级对Actor支持的编程语言Erlang又重新火了起来，Scala也提供了Actor，但是并不是在语言层面支持，Java也有第三方的Actor包，Go语言channel机制也是一种类Actor模型。

Actor模型原理：

Actor模型=数据+行为+消息。

Actor模型内部的状态由自己的行为维护，外部线程不能直接调用对象的行为，必须通过消息才能激发行为，这样就保证Actor内部数据只有被自己修改。

# 问答Q&A

## 程序技术问题

#### 下面这两行ARM汇编，str那句什么意思？

mrc p15, 0, sp, c9, c13, 0

str sp, [pc, #\_\_irq\_cycle\_count - . - 8]

答：

str sp, [pc, <offset>] 其中offset为\_\_irq\_cycle\_count地址减去本指令地址再减去8。

#### WITH宏定义

一个模块，是否编译进内核，最终编译的c/c++头文件是config-kernel.h，它是在make阶段生成的。

源码：$Magenta/magenta/kernel/engine.mk

# generate a config-kernel.h file with all of the KERNEL\_DEFINES laid out in #define format

kernelconfigheader:

$(KERNEL\_CONFIG\_HEADER): kernelconfigheader

@$(call MAKECONFIGHEADER,$@,KERNEL\_DEFINES,"")

源码：$Magenta/magenta/kernel/make/module.mk

# kernel module

# add a local include dir to the global include path for kernel code

KERNEL\_INCLUDES += $(MODULE\_SRCDIR)/include

KERNEL\_DEFINES += $(addsuffix =1,$(addprefix WITH\_,$(MODULE\_SHORTNAME)))

MODULE\_SRCDEPS += $(KERNEL\_CONFIG\_HEADER)

以evlog为例解释一下这些角本的工作：

MODULE\_SHORTNAME：lib/evlog

KERNEL\_DEFINES中增加的内容：WITH\_lib/evlog=1

经过MAKECONFIGHEADER宏，把KERNEL\_DEFINES中增加的内容变为：

#define WITH\_LIB\_EVLOG 1

写到config-kernel.h中，然后其它依赖于这个库的地方，就通过宏WITH\_LIB\_EVLOG判断库evlog（位于目录：kernel/lib/evlog）是否编译到内核里面了。

#### 如何让一个kernel/lib参与内核编译

kernel/lib如下evlog（Magenta/magenta/kernel/lib/evlog/rules.mk），它要想参加编译，需要加到$Magenta/magenta/kernel/lib/magenta/rules.mk 文件中。

MODULE\_DEPS := \

lib/dpc \

lib/mxtl \

lib/evlog \

dev/interrupt \

编译内核时，会定义一个宏

KERNEL\_DEFINES += WITH\_lib/evlog=1

在编译模块时，在该模块的make文件中，加编译需要定义的宏。如evlog，参加编译后，需要提供宏：WITH\_KERNEL\_EVLOG。

源码：$Magenta/magenta/kernel/lib/evlog/rules.mk

LOCAL\_DIR := $(GET\_LOCAL\_DIR)

MODULE := $(LOCAL\_DIR)

KERNEL\_DEFINES += WITH\_KERNEL\_EVLOG=1

MODULE\_SRCS += \

$(LOCAL\_DIR)/evlog.c

include make/module.mk

#### mxtl::move()之于移动构造

RefPtr这样的智能指针本身是一个结构体struct，它里面有成员负责记录被它管理的资源的指针，然后是引用计算之类的东西，当一个RefPtr在函数间传递时，调用者拥有的RefPtr结构体里面的内容，需要装到被调用者的RefPtr结构体里面去，并且调用者的RefPtr的生命周期不再与它管理的资源的指针一致，这就涉及到c++的移动构造的问题。

首先读懂mxtl::move()模板，知道mxtl::move()是如何把类型右值化。

请特别注意T类移动构造和移动赋值函数的参数的类型，它是T&&。没错，是两个&&符号：

* + 如果是T&&（两个&&符号），则表示是T的右值引用类型。
  + 如果是T&（一个&符号），则表示是T的引用类型。和右值引用相比，这种引用也叫左值引用。

什么是左值，什么是右值？

* + 左值是有名字的，并且可以取地址。
  + 右值是无名的，不能取地址。

源码：$Magenta/magenta/system/ulib/mxtl/include/mxtl/type\_support.h

template <typename T>

struct remove\_reference {

using type = T;

};

template <typename T>

struct remove\_reference<T&> {

using type = T;

};

template <typename T>

struct remove\_reference<T&&> {

using type = T;

};

template <typename T>

constexpr typename remove\_reference<T>::type&& move(T&& t) {

return static\_cast<typename remove\_reference<T>::type&&>(t);

}

以下面这段程序为例（注意带下划线的内容）：

UserThread::UserThread(mx\_koid\_t koid,

mxtl::RefPtr<ProcessDispatcher> process,

uint32\_t flags)

: koid\_(koid),

process\_(mxtl::move(process)),

state\_tracker\_(true, mx\_signals\_state\_t{0u, MX\_SIGNAL\_SIGNALED}) {

LTRACE\_ENTRY\_OBJ;

}

// process\_的定义如下：

// a ref pointer back to the parent process

mxtl::RefPtr<ProcessDispatcher> process\_;

我们希望process\_ 这个赋值是一个移动构造，把RefPtr中指向的内容移到我新的RefPtr中，在翻译原RefPtr时，不要把它原来指向的内容指针给释放了。

RefPtr的移动构造函数如下所示：

template <typename T>

class RefPtr final {

public:

// Move

RefPtr(RefPtr&& r) : ptr\_(r.ptr\_) {

r.ptr\_ = nullptr;

}

#### c++右值引入

参考：<https://www.devbean.net/2012/11/biggest-changes-in-c11/>

C++03 的引用类型只能绑定左值。C++11 引入了新的引用类型——右值引用。右值引用可以绑定右值，例如临时变量和字面常量。

增加右值引用的主要原因是移动语义。不同于传统的拷贝，移动的含义是目标对象占有源对象的资源，将源对象设置为“空”状态。在这种情景下，拷贝一个对象既昂贵又不必要，应该使用移动操作符。为感受移动语义在性能上的优势，考虑交换字符串。一个原始的实现类似于：

void naiveswap(string &a, string & b)

{

string temp = a;

a = b;

b = temp;

}

这种实现很昂贵。拷贝字符串需要分配内存，将字符从源对象复制到目标对象。相比而言，移动字符串仅仅意味着交换两个数据成员，不需要分配内存、复制字符数组和释放内存：

void moveswapstr(string& empty, string & filled)

{

// 伪代码，体会思想

size\_t sz = empty.size();

const char \*p = empty.data();

// 移动 filled 的资源到 empty

empty.setsize(filled.size());

empty.setdata(filled.data());

// filled 变成空的了

filled.setsize(sz);

filled.setdata(p);

}

如果你正在实现一个支持移动的类，需要声明一个移动构造函数和移动复制运算符：

class Movable

{

Movable (Movable&&); // 移动构造函数

Movable&& operator=(Movable&&); // 移动复制运算符

};

C++11 标准库大量使用了移动语义。许多算法和容易也为移动语义做了优化。

#### 为什么要定义\_ONE这个宏

看下面一段程序，为什么要定义\_ONE这个宏？因为\_ONE这个宏，使得BIT这个宏可以处理各种整数，8位的、16位的、32位的、64位的，\_ONE“返回”的是一个与x同比特数的整数，而不是与当前编译目标字长有关的整数，这样的32位目标环境，处理64位的整数计算，也不会出错。

/\* Trick to get a 1 of the right size \*/

#define \_ONE(x) (1 + ((x) - (x)))

#define BIT(x, bit) ((x) & (\_ONE(x) << (bit)))

#### gcc内置原子操作函数

原子操作包含三个方面的语义：操作本身是不可分割的（Atomicity），一个线程对某个数据的操作何时对另外一个线程可见（Visibility），执行的顺序是否可以被重排（Ordering）。

gcc从4.1.2提供了\_\_sync\_\*系列的built-in函数，用于提供加减和逻辑运算的原子操作。注意:这是gcc编译器自带的原子函数，不能跨平台交叉编译。

gcc还有这一组内置函数.

//先做操作，再返回变化后的值

* type \_\_sync\_fetch\_and\_add (type \*ptr, type value);
* type \_\_sync\_fetch\_and\_sub (type \*ptr, type value);
* type \_\_sync\_fetch\_and\_or (type \*ptr, type value);
* type \_\_sync\_fetch\_and\_and (type \*ptr, type value);
* type \_\_sync\_fetch\_and\_xor (type \*ptr, type value);
* type \_\_sync\_fetch\_and\_nand (type \*ptr, type value);

//先返回变化前的值，再做操作

* type \_\_sync\_add\_and\_fetch (type \*ptr, type value);
* type \_\_sync\_sub\_and\_fetch (type \*ptr, type value);
* type \_\_sync\_or\_and\_fetch (type \*ptr, type value);
* type \_\_sync\_and\_and\_fetch (type \*ptr, type value);
* type \_\_sync\_xor\_and\_fetch (type \*ptr, type value);
* type \_\_sync\_nand\_and\_fetch (type \*ptr, type value);

按照官方文档的说法，

GCC will allow any integral scalar or pointer type that is 1, 2, 4, or 8 bytes in length. 16-byte integral types are also allowed if `\_\_int128' (see \_\_int128) is supported by the architecture.

也就是说至少支持长度为 1，2，4，8的整形。16位是否支持取决于体系结构

参考：

* Built-in Functions for Memory Model Aware Atomic Operations

<https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html>

* Built-in functions for atomic memory access

<https://gcc.gnu.org/onlinedocs/gcc-4.4.5/gcc/Atomic-Builtins.html>

这里，对\_\_ATOMIC\_RELEASE、\_\_ATOMIC\_RELAXED等关键字有详细的解释。

* Memory model synchronization modes

<https://gcc.gnu.org/wiki/Atomic/GCCMM/AtomicSync>

#### 为什么这个函数生成的数是随机的

**问：**生成随机数的函数的一个实现：

int rand(void)

{

return (randseed = randseed \* 1664525 + 1013904223);

}

为什么这个函数生成的数是随机的？

**答：**

这属于线性同余（linear congruential generator）求伪随机数的一种方法。这里的 randseed 应该定义为 uint32 类型，所以 randseed = randseed \* 1664525 + 1013904223 实际上是 randseed = (randseed \* 1664525 + 1013904223) mod 2^32 。满足线性同余法的一般形式：x\_n = (a \* x\_n-1 + c) mod M

条件是：1. 0<= x\_0, a, c <M ；2. c 和 M 互质； 3. 如果 p 是 M 的质数因子，那么 a-1 也必须能被 p 整除； 4. 如果 M 能被 4 整除，那么 a-1 也必须能被 4 整除。

自己的操作系统内核，如何支持c++异常机制

<http://wiki.osdev.org/C%2B%2B_Exception_Support>

If the g++ that targets your operating system follows the Itanium C++ ABI (GCC does so for almost all architectures), you can follow this article and add C++ exception support to your kernel.

In the ABI, C++ exception is supported by the cooperation of three layers. The first layer is the compiler. The compiler translates the "try" "catch" "throw" statements into calls to specific functions in C++ runtime. The second layer is the C++ runtime. For exception support this layer is more or less a wrapper around the third layer. All the dirty work are done by the third layer: the unwind library. BTW, the second layer also adds support for RTTI & dynamic\_cast

So, in order to add C++ exception support, you just need to port the second and the third layer into you kernel, which means you need to either port libsupc++ or port libcxxrt and an unwind library. I prefer the second option, because libsupc++ is part of GCC, it's even not trivial to compile it separately. For the unwind library part, you have two options, libunwind and libgcc\_eh. libunwind is more difficult to port, it depends on pthread and some type definitions in <elf.h> <link.h> and <ucontext.h>. The major reason you may want libunwind instead of libgcc\_eh is that libgcc\_eh is licensed under LGPL.

If you don't have any of the unix header files when compiling libcxxrt or libgcc\_eh, e.g. <unistd.h>, create an empty one and try again. If you still can't compile libcxxrt or libgcc\_eh, find a copy of the corresponding header file from a unix system and find out exactly what you are missing.

## 计算机体系结构architecture相关

#### IPI： Inter-Processor Interrupt（处理器之间的中断）

IPI（inter-processor interrupt）是一种特别的中断。在对称多处理器 (SMP）环境下，它可以被任意一个处理器用来对另一个处理器产生中断。IPIs典型地被用来实现高速缓存间的一致性同步（Cache Coherency Synchronization）。

ARM多核多线程处理器的中断由PIC(Programmable Interrupt Controller）统一控制。PIC 允许一个硬件线程中断其他的硬件线程，这种方式被称为核间中断（Inter-Processor Interrupts，IPI）。PIC 拥有一个宽度为 32 位的核间中断寄存器IPIBase，该寄存器包含目的线程的编号、中断向量及中断类型(是否中断多个硬件线程）等内容。核间中断可以通过向这个寄存器写入需要的值来产生。若硬件线程 A 想要发送一个核间中断给硬件线程 B，它只需要向寄存器 IPIBase 中写入 B的 Thread ID、中断向量、中断类型等值就可以了，PIC 会通知 B 所在的核挂起它当前的执行序列，并根据中断向量跳转到中断服务例程 ISR 的入口。使用 IPI 进行核间通信的关键在于要利用中断服务例程 ISR 去读取一个事先约好的共享内存区域。发起方首先将消息写到一块共享内存中，然后发起核间中断。被中断的硬件线程在中断服务例程中读取该内存，以获得发起方通知的消息。为防止多核间的竞争导致消息被改写，使用这种方式必须利用锁机制来确保消息的完整性。

源码：$Magenta/magenta/kernel/include/arch/mp.h

/\* send inter processor interrupt, if supported \*/

status\_t arch\_mp\_send\_ipi(mp\_cpu\_mask\_t target, mp\_ipi\_t ipi);

#### SOC VID、PID、DID

System on Chip (SoC) binding variables

vendor ID (VID) and product ID (PID)，设备号DID、制造商号VID

#### 什么是Cache Line

参考：<http://cenalulu.github.io/linux/all-about-cpu-cache/>

Cache Line可以简单的理解为CPU Cache中的最小缓存单位。目前主流的CPU Cache的Cache Line大小都是64Bytes。假设我们有一个512字节的一级缓存，那么按照64B的缓存单位大小来算，这个一级缓存所能存放的缓存个数就是512/64 = 8个。

### ARM

#### Address Space Identifier - ASID

参考：<https://community.arm.com/thread/8219>

**问：**

For ARMv7 -A/R systems, the MMU uses an ASID to distinguish between memory pages which have the same virtual address, but which are used by an individual task ( I.e. A task which uses non-Global memory). The ASID is an eight-bit value, from 0-255, assigned by the Operating System.

So, does that mean, for a normal, e.g. Linux system, there can be no more than 256 non-Global tasks running at one time? Should we assume that a piece of multi-threaded code would have a unique Task/Process Identifier (for scheduling purposes), for each thread, but all its threads would have the same ASID?

And what happens in a Type1 virtual system, with a Hypervisor and say, two Guest OS's? Presumably, each Guest kernel has its own ASID? And which OS is assigning the ASID to user tasks?

**答：**

要清楚一点，只有就绪schedule调度队列才能参与到ASID的分派，有ASID的执行单位，意味着它随时可以被执行到。

Linux uses a rollover mechanism for ASID - there can only be 256 ASIDs allocated within the system, and it maintains a bitmap of ASIDs currently in use. So what it does is allocate them per task until it runs out. When a task exits, the ASID is invalidated from the branch predictor, caches and TLBs and can be reallocated to another task. If all the ASIDs are in use, it will invalidate the branch predictor and caches and TLBs and start again allocating new ASIDs on a first-come first-serve basis (so tasks that never want to run don't waste ASIDs).

This works because even on a system with 100,000 "running" tasks, you probably only have 4-8 cores, and only one task can run at any time per core. Until you can throw 256 cores in an ARM system, the current behaviour will work fine (most systems are pretty far from that and, by the time it is a problem, 16-bit ASIDs will be de jure).

For threading (not just on Linux), since every "process" has an ASID, all "threads" within that process will share the ASID. I quote those because different operating systems call these things different names.

For virtual machines, there is a VMID - each guest OS has a VMID and each guest OS has tasks with an ASID. A cache lookup requires both to match a translation, so the ASID rollover code won't clobber entries for another guest OS.

### X86

#### Avoiding a problem with the SYSRET instruction

On x86-64, the kernel uses the SYSRET instruction to return from system calls. We must be careful not to use a non-canonical return address with SYSRET, at least on Intel CPUs, because this causes the SYSRET instruction to fault in kernel mode, which is potentially unsafe. (In contrast, on AMD CPUs, SYSRET faults in user mode when used with a non-canonical return address.)

Usually, the lowest non-negative non-canonical address is 0x0000800000000000 (== 1 << 47). One way that a user process could cause the syscall return address to be non-canonical is by mapping a 4k executable page immediately below that address (at 0x00007ffffffff000), putting a SYSCALL instruction at the end of that page, and executing the SYSCALL instruction.

To avoid this problem, we disallow mapping a page when the virtual address of the following page will be non-canonical.

For more background, see "A Stitch In Time Saves Nine: A Case Of Multiple OS Vulnerability", Rafal Wojtczuk (

<https://media.blackhat.com/bh-us-12/Briefings/Wojtczuk/BH_US_12_Wojtczuk_A_Stitch_In_Time_WP.pdf> ).

## 高阶逻辑在操作系统开发中的应用

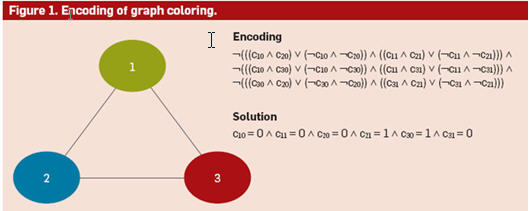
* 我在程序运行过程中，详细记录它的运行时信息，如函数的进、出，中断上下文的切换等等，我可不可能利用高阶逻辑验证这个程序的状态，如是否死机，是否处于安全safety状态？

你说的是软件漏洞检测问题，现在很多归为测试来研究的，被测进程状态可控可测是一个难点。如果是需要做实时检测，面临大状态空间的效率问题，符号模型检测甚至随机模型检测可能需要的，有的还可以结合仿真模型做跟踪性检测，以前用spin的状态机模型，现在这些自动机模型花样很多的，因为运行时环境变化必然导致模型也在改变，必须频繁进行检测，这种检测会受到系统能力的约束，所以倒不如说运行时模型诊断好，诊断系统行为和安全属性是否符合安全方面的需求和规范，建议关注一下这些研究，静态的测试和漏洞挖掘再结合模型检测。

### 布尔可满足性问题

计算机里的位变量都是布尔变量，只取值0或1。两个布尔变量之间的运算只有“与”“或”“非”，表示为“∧”“∨”“¬”。你能猜到 1∧0=0，1∨0=1，¬1=0。现在我有一个包含许多变量的表达式，例如：x∨y∧¬z，我问：这个表达式能等于1吗？如果能，x,y,z分别等于什 么，表达式就等于1，就说这个表达式被满足了。这就是布尔可满足性（简称SAT）问题。但是，你要允许变量个数可以任意，表达式可以任意长。这个问题是一 个计算复杂性很高的问题，已经证明它基本上是一个多项式复杂性算法不可解决的问题。这句话，我无法用通俗的语言来说明。但是没有关系，你只要知道这问题没 有一般的简单算法来彻底解决。本文要说的是：这个问题有实际意义，而且最近在算法上有进展和应用。

一个经典的图包含节点和边，问：给每一个节点着色，使相邻节点的颜色不同，四个颜色够不够？这就是图论中著名的四色问题。这个问题可以转化为布尔可满足性问题。四个颜色可以分别编码为00，01，10，11。例如，从图1中，节点2着色记为（c20,c21）。节点i与节点j不能同色可表示为（ci0∧cj0）∨(ci1∧cj1)=0。你可以看到，对于这个简单的图，约束可以用上面的编码表达式表示，要求其为1。下面就是解。



布尔可满足性问题的扩张就是优化问题。如果变量的取值不只有0和1，目标函数的取值也不只是0和1，而是取最大或最小，而约束条件可以表示为若干个表达式。我们的问题是寻找变量的一组值，在满足约束条件下，目标函数取值最大或最小。这不就是优化问题吗？

布尔可满足性问题有许多应用，许多问题都可以归结为SAT。特别是在集成电路的设计与测试中，到处都是。例如，看两个逻辑函数f1,f2是否等价，只要判断（f1∧¬f2）∨(¬f1∧f2)是否永远不可满足。

一般先把待满足的逻辑表达式变成和之积的形式CNF，即每一项都是“或”，所有的项“与”起来，可称“或与式”。例如， (¬x2∨x3)∧(¬x1∨x3)∧(x1∨x2∨¬x3)。可以证明，任何逻辑函数都可以或与式表示。要让它为1，只要让所有项（子句）皆为1就可以 了。而要让某一项为1，只要其中一个文字为1就可以了。首个变量赋值的基本方法当然是随机，再加上推理、学习、回溯等等功能，可以比较有效地使表达式为 1。但一般要考虑表达式的特殊性，才能使算法更有效。

## 操作系统领域顶级国际学术会议

#### DSN

IEEE The International Conference on Dependable Systems and Networks

IEEE可靠系统和网络会议，是IEEE容错计算技术委员会主办的最重要的国际会议，也是可靠系统和网络领域历史悠久、地位很高的学术会议。

网址：<http://ieeexplore.ieee.org/servlet/opac?punumber=1000192>

#### USENIX Annual Technical Conference

操作系统、体系结构方面最好的会议之一

网址：<https://www.usenix.org/conferences>

#### OSDI

Symposium on Operating System Design and Implementation（OSDI）

操作系统最好的会议和SOSP交替举行，每两年一届，操作系统的旗舰会议。

提到OSDI，就得提到另一个会议： SOSP。这两个是操作系统领域最好的会议，每两年开一次，轮流开，比如今年是OSDI，那么明年就是SOSP。由于这两个会议方向很广，因此影响很大。

#### SOSP

ACM Symposium on Operating Systems Principles （SOSP）

操作系统最好的会议和OSDI交替举行，每两年一届，操作系统旗舰会议操作系统。

网址：<http://sosp.org/>

#### ASPLOS

International Conference on Architectural Support for Programming Languages and Operating Systems （ASPLOS）

操作系统和程序语言最好的会议之一。

网址：<http://www.sigplan.org/Conferences/ASPLOS/>

#### HOTOS

Workshop on Hot Topics in Operating Systems （HOTOS）

操作系统最好的会议之一

网址：<https://www.usenix.org/conferences/byname/155>

#### POPL

POPL: Annual Symposium on Principles of Programming Languages

关注程序设计语言的设计、定义、分析、实现的国际权威会议，始于1973年

网址：<http://www.sigplan.org/Conferences/POPL/>

## 内核调试技术

### 内核探测工具systemtap

#### 什么是systemtap

假如现在有这么一个需求：需要获取正在运行的 Linux 系统的信息，如我想知道系统什么时候发生系统调用，发生的是什么系统调用等这些信息，有什么解决方案呢？

最原始的方法是，找到内核系统调用的代码，加上我们需要获得信息的代码、重新编译内核、安装、选择我们新编译的内核重启。这种做法对于内核开发人员简直是梦魇，因为一遍做下来至少得需要1个多小时，不仅破坏了原有内核代码，而且如果换了一个需求又得重新做一遍上面的工作。所以，这种调试内核的方法效率是极其底下的。

之后内核引入了一种Kprobe机制，可以用来动态地收集调试和性能信息的工具，是一种非破坏性的工具，用户可以用它跟踪运行中内核任何函数或执行的指令等。相比之前的做法已经有了质的提高了，但Kprobe并没有提供一种易用的框架，用户需要自己去写模块，然后安装，对用户的要求还是蛮高的。

systemtap 是利用Kprobe 提供的API来实现动态地监控和跟踪运行中的Linux内核的工具，相比Kprobe，systemtap更加简单，提供给用户简单的命令行接口，以及编写内核指令的脚本语言。对于开发人员，systemtap是一款难得的工具。

#### systemtap 工作原理

systemtap 的核心思想是定义一个事件（event），以及给出处理该事件的句柄（Handler）。当一个特定的事件发生时，内核运行该处理句柄，就像快速调用一个子函数一样，处理完之后恢复到内核原始状态。这里有两个概念：

事件（Event）：systemtap 定义了很多种事件，例如进入或退出某个内核函数、定时器时间到、整个systemtap会话启动或退出等等。

句柄（Handler）：就是一些脚本语句，描述了当事件发生时要完成的工作，通常是从事件的上下文提取数据，将它们存入内部变量中，或者打印出来。

# 结语

vxWorks、QNX之类的比RTOS大，比Linux小的操作系统们要当心了，可能是来抢你们的饭碗的。是不是Google要用到你们这个级别的操作系统，你们开价太高了，气得Google决定自己开发一个？

# 参考文献

1. Software Fault Tolerance <https://users.ece.cmu.edu/~koopman/des_s99/sw_fault_tolerance/>
2. 单比特错误 <https://www.zhihu.com/question/20686057/answer/15909317>
3. 又谈容错——DRAM的差错避免 http://blog.sciencenet.cn/blog-290937-941539.html
4. 实时嵌入式系统中高效定时器算法的实现

<http://www.zte.com.cn/cndata/magazine/zte_communications/2003/5/magazine/200311/t20031127_150200.html>

1. osdev

<http://wiki.osdev.org/Main_Page>

1. 布尔可满足性问题

<http://blog.sciencenet.cn/blog-290937-255116.html>

1. 失效模式与影响分析

<https://zh.wikipedia.org/wiki/%E5%A4%B1%E6%95%88%E6%A8%A1%E5%BC%8F%E4%B8%8E%E5%BD%B1%E5%93%8D%E5%88%86%E6%9E%90>

1. 内核探测工具systemtap简介

<http://blog.jobbole.com/107667/>