

广东中兴新支点

CGEL 用户手册

嵌入式操作平台

声 明

本资料著作权属广东中兴新支点技术有限公司所有。未经著作权人书面许可，任何单位或个人不得以任何方式摘录、复制或翻译。

侵权必究。



是广东中兴新支点技术有限公司的注册商标。广东中兴新支点技术有限公司产品的名称和标志是广东中兴新支点技术有限公司的专有标志或注册商标。在本手册中提及的其他产品或公司的名称可能是其各自所有者的商标或商名。在未经广东中兴新支点技术有限公司或第三方商标或商名所有者事先书面同意的情况下，本手册不以任何方式授予阅读者任何使用本手册上出现的任何标记的许可或权利。

由于产品和技术的不断更新、完善，本资料中的内容可能与实际产品不完全相符，敬请谅解。如需查询产品的更新情况，请联系广东中兴新支点技术有限公司。

若需了解最新的资料信息，请访问网站 <http://www.gd-linux.com/>。

手册说明

CGEL 全称为 Carrier Grade Embedded Linux，本手册主要针对应用 CGEL 3.0 与 CGEL4.0 内核版本的一些基本知识。通过本手册用户可以简单操作 CGEL 3.x\4.x，对于详细的开发指导请参考《广东中兴新支点 CGEL 产品开发指南》与《广东中兴新支点 CGEL API 参考手册》。

内容介绍

章名	概要
第 1 章 欢迎使用 CGEL	简单介绍 CGEL 3.x\4.x
第 2 章 安装 CGEL	介绍如何安装 CGEL 3.x\4.x
第 3 章 开发环境	详细介绍搭建 CGEL 命令行开发环境
第 4 章 CGEL 系统配置	详细介绍 CGEL 系统配置的方法和选项内容
第 5 章 CGEL 支持包开发	略
第 6 章 设备驱动开发	介绍设备驱动相关配置
第 7 章 应用开发	略
第 8 章 版本构建	详细介绍 CGEL 版本构建的相关内容
第 9 章 特色功能	介绍了 CGEL 所提供的特色功能及其使用方法
第 10 章 开源 LICENSE 说明	简单介绍 GPL


版本更新说明

文档版本	发布日期	备 注
CGEL_V4.02.20_P1	2013-3-20	2013 年第一次版本发布

本书约定

介绍符号的约定、键盘操作约定、鼠标操作约定以及四类标志。

1) 符号约定

- 样式  表示按钮名；带方括号 “【属性栏】” 表示人机界面、菜单条、数据表和字段名等；
- 多级菜单用 “->” 隔开，如 【文件】->【新建】->【工程】 表示【文件】菜单下的【新建】子菜单下的【工程】菜单项；
- 尖括号 <路径> 表示当前目录中 include 目录下的 .h 头文件，如 <asm-m68k/io.h> 表示 /include/asm-m68k/io.h 文件。

2) 标志

本书采用二个醒目标志来表示在操作过程中应该特别注意的地方：



提示：介绍提高效率的一些方法；



警告：提醒操作中的一些注意事项。

联系方式

电 话：020-87048510

电子信箱：cgel@gd-linux.com

公司地址：广州市天河区科技园高唐软件园基地高普路 1021 号 6 楼

邮 编：510663

目 录

第 1 章 欢迎使用 CGEL.....	1
1.1 CGEL 简介	1
1.2 交叉开发环境.....	3
1.3 基于 CGEL 的开发流程.....	4
1.4 相关支持.....	5
第 2 章 安装 CGEL	6
2.1 硬件需求.....	6
2.1.1 主机需求	6
2.1.2 目标板需求	6
2.2 安装 CGEL	7
2.3 CGEL 目录结构	7
第 3 章 开发环境.....	10
3.1 引言	10
3.2 Cygwin 开发环境.....	10
3.2.1 安装 Cygwin	11
3.2.2 安装交叉工具链	11
3.2.3 安装 ZDB 调试器	13
3.2.4 拷贝 CGEL 源代码.....	14
第 4 章 系统配置.....	15
4.1 引言	15
4.2 配置途径.....	15

4.3	配置项搜索	18
4.4	配置项介绍	18
第 5 章 最小系统开发		19
第 6 章 设备驱动开发		20
6.1	FLASH	20
6.1.1	NAND FLASH	20
第 7 章 应用开发		21
第 8 章 版本构建		22
8.1	KIDE 中构建	22
8.2	命令行下构建	23
8.2.1	基于 LSP 的编译	23
8.2.2	内核模块编译	25
8.3	分布式构建	25
第 9 章 特色功能		26
9.1	调度机制改进	26
9.1.1	排他性绑定	26
9.1.2	智能迁移功能	28
9.1.3	全局优先级调整	30
9.1.4	软中断线程化	32
9.1.5	负载均衡算法考虑软中断因素	35
9.1.6	内核信号量支持优先级继承	37
9.1.7	简单分区调度	38
9.1.8	保障性任务的静态优先级提升	39
9.1.9	工作队列静态优先级提升支持	41
9.1.10	高精度定时器中断差别处理支持	42
9.1.11	零开销 Linux 内核	43
9.2	调测增强	44

9.2.1	死机死锁检测	44
9.2.2	紧急 shell.....	45
9.2.3	系统黑匣子	49
9.2.4	CPU 占用率精确统计	58
9.2.5	异常输出控制台	59
9.2.6	数据断点调测增强	60
9.2.7	静态插点调测	63
9.2.8	动态插点调测	64
9.2.9	深度睡眠长期不调度检测	68
9.2.10	任务退出监控	70
9.2.11	异常转储 Kdump	72
9.2.12	任务长时间运行或不运行监控	78
9.2.13	内核态回溯用户态堆栈功能	80
9.2.14	查看软中断运行次数	83
9.2.15	printk 显示当前 CPU 号	84
9.3	IPC 增强.....	85
9.3.1	快速信号量	85
9.3.2	快速消息队列	90
9.4	文件系统增强.....	92
9.4.1	JFFS2 启动效率优化	92
9.4.2	JFFS2 主动垃圾回收机制	93
9.4.3	LZMA 压缩与解压功能	94
9.4.4	VFAT 专利规避功能.....	95
9.4.5	NFS 网络文件系统	97
9.4.6	EXT 文件系统	98
9.4.7	UBI fastmap 功能.....	108
9.5	内存管理增强.....	109
9.5.1	内存紧张时流程改造	109
9.5.2	内存即时映射	112
9.5.3	虚拟地址与物理地址映射关系查询	113
9.5.4	虚拟地址与物理内存分配分离	114
9.5.5	页缓存限制	116
9.5.6	代码段、数据段巨页映射	117
9.5.7	为用户态进程保留地址空间	118
9.5.8	即时分配巨页功能	119

9.5.9	homecache 功能	121
9.5.10	hugetlbfs 工具	125
9.6	信息统计.....	130
9.6.1	指定任务的调度信息统计	130
9.6.2	系统级调度与中断信息记录	134
9.6.3	内存信息统计	135
9.6.4	文件系统信息统计	138
9.6.5	调度轨迹信息统计	139
9.7	高精度定时器.....	143
9.8	用户态异常处理框架.....	144
9.9	多核差异化运行.....	154
9.9.1	基于 SMP 系统的多核差异化运行	154
9.9.2	基于 AMP 系统的多核差异化运行	158
9.10	安全机制.....	165
9.10.1	SMACK 机制	165
9.10.2	OCF 框架	165
9.10.3	防 DOS 攻击	167
9.10.4	内核堆栈溢出	168
9.11	启动增强.....	170
9.11.1	优雅重启支持	170
9.12	协议栈增强.....	184
9.12.1	原始套接字接收外发报文（LOP）	184
9.12.2	RPS/RFS	185
9.12.3	IP 地址冲突被动检测	190
9.12.4	IPV6 地址保存	191
9.12.5	arp 老化时间设置参数立即生效	192
9.12.6	多网卡绑定	193
9.12.7	网卡报文截获	197
9.12.8	按网口禁止 LINUX 协议栈报文收发	199

第 10 章 开源 LICENSE 说明 203

10.1	GPL 简介.....	203
10.2	开发指导.....	203
10.3	源码获取方式.....	204

图目录

图 1-1 CGEL 体系结构.....	3
图 1-2 交叉开发环境.....	4
图 3-1 Cygwin 目录结构	12
图 3-2 ZDB 调试器目录结构	14
图 4-1 Cygwin 下内核配置界面	16
图 4-2 选择“内核配置”	17
图 4-3 内核配置界面	17
图 4-4 menuconfig 搜索界面	18
图 8-1 KIDE 中构建工程	22
图 8-2 构建命令查看	23
图 9-1 系统黑匣子	50
图 9-2 本地 home 策略	122
图 9-3 远程 homing 方案.....	123
图 9-4 调度细节统计信息	132
图 9-5 CGEL 新增内存管理功能.....	136
图 9-6 设置 usrexc 库.....	145
图 9-7 设置 pthread 库	145
图 9-8 设置\$(PROC_ROOT)库路径.....	146
图 9-9 基于共享内存通讯机制	161
图 9-10 内核态堆栈	169
图 9-11 优雅重启运行流程	172

第 1 章

欢迎使用 CGEL

1.1 CGEL 简介

欢迎使用 CGEL！

目前，Linux 的主流内核已经全面转向了 Linux 2.6 内核，各 Linux 商用版本也逐步推出基于 2.6 内核的版本，硬件商发布的驱动程序也是基于 2.6 内核。同时，2.6 内核也添加许多新特性，如 O(1) 复杂度的调度器，对海量内存的支持，融入了 ucLinux 的补丁等等，这些新的特性对于最终提高系统性能也是很有帮助的。为了满足用户日益增长的用户需求，CGEL (Carrier Grade Embedded Linux) 操作系统平台在标准 Linux 内核的基础上进行优化改造，融入高性能、高可用、高可靠和高易用等特性，并支持 CGL 4.0 规范，可用于业务功能复杂、扩展性、稳定性、安全性要求高的电信级系统产品。

■ 特色功能

CGEL 3.X 的特色功能：

- 基于 linux 2.6.21.7 内核
- 支持多种业界主流处理器构架：X86/PPC(PowerPC)/MIPS/ARM(ARMNOMMU)等
- 在内核态进行实时性改造，大幅提高应用的性能，已达业界主流水平
- 支持 SMP 下任务的全局优先级调度功能
- 用户态提供 Glibc、uClibc、V2LIN 接口
- 在用户态、内核态设计了异常处理框架，应用程序可以挂接自定义的异常处理函数，实现系统级、进程级、线程级及代码段级的异常捕获及处理
- 可调、可维护性增强，提供内存管理增强、黑匣子、内存安全检查、基于调度检测内存越界等机制

- 提供统一 Shell 管理功能，可统一管理用户态标准 Shell、用户进程自定义 Shell、用户进程监控 Shell 以及内核态 Shell 等多种 Shell，实现灵活切换管理，也可在 Shell 中查看、控制系统运行状态，执行调试函数等
- 可剪裁的内核映像大小
- 提供支持多任务运行环境及 TCP/IP 协议栈的引导装载器 Kuboot
- 主机开发环境支持主机 Windows 和 Linux 操作系统


CGEL 4.X 的特色功能：

- 内核基础版本升级至 linux 2.6.32.
- 将原 V3.X 中具有的功能对齐至 V4.X 版本
- 增强可维护可测性，并整合分散的调试维护功能，形成系统的调测框架支持
- 强化对数据的综合展示分析能力，并增强对多核系统的支持
- 新增嵌入式虚拟化功能
- 提供全面的嵌入式节能降耗解决方案
- 进一步对嵌入式安全性能进行加固
- 调度机制改进与增强，保证系统实时、非实时任务的协调配合运行

■ 体系结构

CGEL 的体系结构和层次关系可见图 1-1，图中可以看出它同标准 Linux 的区别。CGEL 在标准 Linux 基础上通过提供多种增强手段建立内核，包括 IPC 增强，提供了进程间的快速消息能力，同时也支持内核态和用户态任务的消息能力；工程管理通过对 LSP 的工程化增强和工程脚本，提升 CGEL 的产品化能力；运行监控和错误管理，建立内核和应用的运行状态监控和异常处理机制，提升 CGEL 系统运行的稳定性和可测试性；内存管理增强，提供内核即时映射、内存紧张时的嵌入式处理、虚拟地址与物理内存分配分离、页缓存限制、可回收物理内存统计等功能；文件系统增强，提供 JFFS2 的详细节点信息查询、主动垃圾回收等功能；动态补丁代理，支持运行状态下的应用动态修改功能，提升了系统的可用性。CGEL 通过对标准 Linux 内核的多项功能的增强，提供了支持电信级核心应用所需的高性能、高可靠性的 OS 运行环境以及工程化的开发环境。

在接口层，CGEL 对于 VxWorks 和 Linux/Unix 应用程序的移植都作了良好的支持，用户只需做少量修改即可平滑移植。

 提示：由于 CGEL 使用用户态快速等待队列实现快速消息队列，所以其在进程间通讯需要使用进程间的共享内存。Glibc 支持进程间的共享内存分配，而 Uclibc 不支持共享内存（没有 shm_open 接口），所以在使用 Uclibc 工具链编译的快速消息队列的情况下，现只能支持线程间的通讯。

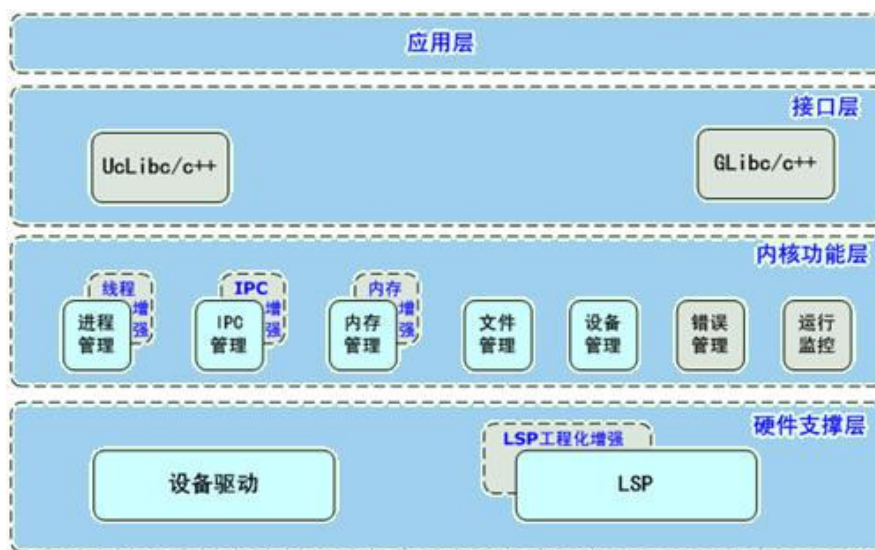


图 1-1 CGEL 体系结构

1.2 交叉开发环境

对一个程序进行编译的过程要通过在一个操作系统平台（编译平台）上运行编译器而完成。被编译的程序也将运行在一个操作系统平台（运行平台）上，这二个平台通常是相同的，如果二者不同，则这个编译过程被称为交叉编译。简单的说就是在一个平台上生成另一个平台上的可执行代码。

交叉开发环境由两部分组成，主机和目标板。

主机是用户通常工作时使用的 PC 机，可以安装开发工具，通常用他来编译、链接、远程调试软件以及其他相关的开发工作。目标板通常是用户准备生产的产品的参考板。一些目标板的存储设备很小甚至没有，没有完善的工具来编译内核和应用程序。

通常主机可以使用 Linux、Windows，目标板也可以为各个型号的 CPU，其交叉开发环境如图 1-2 所示。

▲ 警告：本文如果没有特殊说明，主机默认为 Windows 操作系统，以 Cygwin 作为命令行开发环境。

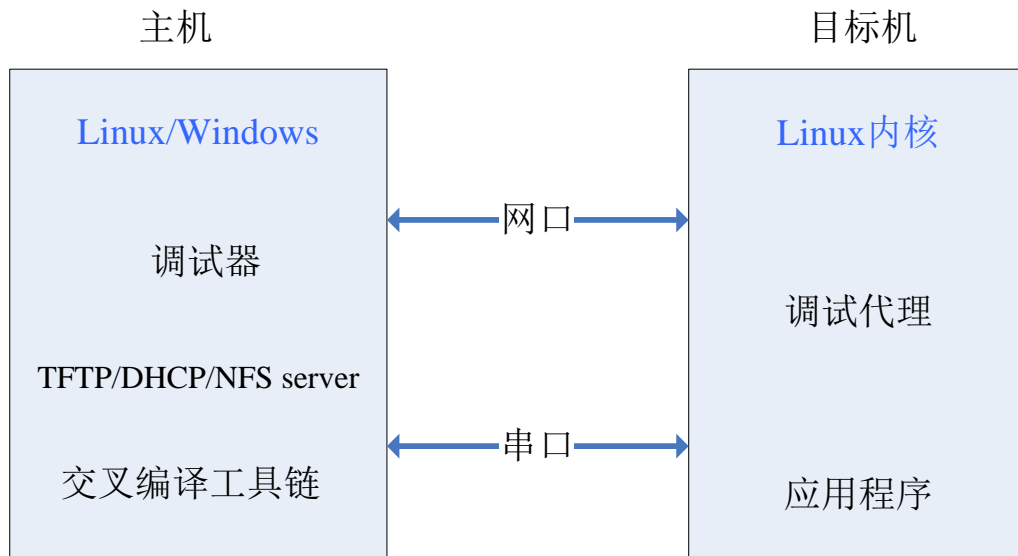


图 1-2 交叉开发环境

1.3 基于 CGEL 的开发流程

CGEL 提供了高度集成的集成开发环境 KIDE (CGEL Integrated Development Environment)，包括工程管理、内核及支撑系统的配置、裁剪、编辑工具、交叉编译工具链、Shell 等一系列适用工具，提供了一个跨平台的统一开发环境。

下面简要介绍使用 CGEL 开发的流程。

■ 项目规划

在开发前推荐对整个项目作简单的规划，这样能提高研发效率。这里推荐做如下的规划：

- 决定使用什么样的硬件。包括目标板、网卡、存储装置等。
- 选择项目需要的资源
- 最后列出硬件清单

■ 建立开发环境

CGEL 可以在集成开发环境 KIDE 和命令行开发环境 Cygwin 两种开发环境中使用，开发前先确定好使用哪种开发环境，然后建立相应的环境，可参见本文[第 3 章](#)。

■ LSP 及应用程序开发

建立好开发环境后，可以开始进行内核及应用程序的开发，下面的章节将详细介绍开发步骤。

■ LSP 及应用程序调试

内核及应用程序开发完毕，开始进行调试，以满足需要，下面的章节将详细介绍调试步骤。

1.4 相关支持

该手册介绍了如何安装 CGEL 3.0/4.0，以及怎样使用它来开发产品。

CGEL 平台提供完整的解决方案，主要包含以下内容：

■ CGEL 3.0/ 4.0 基本开发包

单独的内核源代码包；完整的开发环境软件包（Cygwin 安装包）；优化高效的 GNU 交叉开发工具链；支持多种调试模式的 ZDB 命令行调试器。

■ CGEL 技术支持

为产品线搭建 Linux 的开发环境，设计单板最小系统的 LSP，为产品线提供技术培训等。

■ LSP 模板库

提供丰富的 LSP 模板，范围覆盖了 MIPS、ARM、PPC、X86 架构的多类开发板。

■ CGEL 相关开发文档

包括《广东中兴新支点 CGEL 用户手册》、《广东中兴新支点 CGEL API 参考手册》、《广东中兴新支点 CGEL 产品开发指南》、《广东中兴新支点 CGEL LSP》等。

第 2 章

安装 CGEL

CGEL 的安装过程很简单，下面具体介绍安装过程。

2.1 硬件需求

2.1.1 主机需求

主机只要满足下面几点，就可以正常的安装和使用 CGEL 产品。

- 大于 10G 的单个硬盘空间，因为 KIDE 内包含了 CGEL 2.0、CGEL 3.0、CGEL 4.0、CGEL 2.2 内核，集成了一系列工具，故所需空间较大；
- 内存不小于 512M；
- CPU 主频 2.0G；
- 网卡、串口，最好有双网卡。

2.1.2 目标板需求

目前 CGEL 支持的目标板包括：

- **X86 体系**（支持所有 IA32 的 CPU、虚拟机）
- **X86_64 体系**（intel、amd）
- **PPC 体系**（E300、E500（包括 E500V1 和 E500V2）、E500mc、603e、MPC8XX）
- **ARM-NOMMU**（ARM7 TDMI、ARM7TDMI-S）
- **ARM**（ARM922T、ARM926EJ-S、ARM946E-S、ARM940T、Xscale、ARM11、CortexA9）

- MIPS (R4000、R4000E、R24000)
- MIPS64 (xls、xlr、xlp、octeon)

2.2 安装 CGEL

KIDE V3 中集成了 CGEL 3.0、4.0 内核，安装 KIDE 时勾选内核组件，就能完成 CGEL 3.0、4.0 内核的安装，其位于<\KIDE\target>目录下。CGEL 的详细安装步骤请见《广东中兴新支点 KIDE V3 用户手册》。

CGEL3.0、CGEL4.0 在命令行下安装步骤如下：

■ 下载源代码

CGEL 提供了单独的源代码包 CGEL.tar.bz2，用户可以到 FTP 下载：

- [ftp:// 10.75.8.168/ZX-CGEL/2012 年/CGEL3.X](ftp://10.75.8.168/ZX-CGEL/2012年/CGEL3.X),
- [ftp:// 10.75.8.168/ZX-CGEL/2012 年/CGEL4.X](ftp://10.75.8.168/ZX-CGEL/2012年/CGEL4.X)


其中 FTP 用户名和密码都为 zteuser。然后将这些源代码放到开发主机的某一目录下，如 <d:\CGEL3.0>下，注意要保证您的开发主机上有足够的空间。

■ 解包

在命令行开发环境中，进入源代码所在目录，然后输入如下命令：

```
tar -xjvf CGEL3.X.tar.bz2
```

解压到 d:\CGEL3.0。

 提示：命令行开发环境的搭建请参照本手册的 3.2 节。

2.3 CGEL 目录结构

CGEL 源码中各目录含义如下：

■ bsp

提供各类型开发板的最小系统模板，用户根据需要可以选择使用。

■ include

存放基于 VxWorks 函数接口和库开发所需的头文件。

■ build

该目录主要是工程管理相关目录，里面主要包含 product 目录。product 目录存放由用户维护的源代码，包括从内核提取的供用户修改的 lsp 代码（其中包括编译过程中动态生成的头文件），和用户维护产品的代码，如应用程序，平台 oss 代码等等。另外该目录也包含不同的单板进行编译控制的 makefile 文件，以及对不同单板动态生成的目录。

➤ application

该目录用来存放用户的应用程序。

➤ build

该目录为针对各个单板进行编译控制的 Makefile 和配置文件。

➤ lsp

该目录存放了 LSP 模板，由用户维护。

➤ object

该目录存放各个单板生成的临时文件、目标文件，库以及映像和模块。它会动态生成三个子目录：存放目标文件的 obj 目录，存放最终生成映像文件的 image 目录，以及存放最终生成的库的 libs 目录。

■ kernel-version

<kernel-version\>目录下存放了 CGEL 3.x\4.x 的内核源码，它又分为以下这些子目录。

➤ linux

- ✧ arch: 该目录包含了此核心源代码所支持的硬件体系结构相关的核心代码，如对于 X86 平台就是 i386。
- ✧ BLOCK: 此目录提供块设备驱动程序状态的具体数据。
- ✧ crypto: 此目录包含 Crypto API 的内容。
- ✧ documentation: 此目录存放一些文档，起参考作用。
- ✧ drivers: 系统中所有的设备驱动都位于此目录中。它又进一步划分成几类设备驱动，每一种也有对应的子目录，如声卡的驱动对应于 drivers/sound。

- ✧ fs: Linux 支持的文件系统代码。不同的文件系统有不同的子目录对应, 如 ext2 文件系统对应的就是 ext2 子目录。
- ✧ include: 这个目录包括了核心的大多数 include 文件。另外对于每种支持的体系结构分别有一个子目录。
- ✧ init: 此目录包含核心启动代码。
- ✧ ipc: 此目录包含了核心的进程间通讯代码。
- ✧ kernel: 主要核心代码。同时与处理器结构相关代码都放在 arch/*/kernel 目录下。
- ✧ lib: 此目录包含了核心的库代码。与处理器结构相关库代码被放在 arch/*/lib/目录下。
- ✧ mm: 此目录包含了所有的内存管理代码。与具体硬件体系结构相关的内存管理代码位于 arch/*/mm 目录下, 如对应于 X86 的就是 arch/i386/mm/fault.c。
- ✧ net: 核心的网络部分代码。里面的每个子目录对应于网络的一个方面。
- ✧ scripts: 此目录包含用于配置核心的脚本文件。
- ✧ security: 此目录包含 Linux 安全模块内容。
- ✧ sound: 此目录包含语音子系统的信息。
- ✧ usr: 此目录包含早期用户空间代码。

➤ vxadapt

该目录为 VxWorks 适配函数的具体实现的源代码。

■ tools

该目录存放目标机需要的部分软件, 包括符号表生成工具 anaelf、fakesym 和 Makefile 需要用到的工具 phrase。

第 3 章

开发环境

3.1 引言

CGEL 提供了两种 Windows 下的开发环境，其一是集成开发环境 KIDE，其二是命令行开发环境 Cygwin。集成开发环境 KIDE 的详细介绍和操作[请参照《广东中兴新支点 KIDE V3 用户手册》](#)。

下面具体介绍命令行开发环境 Cygwin 的搭建过程。

3.2 Cygwin 开发环境

Cygwin 是一个在 Windows 平台上运行的 Unix 模拟环境，是 cygnus solutions 公司开发的自由软件，它可以帮助程序开发人员把应用程序从 UNIX/Linux 移植到 Windows 平台，是一个功能强大的工具集。

下面介绍 Cygwin 环境的搭建过程。



提示：如果用户安装了 KIDE，就已经包含了 Cygwin，交叉编译工具链，zdb 调试器和 CGEL 源

代码，进行相关设置后就可进行命令行开发。目录<\KIDE\host\ide\Cygwin>下为 3.2.1 节的

命令行开发环境 Cygwin；目录<\KIDE\host\ide\tools_chain>下为 3.2.2 节的交叉编译工具链

crosstools；目录<\KIDE\host\ide\debugger>为 3.2.3 节的调试器 debugger；目录

<\KIDE\target>下为 3.2.4 节的 CGEL 源代码。用户可以直接使用 KIDE 进行命令行开发，具体方法请参见《广东中兴新支点 CGEL 产品开发指南》。

3.2.1 安装 Cygwin


下面简单介绍 Cygwin 的安装过程：

第 1 步：获取 Cygwin 安装包，到 FTP（[ftp://10.75.8.168/ZX-TSP/2012 年/ToolsChain/VX.XX/windows/cygwin](ftp://10.75.8.168/ZX-TSP/2012年/ToolsChain/VX.XX/windows/cygwin)）下载，其中 FTP 的用户名和密码都为 zteuser。

第 2 步：将安装包 Cygwin.7z 拷贝至开发主机下，并解压到本机某目录下。

第 3 步：运行解压后的<xxx\Cygwin\BuildRuntimeVar.bat>文件（仅在第一次安装时运行），添加注册表相关内容。

到此为止，Cygwin 已安装成功，可进行后面的开发工作。

 提示：安装问题可参考<\crosstools\VX.XX\windows\Cygwin>下的 readme 文件。手工将

xxx\Cygwin\bin 路径添加到系统环境变量 PATH 首位方法为【我的电脑】->【右键】->【属性】->【环境变量】->【系统变量】，找到 Path，双击添加保存即可。

3.2.2 安装交叉工具链

下面按顺序依次介绍安装过程。

第 1 步：获取 crosstools。

在 FTP（[ftp://10.75.8.168/ZX-TSP/2012 年/ToolsChain](ftp://10.75.8.168/ZX-TSP/2012年/ToolsChain)）下载交叉编译工具链，FTP 下载的用户名和密码都为 zteuser，根据自身需要选择主机操作系统对应的工具链，将其拷贝至 Cygwin 安装目录下。此例中主机为 Windows 操作系统，目标板为 X86 类型，所以下载目录为<ftp://10.75.8.168/ZX-TSP/2012 年/ToolsChain/V2.08.20_P2/windows/x86>，选择 x86_gcc4.1.2_glibc2.5.0_V2.08.20_P2.tar.bz2（根据自身需求选择），将其下载到目录<e:\Cygwin>。

第 2 步：解压缩。

通过下面的命令进行解压后，目录结构如图 3-1 所示。

```
tar -xjvf x86_gcc4.1.2_glibc2.5.0_V2.08.20_P2.tar.bz2
```

第 3 步：指定交叉工具链目录。

指定交叉工具链目录，又称设置编译器路径，与 CPU 相关，有两种方法。以此例的 X86 为例来说明。

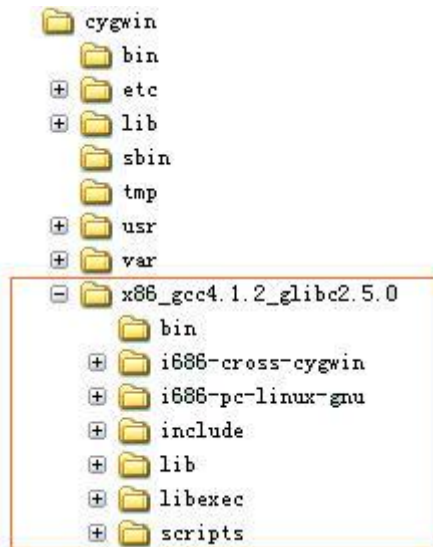


图 3-1 Cygwin 目录结构

第一种方法：修改 Cygwin 安装目录下的 Cygwin.bat 文件，添加或者修改如下粗体代码。

例：

```
@echo off

E:

umount -A
mount E:\Cygwin /
mount E:\Cygwin\bin /usr/bin
mount E:\Cygwin\lib /usr/lib

set PATH=/x86_gcc4.1.2_glibc2.5.0/bin //添加这行代码

set HOME= E:\Cygwin
```

```
bash --login -i
```

如果想添加当前所有支持的交叉编译工具目录，可以用分号作为分隔符将所有工具链列出。例如：

```
set PATH=/x86_gcc4.1.2_glibc2.5.0/bin:/x86_64_gcc4.1.2_glibc2.5.0/bin
```

第二种方法：运行 Cygwin，进入到工程目录 prj，在命令行下输入命令导入。

例：

```
export PATH=$PATH:x86_gcc4.1.2_glibc2.5.0/bin
```

如果想添加当前所有支持的交叉编译工具目录，可以采用如下命令：

```
export PATH=$PATH:/x86_gcc4.1.2_glibc2.5.0/bin:/x86_64_gcc4.1.2_glibc2.5.0/bin
```

▲ 提示：第二种方法在重启 Cygwin 后需要重新设置工具链路径导入。

3.2.3 安装 ZDB 调试器

接下来就是安装 ZDB 调试器了。关于 ZDB 调试器的安装，用户只需到成研所对外 FTP（[ftp://10.75.8.168/ZX-TSP/2012 年/ZDB/](ftp://10.75.8.168/ZX-TSP/2012%20%E5%B9%B4/ZDB/)）下载相应版本的调试器（方法直接拷贝到本机）即可，如目录<e:\ZDB_V2.08.20_P2>下。该目录下包含了目标端调试代理模块和主机端调试控制模块，以及一些辅助工具，如图 3-2 所示。



图 3-2 ZDB 调试器目录结构

目标端调试代理模块运行于目标端，主要功能是接收调试控制模块发送的调试信息，并主动向调试控制模块反馈调试事件；目标端调试代理模块以异步模型为基础进行开发，同时监控调试控制模块的调试命令和被跟踪任务的状态事件。系统级调试的调试代理模块为嵌入到内核异常处理函数的调试代码，不作为单独任务存在。调试代理模块同时支持 X86、PPC、MIPS 和 ARM 四种 CPU 体系架构。

主机端调试控制模块主要功能为接收调试命令进行对目标端的调试控制，并对调试代理模块返回的调试事件进行解析，反馈给用户。调试控制模块在 GDB6.5 的基础上进行开发、升级和融合，延续了 GDB6.6 的符号分析和事件循环机制，增加了调试代理上报的异步消息，以及完善 GDB6.6 的异步处理流程。调试控制模块同时支持 X86、PPC、MIPS 和 ARM 四种 CPU 体系架构。

工具包 paxctl-0.5 的相关说明可以查看根目录下的 paxReadMe 文件。

 **提示：**在 ZDB 调试器目录中，同时提供了 Windows 和 Linux 主机开发环境下所需使用的调试器，

用户只需到相应目录获取即可。

3.2.4 拷贝 CGEL 源代码

用户可以在成研对外 FTP (<ftp://10.75.8.168/ZX-CGEL/2012年>) 下载获取源代码包 CGEL.tar.bz2，按照**本手册 2.3 节**安装完成后 CGEL 目录已经位于 Windows 主机的某一目录下，如 <e:\CGEL3.0>下。注意，最好不要将 CGEL 目录放到 Cygwin 的安装目录下。

至此，CGEL 3.0 的 Cygwin 开发环境搭建完毕，运行 Cygwin，可以在 Cygwin 环境下进行后续所有工作，下面的章节将详细介绍。

第 4 章

系统配置

4.1 引言

开发环境成功搭建完成后，CGEL 源代码已经存在于开发主机，接下来用户就可以进行内核配置了。CGEL3.0\4.0 系统配置方法完全相同，本章主要以 CGEL3.0 为例详细介绍配置方法。

CGEL 内核的配置有两种途径，第一种，从 Cygwin 命令行界面进入配置界面；第二种，从 KIDE 开发界面进入内核配置界面。两种途径的配置方法是相同的，将在 4.2 节分别介绍两种方法的配置途径，4.3 节介绍具体的配置项。

4.2 配置途径

■ Cygwin 命令行进入配置界面

第 1 步：将目标板对应的 LSP 模板从目录<\target\bsp\cgel3.0>拷贝到<\target\build\cgel3.0\product\lsp>下，例如虚拟机的 LSP 模板 ZTE-VERMACHINE-i386。

第 2 步：打开 Cygwin 开发界面。

第 3 步：进入目录<\target\build\cgel3.0\product\build>。

第 4 步：输入以下命令进入内核配置界面，如图 4-1 所示。

```
make ARCH=i386
CROSS_COMPILE=/x86_gcc4.1.2_glibc2.5.0/bin/i686-pc-linux-gnu-
LSPSRC=../lsp/ZTE-VERMACHINE-i386 menuconfig
```

其中，参数 **ARCH**、**CROSS_COMPILE**、**LSPSRC** 三个参数用户可以根据目标系统的实际体系架构进行填写。**ARCH** 是选用目标系统的 CPU 类型；**CROSS_COMPILE** 是目标系统需要选用的交叉工具链，此例位于<Cygwin\x86_gcc4.1.2_glibc2.5.0>目录下；**LSPSRC** 是待编译目标系统 LSP 最终的存放位置，位于<CGEL\product\build\lsp>目录下。

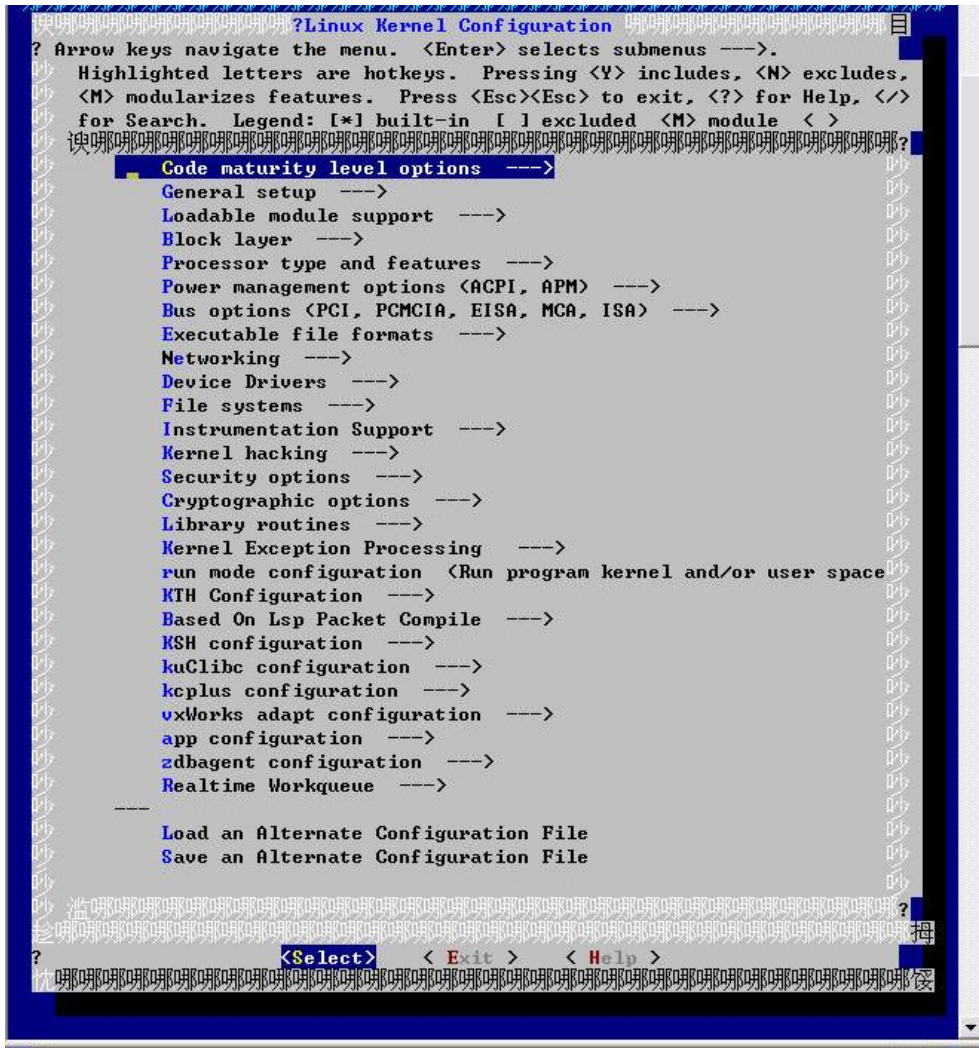


图 4-1 Cygwin 下内核配置界面

提示：如果“make menuconfig”命令失败，很可能是 ncurses 库没有安装。

■ KIDE 进入配置界面

第1步：打开 KIDE 开发界面。

第2步：建立 Kernel 工程，或者 LSP 工程。（具体方法请参照《广东中兴新支点 KIDE V3 用户手册》）

第3步：点【右键】，选择【内核配置】，自动进入配置界面。如图 4-2、图 4-3。

在 KIDE 下内核配置的详细信息请参照《广东中兴新支点 KIDE V3 用户手册》。

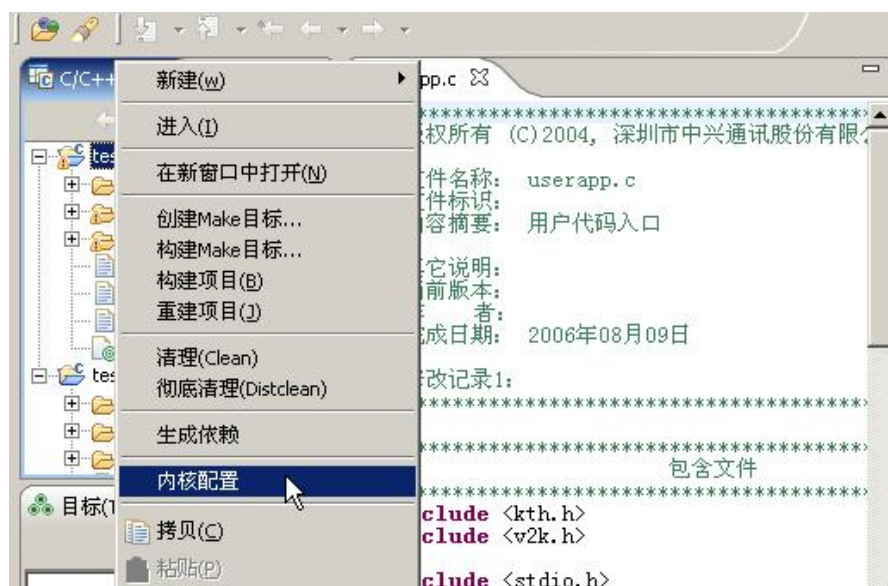


图 4-2 选择“内核配置”

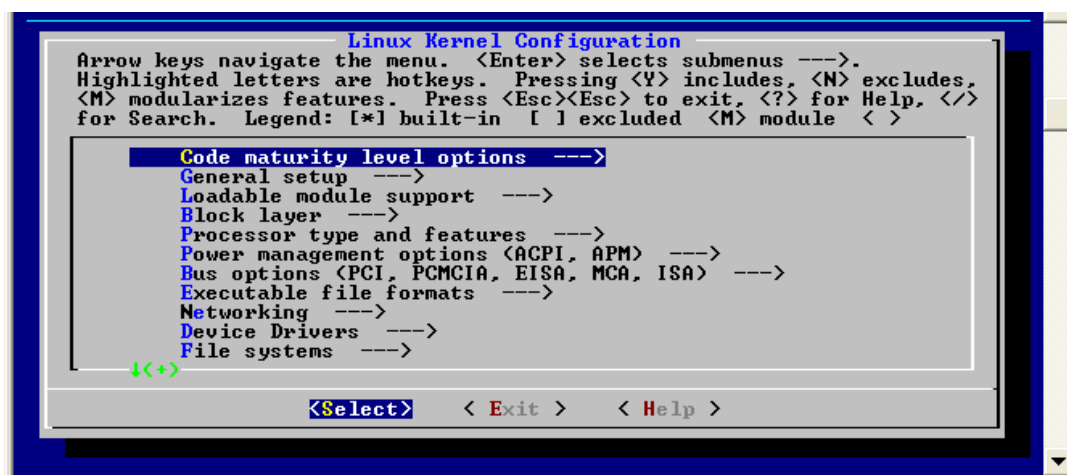


图 4-3 内核配置界面

4.3 配置项搜索

由于 CGEL 内核中内核配置众多，配置选项位置在不同体系架构中也不同，此时可以通过搜索的方式快速定位选项进行配置。

搜索方法：进入 menuconfig 界面后，用键盘敲一下 “/” 键就可以进入搜索界面，如下所示。

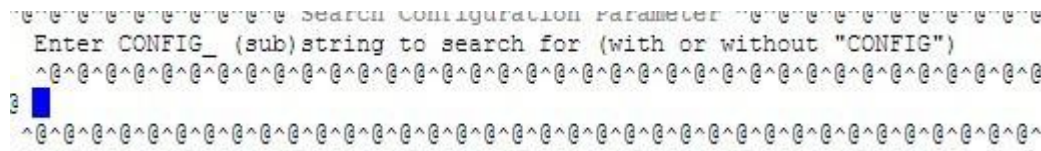



图 4-4 menuconfig 搜索界面

进去搜索界面后即可通过关键字搜索所需的配置项。

4.4 配置项介绍

4.2 节介绍了进入 CGEL 内核配置界面的两种途径，对于标准 Linux2.6 的内核配置项介绍请用户按 “[h]” 查看这个选项的帮助信息，这里就不再作介绍。

使用方向键可以在各选项间移动；使用 “[Enter]” 键进入下一层选单；每个选项上的高亮字母是键盘快捷方式，使用它可以快速地到达想要设置的选单项。在括号中选择 “y” 将这个项目编译进内核中，选择 “m” 编译为模块，选择 “n” 为不选择（按空格键也可在编译进内核、编译为模块和不编译三者间进行切换），按 “[h]” 将显示这个选项的帮助信息，按 “[Esc]” 键将返回到上层选单。

 提示：在 make menuconfig 下，* 表示 Y，M 表示 M，空白表示 N。

第 5 章

最小系统开发

请参考《广东中兴新支点 CGEL LSP 开发指南》。

第 6 章

设备驱动开发

6.1 FLASH

6.1.1 NAND FLASH

6.1.1.1 BBT 配置

NAND 驱动在 probe 的时候会扫描整个 nand flash，生成一张 BBT（坏块表），表中描述了 flash 中所有坏块位置信息。使用此功能，需修改设备驱动源文件，按需增加如下配置：

➤ nand_chip->options:

✧ 配置 NAND_USE_FLASH_BBT，表示 BBT 存在于 flash 中，如：

```
nand_chip->options = NAND_USE_FLASH_BBT;
```

➤ nand_bbt_descr->options:

✧ 配置 NAND_BBT_ABSPAGE，表示 BBT 存在于指定的位置，具体位置可在 lsp 的 nand_bbt_descr 配置中设置；

✧ 配置 NAND_BBT_LASTBLOCK，表示 BBT 放在 flash 尾部；

✧ 配置 NAND_BBT_WRITE，表示可将 BBT 写入 flash 中；

✧ 配置 NAND_BBT_CREATE，表示如果 flash 中不存在 BBT 则可在 flash 中创建 BBT；

✧ 配置 NAND_BBT_SCAN2NDPAGE，表示扫描坏块时只扫描该块的前两页，建议配置；

✧ 配置 NAND_BBT_SCANALLPAGES，表示扫描坏块时扫描该块所有页。

第 7 章

应用开发

请参见《广东中兴新支点 CGEL 产品开发指南》。

第 8 章

版本构建

8.1 KIDE 中构建

由 KIDE 创建的工程，在 KIDE 工程视图的右键菜单中可以找到构建项目：

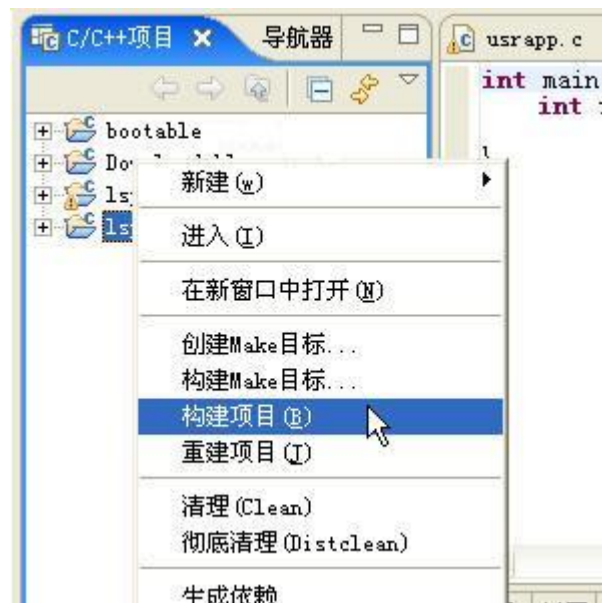


图 8-1 KIDE 中构建工程

之后在控制台视图可以查看所有构建命令：

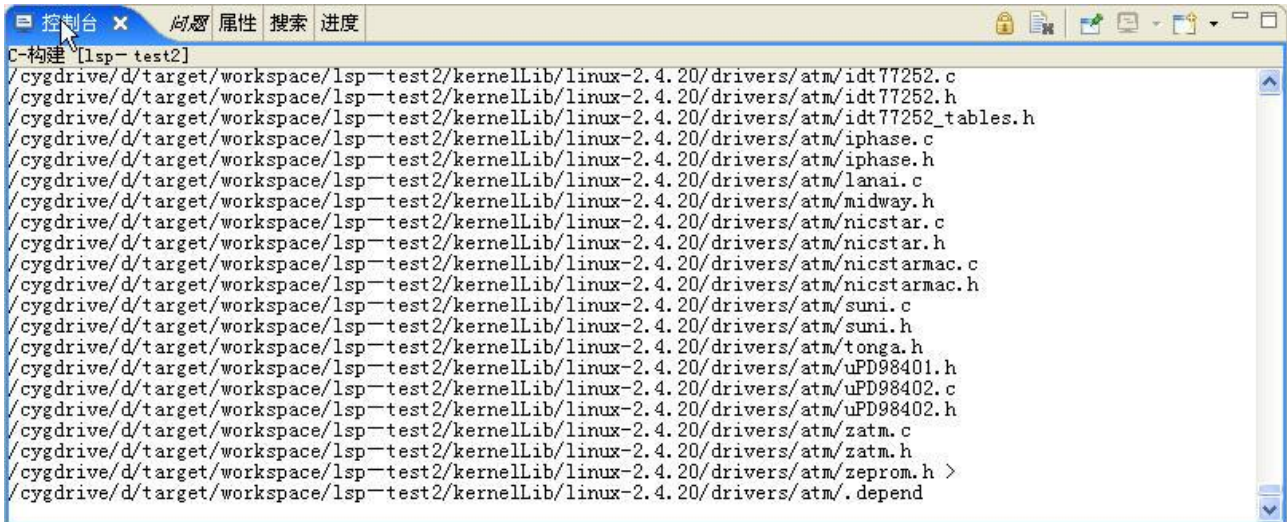


图 8-2 构建命令查看

构建完成后，在<bspConfig\object\images>目录下自动生成内核映像文件，包括 **bzImage**、**zImage**、**uImage** 等，以及符号表文件 **vmLinux**，<bspConfig\object\obj>目录生成编译目标文件。

8.2 命令行下构建

由命令行创建的工程，构建直接在<\target\build\cgel3.0\product\build>目录（CGEL4.0 同此目录）进行，编译完成后内核映像生成在<\target\build\cgel3.0\product\object\xxx\images>目录下，其中 xxx 为你的目标板的 LSP 名称。下面是以 x86 为例的构建命令：


8.2.1 基于 LSP 的编译

■ 获取帮助

```
make ARCH=i386
CROSS_COMPILE=/x86_gcc4.1.2_glibc2.5.0/bin/i686-pc-linux-gnu-
LSPSRC=../../../../../bsp/cgel3.0/i386/i386/ZTE-VERMACHINE-i386 help
```

■ 配置内核：

```
make ARCH=i386
CROSS_COMPILE=/x86_gcc4.1.2_glibc2.5.0/bin/i686-pc-linux-gnu-
LSPSRC=../../../../../bsp/cgel3.0/i386/i386/ZTE-VERMACHINE-i386 menuconfig
```

 提示：内核配置会影响 CGEL 3.0 代码的编译流程，每次配置前都需要 distclean 后再重新编译代码。

■ 编译整个工程

```
make ARCH=i386
CROSS_COMPILE=/x86_gcc4.1.2_glibc2.5.0/bin/i686-pc-linux-gnu-
LSPSRC=../../../../../bsp/cgel3.0/i386/i386/ZTE-VERMACHINE-i386 all
```

■ 只编译内核

```
make ARCH=i386
CROSS_COMPILE=/x86_gcc4.1.2_glibc2.5.0/bin/i686-pc-linux-gnu-
LSPSRC=../../../../../bsp/cgel3.0/i386/i386/ZTE-VERMACHINE-i386 linux_image
```

■ 只编译 LSP

```
make ARCH=i386
CROSS_COMPILE=/x86_gcc4.1.2_glibc2.5.0/bin/i686-pc-linux-gnu-
LSPSRC=../../../../../bsp/cgel3.0/i386/i386/ZTE-VERMACHINE-i386 lsp
```

■ 只编译模块文件

```
make ARCH=i386
CROSS_COMPILE=/x86_gcc4.1.2_glibc2.5.0/bin/i686-pc-linux-gnu-
LSPSRC=../../../../../bsp/cgel3.0/i386/i386/ZTE-VERMACHINE-i386
linux_modules
```

■ 只编译 APP

```
make ARCH=i386
CROSS_COMPILE=/x86_gcc4.1.2_glibc2.5.0/bin/i686-pc-linux-gnu-
LSPSRC=../../../../../bsp/cgel3.0/i386/i386/ZTE-VERMACHINE-i386 app
```

■ 清理整个工程：

```
make ARCH=i386
```

```
CROSS_COMPILE=/x86_gcc4.1.2_glibc2.5.0/bin/i686-pc-linux-gnu-  
LSPSRC=../../../../../bsp/cgel3.0/i386/i386/ZTE-VERMACHINE-i386 clean
```

■ 只清理 APP

```
make ARCH=i386  
CROSS_COMPILE=/x86_gcc4.1.2_glibc2.5.0/bin/i686-pc-linux-gnu-  
LSPSRC=../../../../../bsp/cgel3.0/i386/i386/ZTE-VERMACHINE-i386 clean_app
```

8.2.2 内核模块编译

当用户添加的内核代码不依赖 LSP 工程时，可以通过以下命令编译为模块。命令为：

```
make ARCH=i386  
CROSS_COMPILE=/x86_gcc4.1.2_glibc2.5.0/bin/i686-pc-linux-gnu-  
TYPE_PROJECT=6 -C /cygdrive/d/target/kernel-version/cgel3.0/linux/  
DIR_LSP_SRC=/cygdrive/d/target/bsp/cgel3.0/powerpc/85xx/ZTE-VERMACHINE-  
i386/  
O=/cygdrive/d/target/build/cgel3.0/product/object/ZTE-WPBCB-mpc85xx/obj/linux/  
M=/cygdrive/c/target/workspace/WPBCB_MODULE/ modules
```

命令中主要参数的含义为：

TYPE_PROJECT 表示工程类型，**TYPE_PROJECT=6** 表示基于 CGEL 命令行编译内核的标准模块工程；

-C 指示内核源代码的路径

DIR_LSP_SRC 表示 lsp 包（也称 bsp 包）的源代码路径；

O 表示模块目标文件的输出路径（与内核目标文件的输出路径一致）；

M 表示被编译的内核模块源代码路径。

8.3 分布式构建

待续。

第 9 章

特色功能

9.1 调度机制改进

9.1.1 排他性绑定

9.1.1.1 功能描述

提供一个接口，用于为指定的 CPU 或 CPU 集合设置排他性，从而在设置了排他性的 CPU 上，只有绑定到该 CPU 上的线程才能够在该 CPU 上运行，未被绑定的线程将不会被迁移到该 CPU 上运行。这种行为区别于传统意义的“绑定”概念，实质上是我们常说的“绑定”的反义词，因此，我们又称之为“反向绑定”。

9.1.1.2 应用场景

在 SMP 系统中，可能出现以下问题：

- 在媒体面核 CPU 占有率比较高的情况下，仍然会出现高优先级线程（如看门狗）会被均衡到媒体面核的情况，造成媒体面核性能的波动；
- 在媒体面线程的优先级设置为最高的情况时，如果将看门狗线程迁移到媒体面核，由于媒体面线程不会主动放弃处理器，将导致看门狗线程饿死，会导致单板复位

采用排他性绑定功能，将指定媒体面 CPU 核为排它性 CPU，将媒体面线程绑定到媒体面 CPU 核（控制面线程无须绑定），就能很好的解决上述问题，同时保证了代码的通用性及可移植性。

注意事项：

- 1) 在配置了排他性绑定和智能迁移的前提下，用户若需要对 cpu 进行热插拔操作，则必须先关闭排他性绑定和智能迁移功能，然后才可进行 cpu 热插拔操作。用户若因为未关闭排他性绑定和智能迁移功能而导致热插拔失败的同时，会打印提示信息，返回错误码；
- 2) 热插拔完成后由用户负责重新设置和启用 cpu 排他性绑定和智能迁移功能，系统将不会自动进行排他或者智能迁移；
- 3) 当下线 cpu 被设置为排他性 cpu 集合或者非排他 cpu 集合时，系统将对其进行过滤，即保证排他性 cpu 集合或者非排他 cpu 集合需为在线 cpu 的子集。

9.1.1.3 内核配置

配置宏选项 CONFIG_EXCLUSIVE_BIND。

在 menuconfig 中配置如下

```
CPU Exclusive Binding --->
[*]Exclusive binding threads to cpu
```

9.1.1.4 使用方法

■ 设置排他性 CPU 集合

用法：

```
echo X > /proc/exclusive_cpus
```

输入待设置的排它性 CPU 集合，X 代表输入值。例如：

设置排它性集合为 cpu2、cpu3，那么输入

```
echo c > /proc/exclusive_cpus
```


c 是 CPU 掩码，换算成二进制为 0000 1100，对应 cpu2、cpu3 核。


设置排它性集合为 cpu2、cpu3、cpu5，那么输入

```
echo 2c > /proc/exclusive_cpus
```

2c 是掩码，换算成二进制为 00101100，对应为 cpu2、cpu3、cpu5 核。

X 以 16 进制表示，无需在前面加 0x 前缀。

 提示：用户对线程的绑定优先于排它性 cpu 集合设置的反向绑定。

 提示：用户对线程的绑定会清除反向绑定标志 reverse binding flag。

要取消排它性 cpu 集合，执行

```
echo 0 > /proc/exclusive_cpus
```

■ 查询排他性 CPU 集合

用法：

```
cat /proc/exclusive_cpus
```


输出当前排他性 CPU 集合。

9.1.2 智能迁移功能

9.1.2.1 功能描述

智能迁移功能又将排他 CPU 集合和非排他集合进一步细分，提出独占式 CPU 集合和共享式 CPU 集合。其中独占式 CPU 集合中内核任务会迁移到共享式 CPU 集合中。独占式 CPU 集合是排他 CPU 集合的子集；共享式 CPU 集合是非排他 CPU 集合的子集。

独占式 CPU 集合是指只运行一个或多个高负载进程的 CPU 的集合。该集合内的 CPU 占用率通常为 100%，被一个或多个指定的进程独占，这些进程不与内核发生交互。该集合的进程期望在一块“干净”的环境中执行，不希望被内核线程或者中断干扰。独占式 CPU 集合适用于运行重负载进程，包括长时间 CPU 占用率高和 CPU 占用率间歇性冲高类型。内核线程不会在独占式集合运行。


 提示：由于排他性 CPU 除能运行绑定其上的线程外，还能够运行未绑定的内核线程，而独占式 CPU

只能运行绑定其上的线程，因而独占式集合属于排他性集合的子集，两者可以重叠，但独占式集合

不能超过排他性集合的范围。

共享式 CPU 集合是指没有运行高负载进程，CPU 占用率适中，用户态进程或者内核线程都可以在其上执行的 CPU 的集合。共享式 CPU 集合适用于轻负载进程运行。不仅可自由迁移的内核线程会在该集合中运行，而且从独占式集合迁移过来的内核线程也会在该集合中运行。

将“独占式”CPU 中的内核线程迁移到“共享式”CPU，从而避免“独占式”CPU 上的内核线程得不到调度后出现的问题，以及“独占式”CPU 上的内核线程对业务性能的影响。

 提示：共享式集合属于非排他性集合的子集，两者可以重叠，但共享式集合不能超过非排他性集合的范围。

9.1.2.2 应用场景

目前业务产品线在电信产品规划时，对 CPU 核数较多的单板通常采用控制面和媒体面的区分，媒体面绑定运行高负载进程，不与内核进行交互，CPU 占用率通常高达 100%。但由于 Linux 内核本身的特点，每个 CPU 核都运行了一些内核线程，每个内核线程完成特定的功能，当用户态任务独享 CPU 时，这些内核线程长时间得不到调度。在这种场景下，使用内核线程迁移接口，将“独占式”CPU 中的内核线程迁移到“共享式”CPU，从而避免“独占式”CPU 上的内核线程得不到调度后出现的问题，以及“独占式”CPU 上的内核线程对业务性能的影响。

9.1.2.3 内核配置

配置宏选项 CONFIG_MIGRATE_KTHREAD。在 menuconfig 中配置如下：

```
CPU Exclusive Binding --->
[*]Exclusive binding threads to cpu
[*]Migrate kernel thread to shared CPUs
```

该配置默认关闭，根据用户需要可手动配置开启该功能。

9.1.2.4 使用方法

通过 proc 接口将原本绑定在独占式集合上的内核线程、工作队列，从独占式集合中迁移到共享式集合上：

```
echo X Y Z > /proc/exclusive_cpus
```

输入待设置的排它性 CPU 集合、独占 CPU 集合、共享 CPU 集合。X Y Z 分别代表输入的 CPU 掩码，

以 16 进制表示，中间使用空格隔开，无需在前面加 0x 前缀。独占式 CPU 集合不能超过排他性 CPU 集合的大小；共享式 CPU 集合不能超过非排他性 CPU 集合的大小。

如只输入一个参数则默认为排他性 CPU 集合，而独占 CPU 集合默认等于排他 CPU 集合，共享 CPU 集合默认等于非排他 CPU 集合。

```
echo X > /proc/exclusive_cpus
```

输入两个参数或超过三个参数被认为是非法的输入。

以下是非法的输入格式：

```
echo X Y > /proc/exclusive_cpus      //只指定两种 CPU 集合非法输入。
echo X Y Z A > /proc/exclusive_cpus  //指定超过三种 CPU 集合输入非法输入。
```

9.1.2.5 接口说明

显示迁移操作的执行结果：

```
cat /proc/migrate_info
```

通过 /proc/migrate_info 获取信息，查看上一次设置独占式 CPU 集合和共享式 CPU 集合的迁移结果。

```
cat /proc/exclusive_cpus
```

配置智能迁移功能后，输出当前排他性 CPU 集合、独占 CPU 集合、共享 CPU 集合。

9.1.3 全局优先级调整

9.1.3.1 功能描述

在多核（如 SMP）系统中，标准 Linux 内核对线程的调度并非完全按照优先级进行，每个 CPU 拥有自己单独的运行队列，CPU 在进行调度时只从自己的运行队列中选取任务，因此各核间可能出现中、低优先级任务先于高优先级运行的情况，影响系统实时性能，且可能不符合业务逻辑的预期。

全局优先级调度技术保证多个实时任务在多 CPU 间完全按照优先级进行调度，使得 Linux 在调度机制上更符合实时系统的要求。CGEL 主要从任务唤醒、调度切换和优先级继承这三个方面切入，采取推拉操作的方式在 CPU 之间完成实时优先级任务的平衡。

- 在唤醒任务时，如果被唤醒任务是实时任务，则判断该任务是否为高优先级实时任务，如果它比系统中正在运行的任务优先级高，则将当前运行任务中最低优先级的任务抢占，即主动将任务推到其他 CPU 中。

- 在任务调度前，不是从当前 CPU 运行队列中选择最高优先级任务，而是从其他 CPU 运行队列中，选择就绪的最高优先级任务，并与当前 CPU 运行队列上的最高优先级任务进行比较，执行拉操作；在任务调度后，在当前 CPU 运行队列中选择一个适合推的实时任务，为该任务查找一个适合迁移的 CPU，将其加入该 CPU 的运行队列中，主动将任务推到该 CPU 中。
- 在发生优先级继承时，如果实时任务的优先级变低，则执行一次拉操作；如果从非实时任务提升到实时任务，当前 CPU 运行队列已有实时任务运行，则执行一次推操作；如果从实时任务切换到非实时任务，并且当前 CPU 运行队列已无实时任务运行，则执行一次拉操作。

9.1.3.2 应用场景

在 SMP 系统中，标准 Linux 内核对线程的调度并非完全按照优先级进行，每个 CPU 拥有自己单独的运行队列，CPU 在进行调度时只从自己的运行队列中选取任务，因此这就会造成低优先级任务提前获得调度，而高优先级任务迟迟得不到运行的问题，严重影响了内核的实时性。其主要表现在：

- 当某个高优先级实时线程准备就绪时，可能会抢占当前 CPU 上的运行任务。如果当前运行任务是一个实时任务的话，即使其他核上运行的是非实时任务，也无法迅速飘移到其他核并抢占非实时任务；
- 就绪的实时任务可能没有当前运行任务的优先级高，但是就绪任务无法迅速飘移到其他核上抢占其他核上的非实时任务。


标准 Linux 内核的调度算法可能使得实时任务得不到及时调度，这在嵌入式系统中是不合适的，因而这种场景下需要使用全局优先级调度。

9.1.3.3 内核配置

CGEL3.X 的内核配置：

- 配置宏选项 CONFIG_REAL_PREEMPT

在 menuconfig 中配置如下：



```
Processor type and features --->
[*]RealTime Scheduler sport
```

CGEL4.X 的内核配置：

- 配置宏选项 CONFIG_SMP
 - ✧ BFS 调度器内核配置：CONFIG_BFS
 - ✧ CFS 调度器内核配置：去掉 CONFIG_BFS 配置

在 menuconfig 中配置如下：

```
Processor type and features--->
    [*]Symmetric multi-processing support
#BFS 调度器
General setup --->
    [*]BFS cpu scheduler
File system --->
    [*]Miscellaneous filesystems --->
        [*]BFS file system support <EXPERIMENTAL>
```

CGEL4.X 内核使用内核配置选项 CONFIG_BFS 来选择使用哪种全局优先级调度器，若使用 BFS 调度器则需配置 CONFIG_BFS，若使用标准 Linux 调度器，则不配置此选项。

9.1.3.4 使用方法

CGEL3.x 完成内核配置后，需要打开/proc/rt_global_schedule 接口启动全局优先级调度；CGEL4.X 完成内核配置后，内核自动开启全局优先级调整功能。

9.1.3.5 接口说明

■ CGEL3.x 全局优先级调度功能动态开关

接口：

```
/proc/rt-global-schedule
```

用法：

```
echo 1 > /proc/rt-global-schedule    //打开全局优先级调度功能；
echo 0 > /proc/rt-global-schedule    //关闭全局优先级调度功能。
```

9.1.4 软中断线程化

9.1.4.1 功能描述

在标准 Linux 中，软中断具有比实时任务高的优先级。不论在任何时刻，只要产生软中断事件，内核将立即执行相应的软中断处理程序，等到所有挂起的软中断处理完毕后才能执行正常的任务，因此有可能造成实时任务得不到及时的处理。软中断线程化之后，软中断将作为内核线程运行，而且被赋予不同的实时优先级，实时任务可以有比软中断线程更高的优先级。这样，具有最高优先级的实时任务就能

得到优先处理，即使在严重负载下仍有实时性保证。

软中断线程化具备如下两大特色功能：

➤ 低延时差异化高精度时钟

由于标准内核的高精度定时器处理基本都是关中断进行，因此，为提供实时响应性能，CGEL 对内核的高精度定时器处理进行了修改，使得可灵活设置定时器回调是在中断中回调，还是在软中断中执行，并且能够对软中断中回调的定时器链表进行维护

➤ 软中断线程化管理

在实际的应用场景中，并非所有的软中断都需要在对应的软中断实时任务中处理，例如 timer，该软中断处理所花的时间可能远小于任务切换所花的时间。因此对于某些软中断，应用希望该软中断直接在中断上下文中处理，而不是在软中断实时任务中处理。

CGEL 内核通过为用户提供一个内核启动参数和相关的用户配置选项，指定哪些软中断可以线程化，若用户没有指定该参数时，缺省所有的软中断都线程化，保证原来的 CONFIG_PREEMPT_SOFTIRQS 语义兼容。

 提示：无论是否配置 CONFIG_PREEMPT_SOFTIRQS，cge14.x 内核都会为各软中断创建相应的软中断实

时任务；配置了 CONFIG_PREEMPT_SOFTIRQS，软中断则会在软中断实时任务中处理；没有配置

CONFIG_PREEMPT_SOFTIRQS，软中断则会在中断上下文处理，当中断负载很高时，软中断也可能在软中断线程中处理。

9.1.4.2 应用场景

某些情况下，如网络负荷较重时，软中断的执行会导致系统高优先级实时任务无法及时被调用运行。为了保证系统实时响应能力，避免软中断的执行影响高优先级任务，可使用软中断线程化功能，将系统的软中断处理完全放到任务上下文去运行，以便产品统筹规划各任务的优先级设定。

9.1.4.3 内核配置

CGEL3.X 内核配置：

➤ CONFIG_PREEMPT_SOFTIRQS（可手动配置，无依赖）

在 menuconfig 中配置如下：

```
Processor type and features --->
[*]Thread Softirqs
```

CGEL4.X 内核配置：

- CONFIG_PREEMPT_SOFTIRQS (依赖于 CONFIG_PREEMPT_RT)
 - ✧ CONFIG_THREAD_HRTIMER: 低延时高精度时钟配置，该配置实现高精度时钟线程化和时钟差异化处理
 - ✧ CONFIG_SOFTIRQ_THREADING_MASK 可以分别对网络收发包、普通定时器、RCU 等实现软中断线程化分别进行设置
 - ✧ CONFIG_SOFTIRQ_THREAD_STAT 对软中断线程化的执行情况进行统计

在 menuconfig 中配置如下：

```
Processor type and features --->
  Preemption Model--->
    [X]Complete Preemption<Real-Time>
  Complete Preemption<Real-Time> --->
    [*]Complete Preemption –Thread Softirqs
```

9.1.4.4 使用方法

配置了软中断线程化功能后可以通过 cat /proc/softirqs 查看在实时任务中软中断执行次数；

```
# cat /proc/softirqs

          CPU0      CPU1      CPU2      CPU3      CPU4      CPU5      CPU6
CPU7
HI:        0         0         0         0         0         0         0
TIMER:    4591      4540      4517      4492      4469      4444      4421      4362
NET_TX:        0         0         0         0         0         0         0
0
NET_RX:        0         0         0         0         0         0         0
0
BLOCK:     0         0         0         0         0         0         0
BLOCK_IOPOLL: 0      0         0         0         0         0         0
TASKLET:  0         0         0         0         0         0         0
```

SCHED:	0	0	0	0	0	0	0	0
HRTIMER:	0	0	0	0	0	0	0	0
RCU:	0	0	0	0	0	0	0	0
mask: 0x3fd	CPU0	CPU1	CPU2	CPU3	CPU4	CPU5	CPU6	CPU7
HI:	0	0	0	0	0	0	0	0
TIMER:	0	1	0	0	0	1	0	0
NET_TX:	0	0	0	0	14	0	0	0
NET_RX:	3	4	0	0	1	0	4	0
BLOCK:	0	0	0	0	0	0	0	0
BLOCK_IOPOLL:	0	0	0	0	0	0	0	0
TASKLET:	0	424	0	426	0	426	0	448
SCHED:	4244	2029	4216	4324	4263	4201	4325	4191
HRTIMER:	0	0	0	0	0	0	0	0
RCU:	4571	4541	4943	4493	4469	4445	4421	4361

这里 mask 为 0x3fd 表示线程化除 TIMTER_SOFTIRQ 以外的所有软中断，上半部显示中断上下文中软中断执行的次数，下半部显示进程上下文中软中断执行的次数。

9.1.5 负载均衡算法考虑软中断因素

9.1.5.1 功能描述

标准 Linux 的负载均衡算法，只考虑将任务作为负载，实际应用中与负载均衡算法相关的因素有：任务（处于运行状态）优先级，任务个数，调度域参数，任务的 CPU 掩码。

CGEL 将软中断等因素纳入负载均衡算法的考虑范围。

9.1.5.2 应用场景

主要针对产品遇到的多起负载均衡问题。例如，灌包时网络收发软中断比较高，而表现出来软中断较高的 CPU 上的利用率偏高，造成软中断所在的 CPU 负载偏重，为此产品线要求在软中断（sirq）频繁的情况下，操作系统也能保持各 CPU 间的负载均衡。

9.1.5.3 内核配置

配置宏选项 CONFIG_SIRQ_BALANCE, CONFIG_SYSCTL（此模块 CGEL3.X 默认配置）

宏 CONFIG_SIRQBALANCE_SIRQNUM_BIT 会自动选中，默认值为 6。

在 menuconfig 中配置如下：

```
SIRQ Balancing --->
[*]Considering softirq to kernel load balance
```

9.1.5.4 使用方法

正确配置内核后，即可开启此功能。

同时，提供了 proc 接口用于控制软中断负载均衡。/proc/sys/kernel/中导出 4 个用户控制参数：

- sirqlow: 判断软中断占用率较低的门限，默认为 15，允许范围[5,100]；
- sirghigh: 判断软中断占用率较高的门限，默认为 75，允许范围[5,100]，设置时请保证 sirghigh > sirqlow；
- sirbalance: 算法开启与否的开关，非 0 表示开启，0 表示关闭，默认为开启；
- sirmask: 软中断掩码，默认为 12，即 (1<<NET_TX_SOFTIRQ)|(1<<NET_RX_SOFTIRQ)，可以设置除 TIMER_SOFTIRQ 以外的所有内核允许的软中断。

附 CGEL3.0 内核的软中断掩码位参考：

```
enum
{
    HI_SOFTIRQ=0,
    TIMER_SOFTIRQ,
    NET_TX_SOFTIRQ,
    NET_RX_SOFTIRQ,
    BLOCK_SOFTIRQ,
    TASKLET_SOFTIRQ,
    SCHED_SOFTIRQ,
#ifdef CONFIG_HIGH_RES_TIMERS
    HRTIMER_SOFTIRQ,
#endif
    RCU_SOFTIRQ,
    MAX_SOFTIRQ,
};
```

9.1.5.5 注意事项

- 本功能同软中断线程化，RPS/RFS 补丁，NO_HZ 是互斥的；
- 不能处理“软中断 CPU 的软中断占用率接近于或大于非软中断 CPU 利用率”的情形；如：非软中断的 CPU 利用率为 20%，软中断 CPU 的软中断就有 50%，是不可能做到利用率平衡的；
- 当软中断比较高时，本算法会自动关闭负载均衡，倾向于将此 CPU 专门做软中断；
- 对于普通定时器软中断，本算法不予处理；实际系统应用中，普通定时器软中断的负载通常也是可以忽略的。

9.1.6 内核信号量支持优先级继承

9.1.6.1 功能描述

实时操作系统引入了内核互斥信号量的优先级继承功能，即当低优先级的任务获取到信号量后，高优先级任务获取信号量时会自动升高信号量拥有者的优先级，防止其被其他高优先级的任务抢占运行。此内核信号量优先级继承功能仅适用于互斥信号量。

9.1.6.2 应用场景

在嵌入式实时系统中，任务是依靠优先级来进行调度的，优先级高的保证优先执行，以满足实时性要求。但在具体应用中，很可能会有不同优先级的任务去访问同一资源，这样就需要引入信号量来保证资源访问的互斥。如果使用一般的信号量，则可能会引起优先级反转现象，造成高优先级的实时任务得不到及时运行。

9.1.6.3 内核配置

CGEL3.X 内核配置：

- CONFIG_SEMAPHORE_PI（手动配置）
- CONFIG_RW_SEMAPHORE_PI（读写信号量优先级继承，依赖于 SEMAPHORE_PI）

在 menuconfig 中配置如下：

```
Semaphore support priority inherit --->
[*]Semaphore support priority inherit
[*]RW semaphore support priority inherit
```

CGEL4.X 内核配置：

- CONFIG_SEM_TO_MUTEX (Selected by: CONFIG_PREEMPT_RT_MUTEX [=y] && CONFIG_PREEMPT_RT [=y])
- CONFIG_RWSEM_PI (读写信号量优先级继承, 依赖于 CONFIG_PREEMPT_RT_MUTEX, 目前仅对 mmap_sem 实现读写信号量优先级继承)

在 menuconfig 中配置如下:

```
Processor type and features --->
  Preemption Model--->
    [X]Complete Preemption<Real-Time>
  Complete Preemption<Real-Time> --->
    [*]mutex PI
    [*]rwsem PI
```

9.1.6.4 使用方法

完成内核配置后, 内核自动启动信号量优先级继承机制。

9.1.7 简单分区调度

9.1.7.1 功能描述

简单分区调度是从现有系统中固定分出一片时间份额, 用于所有非实时任务的运行 (非实时任务的选择由 CFS 公平调度器选择), 以此实现实时任务与非实时任务的统一调度, 从而保证在高负荷下, 普通的非实时系统任务能够得到运行的机会, 不会被死循环的实时任务阻死。

该方法具有如下优点:

- 对系统其他任务运行的影响可控, 因为其运行时间被严格限制;
- 保障范围广, 无需用户做过多设置;
- 任务的保障可靠, 同时又可以保障看门狗等必须确保任务的执行;
- 操作开销较小, 修改代码少, 充分利用了公平调度器内部的公平调度特性。

9.1.7.2 应用场景

很多后台任务都是非实时策略, 在通信系统的模式下, 为达到程序的可预测性, 会让应用任务以实时优先级运行, 此时会引起对一些非实时任务的影响。简单分区调度算法可以确保非实时任务能够执行, 并且所有的非实时任务的执行对实时任务和整个系统的影响可控。

9.1.7.3 内核配置

配置宏选项 CONFIG_PART_SCHED。

在 menuconfig 中配置如下：

```
Part Schedule --->
[*]Enable part schedule<NEW>
```

9.1.7.4 使用方法

通过 proc 接口查看或者设置静态优先级。

查看某个核上，非实时任务集合**每秒**可运行的最大微秒数，即运行比例：

```
cat /proc/sys/kernel/part_sched/cpuN/op_interval
```

设置某个核上，非实时任务集合**每秒**可运行的最大微秒数：

```
echo us >/proc/sys/kernel/part_sched/cpuN/op_interval
```

查看某个核上，非实时任务集合**每次**运行的最大微秒数：

```
cat /proc/sys/kernel/part_sched/cpuN/run_interval
```

设置某个核上，非实时任务集合**每次**运行的最大微秒数：

```
echo us >/proc/sys/kernel/part_sched/cpuN/run_interval
```

查看某个核上，不受分区调度影响的实时任务优先级：

```
cat /proc/sys/kernel/part_sched/cpuN/guard_prio
```

设置某个核上，不受分区调度影响的实时任务优先级：

```
echo prio >/proc/sys/kernel/part_sched/cpuN/guard_prio
```

9.1.8 保障性任务的静态优先级提升

9.1.8.1 功能描述

为保障关键任务的可靠运行，需要支持静态优先级提升相关功能，主要包括：

- 提供保障任务的静态优先级提升；
- 提供针对工作队列的静态优先级提升机制；

- 提供针对 Keventd 的工作进行梳理；
- 提供针对静态优先级提升方法的配置支持。

9.1.8.2 应用场景

当系统运行过程中，若出现如下异常情况时，仍希望保证用户监控运行能力（包括 shell, telnet 等任务）。此时，可通过采取静态提升远端监控关键任务的优先级，以确保远端监控能力的切实可靠。异常情况如下：

- 具有实时优先级的用户程序死循环；
- 系统的运行环境比较恶劣，必须使用远程环境来登录。

9.1.8.3 内核配置

配置宏选项 CONFIG_RT_EVENTD, CONFIG_HIGH_PRIO。

在 menuconfig 中配置如下：

```
General setup --->
    [*]Modify scheduling policy for a recently OOM killed thread
Setting Realtime Eventd --->
    [*]Enable realtime eventd
```

9.1.8.4 使用方法

使用如下命令可提升保障性任务的优先级

```
chrt -f -p prio pid
```

参数：

- -f: 设置进程为实时进程
- prio 表示静态优先级级数
- pid: 进程 ID 号

备注：chrt 命令为标准 Linux 命令，busybox 默认提供支持。

9.1.9 工作队列静态优先级提升支持

9.1.9.1 功能描述

工作队列应用广泛，且工作队列优先级各有高低，通过提升工作队列的静态优先级，将原有工作队列机制划分为高、低两种，高优先级以实时优先级执行，低优先级保持原来的非实时优先级，从而保证高优先级的工作能够及时得到执行。

9.1.9.2 应用场景

主要针对 keventd，workqueue 等服务于工作队列机制的后台任务。各后台任务被绑定到各个核上，用于负责这些工作队列上的工作。

9.1.9.3 内核配置

配置宏选项 CONFIG_RT_EVENTD，CONFIG_HIGH_PRIO。

在 menuconfig 中配置如下：

```
General setup --->
    [*]Modify scheduling policy for a recently OOM killed thread
Setting Realtime Eventd --->
    [*]Enable realtime eventd
```

9.1.9.4 使用方法

通过 proc 接口查看或者设置静态优先级。

查看实时工作队列的优先级：

```
cat /proc/sys/kernel/rt_events_high_priority
```

设置实时工作队列的优先级：

```
echo prio >/proc/sys/kernel/rt_events_high_priority
```

其中 prio 为我们设置的静态优先级。

9.1.10 高精度定时器中断差别处理支持

9.1.10.1 功能描述

为提升系统实时性能，可采用了软中断线程化机制，在该机制的原始实现中，高精度定时器软中断必须在硬中断退出时才能进行处理，这使得定时器的处理优先级很高。但在 Linux 内核中，定时器中挂接了大量的回调处理（如内核协议栈的定时路由处理），这将占用大量 CPU 运行时间，从而导致高优先级任务得不到及时调度运行。因此，我们需要对高精度定时器的使用进行清理，将不需要很高实时性的使用分离出来，放到线程中处理。这样需要高实时性的高精度定时器回调会在硬中断退出时，及时得到处理，而其它的高精度定时器回调，会在工作线程中得到调度运行。通过这种处理机制，可以平衡定时器和其它系统实时任务的关系，使系统得以协调运行。

9.1.10.2 应用场景

定时器中挂接大量的回调函数，占用大量的 CPU 运行时间，导致高优先级任务得不到及时调度。此时，就需要高精度定时器对中断进行差别对待。

9.1.10.3 内核配置

内核配置说明：CONFIG_TIMERINTER_DIFPROCESS。

依赖配置：CONFIG_HIGH_RES_TIMERS，CONFIG_PREEMPT_SOFTIRQS。

在 menuconfig 中配置如下：

```
Processor type and features --->
  [*]High Resolution Timer Support
  [*]Thread Softirqs
Diff processes of high-res timer interrupt --->
  [*]Diff processes of high-res timer interrupt
```

9.1.10.4 使用方法

将回调模式设置为 IMER_CB_INSOFTIRQ，则此高精度定时回调在软中断上下文，中断线程化后仍然在软中断上下文。

9.1.11 零开销 Linux 内核

9.1.11.1 功能描述

ZOL 指的 0 开销 Linux 内核。ZOL 功能下的 DATAPLANE 数据面 cpu 上，运行一个单一的用户空间任务，可以不发生 linux 系统开销。没有调度 tick 以及系统调用等干扰。ZOL 的数据面 cpu 用来运行抖动敏感低延时的数据面代码，专职专做，运行负载很大。

标准 Linux 运行 Linux 可抢占调度器，周期定时中断会将抢占任务，以及系统调用等也会进入内核态打断正常任务执行。同时 ZOL 的数据面 cpu 也具备 isolate cpu 的特性，即调度域的相对孤立提高 cpu 独立性能，所以除非设置任务亲和在数据面 cpu，否则任务是不会运行到数据面 cpu 上去的。另外数据面 cpu 抑制了一些内核任务，从而避免用户面被干扰。当数据面任务进入内核，内核转入非数据面模式。将调度内核 scheduler tick。在从内核态返回用户态之前会进行可能的调度，因此会重入内核。

9.1.11.2 应用场景

通常 ZOL 的数据面 cpu 上只运行着单一任务且占用较多 cpu 资源且对实时性要求较高的场合。比如网络应用中，一个 Linux 中断或者调度、系统调用等开销的打断，会消耗成百上千的处理包的周期，从而导致丢包。ZOL 主要意图是保证数据面 cpu 上应用的实时性。

9.1.11.3 使用约束

- 必须指定至少 1 个 cpu 用作非数据面 cpu，保证标准 kernel 空间可以运行；
- 系统一旦 boot，就不能在改变核的数据面属性或反之；
- 用户一旦主动调用系统调用等进入了内核，系统就会当做普通的 linux cpu 那样运行。当返回用户态，又回到 ZOL 模式；
- 数据面 tile 会增长 linux 系统工作的延时，同时严重影响 tile 核上的调度器迁移任务的性能。因此如果应用不是要求很高的实时性，或者需要多线程在一个核上运行，就不要使用数据面 tile。对于那些需要系统调用的应用，或者不特别怕中断影响的应用，最好就用进程亲和性来替代数据面 cpu。

9.1.11.4 内核配置

内核配置说明：CONFIG_DATAPLANE，缺省已配置。

在 menuconfig 中配置如下：

```
Tilera-specific configuration --->
```

[*] Support for Zero-Overhead Linux mode

9.1.11.5 使用方法

通过在 boot 参数中指定 dataplane= 来指定数据面 cpu，实例如下：

```
tile-monitor --dev usb0 --hvx dataplane=1-35
```

9.1.11.6 接口说明

Tile 为应用提供显式绑定任务到数据面 tile 接口用法如下：

获取当前任务所绑定的数据面 cpu 的接口：

```
int tmc_cpus_get_my_cpu (void)
```

设置当前任务要绑定的数据面 cpu 的接口：

```
int tmc_cpus_set_my_cpu (int cpu)
```

9.2 调测增强

9.2.1 死机死锁检测

9.2.1.1 功能描述

CGEL 提供的死机死锁检测与监控功能，是基于 NMI 和 sysrq 的系统宕机信息获取工具。在开启死机死锁检测工具后，当有一个或者多个 CPU 关中断死循环时，系统会通过 NMI 机制触发故障现场的记录与保存，并根据用户需要可进入紧急 shell 进行故障调试与定位；系统运行时，也可通过特殊的 sysrq 命令，进入紧急 shell，进行系统状态的监控。紧急 shell 可以使用相关命令查看系统情况，以便对故障进行定位分析。



提示：目前 CGEL3.X 支持 MIPS64_XLR732、MIPS64_XLS 416 和 Powerpc 以及 x86_64 体系；CGEL4.X

在 CGEL3.X 的基础上，还提供了 MIPS64_XLP 架构支持。

9.2.1.2 触发方式

死机死锁有三种触发方式：NMI 硬件看门狗、NMI 软件看门狗以及 sysrq 三种方式。

开启死机死锁检测需配置 CONFIG_EM_LOCK_MONITOR（默认配置）

- 开启 NMI 硬件看门狗需配置宏选项 CONFIG_EM_HARD_WD
- 开启 NMI 软件看门狗需配置宏选项 CONFIG_EM_SOFT_WD
- 开启 sysrq 需配置宏选项 CONFIG_MAGIC_SYSRQ

➡ 提示：使用硬件、软件看门狗触发死机死锁工具时需要系统支持 NMI 中断。使用 sysrq 开启死机死锁工具时需要系统支持 sysrq。

➡ 提示：硬件触发方式和软件触发方式不可同时工作，只能使用一种。

➡ 提示：在硬件看门狗激活状态下，一旦发生死锁，所有 CPU 都会挂死；在软件看门狗激活状态下，一旦发生死锁，仅发生死锁的 CPU 会进入监控模式。

➡ 提示：配置 sysrq 功能的内核成功启动后，可按 ctrl + Pause Break 键后点击 J 键开启 sysrq 死机死锁检测工具。

9.2.2 紧急 shell

当系统出现关中断死循环、内核死锁、调度不正常等异常情况时，系统进入死机死锁检测状态，此时需要通过紧急 shell，登陆该 shell 与系统交互，以便获取内存、CPU 寄存器等信息。

9.2.2.1 触发方式

配置如下宏选项：

- CONFIG_URGENT_SHELL, CONFIG_RELIABLE_SERIAL (CPU 状态获取调试命令)
- CONFIG_URGENT_SHELL (内核 mutex 信号量的调试命令, mutex 拥有任务功能依赖 CONFIG_SMP)
- CONFIG_URGENT_SHELL, CONFIG_SLAB (内存分配的调试命令)
- CONFIG_URGENT_SHELL, CONFIG_VIRADDR_TO_PHYADDR (指定地址内存查看的调试命令)
- CONFIG_URGENT_SHELL (除以上以外的调试命令)

9.2.2.2 应用场景

当系统出现关中断死循环、内核死锁、调度不正常等异常情况时。

9.2.2.3 使用方法


使用如下命令查看系统情况，以便对故障进行定位分析。

■ CGEL3.X 支持命令

- 支持可恢复运行模式

```
echo j > /proc/sysrq-trigger exit
```

紧急 shell 在调用后，还可以恢复到原来的系统运行。

 提示：在串口都不能正常输入的情况下，通过按 sysrq 键或 NMI 触发方式进入紧急 shell 之后，是无法退出并恢复到原来系统运行的。

- 支持不可恢复运行模式

```
echo j > /proc/sysrq-trigger
```

- 显示当前系统中的高速缓存信息

```
showslab
```


用户可以通过此命令查看系统高速缓存的信息。

- 显示系统内存使用情况

meminfo

用户可以通过此命令查看系统内存使用情况。

- 显示所有 cpu 运行队列的信息

showrunall

用户可以通过此命令查看所有 cpu 上的运行队列信息

- 显示指定 cpu 上运行队列信息

showrun cpuid

输入参数：

cpuid 用户需要查看的 cpuid 号

用户可以通过此命令查看指定 cpu 上的运行队列信息

- 显示当前所有进程

showtskall

用户可以通过此命令查看所有进程的运行情况。

- 显示指定 pid 的进程信息

showtsk pid

输入参数：

pid 用户需要查看的进程 pid 号


用户可以通过此命令查看指定的进程运行情况。

- 切换 nmi 的 shell 到指定 cpu

chcpu cpuid

输入参数：

cpuid 用户切换到的 cpuid 号


 **提示：** sysrq 激活状态下不支持该命令。

- nmi 下显示指定 cpu 的寄存器现场

```
showregs cpuid
```

输入参数：

cpuid 用户需要查看寄存器现场的 cpuid 号

 提示：sysrq 激活状态下不支持该命令。

■ CGEL4.X 支持命令

- 支持可恢复运行模式

```
echo x > /proc/sysrq-trigger exit
```

紧急 SHELL 在调用后，还可以恢复到原来的系统运行。

- 支持不可恢复运行模式

```
echo x > /proc/sysrq-trigger
```


- 获取 cpu 状态

```
echo x > /proc/sysrq-trigger showregs
```

除了 URGENT_SHELL，该功能需要依赖 RELIABLE_SERIAL 可靠串口输出，故目前只有部分单板（具体来说，目前实现了 defined (CONFIG_NLM_XLP) || defined (CONFIG_BOOKE) || defined (CONFIG_X86) || defined (CONFIG_PPC_83xx) 4 种类型单板）。

- 提供用户态的 mq 消息队列信息

```
echo x > /proc/sysrq-trigger showmq
```

 提示：在中断方式触发情况下不支持该命令。

- 提供用户态的 FUTEX 信号量信息

```
echo x > /proc/sysrq-trigger showfutex
```

- 提供内核 mutex 信号量信息

```
echo x > /proc/sysrq-trigger showmutex
```

- 提供内核 semaphore 信号量信息

```
echo x > /proc/sysrq-trigger shosem
```

- 提供内存分配信息

```
echo x > /proc/sysrq-trigger showslab showmeminfo
```

- 提供任务状态获取

```
echo x > /proc/sysrq-trigger showbstk
```

- 显示指定任务堆栈回溯

```
echo x > /proc/sysrq-trigger $showpidstack
```

- 显示 CPU 运行队列

```
echo x > /proc/sysrq-trigger showrunning
```

- 各 CPU 运行队列当前任务堆栈获取


```
echo x > /proc/sysrq-trigger showcurstack
```

- 查看各 CPU 当前任务运行状态

```
echo x > /proc/sysrq-trigger showcurstate
```

- 指定地址内存查看

```
echo x > /proc/sysrq-trigger $showvmvalue
```

 提示：在中断方式触发情况下不支持该命令。

9.2.3 系统黑匣子

9.2.3.1 功能描述

系统黑匣子作为内核监控机制的一部分，所完成的主要功能是当 CGEL 检测到系统故障并进入到故障处理流程的时候保存故障信息，之后再在合适的时机获取所保存的信息，便于定位故障。同时黑匣子也可以用来主动保存各种信息，为其他模块调试做支撑。

黑匣子将所控制的区域分为滚动区域和非滚动区域，在滚动区域中当信息写满的时候继续写入会自动覆盖过去的信息，而在非滚动区域写满之后则不能继续写入。

9.2.3.2 应用场景

系统黑匣子可以用于紧急情况的处理，例如发生死机，调度不正常，信号量死锁，内存耗尽等。当系统出现这些异常情况的时候就会将相关信息保存到系统黑匣子之中，为之后故障的分析和定位提供信息。

9.2.3.3 内核配置

配置宏选项 CONFIG_BLACK_BOX，该配置默认关闭，根据用户需要可手动配置开启该功能。

在 menuconfig 中配置如下：

```
Kernel hacking --->
  Emergency for System Deadlock --->
    [*] Black box support
```

9.2.3.4 使用方法

系统黑匣子由一块保留出来的内存或者 NVRAM 区域，该区域一般是由 boot 或者 BSP 划分出来给 LINUX 的，专门用于保存异常信息，并且在系统重启后，数据不能丢失或被改写。此区域除了系统黑匣子以外，无论内核还是用户态程序都不可以使用。

在初始化好黑匣子之后，当系统出现异常状态时会将相关信息记录到相应区域中。之后用户就可以根据自己的模块是用户态程序，还是内核模块来调用相应的内匣子接口获取信息。其示意图如下：

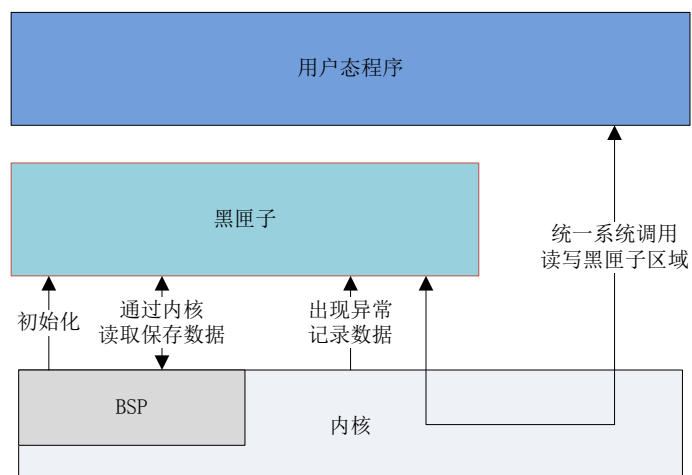


图 9-1 系统黑匣子

■ 黑匣子模块的初始化

黑匣子需要 Boot 及 BSP 来提供存储空间，同时，Boot 和 BSP 应注意不能覆盖提供作为黑匣子的存储空间，BSP 中一般是在内存初始化的时候把事先规划好的黑匣子内存空出来，而不分配给内存分配器管理。CGEL 负责提供相关接口用来初始化黑匣子模块相关的全局数据。

BSP 提供黑匣子非滚动区的起始物理地址和大小和滚动区的大小，其中滚动区大小可以为 0（不存在滚动区），并且非滚动区和滚动区必须物理上连续。

在获取非滚动区的起始物理地址和非滚动区大小和滚动区的大小之后，便可调用 `kbbox_init` 函数。

BSP 中初始化函数的 demo 代码如下：

```
#include <linux/bbox.h>
static int init_black_box (void)
{
    bbox_base_phyaddr = BOX_PHY_BASE; /* 初始化物理基地址 */
    bbox_const_size = CONST_SIZE; /* 初始化非滚动区大小 */
    bbox_loop_size = LOOP_SIZE; /* 初始化滚动区大小 */

    return kbbox_init();
}
```

■ 读写黑匣子

黑匣子的读写分为用户态和内核态两种情况，用户态下是通过系统调用接口，内核态下则是调用黑匣子的内核态接口。下面是一个黑匣子读写程序的示例，示例了一些关键函数的用法。

1) 用户态示例

```
#define RESERVE_REGION 0
#define SHARE_REGION 1
static int ubbox_rdwr(void * buf, int size)
{
    unsigned int offset = 0;
    offset = syscall(nr, SYSCALL_KBBOX_GETLEN, RESERVE_REGION);
    if (0 > syscall(nr, SYSCALL_KBBOX_STOREINFO, buf, size, RESERVE_REGION)) {
        printf("ERROR: SYSCALL_KBBOX_STOREINFO\n");
        return -1;
    }
}
```

```

/* 清除 buf 区内容 */
memset(buf, 0, size);
/* 假定这期间没有其他执行流程往黑匣子的非滚动区写入，则可在 offset 处读取刚刚写入的信息 */
if (0 > syscall(nr, SYSCALL_KBBOX_GETINFO, buf, offset, size, RESERVE_REGION)) {
    printf("ERROR: SYS_KBBOX_GETINFO\n");
    return -1;
}
/* 清除黑匣子区域所有存储的信息 */
if(0 != syscall(nr, SYSCALL_KBBOX_CLEAN)) {
    printf("ERROR: SYS_KBBOX_CLEAN\n");
    return -1;
}

/* 查看 buf 区信息内容 */
printf ("%s\n", buf);
return 0;
}

```

2) 内核态示例

```

static int kbbox_rdwr(void * buf, int size)
{
    unsigned int offset = 0;
    int ret = 0;
    offset = kbbox_get_len(RESERVE_REGION);
    if (0 > (ret = kbbox_store_info(buf, size, RESERVE_REGION))) {
        printf("ERROR: SYSCALL_KBBOX_STOREINFO\n");
        return ret;
    }
    /* 清除 buf 区内容 */
    memset(buf, 0, size);
    /* 假定这期间没有其他执行流程往黑匣子的非滚动区写入，可在 offset 处读取刚刚写入的信息 */
    if (0 > (ret = kbbox_get_info(buf, offset, size, RESERVE_REGION))) {
        printf("ERROR: SYS_KBBOX_GETINFO\n");
        return ret;
    }

    /* 清除黑匣子区域所有存储的信息 */
}

```

```

if (0 != (ret = kbbox_clean())) {
    printf("ERROR: SYS_KBBOX_CLEAN\n");
    return ret;
}

/* 查看 buf 区信息内容 */
printk ("%s\n", buf);
return 0;
}

```

9.2.3.5 接口说明

■ 内存块设备创建接口

接口：

```

syscall(unify_syscall, SYSCALL_DEV_MEMBLK, unsigned long addr, unsigned long size, char *path);
#define SYSCALL_DEV_MEMBLK ((5<<24) | (1<<16))

```

功能：应用程序通过统一系统调用创建内存块设备。支持将保留内存作为块设备供用户使用；同时也可将 ext 文件系统 mount 到该设备上，供用户以文件方式访问该块设备。

内核配置：CONFIG_MEMORY_BLOCK

输入参数：

- SYSCALL_DEV_MEMBLK：内存块设备功能号，为固定值；
- Addr：内存块设备基址；
- size：内存块设备容量；
- path：创建的设备节点路径，可以对这个设备节点格式化和挂载文件系统。

返回值：大于 0，成功；其它，失败。

■ 黑匣子区域基地址、滚动区大小和非滚动区大小

```

unsigned long bbox_base_phyaddr;    /* 黑匣子物理基地址*/
unsigned int  bbox_const_size;      /* 黑匣子非滚动区大小 */
unsigned int  bbox_loop_size;       /* 黑匣子非滚动区大小 */

```

说明：BSP 须提供以上三个变量的值，并保证 4 字节对齐。

■ 黑匣子模块初始化接口 `kbbox_init`

接口类型：内核态接口。

接口名称：

```
int kbbbox_init(void);
```

功能：通过调用此接口进行系统级数据初始化，该接口完成物理地址到虚拟地址的映射，并初始化系统级的黑匣子控制区数据结构，黑匣子滚动区和非滚动区数据结构的指针。用户在初始化 `bbox_base_phyaddr`、`bbox_const_size`、`bbox_loop_size` 三个变量后可直接调用该接口，该接口初始化成功后，用户便可使用黑匣子功能提供的各种使用接口。

返回值：

- 等于 0 表示初始化成功；
- 小于 0 表示调用失败，并返回错误码（-EINVA 参数错误；-ENOMEM 临时缓冲申请失败）。

■ 获取系统黑匣子信息接口 `kbbox_get_info`

接口：

```
int kbbbox_get_info(void * buf, unsigned int offset, unsigned int size, int region);
```

功能：用于内核态获取区域信息的接口。

输入参数：

- `buf`：指明要保存获取信息的缓冲的起始地址；
- `offset`：指明要获取的信息的起始偏移，该偏移依赖于黑匣子的区域；
- `size`：要获取信息的长度，为 0 时代表获取整个区域的信息；
- `region`：指明具体存储的区域是非滚动区还是滚动区。

返回值：

- 大于 0 表示调用成功，并返回读取的字节数，若需要读取的字节数大于实际的返回值，则表示已经读取到信息的末尾；
- 等于 0 表示已经读取到区域中信息的末尾，无新数据可以读；
- 小于 0 表示调用失败，并返回错误码（-EINVAL 参数错误；-EFAULT 其他错误）。

注意：参数中提供的 `buf` 缓冲的长度必须大于等于参数 `size` 的大小，否则将可能引起越界。

说明：调用 `int syscall(int nr, int id, char __user * buf, unsigned int offset, unsigned int size, int region)` 接口

用于用户态获取指定的区域偏移处开始一定大小的信息段。

■ 向系统黑匣子存储一条信息接口 `kbbox_store_info`

接口：

```
int kbbbox_store_info(void * buf, unsigned int size, int region);
```

功能：该接口用于内核态向黑匣子中存储一条信息。

输入参数：

- `buf`：指明要保存信息的缓冲区起始地址；
- `size`：要保存的信息的长度；
- `region`：指明具体存储的区域是非滚动区还是滚动区。

返回值：

- 大于 0 表示调用成功，并返回读取的字节数，当要保存的信息长度大于返回值时，表示区域（非滚动区）已经写满，剩余信息已经不能保存；
- 等于 0 表示区域（非滚动区）已经填满，无空间可以保存信息；
- 小于 0 表示调用失败，并返回错误码（-EINVAL 参数错误；-EFAULT 其他错误）。

注意：参数中提供的 `buf` 缓冲的长度必须大于等于参数 `size` 的大小，否则将可能引起越界。

说明：调用 `int syscall(int nr, int id, char __user * buf, unsigned int size, int region)` 接口用于用户态向黑匣子中存储一条信息。

■ 获取存储区域当前记录长度接口 `kbbox_get_len`

接口：

```
int kbbbox_get_len(region);
```

功能：该接口用于内核态获取指定区域已有记录信息的总长度。

输入参数：`region` 指明具体存储的区域是非滚动区还是滚动区。

返回值：返回指定区域的当前偏移值，如果是滚动区已经产生回环则返回整个滚动区大小。

说明：调用 `int syscall(int nr, int id, int region)` 接口用于用户态获取指定区域的总长度和用户态获取指定区域已有记录信息的总长度。

■ 获取存储区域总长度接口 `kbbox_get_size`

接口:

```
int kbbox _get_size(int region);
```

功能: 该接口用于内核态获取指定区域已有记录信息的总长度。

输入参数: **region** 指明具体存储的区域是非滚动区还是滚动区。

返回值: 返回当前存储区域的总大小。

说明: 调用 `int syscall(int nr, int id, int region)` 接口用于用户态获取指定区域的总长度和用户态获取指定区域已有记录信息的总长度。

■ 常规信息保存接口 `kbbox_store_context`

接口:

```
int kbbox_store_context(struct pt_regs *regs, int region);
```

功能: 该接口用于向黑匣子中记录异常的寄存器现场、堆栈、内存和运行队列等信息。

输入参数:

- **regs**: 该指针指向异常时保存的寄存器现场;
- **region**: 指明具体存储的区域是非滚动区还是滚动区。

返回值:

- 等于 0 表示调用成功;
- 小于 0 表示调用失败, 并返回错误码 (-EINVAL 参数错误; -EFAULT 其他错误)。

■ 黑匣子回调的注册接口

接口:

```
typedef int (*kbbox_callback)(void *);  
int kbbox_register_callback(kbbox_callback callback);
```

功能: 该接口用于注册黑匣子回调函数。

输入参数: **callback** 指向要注册的回调函数的指针。

返回值:

- 等于 0 表示调用成功;

- 小于 0 表示调用失败，并返回错误码（-EINVAL 参数错误；-EFAULT 其他错误）。

■ 黑匣子回调注销接口 `kbbox_unregister_callback`

接口：

```
int kbbox_unregister_callback(void);
```

功能：该接口用于注销黑匣子回调函数。

返回值：

- 等于 0 表示调用成功；
- 小于 0 表示调用失败，并返回错误码（-EINVAL 参数错误；-EFAULT 其他错误）。

■ 调用黑匣子回调的接口

接口：

```
int kbbox_callback(void * ptr);
```

功能：调用已注册的黑匣子回调函数，若未注册任何黑匣子回调函数，则返回。

输入参数：ptr 指向回调使用的参数，无需传参数时直接填写 NULL。

返回值：

- 等于 0 表示调用成功；
- 小于 0 表示调用失败，并返回错误码（-EINVAL 参数错误；-EFAULT 其他错误）。

■ 清空系统黑匣子的统一系统调用接口

接口：

```
int syscall(int nr, int id);
```

功能：黑匣子信息保存到非易失性物理介质后，就可用该接口来清除黑匣子区域了，主要是清除黑匣子状态相关的数据，清除黑匣子区域的数据。

输入参数：

- nr：对应于架构下的统一系统调用号；
- id：扩展子功能号，获取区域长度子功能号为 SYSCALL_KBBOX_CLEAN。

返回值：

- 0 表示调用成功；
- 小于 0 表示调用失败并返回错误码（-EINVAL 参数错误；-EFAULT 其他错误）。

9.2.4 CPU 占用率精确统计

9.2.4.1 功能描述

在实际项目应用中常常需要对 IDLE 任务调度精确计时，但标准 Linux 基于打点采样统计 idle 执行时间，不支持对 idle 执行时间的精确统计，导致 IDLE 任务的执行时间统计只能精确到 TICK，甚至在极端情况下 IDLE 任务的实际执行时间与统计结果之间误差较大。

CGEL 对此功能进行改造，内容包括 idle 任务 CPU 占用率精确统计、中断/软中断 CPU 占用率精确统计，并在此基础上，通过 linux top 命令提供系统 CPU 占用率的精确统计与显示。

9.2.4.2 应用场景

标准内核的 IDLE 任务运行时间、硬中断、中断运行时间是通过打点采样的方式计算出来的。在某些应用场景中，可能出现 CPU 实际利用率较小，但是采样结果显示出 CPU 利用率很高的现象。此功能模块就是用于解决这种问题，统计出精确的 CPU 占用率。

9.2.4.3 内核配置

配置宏选项 CONFIG_IDLESTATS。

在 menuconfig 中配置如下：

```
IDLE statistics --->
[*]Enable idle statistics<NEW>
```

9.2.4.4 运行使用

■ 查看系统统计信息

```
cat /proc/stat
```

可查看 hard_irq_run_time_cpu、irq_run_time_cpu 字段输出了每个核上硬中断、中断的运行时间，以及所有核上硬中断、中断运行时间总和。idle_exec_runtime_cpu 字段减去了 IDLE 任务被中断打断的时间。

■ 统计进程被中断打断的时间

```
cat /proc/<pid>/intr
```

可查看进程被中断打断的时间，包括四个值：被硬中断打断的时间（以 clock_t 为单位）、被中断打断的时间（以 clock_t 为单位）、被硬中断打断的时间（以微秒为单位）、被中断打断的时间（以微秒为单位）。

■ 统计线程被中断打断的时间

```
cat /proc/<pid>/task/<tid>/intr
```

查看线程被中断打断的时间，包括四个值：被硬中断打断的时间（以 clock_t 为单位）、被中断打断的时间（以 clock_t 为单位）、被硬中断打断的时间（以微秒为单位）、被中断打断的时间（以微秒为单位）。

9.2.5 异常输出控制台

9.2.5.1 功能描述

异常输出控制台，是针对内核出现异常时，能够可靠的将异常信息输出到用户模块注册的控制台，以便用户获取这些信息写入黑匣子或者转储到别的存储介质。异常输出控制台提供以下功能：

- 在系统异常处：panic、die、OOM、WARN 等地方，截获系统打印
- 把某种级别以上的打印向黑匣子共享区域保存，其中要求的级别可配置，提供调用 printk 的文件名和代码行和打印内容；当内核产生死锁时，产生死锁检测的输出
- 提供两种类型的控制台：一种控制台获取所有 printk 输出。这样，所有打印信息可能会同时送往串口、自定义控制台。另一种控制台只截获系统异常信息（即系统进入 panic、die、OOM、WARN 流程时的打印信息）。这样，所有异常信息可能会同时送往串口、自定义控制台
- 提供“异常打印不向串口输出”的功能。设置了该功能后，系统异常信息不再向串口输出。主要是为了防止在异常流程时，因串口打印过多造成看门狗过早复位

CGEL 在内核的 panic、die、OOM 等异常信息输出完毕后，需要采用可靠手段将设备复位。

9.2.5.2 应用场景

异常输出控制台，实现在内核出现异常时能够可靠的将异常信息输出到用户模块注册的控制台，以便用户获取这些信息写入黑匣子或者转储到别的存储介质。

9.2.5.3 内核配置

配置宏选项：

CONFIG_KERN_EXC_RELIABLE_RESET

CONFIG_BBOX_PRINTK_INFO (printk 死锁检测输出)

CONFIG_BBOX_LOGLEVEL_PRINTK (printk 打印级别)

在 menuconfig 中配置如下:

```
Kernel Exception Processing --->
  [*]Kernel Exception Callback Support
  [*]Kernel Exception Reliable Reset Support<NEW>
Kernel hacking --->
  Emergency for System Deadlock--->
    [*]Black box support
    [*]support printk
```

9.2.5.4 运行使用

■ 异常重启时的回调处理

```
void (*callback_system_reset)(void)
```

说明: 异常重启时, 可以通过该函数指针挂接相关的处理函数。

■ 在线控制异常输出功能

/proc/sys/kernel/excep_not_output 接口

说明: 控制异常信息是否输出到控制台。

9.2.6 数据断点调测增强

9.2.6.1 功能描述

针对数据断点调测增强, 主要分为基于硬件数据断点的调测功能和基于软件数据监控的调测功能。

■ 基于硬件数据断点调测功能

主要包括以下两方面:

- 1) 用户态应用程序通过系统调用, 在正常访问数据前可清除数据断点, 正常访问完毕后, 再恢复数据断点;

- 2) 若存在数据被意外修改的情况，内核向应用程序发送 SIGTRAP 信号。应用程序在 SIGTRAP 的处理钩子中，打印堆栈回溯等现场信息，供事后故障分析参考。如果是内核态断点异常，终止该进程并打印相关信息。

■ 基于软件数据监控调测功能

基于软件数据监控的调测功能只适用于单核，主要监控全局的用户态数据区，并且该数据区之前被写操作过，即监控的虚拟地址存在物理地址，不支持监控文件系统缓存，不支持有 mmap 映射，但没有初始化该区域的情况，不支持有 fork，但没有 exec 的情况。

9.2.6.2 应用场景

此功能适用于内存被异常修改时。

对于基于软件数据监控的调测功能自身不存在使用限制，但在某些使用场景下此功能可能无效：

- 1) 需要监控的虚拟地址没有对应的物理地址。例如 mmap 调用后未分配物理地址情况，无法监控。由于软件监控区间物理地址需要提供原始值，如果存在原始值，物理地址已分配。
- 2) 需要监控的虚拟地址对应的物理地址会自动变动。

例如 fork 创建子进程的情况，可能原来监控父进程空间突然变成监控子进程空间。这种情况下监控可能无效。

9.2.6.3 内核配置

配置宏选项：

- CONFIG_USER_SIMPLE_WATCHPOINT
- CONFIG_BOOKE_USER_WATCHPOINT
 - ✧ CONFIG_BOOKE_KERNEL_WATCHPOINT（仅 BOOKE 架构支持）
- CONFIG_BOOKE_KERNEL_WATCHPOINT（powerpc 85xx 以上）
 - ✧ CONFIG_BOOKE_KERNEL_WATCHPOINT（仅 BOOKE 架构支持）

在 menuconfig 中配置如下：

```
Watchpoint Support--->
  Watchpoint Type<No Watchpoint Support >---->
    [X]Simple Watchpoint Support
```

9.2.6.4 运行使用

通过 Proc 文件系统接口实现数据断点的设置与删除。

■ 设置或取消数据断点

用法:

```
echo tid idx addr flag > /proc/watchpoint/hard_regs
```

参数说明:

- tid 用户态时指线程号
- idx 设置数据断点时使用的硬件断点资源号
- addr 数据断点监控的地址 (addr 是 16 进制, 且必须以 0x 开头, 其他参数都是 10 进制)
- flag 用于标志设置内核态还是用户态数据断点, 0 代表用户态, 1 代表内核态

■ 设置或取消数据区间断点

用法:

```
echo tid start len flag > /proc/watchpoint/hard_rang
```

参数说明:

- tid 用户态时指线程号
- start 用户态区间起始虚拟地址 (start 是 16 进制, 且必须以 0x 开头, 其他参数都是 10 进制)
- len 区间长度, 不超过一个页面

■ 设置或取消软件区间监控

```
echo tid start len ignore > /proc/watchpoint/soft_range
```


参数说明:

- tid 用户态时指线程号
- start 用户态区间起始虚拟地址 (start 是 16 进制, 且必须以 0x 开头)
- len 区间长度, 不超过一个页面
- ignores 该参数为指向扩展参数的指针, 主要是为了方便以后扩展一些具体的功能, 目前该参数的功能是指定哪些线程不参与监控, 其中指定忽略的 tid 数小于 10。不使用该参数时输入 0x0

■ 软件区间监控结果的查询

用法:

```
cat /proc/watchpoint/soft_range
```

 提示：在内核配置了内核态数据断点但不使用的情况下，不影响用户态调试器数据断点的使用，但两类数据断点不能同时使用。

9.2.7 静态插点调测

9.2.7.1 功能描述

静态插点调测，主要提供对静态插点方式的机制支持，提供预设的针对内核部分的静态插点。此功能仅 CGEL4.X 提供。

9.2.7.2 应用场景

在内核的一些性能关键，使用频繁的模块中添加静态探测点。

9.2.7.3 内核配置

配置宏选项 CONFIG_TRACEPOINTS。

内核默认提供支持，menuconfig 下无该配置项。

9.2.7.4 运行使用

■ 插点函数

接口:

```
TRACE_EVENT (name, proto, args)
```

功能：用来定义一个静态检查点的宏模板，通过该模板，可以扩展出静态检查点调测动作钩子挂接函数，调测动作钩子卸载函数和调测动作钩子调用管理函数。

输入参数：

- name 静态检查点名字
- proto 输入钩子的参数类型声明
- args 输入钩子的参数声明

返回值：无

■ 插点钩子注册函数

```
int register_static_trace (name, void (*probe)(void))
```

功能：用来向指定名字的静态检查点注册一个调测动作钩子。

输入参数：

- name 静态检查点名字
- probe 调测动作函数钩子地址

返回值：0：成功；负数：错误原因。

■ 插点钩子注销函数

```
int unregister_static_trace (name, void (*probe)(void))
```

功能：用来从指定名字的静态检查点注销一个调测动作钩子。

输入参数：

- name 静态检查点名字
- probe 调测动作函数钩子地址

返回值：0：成功；负数：错误原因。

9.2.8 动态插点调测

9.2.8.1 功能描述

动态插点调测，主要是提供断点替换，跳转替换，PG 编译选项插点替换等动态机制的支持。

9.2.8.2 应用场景

动态插点调测可以在内核正在运行的时候，动态的添加调试代码，输出调试信息。

9.2.8.3 内核配置

配置宏选项 CONFIG_KPROBES。

在 menuconfig 中配置如下：

```
Instrumentation Support--->  
  [*]Kprobes <EXPERIMENTAL>
```

9.2.8.4 运行使用

■ 注册动态检查点

```
int register_kprobe(struct kprobe *p)
```

功能：用来在任一地址注册插入一个动态检查点。

输入参数：

- p 该参数包含了插入地址和调测动作的所有信息

返回值：0：成功；负数：错误原因。

■ 注销动态检查点

```
int unregister_kprobe(struct kprobe *p)
```

功能：用来注销卸载的一个动态检查点。

输入参数：

- p 该参数指向了待注销卸载的动态检查点

返回值：0：成功；负数：错误原因。

■ 多个地址注册一组动态检查点

```
int register_kprobes(struct kprobe **kps, int num)
```

功能：用来在多个地址注册插入的一组动态检查点。

输入参数：

- kps 该参数指向一组待插入动态检查点
- num 动态检查点的个数

返回值：0：成功；负数：错误原因。

■ 卸载一组动态检查点

```
int unregister_kprobes (struct kprobe **kps, int num)
```

功能：用来注销卸载的一组动态检查点。

输入参数：

- kps 该参数指向一组待注销卸载的动态检查点
- num 动态检查点的个数

返回值：0：成功；负数：错误原因。

■ 向任一函数入口注册动态检查点

```
int register_jprobe(struct jprobe *p)
```

功能：用来向任一函数入口注册插入一个动态检查点。

输入参数：

- p 该参数包含了函数入口地址和调测动作的所有信息

返回值： 0：成功；负数：错误原因。

■ 卸载函数入口动态检查点

```
int unregister _jprobe(struct jprobe *p)
```

功能：用来注销卸载的一个函数入口动态检查点。

输入参数：

- p 该参数指向了待注销卸载的函数入口动态检查点

返回值： 0：成功；负数：错误原因。

■ 多个函数入口注册动态检查点

```
int register_jprobes(struct jprobe **jps, int num)
```

功能：用来注册插入多个函数入口的一组动态检查点。

输入参数：

- `jps` 该参数指向一组待插入函数入口动态检查点
- `num` 动态检查点的个数

返回值：0：成功；负数：错误原因。

■ 卸载一组函数入口动态检查点

```
int unregister_jprobes (struct jprobe **jps, int num)
```

功能：用来注销卸载的一组函数入口动态检查点。

输入参数：

- `jps` 该参数指向一组待注销卸载的函数入口动态检查点
- `num` 动态检查点的个数

返回值：0：成功；负数：错误原因

■ 向任一函数返回点注册动态检查点

```
int register_kretprobe(struct kretprobe *p)
```

功能：用来向任一函数返回点注册插入一个动态检查点。

输入参数：

- `p` 该参数包含了函数返回点地址和调测动作的所有信息

返回值： 0：成功；负数：错误原因。

■ 卸载函数返回点动态检查点

```
int unregister _kretprobe(struct kretprobe *p)
```

功能：用来注销卸载的一个函数返回点的动态检查点。

输入参数：

- `p` 该参数指向了待注销卸载的函数返回点动态检查点

返回值： 0：成功；负数：错误原因。

■ 多个函数返回点注册动态检查点

```
int register_kretprobes(struct kretprobe **rps, int num)
```

功能：用来注册插入多个函数返回点的一组动态检查点。

输入参数：

- `jps` 该参数指向一组待插入函数返回点动态检查点
- `num` 动态检查点的个数

返回值：0：成功；负数：错误原因。

■ 卸载一组函数返回点动态检查点

```
int unregister_kretprobes (struct kretprobe **rps, int num)
```

功能：用来注销卸载的一组函数返回点的动态检查点。

输入参数：

- `jps` 该参数指向一组待注销卸载的函数返回点的动态检查点
- `num` 动态检查点的个数

返回值：0：成功；负数：错误原因。

9.2.9 深度睡眠长期不调度检测

9.2.9.1 功能说明

为每个 CPU 建立内核线程，用于定期检测任务的切换次数，一旦发现某个任务的切换次数在指定时间之内未被更新，则说明该核任务一直未被调度，则打印告警信息。

9.2.9.2 应用场景

检测长期处于深度睡眠的任务，并且此任务一段时间内不调度。

9.2.9.3 内核配置

配置宏选项 `CONFIG_DETECT_HUNG_TASK`。

在 `menuconfig` 中配置如下：

```
Kernel hacking--->
  [*]Kernel debugging
  [*]Detect Hung Tasks
```

9.2.9.4 使用说明

■ 开启深度睡眠长期不调度检测功能

```
echo x > /proc/sys/kernel/hung_task_panic
```

缺省内核未配置，值为 0；若配置，则为 1。

■ 监控系统内所有线程

```
echo all > /proc/hung_task
```

■ 监控指定 pid 范围的线程（数字——数字）

```
echo pid1... pid2 > /proc/hung_task
```

■ 设置每次遍历线程数目的上限

```
echo pid > /proc/sys/kernel/hung_task_check_count
```

默认值为 pid 的上限，完成一次遍历之后，下一次将重新开始遍历。

■ 设置检测间隔时间

```
echo 120 > /proc/sys/kernel/hung_task_timeout_secs
```

检测间隔时间，单位为秒，通常值是 120 秒。

只有睡眠时间超过 hung_task_timeout_secs 设置值 2 倍的任务才会被有效监控。即是说，若设置检测间隔时间为 120 秒，则能保证所有睡眠时间超过 240 秒的任务都被监控，相反，睡眠时间处于 120-240 秒之间的任务则可能被监控或不被监控，不能保证。

■ 设置打印告警次数

当检测出有睡眠任务不调度时，总共打印告警的次数，默认值为 10。

```
echo 10 > /proc/sys/kernel/hung_task_warnings
```

9.2.10 任务退出监控

9.2.10.1 功能描述

当指定的目标任务退出时，判断该任务是否异常退出，若为异常退出，则清除该任务 ID，并输出相关的打印信息，否则不会清除该任务 ID。

9.2.10.2 应用场景

需要监控某些任务是否异常退出，如果异常退出需打印该任务的内核栈回溯信息。

9.2.10.3 内核配置

配置宏选项 CONFIG_CAPTURE_THREAD。

在 menuconfig 中配置如下：

```
Device Drivers--->
  Misc devices--->
    [*]capture thread exit
```

9.2.10.4 运行使用

■ /proc/capture/enabled

用途：

/proc/capture/enable 接口作为输入和输出接口用于激活任务退出监控功能和查询当前的任务退出功能是否激活。

用法：

```
echo <va> > /proc/capture/enable
```

输入参数：

- <va>为 1： 激活任务退出监控功能
- <va>为 0： 关闭任务退出监控功能

```
cat /proc/capture/enable
```

输出参数：

- 输出为 1：任务退出监控功能已激活
- 输出为 0：任务退出监控功能未激活

■ /proc/capture/thread_ids

用途：

/proc/capture/thread_ids 接口作为输入和输出接口。

用法：

```
echo va > /proc/capture/thread_ids
```

输入参数：

- 如果 id 为一个进程的 pid，则所有该进程的线程都将被监控
- 如果 id 为一个线程的 id，则仅仅监控该线程

将需要监控的单个任务的 id 写入 thread_ids 文件中。该功能可以同时监控 TRHEAD_MAX 个 id，该宏的缺省值为 100；

```
cat /proc/capture/thread_ids
```

输出参数：

- 如果任务是正常退出，内核不会自动清除 thread_ids 中对应任务的 pid
- 如果任务是异常退出，则内核自动清除 thread_ids 中对应任务的 pid

读出当前正在被监控的所有任务 id。任务退出监控模块输出的打印信息非常多，如果用户使用串口调试，请用户在使用时将串口终端的当前打印级别调至 4（dmesg -n4），防止串口终端过于频繁，导致单板复位。

■ /proc/capture/reset

用途：

/proc/capture/reset 接口作为输入清空所有/proc/capture/thread_ids 中的所有 id。

用法：

```
echo 1 > /proc/capture/reset
```

清空当前所有带监控的任务 id。

■ /proc/capture/delid

用途:

/proc/capture/delid 接口作为输入清除待监控任务 id 列表中的某个 id。

用法:

```
echo va > /proc/capture/delid
```

写入某个不要监控的 id。

9.2.11 异常转储 Kdump

9.2.11.1 功能描述

异常转储功能 Kdump 是一种基于二次加载功能 (Kexec) 的可靠宕机转储 (crash dump) 解决方案, 该方案增强了内核的可维护性, 目前已合入主流内核分支。

Kdump 正常工作时需要两个内核:

- 1) 系统内核 (system kernel), 即系统正常工作时运行的内核;
- 2) 捕获内核 (dump-capture kernel), 即系统内核崩溃时, 用来进行数据转储的内核。

系统内核启动时, 通过修改启动选项保留出一部分内存供捕获内核使用, 这部分内存被称作保留区域 (reserved region)。在启动捕获内核之前, 需要加载该内核, 即把该内核映像拷贝到保留区域的最开始, 加载后当系统内核宕机时, Kdump 会在保存各种信息后利用 kexec 启动捕获内核, 并且就从保留区域中启动捕获内核, 而不是从系统默认的位置 (物理内存的开始), 这也是 Kump 可靠性的体现。在捕获内核中, 系统内核的内存可以通过 /proc/vmcore 文件进行访问。该方式是将转储输出为一个 ELF 格式的文件, 并且可以方便使用 crash 工具对其进行深入分析。

9.2.11.2 应用场景

应对各种内核宕机事件, 例如小概率发生的异常, 或者不明情况的内核宕机, 都可以借助 Kdump 功能来帮助分析。捕获内核仅在系统内核异常时被触发启动, 且运行在保留内存区间内启动, 可以完整保留系统内核宕机的第一现场, 通过 /proc/vmcore 文件, 借助 crash 等用户态工具, 可以对异常现场进行分析, 进而确定问题原因。

9.2.11.3 内核配置

由于内核配置选项在不同体系架构中的位置不同, 以下示例只是以 ARM 为例进行介绍, 其他体系架构的内核项位置可以通过搜索的方式 (“\” + 关键字) 定位配置项进行配置。

■ CGEL 系统内核配置

1) CONFIG_KEXEC=y

该选项使内核支持 kexec 系统调用。

若是 ARM 架构，还需同时选中 CONFIG_CONFIG_ATAGS_PROC=y，表示 proc 目录下注册 atags 文件系统。以 ARM 为例，在 menuconfig 中配置如下：

```
Boot options--->
  Kexec system call ( EXPERIMENTAL ) ( KEXEC [ = y ] )
```

2) CONFIG_DEBUG_INFO=y

该选项使构建出的内核包含调试符号。转储分析工具 crash 需要一个包含调试符号的 vmlinux，以便读取并分析转储文件。以 ARM 为例在 menuconfig 中配置如下：

```
Kernel hacking
```

3) CONFIG_SYSFS=y

该选项使能 sysfs 文件系统，使用户空间可以存取到系统设备和总线等信息。以 ARM 为例在 menuconfig 中配置如下：

```
FILE systems--->
  Pseudo filesystems--->
    Selected by : GFS2_FS [ =n ] && BLOCK [ = y ] && EXPERIMENTAL [ = y ]
```

4) 启动选项的设置

在系统内核的启动参数末尾添加如下内容，其中 Y 是为捕捉内核保留的内存，X 是保留部分内存的起始位置。

```
crashkernel = Y @ X
```

■ 捕获内核配置

CGEL 3.x\4.x 各个架构公共配置：

1) CONFIG_CRASH_DUMP=y

该选项使内核支持 kdump 机制。以 ARM 为例在 menuconfig 中配置如下：

```
Device Drivers--->
```

```
Misc devices--->
    [*]capture thread exit
```

2) CONFIG_PROC_VMCORE=y

该选项设置/proc/vmcore 支持。以 ARM 为例在 menuconfig 中配置如下：

```
Device Drivers--->
    Misc devices--->
        [*]capture thread exit
```

3) CONFIG_SMP=n

对于多 SMP 架构，需要关闭该选项，捕获内核仅支持单核。以 ARM 为例在 menuconfig 中配置如下：

```
Device Drivers--->
    Misc devices--->
        [*]capture thread exit
```

不同配置：对于捕获内核来说，还需要配置内核加载的地址。该地址必须与系统内核启动时传入的“crashkernel=Y@X”参数保持一致。捕获内核将在该地址范围内运行。不同的架构设置加载地址的方法不同，具体如下。

CGEL3.x :

MIPS32/MIPS64: (目前支持 ZTE-MSU4-XLS416/ZTE-MSU2-XLS408 小系统)

1) CONFIG_PHYSICAL_START = 0x84000000/0xffffffff84000000

表示捕获内核运行的起始物理地址。以 ZTE-MSU4-XLS416 为例在 menuconfig 中配置如下：

```
Kernel type--->
    kernel crash dumps ( EXPERIMENTAL ) (CRASH_DUMP [= y ])
```

PPC: (目前支持 ZTE-BBU-CCC-MPC8569E 小系统)

1) CONFIG_PHYSICAL_START=0x04000000

表示捕获内核运行的起始物理地址。在 menuconfig 中配置如下：

```
Advanced setup--->
```

```
Prompt for advanced kernel configuration options ( ADVANCED_OPTIONS [ = y ] ) --->
Set physical address where the kernel is loaded ( PHYSICAL_START_BOOL [ = y ] )
```

2) CONFIG_KERNEL_START=0x84000000

表示捕获内核运行的虚拟地址，该地址将与 CONFIG_PHYSICAL_START 配置的物理地址建立映射关系。在 menuconfig 中配置如下：

```
Advanced setup--->
Prompt for advanced kernel configuration options ( ADVANCED_OPTIONS [ = y ] ) --->
Set custom kernel base address ( KERNEL_START_BOOL [ = y ] )
```

X86/X86_64:

1) CONFIG_PHYSICAL_START=0x1000000

表示捕获内核运行的起始物理地址。在 menuconfig 中配置如下：

```
Processor type and features
```

CGEL4.x :

ARM: (目前支持 TI-DM816X-AM3892 小系统)

1) CONFIG_PHYS_OFFSET = 0x84000000

表示捕获内核运行的起始物理地址。在 menuconfig 中配置如下：

```
Boot options--->
Build kdump crash kernel ( EXPERIMENTAL ) (CRASH_DUMP [ = y ] )
```

MIPS64: (目前支持 ZTE-AFMPFUA-xlp316 小系统)

1) CONFIG_PHYS_LOAD_ADDRESS = 0xffffffff84600000

表示捕获内核运行的起始物理地址。在 menuconfig 中配置如下：

```
Machine selection--->
Mapped kernel ( MAPPED_KERNEL [ = y ] )
Kernel type--->
```

Mapped kernel (MAPPED_KERNEL [= y])

2) CONFIG_NLM_COMMON_LOAD_ADDRESS = 0xffffffffc4600000

表示捕获内核运行的虚拟地址，该地址将与 CONFIG_PHYS_LOAD_ADDRESS 配置的物理地址建立映射关系。在 menuconfig 中配置如下：

Machine selection

PPC: (目前支持 ZTE-CDMA-CC0-mpc8360 小系统)

1) CONFIG_PHYSICAL_START = 0x02000000

表示捕获内核运行的起始物理地址。在 menuconfig 中配置如下：

Advanced setup--->

Prompt for advanced kernel configuration options (ADVANCED_OPTIONS [= y]) --->

Set physical address where the kernel is loaded (PHYSICAL_START_BOOL [= y])

2) CONFIG_KERNEL_START = 0xC2000000

表示捕获内核运行的虚拟地址，该地址将与 CONFIG_PHYSICAL_START 配置的物理地址建立映射关系。在 menuconfig 中配置如下：

Advanced setup--->

Prompt for advanced kernel configuration options (ADVANCED_OPTIONS [= y]) --->

Set custom kernel base address (KERNEL_START_BOOL [= y])

3) 用户态使用 kexec-tools 加载捕获内核是，需要使用 -append 参数添加 nobats 选项。

X86/X86_64:

1) CONFIG_PHYSICAL_START=0x1000000

表示捕获内核运行的起始物理地址。在 menuconfig 中配置如下：

Processor type and features

2) CONFIG_RELOCATABLE=y

在 menuconfig 中配置如下：

9.2.11.4 触发条件

- 如果内核异常流程中调用了函数 `die()`，并且该线程的 `pid` 为 0 或 1；或者在中断上下文中调用 `die()`；或者设置了 `panic_on_oops` 并调用了 `die()`；上述情况下，系统将启动 `crash_kexec` 进入转储捕捉内核；
- 当内核发生无法恢复的错误时，会调用函数 `panic`，系统将启动 `crash_kexec` 进入转储捕捉内核；
- 为了测试目的，可以使用 “`echo c > /proc/sysrq-trigger`” 触发一个崩溃。

9.2.11.5 运行使用

CGEL 及各个体系架构 `Kdump` 的使用方法基本相同，现以 CGEL4.x arm 架构为例，描述 `Kdump` 功能的使用方法：

- 1) 按照配置要求，分别配置并编译得到系统内核 `uImage` 与捕获内核 `uImage_dump`。
- 2) `boot` 中设置启动参数 `crashkernel=Y@X`，并启动系统内核 `uImage`。

需要根据各个单板的内存使用和划分的情况，设置保留内存。如果设置成功，将在内核启动阶段有如下打印：

```
Reserving Y MB of memory at X MB for crashkernel (System RAM: xxx MB)
```

- 3) 在系统内核 `uImage` 下挂在 `sysfs`。

```
#mount -t sysfs /sys /sys
```

- 4) 在系统内核 `uImage` 下，使用开源工具 `kexec-tools`，将捕获内核加载到系统内核保留的内存中。

```
#!/kexec-tools -p uImage_dump --ramdisk=initramfs_data.cpio.gz --append="console=ttyO2,115200  
root=/dev/ram rw"
```

其中 `-p` 表示加载捕获内核为 `uImage_dump`，`--ramdisk` 表示捕获内核使用的根文件系统，`--append` 表示用户追加的捕获内核的启动参数。

- 5) 如没有任何异常提示，则表示捕获内核加载成功，此时可以在系统内核上正常运行各种应用程序。

6) 如果出现异常导致内核宕机，系统内核将自动在保留内存区间中运行捕获内核。在捕获内核下，可以使用 `crashdumpfile` 等工具，将 `/proc/vmcore` 文件转储为 ELF 格式的 `dump` 文件，并使用 `crash` 工具进行分析异常现场。

9.2.12 任务长时间运行或不运行监控

CGEL 3.x 提供针对任务长时间不运行(即未获取到 CPU 资源)与任务长时运行两种情况的监控功能。

9.2.12.1 功能描述

■ 监控任务长时间不运行功能（即未获取到 CPU 资源）

该功能通过提供 proc 接口可监控一个任务在指定时间间隔中是否未运行。当检测到指定的任务不运行超过用户设定的时间间隔时，如果该任务为阻塞状态，打印出该任务调用链；如果为 R 状态，打印出该任务所在核上获得 CPU 资源的任务 PID 号，任务名字，抢占计数，优先级。通过 proc 接口方式提供给用户最多不超过 5 个任务的长时间不运行监控功能。

该功能默认不启动，通过 proc 接口在设置监控任务后启动，取消时停止运行。该功能是基于 timer 来检测，精度最小为 1 个 jiffies。

■ 监控任务长时运行功能

任务长时间运行，是指一个任务在指定 CPU 上连续运行时间超过一个用户指定的时间间隔。

当检测到任务连续运行超过用户设定的时间间隔时，打印出该任务所在 CPU，以及任务名字，PID 号，优先级，抢占计数信息。设置监控时间为有效值（大于 0）时，开启对所有任务监控，设置为其他值（小于等于 0）时，关闭该功能。该功能是基于 timer 来检测，精度最小为 1 个 jiffies。

9.2.12.2 应用场景

■ 监控任务长时间不运行功能（即未获取到 CPU 资源）

有些系统出现的问题，是因为有些任务长时间没有得到执行造成的。而这些任务具体在什么时刻需要运行是不确定的，预先能确定的只是这些任务在一个确定的时间段内应该得到执行。排查这种问题，可以使用监控任务长时间不运行功能。

用户通过指定监控任务的 PID 和最大允许的不运行时间段来启用监控任务长时间不运行功能。此后一旦当检测到指定的任务不运行超过用户设定的时间间隔时，如果该任务为阻塞状态，打印出该任务调用链；如果为 R 状态，打印该任务所在核上，获得 CPU 资源的任务 PID 号，任务名字，抢占计数，优先级。

■ 监控任务长时运行功能

在一个 CPU 上，如果一个任务长期占用 CPU，将会造成在这个 CPU 上的其他任务得不到及时执行，在某些情况下由于这些任务得不到及时执行，将会引发一些问题。对于这种情况造成的问题，可以使用监控任务长时间运行功能来确认和协助定位。

用户首先设定一个时间间隔来启用监控功能，此后一旦系统中有任务连续运行的时间超过这个设定的时间间隔，将会被监控功能捕获，并且打印出这个任务的所在 CPU，以及任务名字，PID 号，优先级，抢占计数，以使用户在第一时间发现这些问题。

9.2.12.3 内核配置

配置宏 CONFIG_TASK_MONITOR，在 menuconfig 中配置如下：

```
Kernel hacking--->
[*]Monitor tasks which run too long or too little time
```

9.2.12.4 运行使用

■ 监控任务长时间不运行功能（即未获取到 CPU 资源）

监控任务长时间不运行功能默认是没有启动的，需要在设置监控任务后启动。可以通过以下方法设置 Proc 接口实现：

1) 设置需要监控的任务 PID，这个是必须提供的；

```
cat /proc/sys/kernel/moni_nonrunning/pids          //输出监控任务。
echo xx[,xx,xx,xx,xx] >/proc/sys/kernel/moni_nonrunning/ pids    //设置监控任务。
```

默认值为-1，表示关闭状态，最多设置 5 个，后续设置覆盖前面设置，输入有效监控任务，启动该功能，系统只监控用户数据的有效任务 id，忽略无效任务 id。

2) 设置任务的最大不运行时间，即一旦超出这个时间指定的任务还没有得到执行，监控功能就会捕获这个任务。

```
cat /proc/sys/kernel/moni_nonrunning/interval      //输出监控间隔时间。
echo 40 >/proc/sys/kernel/ softlockup_print_interval //设置监控时间，示例为 40ms。
```

该接口单位为 ms，默认值为 1000，精度为 1 个 jiffies，四舍五入。

3) 当检测到指定的任务不运行超过用户设定的时间间隔时，如果该任务为阻塞状态，打印出该任务调用链；如果为 R 状态，打印出该任务所在核上获得 CPU 资源的任务 PID 号，任务名字，抢占计数，优先级。

■ 监控任务长时运行功能

任务长时间运行，是指一个任务在指定 CPU 上连续运行时间超过一个用户指定的时间间隔。此功能默认不启动，在设置监控时间后启动。可以通过以下方法设置 Proc 接口实现：

1) 设置任务最大允许运行的时间，这个时间表示系统所允许一个任务连续运行的最长时间。即一个任务一旦连续运行的时间超过这个指定的时间，就会被监控功能捕获。

```
cat /proc/sys/kernel/moni_running/interval          //输出监控时间。
echo 40 >/proc/sys/kernel/moni_running/interval      //设置监控任务，示例为 40ms。
```

设置监控时间为有效值（大于 0）时，开启对所有任务监控，设置为小于等于 0 时，关闭该功能（默认值为-1）。该接口单位为 ms，基于 timer 来检测，精度最小为 1 个 jiffies，四舍五入。

2) 在多核系统中，可以指定在哪些 cpu 上进行监控。

```
cat /proc/sys/kernel/moni_running/cpus_allowed      //输出 cpu 掩码。
echo xx > /proc/sys/kernel/moni_running/cpus_allowed //设置 cpu 掩码，默认全部监控。
```

3) 当检测到任务连续运行超过用户设定的时间间隔时，打印出该任务所在 CPU，以及任务名字，PID 号，优先级，抢占计数信息。

9.2.13 内核态回溯用户态堆栈功能

9.2.13.1 功能描述

从当前任务的内核栈中获取该任务陷入内核时保存的用户态栈信息，包括用户态栈指针、指令指针、寄存器值等，并根据相应的堆栈组织规则回溯用户态堆栈调用链。

- 打印当前任务的用户态堆栈调用链到串口；
- 获取当前任务的用户态堆栈调用链，并保存到相关结构中，用户可以自由使用相关数据，如输出到指定文件或存储到黑匣子。

9.2.13.2 应用场景

■ 监视任务的异常退出，查看用户程序的退出路径

复杂的应用程序，退出点可能比较多，程序执行起来后用户不一定知道程序会从哪个地方退出，特别是调用了第三方库时，在未知的代码段中可能存在调用 exit 的情况，这种情况下可在内核 do_exit 接口中调用回溯用户态堆栈的接口，回溯当前任务用户态堆栈调用链。

■ 监视临界区内发生调度时用户程序的调用路径

应用程序在临界区内执行了错误的代码，可能导致任务发生调度，进而引发死锁等问题，这种情况下可以在调度上下文检测当前任务是否在临界区内，如果在临界区可以回溯用户态堆栈，进而确定进入临界区的代码所在的调用路径。

9.2.13.3 使用约束

1) powerpc、arm、x86、x86_64 架构使用该功能的用户态程序需要包含完整的栈帧结构，如果使用优化编译选项，可能会导致栈帧信息不完整，建议应用程序编译时添加-fno-omit-frame-pointer 选项以保留堆栈回溯相关信息；

2) x86_64、arm 架构目前 TSP 提供的工具链对 c 库函数做了优化，调用 c 库时可能导致堆栈回溯不成功；

3) mips 架构下，函数调用路径存在函数体较大的函数时可能回溯失败（函数体较大指汇编指令超过 4000 条）。

9.2.13.4 内核配置

配置宏 CONFIG_BACKTRACE_USRSTACK，在 menuconfig 中配置如下：

```
Kernel hacking --->
  [*]User-stack trace enable
```

该配置默认关闭，根据用户需要可手动配置开启该功能。

9.2.13.5 运行使用

下面以一则实例介绍内核态回溯用户态堆栈功能的使用方法。

实例功能：确认应用程序的退出路径。

应用程序代码：

```
Int main(int argc, char** argv){
    Func1();
}

Int func(void){
    Thirdpart_libfunc()
}

Int Thirdpart_libfunc(){ //三方库函数、实现对应用不可见或不可修改
```

```
Exit(1);
}
```

方法步骤：

1) 编译应用程序，可执行文件 a.out；

2) 在内核 do_exit 内添加回溯用户栈代码 backtrace_usrstack ()，可添加进程号、调用次数等调用条件进行限制；

3) 应用执行到 Thirdpart_libfunc 中 exit(1)时，触发系统调用进入内核 do_exit()流程，在满足条件情况下会调用 backtrace_usrstack () 打印用户栈的调用链；格式如：

```
Usrstack Call Trace:
[<7f9ed988>] 0x4004b4
[<7f9ed9e8>] 0x400420
.....
[<7f9edb98>] 0x400804
[<7f9edc28>] 0x400b3c
```

4) 通过 objdump 工具反汇编 a.out，在反汇编文件中查找 0x400xxx 地址对应的函数，即可确认应用程序的退出路径。

9.2.13.6 接口说明

■ 内核态下回溯 current 任务的用户态堆栈

接口：

```
void backtrace_usrstack()
```

说明：打印当前任务的用户态堆栈信息。

■ 获取当前任务的用户态堆栈栈帧信息

接口：

```
void usr_stack_trace(CALL_STACK_INFO * stackinfo)
```

说明：获取当前任务的用户态堆栈栈帧信息，存入 stackinfo 指向地址。

参数：输出参数 CALL_STACK_INFO 定义如下。

```
#define MAX_FUNC_TRACK    32 /* max backtrace level */
typedef struct
{
    unsigned long RetAddr; /* Return address*/
    unsigned long sfp; /* The stack frame pointer*/
}CALL_STACK_FUNC_INFO;
typedef struct
{
    unsigned long tid; /* Task ID */
    CALL_STACK_FUNC_INFO Funcs[MAX_FUNC_TRACK]; /*back trace result*/
    unsigned long NestCount; /* The actual fucntion called level */
}CALL_STACK_INFO;
```

9.2.14 查看软中断运行次数

9.2.14.1 功能描述

CGEL 上提供软中断运行次数查看功能，可显示各 CPU 上各类型软中断运行次数，如下所示。

9.2.14.2 应用场景

用于显示各 CPU 上各类型软中断运行次数功能（不区分软中断上下文和线程上下文，也即对于软中断线程化的情况也统计在这里，不做单独统计）。此功能对于分析定位问题有所帮助。

9.2.14.3 内核配置

无。

9.2.14.4 使用方法

通过 proc 接口来显示。

以下面两个 CPU 的单板为例，第一列表示软中断类型，第二列表示 CPU0 上的各类型软中断运行次数，第三列表示 CPU1 上各类型软中断运行次数。

```
# cat /proc/softirqs
          CPU0      CPU1
high:      0        0
```

timer:	20489	18901
net-tx:	0	0
net-rx:	14	1
block:	0	0
tasklet:	4137	366
sched:	3540	3389
rcu:	0	0

9.2.15 printk 显示当前 CPU 号

9.2.15.1 功能描述

CGEL 提供在内核下调用 `printk` 输出信息中包含当前 CPU 号。

9.2.15.2 应用场景

主要是进行内核开发、调试、跟踪定位时需要调用 `printk` 进行信息输出的地方，特别是在多核环境下，需要准确知道每行输出具体是由哪个 CPU 来处理的时候就很有效果。

9.2.15.3 内核配置

配置宏 `CONFIG_PRINTK_CPUID`，在 `menuconfig` 中配置如下：

```
Kernel hacking --->
[*] Show cpu id information on printks
```

该配置默认关闭，根据用户需要可手动配置开启该功能。

9.2.15.4 使用方法

该功能有两种使用方法：

- 1) 配置内核方式：配置宏为 `CONFIG_PRINTK_CPUID`；
- 2) 指定启动参数方式：`cpuid`

例如：`root=/dev/ram rw console=ttyS0,115200 cpuid`

当上述方式之一指定之后，从控制台上可以直观地看到输出信息如下所示（行首[]中的数字表示的就是该行输出时所在的 CPU）：

```
[ 0] CPU: L1 I Cache: 64K (64 bytes/line), D cache 64K (64 bytes/line)
[ 0] CPU: L2 Cache: 512K (64 bytes/line)
[ 0] CPU 0/0 -> Node 0
[ 0] mce: CPU supports 0 MCE banks
[ 0] SMP alternatives: switching to UP code
[ 0] Using local APIC timer interrupts.
[ 0] APIC timer disabled due to verification failure.
[ 0] SMP alternatives: switching to SMP code
[ 0] Booting processor 1/4 APIC 0x1
[ 1] Initializing CPU#1
[ 1] Calibrating delay using timer specific routine.. 24713.51 BogoMIPS (lpj=12356758)
[ 1] CPU: L1 I Cache: 64K (64 bytes/line), D cache 64K (64 bytes/line)
[ 1] CPU: L2 Cache: 512K (64 bytes/line)
[ 1] CPU 1/0 -> Node 0
[ 1] mce: CPU supports 0 MCE banks
[ 1] QEMU Virtual CPU version V2.01.20 build by annie stepping 03
```

9.3 IPC 增强

9.3.1 快速信号量

9.3.1.1 功能描述

快速信号量，提供一种比标准 Linux 的 POSIX 信号量接口效率更高的同步互斥机制，提供信号量创建，删除，以及 P/V 操作，同时对于互斥信号量，支持优先级继承协议（PIP），以防止优先级反转问题。

用户态快速信号量优先级继承功能是为了使用户能够在用户态环境下通过快速信号量实现第一级优先级继承。

9.3.1.2 应用场景

在一些单核 CPU 上，为了满足用户态应用对性能的需求，需要提供一些进程间快速通信机制（UFIPC），包括：快速消息机制、快速信号量机制、用户态开关中断，以提升应用程序在用户态的执行效率。

9.3.1.3 内核配置

配置宏选项 CONFIG_UF_WQ 、 CONFIG_UF_SEM_PIP1 （信号量优先级继承）。

在 menuconfig 中配置如下：

```
userspace fast ipc--->
    [*]Enable userspace fast wait queue
    [*]userspace fast wait queue support PIP1
```

9.3.1.4 运行使用

■ B 信号量初始化接口

结构：

```
int uf_semb_init(uf_sem_t *p_sem, int opts, semb_state_t ini_state)
```

输入参数：

- p_sem 初始化信号的结构体地址
- opts 信号量选项
- ini_state 初始状态

返回值：成功，0；失败，非 0。

■ M 信号量初始化接口

结构：

```
int uf_semm_init(uf_sem_t *p_sem, int opts)
```

输入参数：

- p_sem 信号量结构体地址
- opt 信号量初始化选项

返回值：成功，0；失败，非 0。

■ C 信号量初始化接口

结构：

```
int uf_semc_init(uf_sem_t *p_sem,int opts, int ini_cnt)
```


输入参数:

- p_sem 初始化信号的结构体地址
- opts 信号量选项
- ini_cnt 信号量初始计数值

返回值: 成功, 0; 失败, 非 0。

b)

■ 信号量删除接口

结构:

```
int uf_sem_del(uf_sem_t* p_sem);
```

输入参数:

- p_sem 删除信号量的结构体地址

返回值: 成功, 0; 失败, 非 0。

■ B 信号量获取接口

结构:

```
int uf_semb_take(uf_sem_t* p_sem, int timeout)
```

输入参数:

- p_sem 信号量的结构体地址
- timeout 超时时间

返回值: 成功, 0; 失败, 非 0。

■ M 信号量获取接口

结构:

```
int uf_semm_take(uf_sem_t* p_sem, int timeout)
```

输入参数:

- p_sem 信号量的结构体地址
- timeout 超时时间

返回值: 成功, 0; 失败, 非 0。

■ C 信号量获取接口

结构:

```
int uf_semc_take(uf_sem_t* p_sem, int timeout)
```

输入参数:

- p_sem 信号量的结构体地址
- timeout 超时时间

返回值: 成功, 0; 失败, 非 0。

■ B 信号量释放接口

结构:

```
int uf_semb_give(uf_sem_t* p_sem)
```

输入参数:

- p_sem 信号量的结构体地址

返回值: 成功, 0; 失败, 非 0。

■ M 信号量释放接口

结构:

```
int uf_semm_give(uf_sem_t* p_sem)
```

输入参数:

- p_sem 信号量的结构体地址

返回值: 成功, 0; 失败, 非 0。

■ C 信号量释放接口

结构:

```
int uf_semc_give(uf_sem_t* p_sem)
```

输入参数:

- p_sem 信号量的结构体地址

返回值: 成功, 0; 失败, 非 0。

编译用户态测试程序时, 修改 makefile, 加上 CFLAGS += -DCONFIG_UF_SEM_PIP1, 这样用户态应用也支持优先级继承了。

运行可执行程序，在 `busybox` 上观察主任务的优先级，当新任务挂到信号量等待队列时，任务优先级会改变。

```
# cat /proc/827/sched
-----
se.exec_start      :      0.000000
se.vruntime        :      55353.267354
se.sum_exec_runtime :      1536.857865
nr_switches        :           36
se.load.weight     :      177522
policy             :           1
prio               :          36
clock-delta        :      38000

# cat /proc/827/sched
ufiptest.exe (827, #threads: 6)
-----
se.exec_start      :      0.000000
se.vruntime        :      55353.267354
se.sum_exec_runtime :      1536.857865
nr_switches        :           36
se.load.weight     :      177522
policy             :           1
prio               :          29
clock-delta        :      140800

# cat /proc/827/sched
-----
se.exec_start      :      0.000000
se.vruntime        :      55354.609434
se.sum_exec_runtime :      1721.613267
nr_switches        :           38
se.load.weight     :          1024
policy             :           0
prio               :         120
clock-delta        :      24600
```

9.3.2 快速消息队列

9.3.2.1 功能描述

快速消息机制，实现一个基于共享内存和快速用户态同步系统调用的消息队列，从而提供消息队列的创建，删除，发送和接收的接口，并提供高效的消息唤醒机制。

9.3.2.2 应用场景

在一些单核 CPU 上，为了满足用户态应用对性能的需求，需要提供一些进程间快速通信机制（UFIPC），包括：快速消息机制、快速信号量机制、用户态开关中断，以提升应用程序在用户态的执行效率。

9.3.2.3 内核配置

配置宏选项 CONFIG_UF_FSA。

在 menuconfig 中配置如下：

```
userspace fast ipc--->
    [*]Enable userspace fast supervisor access
```

9.3.2.4 运行使用

消息队列提供了消息队列的基本接口，包括消息队列初始化，消息队列发送，消息队列接收，消息队列删除。

■ 创建消息队列

结构：

```
Uf_msgq_t * Uf_msgq_create(int max_msgs, int max_msglen, int opt)
```

输入参数：

- max_msgs 最大消息数
- max_msglen 消息最大长度
- opt 消息选项

返回值：成功，非空；失败，NULL。

■ 删除消息队列

结构:

```
int uf_msgq_delete(uf_msgq_t * pMsgQ)
```

输入参数:

- pMsgQ 要删除的消息队列指针

返回值: 成功, 零; 失败, 非零。

■ 消息队列接收函数

结构:

```
int uf_msgq_recv(uf_msgq_t * pMsgQ, char *buffer, unsigned int buf_len, int timeout)
```

输入参数:

- pMsgQ 消息队列结构指针
- buffer 接收缓冲区
- buf_len 接收长度
- timeout 超时时间

返回值: 成功, 0; 失败, -1。

■ 消息队列发送接口

结构:

```
int uf_msgq_send(uf_msgq_t * pMsgQ, char *buffer, unsigned int nBytes, int timeout, int pri)
```

输入参数:

- pMsgQ 消息队列结构指针
- buffer 发送缓冲区
- nBytes 发送长度
- timeout 发送超时时间
- pri 发送消息优先级

返回值: 成功, 0; 失败, 非零。

9.4 文件系统增强

9.4.1 JFFS2 启动效率优化

9.4.1.1 功能描述

由于文件系统在 mount 后 gc 线程会校验所有数据节点数据 crc，其校验过程比较耗时，以及在系统较脏情况下，“写”性质操作必须要等 gc 校验结束，导致第一次“写”操作延时很大。为了减少第一次“写”操作的延时，改集中校验为分散校验，分散到每个文件的第一次读、写或者数据节点的搬运操作中，从而提高 JFFS2 的启动效率。

9.4.1.2 应用场景

gc 校验后会导致随后的第一次“写”操作延时很大，无法满足应用需求，JFFS2 启动优化即是应用于此场景下，对 gc 检验进行优化，从而解决 gc 检验导致的第一次“写”操作延迟问题。

9.4.1.3 内核配置

配置宏选项 CONFIG_JFFS2_GC_EXT。

依赖配置：CONFIG_JFFS2_FS。

在 menuconfig 中配置如下：

```
File system--->
  [*]Miscellaneous filesystem--->
    [*]Journalling Flash File System V2<JFFS2>support
    [*]JFFS2 speed up starting
```

9.4.1.4 运行使用

正确完成内核配置后，在 gc 校验时系统便会自动触发 JFFS2 启动效率优化。

9.4.2 JFFS2 主动垃圾回收机制

9.4.2.1 功能描述

CGEL 下 JFFS2 文件系统垃圾主动回收机制是为了解决文件系统垃圾过多时影响文件系统问题的一种方法。当文件系统垃圾过多时，每次操作前文件系统必须先收集垃圾，然后才能进行空间分配，大大影响了 JFFS2 文件系统的效率。主动回收机制通过应用层触发，在文件系统空闲时，进行垃圾回收，使文件系统处于干净状态，这样速率得到很大提高。

9.4.2.2 应用场景

JFFS2 文件系统在使用过程中，随着垃圾的增多，写入性能会受到一定的影响，尤其当系统负载较重时，导致内核非实时的垃圾回收线程频繁被抢占，不能及时回收足够的空间。JFFS2 用户态主动垃圾回收则是用于解决此问题的。

9.4.2.3 内核配置

配置宏选项 CONFIG_JFFS2_FS。

在 menuconfig 中配置如下：

```
File system--->
  [*]Miscellaneous filesystem--->
    [*]Journalling Flash File System v2<JFFS2>support
```

9.4.2.4 运行使用

- 查看 jffs2 文件系统参数

```
cat /proc/sys/fs/jffs2/分区名/info_N
```

查看该分区下 jffs2 文件系统参数。

- 制定合适的垃圾回收策略

```
echo N > /proc/sys/fs/jffs2/分区名/start_gc_N
```

制定合适的垃圾回收策略，在用户态主动触发垃圾回收，当 start_gc_N 不为 0 的时候触发垃圾回收。

用户还可以使用 sysctl 系统调用接口同样可以获取 jffs2 文件系统的信息参数并制定垃圾回收策略。来决定什么时候通过 sysctl 主动触发垃圾回收。在使用 sysctl 时，需要用户定义相应的数据结构 struct jffs2_flash_info，以及 FS_JFFS2=22, JFFS2_PART0 = 1,PART_INFO = 1,PART_START_GC = 2 等信息，并

和内核保持一致。

9.4.3 LZMA 压缩与解压功能

9.4.3.1 功能描述

LZMA, (Lempel-Ziv-Markov chain-Algorithm 的缩写), 是一个 Deflate 和 LZ77 算法改良和优化后的压缩算法, 开发者是 Igor Pavlov, 2001 年被首次应用于 7-Zip 压缩工具中, 是 2001 年以来得到发展的一个数据压缩算法。LZMA 算法是 7z 格式的默认标准算法, 它的主要特征有:

- 高压缩比率;
- 可变的字典大小 (高达 4GB);
- 压缩速度: 在 2 GHz CPU 上, 大约 1 MB/s;
- 解压缩速度: 在 2 GHz CPU 上, 大约 10-20 MB/s ;
- 较小解压缩内存 (依赖于所选的字典大小);
- 较小的解压缩代码, 大约 5KB;
- 支持多线程。

基于以上优点, LZMA 压缩算法适合嵌入式应用, CGEL 内核提供对根文件系统进行 LZMA 压缩及解压支持。



提示: 使用 LZMA 进行压缩的优势是压缩比高, 比使用 gzip 方式进行压缩, 压缩比要高一些, 但是相应的, 解压 LZMA 格式文件的时候解压时间就比 gzip 多一些。

9.4.3.2 应用场景

由于一般使用 LZMA 压缩后的文件大小是原来的 1/3, 所以适用于对根文件系统大小比较敏感, 希望尽量减少压缩后根文件系统体积的场景。

9.4.3.3 内核配置

配置宏选项 CONFIG_DECOMPRESS_LZMA。

在 menuconfig 中配置如下：

```
General setup--->
  Initial RAM filesystem and RAM disk (initramfs/initrd) support
  Built-in initramfs compression mode (None) --->
    ( ) None
    ( ) Gzip
    (X) LZMA
```

在内核配置时，需要首先选中【Initial RAM filesystem and RAM disk (initramfs/initrd) support】，再选中【LZMA】即可。

9.4.3.4 运行使用

要生成 LZMA 压缩格式的根文件系统，需要编译环境配置 LZMA 工具。然后在 CGEL 内核配置中选中宏 CONFIG_DECOMPRESS_LZMA，内核即可支持压缩和解压缩 LZMA 格式的根文件系统。

9.4.4 VFAT 专利规避功能

9.4.4.1 功能描述

CGEL 内核提供以下两个宏选项用于规避微软 VFAT 相关专利：

- CONFIG_VFAT_FS_FAKE_SHORTNAME：如果配置则创建长名、虚假短名目录项；
- CONFIG_VFAT_NO_CREATE_WITH_LONGNAMES：不创建长名目录项，对文件名长度大于 11 的创建动作返回失败。但能够浏览磁盘中已存在的长名文件。

9.4.4.2 应用场景

在实际应用中，此功能宏可用于规避对微软 VFAT 专利侵权敏感的场景。

9.4.4.3 内核配置

配置【VFAT fake short names support (NEW)】启用宏选项 CONFIG_VFAT_FS_FAKE_SHORTNAME；配置【Disable creating files with long names (NEW)】启用宏选项 CONFIG_VFAT_NO_CREATE_WITH_LONGNAMES。

在 menuconfig 中配置如下：

```
File systems--->
```

VFAT (Windows-95) fs support--->
VFAT fake short names support (NEW)
Disable creating files with long names (NEW)

另外，在配置 VFAT 时还需根据实际情况配置输入页码号和字符集。

1) 通常页码号英语地区使用的“437”:

File systems--->
DOS/FAT/NT Filesystems--->
VFAT (Windows-95) fs support --->
(437) Default codepage for FAT

同时配置该页码的对应软件模块:

File systems--->
Native language support--->
Codepage 437 (United States, Canada)

2) 通常字符集覆盖英语地区使用的“iso8859-1”:

File systems --->
DOS/FAT/NT Filesystems --->
VFAT (Windows-95) fs support --->
(iso8859-1) Default iocharset for FAT

同时配置该字符集的对应软件模块:

File systems --->
Native language support --->
NLS ISO 8859-1 (Latin 1; Western European Languages)

⚠ 警告：只要是 VFAT 文件系统，就必须根据实际情况配置输入页码号和字符集。

9.4.5 NFS 网络文件系统

9.4.5.1 功能描述

网络文件系统，英文 Network File System(NFS)。是由 SUN 公司研制的 UNIX 表示层协议 (presentation layer protocol)，能使使用者访问网络上别处的文件就像在使用自己的计算机一样。NFS 是基于 UDP/IP 协议的应用，其实现主要是采用[远程过程调用](#) RPC 机制，RPC 提供了一组与机器、操作系统以及低层传送协议无关的存取远程文件的操作。RPC 采用了 XDR 的支持。XDR 是一种与机器无关的数据描述编码的协议，他以独立与任意机器体系结构的格式对网上传送的数据进行编码和解码，支持在异构系统之间数据的传送。linux 内核也支持 NFS。NFS 特性如下：

- 提供透明文件访问以及文件传输；
- 容易扩充新的资源或软件，不需要改变现有的工作环境；
- 高性能，可灵活配置。

9.4.5.2 使用约束

由于 yaffs 文件格式在对 NFS 支持不够完善，所以当 NFS 服务端使用 yaffs 文件系统时，需要注意：如果出现了服务端重启或者重新挂载的情况，客户端应先卸载并重新挂载后再操作。

由于 ubifs 文件格式在对 NFS 支持不够完善，所以当 NFS 服务端使用 ubifs 文件系统时，需要注意：如果出现了服务端重启或者重新挂载的情况，客户端应先卸载并重新挂载后再操作。

9.4.5.3 内核配置

配置宏 CONFIG_NETWORK_FILESYSTEMS，在 menuconfig 中配置如下：

```
File systems --->
[*] Network File Systems
```

9.4.5.4 运行使用

NFS 详细用法见文档：

ftp://10.75.8.168/ZX-TSP/2010-2011年/OtherTools/nfs-utils/NfsUtil_V1.02.20/nfs-utils用户手册.doc

9.4.6 EXT 文件系统

9.4.6.1 功能介绍

EXT4 是在 EXT2、EXT3 之后的第四代扩展文件系统，支持最大 1EB 的文件系统 (1EB=1024PB=1024*1024TB=1024*1024*1024GB)，相比 EXT3，在功能、性能、可靠性上都有了很大增强，如引入了元块组和 extent 设计、增加日志数据校验、磁盘空间分配算法优化、支持纳秒精度级别的文件访问时间、支持创建无限多个子目录。

EXT4 文件系统在开源 linux 2.6.28 正式合入，也就是 CGEL3.x 还不支持 EXT4，CGEL4.x 是支持 EXT4 文件系统的。

EXT4 在设计时考虑了向前兼容，EXT2 文件系统或 EXT3 文件系统可以平滑升级到 EXT4，并且原来已有的数据还可以正常访问。

EXT4 设计了多种功能，用户可以在 umount 状态下可以重新进行功能选择配置；EXT4 也支持在文件系统访问过程中突然异掉电后，可以在 fsck 修复后正常访问。不过这都需要借助 e2fsprogs 套件中的工具来实施，e2fsprogs 套件的各种工具已经由 TSP 在维护了。

9.4.6.2 应用场景

同 EXT2 和 EXT3 文件系统一样，EXT4 主要用作硬盘文件系统，在初次 mount 块设备分区为 ext4 之前，必须要进行格式化。

9.4.6.3 使用约束

只有 CGEL4.x 支持 EXT4，CGEL3.x 不支持 EXT4。

在异常掉电等非正常 umount 文件系统的情况下，再次 mount 文件系统前，必须用 e2fsprogs 中的 e2fsck 修复文件系统；否则很可能会出现文件系统访问错误，文件系统变成只读。

9.4.6.4 内核配置

必须配置宏 CONFIG_EXT4_FS，其它与 EXT4 相关的配置可选，在 menuconfig 中显示如下：

```
<*> The Extended 4 (ext4) filesystem
[*] Ext4 extended attributes          /*支持用户设置/查询文件扩展属性*/
[] Ext4 POSIX Access Control Lists    /*支持 POSIX 文件标准访问权限控制列表，依赖上面第二个配置*/
[] Ext4 Security Labels               /*利用安全模块的接口去访问文件的扩展属性，依赖于上面第二个配置*/
```

```
[ ] EXT4 debugging support      /*支持输出分配器 mballoc 调试信息*/
[ ] JBD2 (ext4) debugging support /*支持输出日志调试信息*/
```

9.4.6.5 使用方法

注意，本文中多次会提到“块”的概念：它由用户在用 `mke2fs` 格式化 EXT 文件系统时指定，如果不指定默认为 4K 大小，在分配磁盘空间时也是按块为最小单位分配，内核中管理 EXT 文件系统的读写也是按整块对齐来管理。

■ 简单用户使用方法

1) 初次使用，首先用 `e2fsprogs` 套件中的 `mkfs.ext4` 工具进行格式化，如：

```
touch    /etc/mtab          //格式化工具要在 mtab 中检查是否已经 mount 了
mke2fs -t ext4 /dev/sda1
```

2) `mount` 为 EXT4 文件系统，就可以开始访问 `/mnt` 目录了

```
mount -t ext4 /dev/sda1 /mnt
```

3) 如果不是格式化后第一次 `mount`，有可能因为之前的某些情况(如异常掉电)现在文件系统上的数据有错误，`mount` 之前最好用 `e2fsprogs` 套件中的 `e2fsck` 工具进行修复，否则有可能在 `mount` 后访问文件系统时会报错。

```
e2fsck    /dev/sda1
mount -t ext4 /dev/sda1 /mnt
```

■ 高级用户使用

EXT4 设计多种可配功能，可以在格式化时配置或 `mount` 前使用 `e2fsprogs` 套件中的 `tune2fs` 工具进行配置，也可以在 `mount` 时通过“-o”指定多个选项来配置本次 `mount` 后要使用的功能，用户可以根据实际应用需求进行配置。

1) 配置文件系统特色功能

在使用 `mke2fs` 格式化时可以指定“-O 特色功能名称 1, ..., 特色功能名称 n”来配置特色功能。

下表是所有可选特色功能以及 EXT2、EXT3、EXT4 各自的支持情况，**格式化 EXT4 时默认的特性功能选项用加黑标注了**，不同的 `e2fsprogs` 版本可能有变化，下面描述的版本是 1.41.14。

特色功能名称	Ext2	Ext3	Ext4	意义
--------	------	------	------	----

has_journal	无	有	有	使用日志功能
ext_attr	有	有	有	支持文件扩展属性,如支持 acl
resize_inode	无	有	有	用来扩展文件系统大小,前提是在格式化时用户指定了专门用于存放块组描述符的保留块数
dir_index	有	有	有	ext4 使用此特征,才可以创建无限多个子目录
sparse_super	有	有	有	在多个块组中稀疏备份超级块和组描述符,可以节省空间
large_file	有	有	有	支持 2G 以上的大文件
huge_file	无	无	有	支持 128PB 以上的巨型文件
uninit_bg	无	无	有	增加块组描述符校验,增强元数据的可靠性
extra_isize	无	无	有	文件 inode 属性大小需要额外扩展的,在初始化时 EXT4 一个 inode 结构在硬盘占 256 字节
filetype	有	有	有	支持在目录项中包含文件系统类型,可以加快修复文件系统的时间
needs_recovery	无	有	有	用日志修复文件系统功能, mount 完成时设置,正常 umount 时清除,如果异常掉电后 mount 会自动检查此标志并修复文件系统
journal_dev	无	有	有	允许日志保存在其它分区或设备上
extent	有	有	有	使用元块组,每个元块组包含 64 个块组,其中只有块组 0,1,63 包含这 64 个块组描述符的备份,不再备份整个分区其它块组的描述符
meta_bg	无	无	有	使用元块组,可以很轻易的扩展文件系统的大小,文件系统大于 256TB 时,必须使用元块组
64bit	无	无	有	Ext4 的块组描述符为 64 字节大小可以表示更多信息(ext2/3 是 32 字节),inode 结构的 i_file_acl 的大小也是 64 位
flex_bg	无	无	有	支持弹性组,能使文件分配的磁盘空间更连续,访问磁盘效率更高,也减少了磁盘空间碎片
journal_checksum	无	无	有	日志数据校验功能,提高日志数据的可靠性
journal_incompat_revoke	无	无	有	支持撤销过时的日志防止被回滚,ordered、journal 日志模式必须要这个特性

journal_async_commit	无	无	有	提升写日志数据的效率
dir_nlink	无	无	有	没有看出明显用处，可以不用，这里不做介绍

在正常 umount 状态(如果不是正常 umount 需要先用 e2fsck 修复文件系统)下，通过 e2fsprogs 套件中的 tune2fs 工具可以查看/配置 EXT2、EXT3、EXT4 文件系统多个特色功能。

通过 tune2fs 可以查看 ext4 文件系统当前已经配置了哪些特性功能（如果查看失败需要先用 e2fsck 修复文件系统或重新格式化），如：

```
[root@cge1 build]# tune2fs -l /dev/sda1
tune2fs 1.41.12 (17-May-2010)
Filesystem volume name:   <none>
Last mounted on:         /
Filesystem UUID:          dc1c79d2-42e2-4179-8812-5bbc4d3aa3ff
Filesystem magic number:  0xEF53
Filesystem revision #:    1 (dynamic)
Filesystem features:      has_journal ext_attr resize_inode dir_index filetype needs_recovery extent
                          flex_bg sparse_super large_file huge_file uninit_bg dir_nlink extra_isize
```

通过下面的命令格式可以添加部分新的特色配置或撤销部分原来已有的特色配置：

```
tune2fs -O [^]特色功能名称 1, [^]特色功能名称 2, ..., [^]特色功能名称 n /dev/sda1
//若在特色功能名称前加“^”表示撤销此特色功能配置。
```

可以添加的特色功能有：has_journal, dir_index, filetype, extent, flex_bg, large_file, huge_file, extra_isize, uninit_bg, sparse_super, dir_nlink

可以撤销的特色功能有：has_journal, dir_index, resize_inode, filetype, flex_bg, large_file, huge_file, extra_isize, uninit_bg, dir_nlink

2) 配置 mount 选项

这里不介绍从 man mount 看到的多个文件系统都有的 mount 选项，这里只介绍 EXT 文件系统系统特有的 mount 选项，下面所介绍的 mount 选项以 EXT4 文件系统为准，内核中使用的默认选项加黑标注。

mount 选项	互斥选项	选项功能说明
bsddf	minixdf	statfs 系统调用返回的结果中 f_blocks 域，表示用户可用的文件系统空间，不包括文件系统控制数据所占的空间

minixdf	bsddf	statfs 系统调用返回的结果中 f_blocks 域, 表示整个文件系统的空间, 包括用户可用的文件系统空间和文件系统控制数据所占的空间两部分
grpuid	nogrpuid, sysvgroups	新创建文件的 group id 继承父目录的 group id
nogrpuid	grpuid, bsdgroups	如果父目录 setgid 位置位, 新创建文件的 group id 继承父目录的 group id; 否则使用当前进程的 fsgid
bsdgroups	nogrpuid, sysvgroups	同 grpuid
sysvgroups	grpuid, bsdgroups	同 nogrpuid
resgid =%u	无	指定可以使用格式化时保留的空间的特权组 id, 默认为 0
resuid =%u	无	指定可以使用格式化时保留的空间的特权用户 id, 默认为 0
sb =%u	无	当主超级块(块编号为 1)被破坏时 mount 会失败, 可以尝试指定其他备份超级块来 mount
errors=continue	errors=panic , errors=remount-ro	当文件系统发生可恢复错误时, 文件系统系统什么也不做, 就当没有发生错误一样继续运行
errors=panic	errors=continue , errors=remount-ro	当文件系统发生错误时, 内核直接死掉
errors=remount-ro	errors=continue , errors=panic	当文件系统发生错误时, 将文件系统系统 mount 为只读
noid32	无	不使用 32 位的 user id 和 group id, 为了与使用 16 位 user id 和 group id 的老内核兼容
debug	无	在 mount/remount 时打印一些调试信息
oldalloc	orlov	老的分配目录文件的方法
orlov	oldalloc	新的分配目录文件的方法, 在老的分配方法的基础

		上，优先选择在父目录所在的组分配
user_xattr	nouser_xattr	让用户设置/查询文件的扩展属性；编译内核前需要配置 CONFIG_EXT4_FS_XATTR
nouser_xattr	user_xattr	不让用户设置/查询文件的扩展属性；编译内核前需要配置 CONFIG_EXT4_FS_XATTR
acl	noacl	支持 POSIX 文件标准的访问权限控制列表；编译内核前需要配置 CONFIG_EXT4_FS_POSIX_ACL 和 CONFIG_EXT4_FS_XATTR
noacl	acl	不支持 POSIX 文件标准的访问权限控制列表；编译内核前需要配置 CONFIG_EXT4_FS_POSIX_ACL 和 CONFIG_EXT4_FS_XATTR
noload	无	不使用日志功能
norecovery	无	同 noload
nobh	bh	不在页缓存中使用块缓冲区 buffer_head，在读写硬盘时才用块缓冲区
bh	nobh	在页缓存中使用块缓冲区 buffer_head
commit=%u	无	设置至少间隔多少秒同步一次脏文件数据和脏元数据；默认是 5 秒
min_batch_time=%u	无	当 fsync 脏数据到日志的线程不是上次同一个线程时，睡眠一会等待其它线程提交更多脏数据，然后批量同步脏数据，提升写日志的效率。该选项指定最小的睡眠等待时间，单位微秒，默认为 0
max_batch_time=%u	无	同上，该选项指定最大的睡眠等待时间，单位微秒，默认为 15000
journal=update	无	mount 加载日志时，更新老的日志格式 V1 到新的日志格式 V2
journal_dev=%u	无	指定日志存放的设备号(包括主设备号和次设备号)

journal_checksum	无	在日志中记录写入日志空间的元数据的校验结果；提高日志数据的可靠性
journal_async_commi t	无	事务完成记录块提交可以在元数据写入到日志完成之前进行，可以提高写日志的效率
abort		用户 remount 时使用此选项，主动去设置了文件系统错误标记，并且文件系统变为只读
data=journal	data=writeback data=ordered	， 文件元数据和数据都写入日志
data=ordered	data=writeback data=journal	， 只有文件元数据才写入日志，不过文件数据写入磁盘目标位置在文件元数据写入日志前进行
data=writeback	data=journal data=ordered	， 只有文件元数据才写入日志
data_err=abort	data_err=ignore	一旦在日志线程中数据写到原磁盘目标位置失败，整个日志标记为错误，再不能向日志写错误。
data_err=ignore	data_err=abort	在日志线程中数据写到原磁盘目标位置失败，还是继续往日志写数据，当没有发生错误一样
usrjquota	usrjquota=%s	关闭使用用户配额文件
usrjquota=%s	usrjquota	指定用户配额文件
grpjquota	grpjquota=%s	关闭使用组配额文件
grpjquota=%s	grpjquota	指定组配额文件
jqfmt=vfsold	jqfmt=vfsv0	设置配额文件格式为老格式
jqfmt=vfsv0	jqfmt=vfsold	设置配额文件格式为新格式
grpquota	noquota	使用组配额
noquota	grpquota , quota ,	不使配额

	usrquota	
quota	noquota	同 qutoa
usrquota	noquota	使用用户配额
barrier=%u	见后面	等于 1 时同 barrier，等于 0 时同 nobarrier
barrier	nobarrier	提交日志事务完成记录块时带 barrier 标记，既可以保证之前先提交的日志数据都先写到日志上，也可以保证此次事务的日志都是写到存储介质上的
nobarrier	barrier	不保证日志数据在存储介质上的的写入顺序，也不保证日志数据是写入到存储介质上，如果突然掉电这些日志数据会丢失，进而导致文件系统数据或元数据丢失。
i_version	无	文件被修改后 inode->i_version++; 此域用在检查目录文件是否修改以便即时更新
stripe=%u	无	块分配器分配磁盘空间时按此指定大小对齐，对于建立在 RAID5 或 RAID6 之上的文件系统可以提高访问效率；默认为块设备的条带宽（block 数）
resize	无	remount 时指定希望扩展成包含多少块的文件系统
delallo	nodelallo	不象 ext3 那样从用户态写入页缓存时立即分配磁盘空间，而是等到从页缓存写入磁盘空间才分配，可以提高分配空间的效率，也可以增加分配到大块连续空间的概率，提高以后访问此文件的效率
nodelallo	delalloc	象 ext3 那样从用户态写入页缓存时就立即分配磁盘空间
block_validity	noblock_validity	将磁盘元数据区专门用红黑树组织起来，在文件访问过程中动态检查文件数据块如果在磁盘元数据区，就报文件系统发生了错误
noblock_validity	block_validity	不使用上面的功能

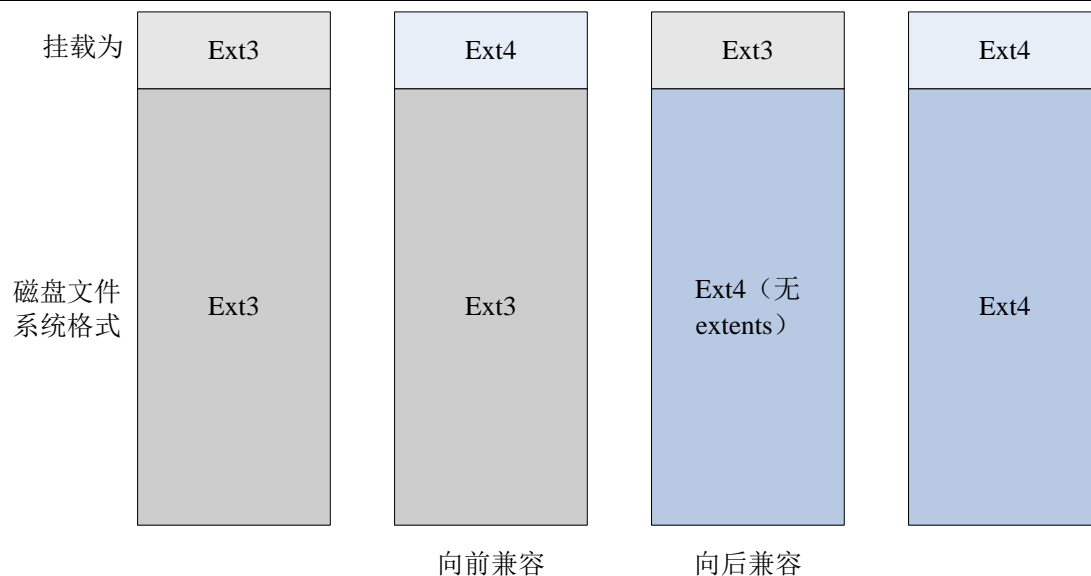
inode_readahead_blks	无	在磁盘中从 inode 表中读取一个 inode 结构时多读一些挨着的 inode 结构，对于连续访问多个文件的情况可以提升效率；此选项默认读 32 个块
journal_ioprio=%u	无	指定日志数据在 IO 排队时的优先级，默认是 3，比其它 IO 数据的优先级高些
auto_da_alloc=%u	见后面	等于 1 时同 auto_da_alloc，等于 0 时同 noauto_da_alloc
auto_da_alloc	noauto_da_alloc	在使用延迟分配空间的情况下，保证某些文件操作结果的一致性得到保证。如有进程这样操作文件 open-write-close-rename，在 rename 结束前保证所有数据写到日志或原磁盘目标位置，防止异常掉电后文件内容还是空的。
noauto_da_alloc	auto_da_alloc	不使用上面的功能
discard	nodiscard	当释放块设备空间时，发送 discard 请求给块设备让他们“忘记”这些空间的内容，这对某些块设备来说是有好处的
nodiscard	discard	不使用上面的功能

■ EXT 文件系统向高版本升级

1) 兼容性

Ext4 向下兼容于 Ext3 与 Ext2，因此可以将 Ext3 和 Ext2 的文件系统升级后挂载为 Ext4 分区。由于某些 Ext4 的新功能可以直接运用在 Ext3 和 Ext2 上，直接挂载即可提升少许效能。

Ext3 文件系统可以部分向上兼容于 Ext4（也就是说 Ext4 文件系统可以被挂载为 Ext3 分区）。然而若是使用到 extent 技术的 Ext4 将无法被挂载为 Ext3 或 Ext2。如下图所示。



Ext3 文件系统都可以升级到 Ext4 文件系统，只需要在只读模式下运行几条命令即可。这就意味着你完全可以不格式化硬盘、不重装操作系统、不重装软件环境，就能够顺利的升级到 Ext4 文件系统。这种升级方法不会损害到你硬盘上的数据和资料，因为 Ext4 仅会在新的数据上使用，而基本不会改动原有数据。后面介绍升级方法。

2) Ext2 向 ext3 升级

升级前提：文件系统处在 `umount` 状态下，如果是异常掉电等其它原因破坏了文件系统，一定要用 `e2fsck` 进行修复了再升级。

只需要执行下面这个命令，其它的都不用做，`/dev/sda1` 上的文件系统就由原来的 `ext2` 变成 `ext3`。

升级后原来的文件数据还保持原样。

```
# tune2fs -j /dev/sda1
tune2fs 1.24a (02-Sep-2001)
Creating journal inode: done
This filesystem will be automatically checked every 31 mounts or
180 days, whichever comes first. Use tune2fs -c or -i to override.
```

3) EXT2 或 EXT3 向 EXT4 升级

升级前提：文件系统处在 `umount` 状态下，如果是异常掉电等其它原因破坏了文件系统，一定要用 `e2fsck` 进行修复了再升级。

Ext2 可以不经 Ext3 直接升级到 Ext4。

Ext2 或 Ext3 向 EXT4 升级只需要执行下面的指令，其它的都不用做：

```
tune2fs -O extents,uninit_bg,dir_index /dev/sda1
```

升级后原来的文件数据还保持原样。

9.4.6.6 接口说明

查看或修改 EXT4 文件系统的相关功能、状态，需要借助 e2fsprogs 套件中多种很有用的工具：如 mke2fs、e2fsck、tune2fs、dumpe2fs、e4defrag、badblocks、debugfs 等。

9.4.7 UBI fastmap 功能

9.4.7.1 功能介绍

UBI/UBIFS 文件系统在加载过程中需要建立逻辑擦除块和物理擦除块的对应关系。目前需要扫描所有的擦除块才能建立此对应关系，扫描时间与 flash 大小线性相关。

为了提高 UBI/UBIFS 文件系统的加载速度，UBI 提供 fastmap 机制。该机制能加快 UBI 的初始化，使得 UBI 的初始化时间是基本为确定值，和 flash 大小无关。

9.4.7.2 应用场景

由于产品线业务模型越来越复杂/flash 价格逐渐下降等因素，产品线会选择更大容量的 flash，原有的 UBI/UBIFS 文件系统扫描速度与 flash 大小线性相关影响文件系统加载速度。

对于使用大容量 flash 的场景，CGEL4.x 内核针对 UBI 提供 fastmap 机制，启用 fastmap 之后，在 UBI 加载过程中不需要扫描所有的擦除块就可以建立逻辑擦除块和物理擦除块的关系并完成 UBI 初始化，这样就能提高文件系统加载速度。

9.4.7.3 使用约束

内核需要开启 fastmap 对应选择，同时要求 flash 要有足够空间，对应特别小的 flash，比如小于 20M，fastmap 功能会自动关闭。

9.4.7.4 内核配置

配置宏选项：CONFIG_MTD_UBI_FASTMAP，在 menuconfig 中配置如下：

```
Device Drivers --->
  Memory Technology Device (MTD) support --->
```

```
Enable UBI - Unsorted block images--->
[*]UBI Fastmap ( Experimental feature)
```

9.4.7.5 使用方法

以前的 UBI 是不支持 fastmap 的，新版本的 fastmap 可以支持将老的 UBI 格式转换为支持 fastmap 格式的。使用配置 fastmap 的内核版本，然后在 attach UBI 之前开启下面的内核模块参数即可：

```
echo Y > /sys/module/ubi/parameters/fm_autoconvert
```

注意，这个模块参数需要先挂着 sysfs 文件系统，挂载方式如下：

```
mount -t sysfs sysfs /sys
```

另外需要说明一点，只有开启模块参数，并且正常 ubi attach 的 dettch 之后才会生成 fastmap。

9.5 内存管理增强

9.5.1 内存紧张时流程改造

9.5.1.1 功能描述

标准 Linux 在出现内存耗尽情况时，会触发 OOM 流程，在该流程中，内核根据任务运行时间、内存占用情况等，选择杀死一个合适的进程，以释放该进程占用的内存。这种处理方式不适合嵌入式处理系统：

- 1) 嵌入式处理系统不会打开磁盘交换功能，在内存不足时不能通过磁盘交换获得足够的内存。这样，系统比较容易进入 OOM 流程。
- 2) 嵌入式处理系统中的进程不允许随意杀死，否则将影响系统性能、可靠性等诸多方面。
- 3) 在内存不足时，嵌入式系统中的控制任务希望得到及时的通知，由该任务自行决定释放可用内存或者复位系统。

为符合嵌入式系统的需求，CGEL 对内存紧张时的流程进行改造，改造包括如下各方面：

- 1) 提供关键任务设置功能，关键任务内存分配水准低于普通任务，以便尽量保证管理监控类任务能够在内存紧张时仍能分配到内存；
- 2) 增加水准级次，以便根据产生内存分配的执行路径，精细控制水准下内存的分配，例如关键任务与中断具有不同的水准；
- 3) 内存不足时，不再进入 OOM 流程，而是通过信号通知应用程序（并附带分配失败原因等信息），

由应用程序自行决定处理方式；

- 4) 避免内核执行路径陷入无限内存回收循环

9.5.1.2 应用场景

当内存不足时，需要了解引起内存不足的原因并及时进行相关处理。以及在配置 NUMA 后，如何应对内存不足的情况。

9.5.1.3 内核配置

CGEL3.X:

配置宏选项 CONFIG_MAPPING_ALLOC

在 menuconfig 中配置如下：

```
Allocate pages at brk/mmap--->
[*]Enable allocating pages when process address mapped
```

CGEL4.X:

配置宏选项 CONFIG_OOM_EMBEDDED_PROCEDURE。

在 menuconfig 中配置如下：

```
Embedded processing procedure when OOM--->
[*]Enable embedded processing procedure when out of memory
```

9.5.1.4 运行使用

在内存紧张时，内核发送信号通知上层应用（SIGURG），这样就给予了上层应用对“系统内存紧张”这个事件的“知情权”，应用程序接收到这个信号之后，可以进行内存的释放、系统状态的记录、各进程内存占用情况的记录等等。

■ 设置关键任务

```
echo -17 > /proc/<pid>/oom_adj
```

设置该线程为关键任务。

■ 设置非关键任务内存不足时最大重试次数


```
echo N > /proc/sys/vm/fail_retry_time
```

设置非关键任务内存不足时尝试分配内存的最大重试次数,将需要重试的次数 N 写入 fail_retry_time。系统默认为 5 次。

■ 设置非关键任务睡眠时间

```
echo 10 > /proc/sys/vm/oom_wait_ms
```

设置非关键任务在尝试第二次内存分配之前等待的时间,单位为毫秒,默认值为 10ms。

■ 设置处理内存不足信号的进程 ID

```
echo ID > /proc/sys/vm/oom_handleurg_process
```

用以设置内核向哪个进程发送内存不足的信号,如果 ID 为 0,则内核此时向所有进程发送此信号。如果非 0,就向此 ID 的进程发送此信号。默认值为 0。

■ 设置三级水线分成比率

```
echo a b c d > /proc/sys/vm/watermarks_alloc_ratio
```

设置内核三级水线 ALLOC_HARDER、ALLOC_HARDEST 和 ALLOC_HIGH 的分成比例。

■ siginfo 提供用户态数据

用户态定义一个如下的结构体变量,并设置类型为 SA_SIGINFO,在信号处理函数 sig_proc()里就可以获取内核传递到用户态的 siginfo 信息,如下:

```
struct sigaction action;  
sigemptyset(&action.sa_mask);  
action.sa_sigaction = sig_proc;  
action.sa_flags = SA_SIGINFO;
```

sig_proc()函数原型如下,第二个参数即为 siginfo

```
static void sig_proc(int signo, siginfo_t *info, void *pvoid)
```

输出结果查看:

分配类型占用 3 位,使用 si_code 字段的第 28~30 位; gfp_mask 占用 25 位,使用 si_code 字段的低 25 位(预留位域用于若内核分配标志的扩展); si_code 字的最高位为 1。


➤ siginfo_t.si_code & ALLOC_KIND_MASK: 取出内存不足时分配失败的分配类型

- `siginfo_t.si_code & GFP_MASK`: 取出本次分配的分配标志
- `siginfo_t.si_pid`: 取出发送信号的进程 pid
- `siginfo_t.si_int`: 取出内核执行路径

9.5.2 内存即时映射

9.5.2.1 功能描述

应用程序申请内存（如调用 `malloc`）时，内核并不会直接为应用程序分配物理内存，而是为其分配一段虚拟地址区间，直到应用程序访问虚拟内存时，内核才会为其分配实际的物理内存，并建立映射。应用程序通过 `malloc` 分配内存返回成功，但在访问虚拟地址区间的时候，可能由于物理内存耗尽而分配不出内存，出现 `oom-kill` 错误。为了使得内存分配更加可控，满足嵌入式实时系统对系统确定性的要求，CGEL 支持物理内存的即时映射功能，也即在分配虚拟内存的时候立即分配物理内存并建立映射关系，如果物理页面不足则分配虚拟内存时立即返回失败，避免将内存不够的错误延迟到内存的使用阶段（发生缺页异常时）而导致系统的不确定性。

 **提示：**即时映射没有对 `fork` 流程进行即时映射，而是在 `exec` 流程中实现

9.5.2.2 应用场景

系统给应用程序分配内存时，在内存紧张的情况下，物理页面分配可能失败而出现 `oom-kill` 错误，为使得内存分配更加可控和可预测，使用此功能模块来实现在分配虚拟内存时立即建立映射。

9.5.2.3 内核配置

配置宏选项 `CONFIG_MAPPING_ALLOC`。

在 `menuconfig` 中配置如下：

```
Allocate pages at brk/mmap--->
[*]Enable allocating pages when process address mapped
```

9.5.2.4 运行使用

增加启动参数 “mapping_alloc=”，当值为 1 时代表系统开启即时映射，值为 0 时代表系统关闭即时映射，默认值为 1。

这样当配置了即时映射配置宏后，仍然可以通过调整启动参数控制是否开启即时映射功能，无需重新编译内核。

注意：

madvise 系统调用是用于提醒内核一个线性区的用途，例如标志线性区为正常、随机访问、顺序访问、近期不会访问等。内核将会根据 madvise 的标志进行相应的文件预读或者内存删除操作。但由于即时映射页面是需要锁住内存，不允许内存被删除或者换出，因此在此页面设置线性区的用途为 MADV_REMOVE 或 MADV_DONTNEED 标志都不会生效。

9.5.3 虚拟地址与物理地址映射关系查询

9.5.3.1 功能描述

通过出错的虚拟地址—>物理地址—>内存条（通道位置）的对应关系可以发现缺陷内存。因而，提供的虚拟地址与物理地址映射关系查询，有助于定位存在缺陷的内存设备。

同时，还可以根据指定的物理地址查到该地址的使用情况，在调试时便于与数据断点等功能配套使用，从而监控此物理地址相关的进程及其虚拟地址。

9.5.3.2 应用场景

在识别有缺陷的内存条时，需要定位存在缺陷的内存设备。定位存在缺陷的内存设备时，需使用虚拟地址与物理地址映射关系查询来得到物理地址与用户态虚拟地址的对应关系。

CGEL 在应用数据断点时，需要内核提供根据物理地址反查对应的进程以及进程的虚拟地址功能。

9.5.3.3 内核配置

虚拟地址与物理地址映射关系查询功能需配置宏选项 CONFIG_VIRADDR_TO_PHYADDR。CGEL3.X 默认编入内核。

CGEL4.X 在 menuconfig 中配置如下：

```
Translate VirAddr to PhyAddr --->
[*]Translate virtual address of user mode to physical address
```

根据物理地址反查相关进程以及进程的虚拟地址的功能需配置宏选项：

CONFIG_PHYADDR_TO_VIRADDR, CONFIG_MMU。CGEL3.X 默认编入内核。

CGEL4.X 在 menuconfig 中配置如下：

```
Translate PhyAddr to VirAddr--->
[*]Translate physical address to virtual address of user mode
```

9.5.3.4 运行使用

■ 虚拟地址与物理地址映射关系查询

将待查询的虚拟地址写入所在进程的 vir2phy 文件

```
echo va > /proc/<pid>/vir2phy
```

读出该虚拟地址对应的物理地址

```
cat /proc/<pid>/vir2phy
```

■ 根据物理地址反查对应的进程以及进程的虚拟地址

将需要查询的物理地址写入所在进程的 pageinfo 文件

```
echo pa > /proc/pageinfo
```

读出该物理地址对应的虚拟地址与 pid

```
cat /proc/pageinfo
```

9.5.4 虚拟地址与物理内存分配分离

9.5.4.1 功能描述

虚拟内存管理中，标准 Linux 对于虚拟地址与物理内存的分配，是采用先分配虚拟地址空间，然后在访问时以缺页方式、以页为单位为该虚拟地址映射物理内存。由于 CGEL 提供内存即时映射功能，这使得虚拟地址空间的分配与物理内存的映射对于用户而言是同步的，但在面对某些应用场景时，需要在两者间进行折中，从而提出虚拟地址空间分配与物理内存映射分离的功能需求。

9.5.4.2 应用场景

应用中内存数据库，在创建库时指定一个当前容量，同时指定一个最大允许扩容容量，比如当前容量 2G，最大允许扩容容量 8G，则在申请时一次性占用 8G 的连续地址空间，但是只对前面 2G 提交物理内存，需要扩容时，只要物理内存足够、且扩容不超过 8G，那么数据库内存还是连续的。

9.5.4.3 内核配置

配置宏选项 CONFIG_PHY_MMA。

在 menuconfig 中配置如下：

```
Uniform System call--->
  Enable uniform system call framwork--->
    [*]Allocation or release physical memory without interfering virtu
```

9.5.4.4 运行使用

使用 syscall 接口来进行物理内存的分配和释放。

■ 不影响虚拟地址空间的前提下分配物理内存的函数

声明：

```
long syscall (int NR_UNIFY_SYSCALL, unsigned int cmd, unsigned long addr,
              unsigned long len)
```

参数：

- cmd，代表具体的子功能号，此处对应 SYSCALL_EMMAP
- addr，待分配物理内存的线性地址起点
- len，待分配物理内存的长度

返回值： 成功返回线性区的起始地址，失败返回-1。

■ 不影响虚拟地址空间的前提下释放物理内存的函数


声明：

```
long syscall (int NR_UNIFY_SYSCALL, unsigned int cmd, unsigned long addr,
              unsigned long len)
```

参数:

- cmd, 代表具体的子功能号, 此处对应 SYSCALL_EMUNMAP
- addr, 待删除物理内存的线性地址起点
- len, 待删除物理内存的长度

返回值: 成功返回线性区的起始地址, 失败返回-1。

 提示: 以上两个函数执行的前提条件是不影响虚拟地址空间, 可通过事先分配一段虚拟地址, 然后在这段地址空间执行函数来保证。

9.5.5 页缓存限制

9.5.5.1 功能描述

标准 Linux 页缓存 (page cache) 机制, 使用贪婪算法, 会无限度申请并耗尽空闲的物理内存, 在一些场景下不满足嵌入式实时系统的确定性要求。CGEL 进行改造, 支持对页缓存大小的限制功能。

9.5.5.2 应用场景

为防止页缓存增长过多而导致内存耗尽的问题, 使用页缓存限制功能来限制页缓存在内存中的比例, 从而避免此问题。

9.5.5.3 内核配置

配置宏选项:

CONFIG_PAGECACHE_LIMIT

CONFIG_PAGECACHE_RATIO

在 menuconfig 中配置如下:


```
Page cache limit configuration--->
[*]Enable control the size of page cache
```

9.5.5.4 运行使用

写入参数以修改页缓存占总物理内存的比例

```
echo pagecache_ratio > /proc/sys/vm/pagecache_ratio
```

设置页缓存占总物理内存的比例，取值为：5 <= pagecache_ratio <= 100。

 提示：在多核情况下，为优化内核性能，减少更新 atomic 变量带来的大量 cache miss，在更新 zone

结构的页缓存统计计数时，以 pcp 统计阈值作为页缓存限制触发点，从而可能出现页缓存超出限制

值若干页面的状况，但并不会对页缓存功能带来影响，请用户知晓。

9.5.6 代码段、数据段巨页映射

9.5.6.1 功能描述

当应用程序对共享内存频繁访问后，内核会产生大量的 tlb miss 异常，系统性能无法满足需求。由于应用程序的代码段、数据段会造成 tlb miss 异常，因此为进一步提升系统性能，CGEL 实现了应用程序代码段、数据段巨页映射功能。目前，我们仅针对特定处理器（mips64 xlr732、mips64 xlp832）实现了该功能。

最初代码段和数据段的大小限制为 32M，内核实现了“32M 代码数据段巨页映射功能”。但之后在使用过程中发现，随着平台版本代码的增加，代码段已经超过了 32M 的限制，需要内核实现支持“64M 代码数据段巨页映射功能”。为了不影响原有 32M 功能，需要做到 32M&64M 代码数据段巨页映射功能兼容。

9.5.6.2 应用场景

CPU 处理器的 tlb 条目数较少，业务关键处理流程的执行存在大量的 tlb miss，导致业务性能大幅降低。

9.5.6.3 内核配置

配置宏选项：CONFIG_HUGETLB_ELF。

在 menuconfig 中配置如下：

```
Hugetlb for code segment and data segment--->
[*]Enable hugetlb for code segment and data segment
```

9.5.6.4 运行使用

配置内核后，应用程序必须使用 TSP 提供的 32M 对齐或 64M 对齐的链接脚本进行链接，且应用程序需要静态编译。应用程序使用 32M 对齐链接脚本进行链接，则内核会启用 32M 代码数据段巨页功能；应用程序使用 64M 对齐链接脚本进行链接，则内核会启用 64M 代码数据段巨页功能。

9.5.7 为用户态进程保留地址空间

9.5.7.1 功能描述

CGEL4.x 在加载业务应用时，提供了预留用户态虚拟地址空间的功能。应用进程将共享内存映射到虚拟地址空间，使得多进程能够使用相同的虚拟地址对此共享内存进行访问，从而降低了编程的复杂度。

9.5.7.2 应用场景

用于多进程共享访问同一块内存区域，且要求此被访问的共享内存区域的虚拟地址一样的场景。

9.5.7.3 内核配置

配置宏选项：CONFIG_RESERV_VMA_SPACE。

在 menuconfig 中配置如下：

```
reserve the vma space--->
[*]reserve the vma space
```

9.5.7.4 运行使用

配置内核后，添加进程启动参数-resvV:[size]@[base]。

- size：需保留地址的长度；
- base：需保留的起始地址。

备注：size 和 base 需要页对齐。

9.5.8 即时分配巨页功能

9.5.8.1 功能描述

开源内核提供的巨页功能一般是通过文件映射的方式访问，当用户挂载巨页文件系统后，并在巨页内存池中预留出一定数目的巨页后，便可以通过在巨页文件系统下创建巨页文件并调用 `mmap` 进行内存映射使用巨页。

`mmap` 映射巨页的方式可以分为：`MAP_SHARED` 和 `MAP_PRIVATE`，当用户以其中的一种标志映射巨页时，内核会比较此次 `mmap` 的巨页大小和已经预留出的巨页内存池中的巨页大小，如果巨页内存池中的巨页数目不能满足分配需求，则 `mmap` 立即返回失败，避免用户进程陷入到缺页异常中去获取巨页失败而导致被 OOM 杀死。本功能提供的即时分配巨页接口对 `MAP_SHARED` 和 `MAP_PRIVATE` 方式映射巨页均适用。

9.5.8.2 应用场景

■ 用户需要使用巨页来保存数据库、网络报文等所需内存空间较大的业务

在 Linux 操作系统上运行内存需求量较大的应用程序时，由于其采用的默认页面大小为 4KB，因而将会产生较多 TLB Miss 和缺页中断，从而大大影响应用程序的性能。例如当操作系统以 2MB 甚至更大作为分页的单位时，将会大大减少 TLB Miss 和缺页中断的数量，显著提高应用程序的性能。

■ 用户映射巨页的方式分为 `MAP_SHARED` 和 `MAP_PRIVATE`

用户在不同的场景需要以不同的标志映射巨页文件，如 `MAP_SHARED` 共享方式和 `MAP_PRIVATE` 私有方式。内核对这两种方式映射巨页内存采取相同的即时分配策略。

9.5.8.3 使用约束

- 1) 内核支持巨页文件系统；
- 2) 目前不支持的架构有 arm、mips32、mips64-xlp 系列。

9.5.8.4 内核配置

配置宏 `CONFIG_HUGETLBFS`，在 `menuconfig` 中配置如下：

```
File systems
-> Pseudo filesystems
```

```
[*] HugeTLB file system support
```

如果架构支持巨页，该配置默认打开。

9.5.8.5 运行使用

在配置了巨页功能的内核启动后，首先需要 mount hugetlbfs 文件系统：

```
mount -t hugetlbfs none /mnt/huge
```

此后，只要是在 /mnt/huge/ 目录下创建的文件，将其映射到内存中时都会使用默认的支持的巨页大小作为分页的基本单位。hugetlbfs 中的文件不支持读、写系统调用（如 read()或 write()等），一般对它的访问都是以内存映射的形式进行。

9.5.8.6 接口说明

■ 设置巨页内存池动态增长巨页的数目

功能：设置巨页内存池动态增长巨页的数目，默认为 0，不能动态增长。

接口：

```
/proc/sys/vm/nr_overcommit_hugepages
```

用法：

```
echo N > /proc/sys/vm/ nr_overcommit_hugepages //设置动态增长巨页的数目
```

其中，N 值表示为：

- 0：不能动态增长；
- >0：可动态增长，增长的数目 $\leq N$ 。

■ 查询分配巨页成功和失败的次数

功能：查询从伙伴系统分配巨页成功和失败的次数。

接口：

```
/proc/vmstat
```

其中，输出输出字段 htlb_buddy_alloc_success 与 htlb_buddy_alloc_fail 分别用于表示从伙伴系统分配

巨页成功和失败的次数。

用法：

```
cat /proc/vmstat
```

9.5.9 homecache 功能

9.5.9.1 功能描述

Homecache 功能允许应用开发者指定内存的 cache 分配策略，从而提升内存访问性能。可以指定数据保存的 cache 芯片，指定使用哪个内存控制器，指定使用的 tlb 页的大小。TMC（TILERA 多核 组件）库的 tmc/alloc.h 文件提供了统一的 API，可以通过 tmc_alloc_t 结构的参数来进行控制。根据性能的需要，可以有很多的修改默认策略的方式。

tilera 系统内存的每个物理地址都关联一个 home tile。home tile 是为特殊的物理地址保持和追踪共享一致性设定的 tile 处理器，比如两个 tile 处理器同时缓存相同的物理地址 P，同时这个信息追踪就保存在该物理地址的 home tile 处理器上。

Homecache 主要的 3 种策略：

1) 本地 home 策略

对于本地 homing 方案，一个完整的内存页（64k 或 16M）home 所在本地 tile 上，这个场景下，当访问物理地址 P 在 L2cache 上 miss，则直接向 DDR 内存发送请求获取数据。

这个方案与远程 homing 和 hash home 方案不同：其他的方案，L2miss 后会向其他 tile 发送信息而不是向内存。

本地 homing 具有本地较低的 L2 miss 延时的优点（因为需要的数据都在本地 cache）。但对于本地 homing，你不能使用其他 tiles 上的 cache 作为备用 L3cache，所以 cache 范围大小仅局限在本地的 L2。

下图显示了在本地 home 内存下 L2 读写 miss，直接发送请求到 DDR 内存。

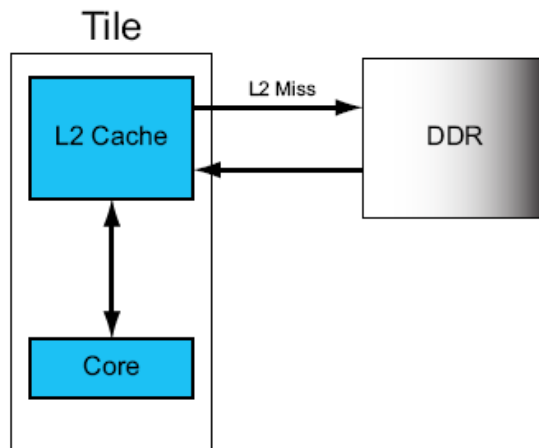


图 9-2 本地 home 策略

2) 远程 homing 方案

远程 homing 方案，访问数据时，一个完整的内存页被 home 在本 tile 以外的其他 tile 之上。在这个场景下，当访问 L2cache miss 时，会申请访问 home tile（也叫 L3），在 home tile 上，相关信息被更新。另外，如果数据存在在 home tile 上，那么请求数据将直接从 home tile 得到。如果数据不存在在 home tile 上，则 home tile 向内核向内存发送请求。

如果数据存在在 home tile，则 L2 miss 访问 home tile 的延时要比访问内存的小。当然，如果数据不存在 home tile，L2 miss 的延时就要比本地 home 的场景要大。在 L2 写 miss 的场景里，数据直接写到 home，更新 home 的值。

除了其他 tile 访问该数据（一个线程的栈空间被其他线程访问），本地访问本地的远程 homing 方式可以被看做同本地 home 方式一样。

主要用在模型如生产消费者模型。比如一个共享内存 fifo，有一个消费者读 fifo，最好将 home fifo 数据设置在消费者 tile 上，通过配置生产者 tile 将数据直接更新到消费者 tile 的 cache。

下图显示了远程 home page 的场景。当 L2 读写 miss，则发送请求到 home tile，数据存在则请求相应，数据不存在则向 ddr 内存发送请求。

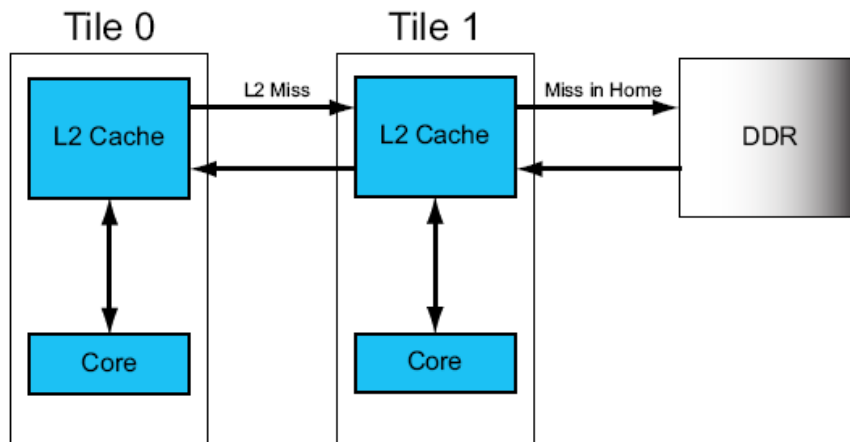


图 9-3 远程 homing 方案

3) Hash-for-home 方法

Hash-for-home 方法很像 homing 方法。然而，远程 home 方法是将一个完整的页映射到一个单独的 home tile，Hash-for-home 将一个完整的页映射到系统的一组 tile 中，以 cacheline 为粒度。将内存分布在各个 tile 上，可以平坦芯片网络得通信量，有效得均衡 tile 得 cache 间数据量。

同其它策略相比，tlb 将虚拟地址转换成物理地址。如果一个页标记上了 Hash-for-home，物理地址的高 bit 被 hash 在一块，并用作序列列表，这个表用来列出合适的 home。

Hash-for-home 是一个不错得策略，为多线程共享的数据与指令内存提供了优秀的性能，也为私有数据提供了合理的性能。Hash-for-home 性能优秀的原因在于它有效的使用完整芯片上聚合的 L2cache 带宽，同时避免了单个 tile 被大量过载访问。

总的来说，理想的设置是 Hash-for-home 用在共享指令和数据内存。

9.5.9.2 应用场景

home tile 机制用在需要通过使用灵活的 L3 缓存提升内存访问性能的场所。

9.5.9.3 使用约束

1) 本地 home 策略

使用：本地 homing 用于本核私有数据，不会被其他核的 cache 作为备用 L3

优点：小的私有数据，本地 homing 可以减少数据在 cache 上的多处拷贝而带来的 cache 占用。同时对于那些命中 cache 的数据，本地 home 方式提供了低时延。

缺点：对于高共享数据不是太好，限制了 cache 的范围。常用于私有的栈空间。

2) 远程 homing 方案

使用：主要用在模型如生产消费者模型。比如一个共享内存 fifo，有一个消费者读 fifo，最好将 home fifo 数据设置在消费者 tile 上，通过配置生产者 tile 将数据直接更新到消费者 tile 的 cache。

优点：共享内存 fifo，生产者直接写入消费者的 cache，而不需要污染生产者的 cache。

缺点：适合只有一个读者的共享内存 fifo 模型，常用于可被多线程访问的栈空间。

3) Hash-for-home 方法

使用：多线程共享的数据与指令内存。

优点：通过将内存分散在一组 tile 上，网络冲突得到缓和，避免 L2cache 控制器被其它核的风暴淹没。

缺点：Hash-for-home 不适合用在私有的或者单个读的数据，常用于静态和动态数据和指令内存，以及各种应用。

9.5.9.4 内核配置

内核配置说明：CONFIG_HOMECACHE，缺省已配置。

在 menuconfig 中配置如下：

```
Tilera-specific configuration --->
[*]Support for dynamic home cache management
```

9.5.9.5 使用方法

Tile 处理器允许应用开发者指定内存分配策略来提升应用性能。用户可以指定数据保存的 cache 芯片，指定使用哪个内存控制器，指定使用的 tlb 页的大小。TMC（TILERA 多核 组件）库的 tmc/alloc.h 文件提供了统一的 API，可以通过 tmc_alloc_t 结构的参数来进行控制。

9.5.9.6 接口说明

常用的接口与使用方法：

```
// Allocate memory homed on this task's CPU, and dynamically rehome
// the pages if this task moves to another CPU.
tmc_alloc_t alloc = TMC_ALLOC_INIT;
```

```
tmc_alloc_set_home(&alloc, TMC_ALLOC_HOME_TASK);
p1 = tmc_alloc_map(&alloc, size);
// Allocate 'hash-for-home' memory.
tmc_alloc_set_home(&alloc, TMC_ALLOC_HOME_HASH);
p2 = tmc_alloc_map(&alloc, size);
```

TMC_ALLOC_INIT 用来初始化内存分配器 alloc, tmc_alloc_set_home 用来设置此次分配的 homecache 的策略。常使用策略如下:

```
TMC_ALLOC_HOME_HERE, home cache;           //在当前 cpu
TMC_ALLOC_HOME_INCOHERENT, //各个 tile 认为自己上的 cache 为 home cache, 一般用在只读
TMC_ALLOC_HOME_NONE,           //读写直接通过内存, 没有 home cache
TMC_ALLOC_HOME_SINGLE,         //系统指定一个 cpu 作为 home cache
TMC_ALLOC_HOME_HASH,           //home 的 hash 策略
TMC_ALLOC_HOME_TASK,           // home cache 跟随任务迁移而变化
```

tmc_alloc_map 按照所设置的策略分配 size 大小页的内存。

同样的使用方式用于巨页的 homecache 分配:

```
// Allocate default-sized huge page, 'hash-for-home' memory.
tmc_alloc_t alloc = TMC_ALLOC_INIT;
tmc_alloc_set_huge(&alloc);
tmc_alloc_set_home(&alloc, TMC_ALLOC_HOME_HASH);
p3 = tmc_alloc_map(&alloc, size);
```

9.5.10 hugetlbfs 工具

9.5.10.1 功能描述

CGEL 内核采用基于 hugetlbfs 特殊文件系统的方式来实现对巨页的支持, 应用程序要使用内核的巨页特性, 必须基于该文件系统进行文件的创建和 mmap 映射, 也即需要对源代码进行修改。Libhugetlbfs 改变了这一使用方式, 其提供的库函数接口可为用户应用程序提供系统巨页的透明访问, 而无需进行任何源代码的修改。

libhugetlbfs 是一个开源社区项目, 迄今该库提供了以下三个主要的功能:

- hugepage heap: 堆扩展, 指定让运行时 malloc 调用使用由巨页分配的内存;
- hugepage text/data/bss: 可以指定将所有三个段 (.bss 段、.data 段 和 .text 段) 均加载进巨页

内存。为了指定这些应用程序段由系统巨页备份，所需做的就是重新链接执行体；

- **library functions:** libhugetlbfs 提供的库函数允许通过 hugetlbfs 文件系统来分配和使用巨页。

同时也提供基于 XLP 系列 cpu 的 hugetlb 支持。注意，本 hugetlb 功能不是 CGEL3.x 或者 CGEL4.x 之前开发的代码数据段巨页，或者产品线开发的定制巨页，而是在标准内核 hugetlb 功能基础上的改进。

- XLP 架构提供 2M, 8M, 32M, 128M 和 512M 的标准巨页功能，具体使用方法参考内核说明文档 `documentation/vm/hugetlbpage.txt`。

9.5.10.2 应用场景

在用户需要更方便地使用内核巨页来保存堆、代码段、数据段和 bss 段文件的时候使用。

9.5.10.3 使用约束

1) 内核支持巨页文件系统。

2) 目前不支持的架构有 arm、mips32 系列。

3) XLP 架构的巨页大小只能在 2M, 8M, 32M, 128M 和 512M 里选取，因为这是 mips 硬件决定。不同于 x86_64 或者 powerpc，系统中同时只能存在一种大小的巨页，这也是目前内核代码决定的。注意，对于大于伙伴系统连续物理地址的页（即大于 order 的页），需要用 bootmem 区分配，也就是说，对于一个普通页大小为 16K 的系统，如果需要 512M 的巨页，则需要在内核启动参数添加巨页大小、巨页个数。

4) 如果要使用 libhugetlbfs 库，则应用程序必须动态编译。测试前需获取 C 动态库文件，如果没有的话，可通过成研交叉工具链 bin 目录下的 GetSharelib.sh 来提取。

9.5.10.4 内核配置

Libhugetlbfs 是一个巨页相关的用户态工具，只有在内核配置了以下相关功能后才能正常运行：


配置宏 CONFIG_HUGETLBFS，相关配置说明：

```
Symbol: HUGETLBFS [=n]
  Prompt: HugeTLB file system support
  Defined at fs/Kconfig:1043
  Depends on: X86 [=X86] || IA64 [=IA64] || SPARC64 [=SPARC64] || S390 [=S390] && 64BIT [=y]
  || SYS_SUPPORTS_HUGETLBFS [=y] || BROKEN [=n]
  Location:
    -> File systems
```


-> Pseudo filesystems

如果是 XLP 系列 CPU，则还需要进一步选择巨页大小，相关配置说明：

```
Symbol: HUGE_PAGE_SIZE_8M [=y]
  Prompt: 8MB
  Defined at arch/mips/Kconfig:1771
  Depends on: <choice>
  Location:
    -> Kernel type
        -> Huge page size (<choice> [=y])
```

 提示：配置中 Symbol 为配置宏名，全称加前缀“CONFIG_”；Prompt 为配置项标题；Depends on 为依赖的其他配置项；location 为配置项在配置菜单中的位置。

9.5.10.5 使用方法

■ 编译

libhugetlbfs 包编译会生成以下三类文件：

- hugeadm 和 hugectl 等工具：编译完成后将会放在 ./obj 目录中；
- libhugetlbfs.so、libhugetlbfs_privutils.so 等库：这些库在运行时调用，编译完成后将会根据编译的是 32 位应用程序或是 64 位应用程序放在 ./obj32 或 ./obj64 中；
- 自动测试包：自动测试包是 libhugetlbfs 用于检测系统是否支持巨页功能，这些测试包可通过 make 在编译完成后调用。如果运行的内核支持巨页，则无需进行测试，因为测试输出结果比较晦涩难懂。

下载代码后解压，进入 libhugetlbfs-2.12 代码路径下，执行如下命令：

```
make ARCH=xxx CC32([CC64]=xxx-gcc LD=xxx-ld libs tools (BUILDTYPE=NATIVEONLY)
```

其中，输入各参数说明为：

- ARCH 为 CPU 架构，目前 libhugetlbfs 支持如下架构：
 - ✧ i386

✧ x86_64

✧ ppc //32 位

✧ tilegx

✧ mips64

- CC32 或 CC64 为 CPU 架构所对应的交叉工具链编译器；
- LD 为交叉工具链的连接器；
- libs 表示编译生成 libhugetlbfs 库文件；
- tools 表示编译生成 hugeadm, hugectl 等工具。

注意：

1) 编译 64 位架构时，需要添加 BUILDTYPE=NATIVEONLY，表示只编译 64 位架构，否则会将 32 位架构一并编译，导致在链接时出错；

2) 某些 ppc32 的工具链在链接时会出现 relocation truncated to fit 的错误，原因是代码段大小超过指令跳转距离了，这时需要在出错的编译命令处添加 --relax 链接参数进行自动调整。

■ 安装

make install 会将上一步编译出的工具、库、以及链接器脚本等复制到设定的合适位置，方便后续对应用程序的重链接编译。命令如下：

```
make ARCH=xxx CC32([CC64]=xxx-gcc LD=xxx-ld install PREFIX=/path/to/install/ <LIB64=lib64  
BUILDTYPE=NATIVEONLY> <LIB32=lib32>
```

建议通过 PREFIX=/path/to/install/ 指定安装目录。

编译 64 位应用程序需要在安装路径下建立 lib64 文件夹，并在命令后添加”LIB64=lib64 BUILDTYPE=NATIVEONLY”

编译 32 位应用程序需要在安装路径下建立 lib32 文件夹，并在命令后添加”LIB32=lib32”。

安装完成后，到安装目录查看：

1) \$PREFIX/share/libhugetlbfs/目录，会发现 ld，它是新的链接器命令。ld 命令被软链接到 ld.hugetlbfs 以方便 GCC 编译器调用此链接器。

```
-bash-4.1$ ls -l /CGEL/wangqiang/lib64/share/libhugetlbfs/
total 8
lrwxrwxrwx 1 wangqiang wangqiang 12 Aug 20 10:36 ld -> ld.hugetlbfs
-rwxr-xr-x 1 wangqiang wangqiang 1938 Aug 20 10:36 ld.hugetlbfs
drwxr-xr-x 2 wangqiang wangqiang 4096 Aug 20 10:36 ldscripsts
-bash-4.1$
```

上图所示的 ldscripsts 子目录包含所有修改后的链接器脚本，这些脚本是处理重链接.bss、.data 和.text 段所必需的。目前，libhugetlbfs 支持的链接脚本仅有 ppc32、ppc64、i386 和 x86_64。

- 2) \$PREFIX/bin 目录：存放 libhugetlbfs 库的工具，包含 hugeadm、hugectl 等。
- 3) \$PREFIX/lib64 目录：存放 libhugetlbfs 动态库，包含 libhugetlbfs.so、libhugetlbfs_privutils.so 等。
- 4) \$PREFIX/include 目录：存放 libhugetlbfs 对外提供库函数的头文件 hugetlbfs.h。

■ 使用

将编译生成的目标文件（包括二进制工具以及 libhugetlbfs.so 库）拷贝到单板环境中，单板运行的内核必须支持巨页功能。在程序运行前必须完成巨页文件系统的挂载以及巨页内存预留的工作，具体方法请参考本文档巨页相关章节。

- 1) 堆扩展：

Libhugetlbfs 允许动态链接的二进制可执行文件在调用 malloc()扩展堆时使用巨页页面，二进制可执行文件必须使用动态链接。

```
./hugectl --heap ./your_program
```

使用 hugectl 工具，无需指定环境变量，达到使用巨页保存 malloc 扩展的堆。

- 2) 代码段、数据段和 bss 段

Libhugetlbfs 可以指定将.bss 段、.data 段和.text 段加载进巨页内存中，为了指定这些应用程序段由系统巨页页面备份，所需做的就是重新链接可执行文件。

将如下参数添加到 C 编译链接命令行：

```
-B $PREFIX/share/libhugetlbfs -Wl,--hugetlbfs-align
```

\$PREFIX/share/libhugetlbfs 为 make install 的安装目录，share/libhugetlbfs 是链接器脚本。

 提示：如果编译的时候提示找不到-lhugetlbfs 库，可加上 -L \$PREFIX/libxx 该目录是安装时

libhugetlbfs.so 文件所在的目录

使用 hugectl 工具可以更方便地使用巨页来备份三个段：

```
./hugectl --text --data --bss ./ your_program
```

9.6 信息统计

9.6.1 指定任务的调度信息统计

9.6.1.1 功能描述

CGEL 提供更为细化的调度、阻塞相关信息，以便于快速分析与调度系统相关功能，定位性能问题。例如定位系统处于 `runing` 即就绪态的任务长时间得不到 `cpu` 上执行的机会的原因。这些信息从次数、时间点、持续时间三个角度对各任务调度、阻塞、睡眠等动作或者状态进行信息记录及统计，同时以 `proc` 文件系统方式输出到用户态。

- 任务的总调度次数：包含主动调度次数即睡眠、等待消息、等待信号量等原因，被动调度的次数即高优先级任务抢占次数；
- 任务被抢占时间统计：包含抢占开始时间，抢占最长时间，抢占总时间，抢占进程 `id`；
- 任务阻塞时间统计：包含阻塞开始时间；阻塞最长时间，阻塞总时间，阻塞原因；
- 信号量阻塞时间统计：包含信号量（`futex`）阻塞起始时间，信号量（`futex`）阻塞最大时间，信号量（`futex`）阻塞总时间，信号量（`futex`）当前阻塞时间（如果当前阻塞于此）；
- 消息队列阻塞时间统计：包含消息队列阻塞起始时间，消息队列阻塞最大时间，消息队列阻塞总时间，（如果当前阻塞于此）`mq_receive` 阻塞起始时间，`mq_receive` 阻塞最大时间，`mq_receive` 阻塞总时间，`mq_receive` 当前阻塞总时间（如果当前阻塞于此）；
- 主动睡眠的阻塞时间统计：包含 `sleep` 阻塞起始时间，`sleep` 阻塞最大时间，`sleep` 阻塞总时间，`sleep` 当前阻塞总时间，（如果当前阻塞于此）`select` 阻塞起始时间，`select` 阻塞最大时间，`select` 阻塞总时间，`select` 当前阻塞总时间，（如果当前阻塞于此）
- 任务所阻塞的消息队列 `id` 以及阻塞的信号量显示：包含任务所阻塞的消息队列 `id`，任务阻塞的信号量（`futex`）。

同时，还提供了 `taskinfo` 功能，即在内核中将统计信息写入到一块共享内存中，然后在用户态将其读出，使得用户能够快速获取系统各线程的统计信息。

9.6.1.2 应用场景

调度信息统计功能提供更为细化的调度相关信息，便于快速分析与调度系统相关功能，定位性能问题。


9.6.1.3 内核配置

关于调度信息统计，配置宏选项 `CONFIG_SCHED_DETAILS_DEBUG`。

关于 `taskinfo` 共享调度信息功能，内核需要配置 `CONFIG_TASK_INFO`。

在 `menuconfig` 中配置如下：


```
Kernel hacking--->
  [*]Kernel debugging
  [*]Collect scheduler debugging info
  [*]Collect scheduler details debugging info
Processor type and feature--->
  [*]Export kernel task info to userspace
```

 提示：目前 `taskinfo` 仅支持 Powerpc 和 mips 体系架构。

■ `taskinfo` 的初始化

用户在首次使用 `taskinfo` 模块时需要初始化该模块，使用如下命令可完成 `taskinfo` 模块的初始化。

```
echo tasknum > /proc/task_info
```

 提示：命令中 `tasknum` 是用户传入的需要监控的任务数（十进制）。用户只能初始化一次。

9.6.1.4 运行使用

在配置了调度细节统计功能的内核上执行用户态程序，然后执行 `cat /proc/pid/task/tid/sched` 命令就可以查看用户态程序的详细调度信息了。如图：

```

involuntary sched details
involuntary_preempt_times      :          4
se.involuntary_preempt_start   :      0.000000
se.involuntary_preempt_max     :      0.917135
sum_involuntary_preempt_time   :      2.694369
involuntary_preempt_pid       :          0

voluntary sched details
voluntary_switch_times         :          1
se.voluntary_switch_start      :    434904.674137
se.voluntary_switch_max        :      0.000000
curr_voluntary_switch_time     :      20588.092021
sum_voluntary_switch_time      :      20588.092021
voluntary_switch_reason        : 6 <0:others 1:sleep 2:compslp 3:select 4:futex
                                :msgq 6:mq>

se.voluntary_sleep_start       :      0.000000
se.voluntary_sleep_max         :      0.000000
sum_voluntary_sleep_time       :      0.000000

se.voluntary_compslp_start     :      0.000000
se.voluntary_compslp_max       :      0.000000
sum_voluntary_compslp_time     :      0.000000

se.voluntary_select_start      :      0.000000
se.voluntary_select_max        :      0.000000
sum_voluntary_select_time      :      0.000000

se.voluntary_futex_start       :      0.000000
se.voluntary_futex_max         :      0.000000
sum_voluntary_futex_time       :      0.000000
futex_uaddr                    :          0

se.voluntary_msgq_start        :      0.000000
se.voluntary_msgq_max          :      0.000000
sum_voluntary_msgq_time        :      0.000000
se.msgq_msgqid                 :          0

se.voluntary_mq_start          :    434904.663933
se.voluntary_mq_max            :      0.000000
curr_voluntary_mq_runtime      :      20589.103012
sum_voluntary_mq_time          :      20589.103012

curr_rq_clocktime              :    455493.766945
clock_delta                    :          327
#

```

图 9-4 调度细节统计信息

taskinfo 共享调度信息提供了如下信息的查看：

■ 线程统计信息的数据结构

```


typedef tagMonitorStatEntry
{
    BYTE  isUse;
    BYTE  state;
    WORD16 reserved;

```

```
WORD32  tid;
WORD64  activeScheCnts;
WORD64  enterTime;
WORD64  runTime;
}
```

其中各字段含义如下：


- **issue**：初始化为 0，线程创建时关联一项，置位 1，线程退出时置 0。
- **state**：线程的状态机状态,每次被切换走时要记录在这里
- **reserved**：保留
- **tid**：线程 id，线程创建时写一次即可
- **activeScheCnts**：线程的主动调度次数，每次线程被切走时，如果不是被高优先级的任务或者中断等抢占，说明是主动放弃 cpu，则累加到这个字段
- **enterTime**：线程最近一次调度进入的时刻，每次切换时，内核都要把被调度的新线程进入的时刻记在这里
- **runTime**：线程运行的总时间，每次线程被切换走时，都要更新这个值，把该次调度的时间累加到这里

 **提示**：该数据结构是按照用户的需求编写的。所有时间的单位均是纳秒。

■ 获取起始物理地址

初始化完成后，初始化后，用户可以通过如下命令获取共享内存的起始物理地址：

```
cat /proc/task_info
```

 **提示**：只有在初始化后，这个输出才是有效的。起始物理地址是按照十六进制格式输出的。

这个命令的输出是的一个值起始物理地址，按照十六进制格式输出。获取到起始物理地址后，用户可以通过 mmap 的方式读取该物理地址的内容（打开/dev/mem 设备，然后 mmap）。

对于 POWERPC 架构，输出的第二、三个值是 tb_to_ns_scale 和 tb_to_ns_shift。用户在通过寄存器读

取到 tb 值后，可以通过这两个参数来计算当前时间。详细寄存方法可以参考内核中的 sched_clock 函数（POWERPC）架构。

■ 获取线程的偏移量

需要获取线程的偏移量可使用如下命令：

```
cat /proc/<pid>/task_info_offset
```

用户将偏移量加上映射的起始虚拟地址，就可以获得该线程统计数据结构的位置了。偏移量也是按照十六进制格式输出的。

9.6.2 系统级调度与中断信息记录

9.6.2.1 功能说明

对于 CPU 挂死等导致的看门狗复位问题，目前缺乏定位信息。为了缩小问题范围，便于定位故障，CGEL 提供实时记录进程调度信息和中断信息功能，并将其保存在系统预留的黑匣子共享区域中，供用户在系统重启后访问。当然，系统运行期间也可以对中断信息和调度信息进行查看。

9.6.2.2 应用场景

用于定位分析 CPU 挂死等导致的看门狗复位问题。

9.6.2.3 内核配置

配置宏选项：CONFIG_BLACK_BOX（黑匣子），CONFIG_SCHED_IRQ_RECORD。

在 menuconfig 中配置如下：

```
Kernel hacking --->
  Emergency for System Deadlock--->
    [*]Black box support<NEW>
Schedule-IRQ info record--->
  [*]Schedule-Interrupt info record
  [50]Record times of sched irq info<default 50>
```


9.6.2.4 使用说明

■ 开启实时记录

```
echo 1 > /proc/sys/kernel/sched_irq_record_on
```

由于实时记录线程运行信息和中断信息可能对性能有影响，在配置了功能宏后，另外设置功能的开关接口，用于系统启动后的控制。

■ 设置记录信息的条目数

```
echo 50 > /proc/sys/kernel/sched_irq_record_times
```

记录最近发生的调度和中断次数。内核在记录信息时采用轮转法，在超过配置的次数后将会覆盖最老的记录。

■ 用户获取信息接口

```
#define SYSCALL_SCHED_IRQ_GETINFO SYS_ID(SYS_VM, SYS_SCHED_IRQ_GETINFO)
syscall(nr, SYSCALL_SCHED_IRQ_GETINFO, getstr, len, region, index)
```

输入参数：

- `getstr` 指明预分配保留要获取信息的 buf 的起始地址
- `len` buf 的长度(至少是 4KB，否则返回错误)
- `region` 指明具体调度信息或中断信息存储的区域（0：调度信息，1：中断信息）
- `index` 指明要获取的信息的索引号

返回值：0 成功；非 0 失败。

9.6.3 内存信息统计

9.6.3.1 功能描述

在标准 Linux 内核中，未对内存的各项信息提供一个完整的统计显示。为方便用户对内存信息有一个整体的驾驭，CGEL 内核在 `proc` 文件系统中新增以下六类内存信息统计功能，以便于系统内存相关功能、性能问题的定位：

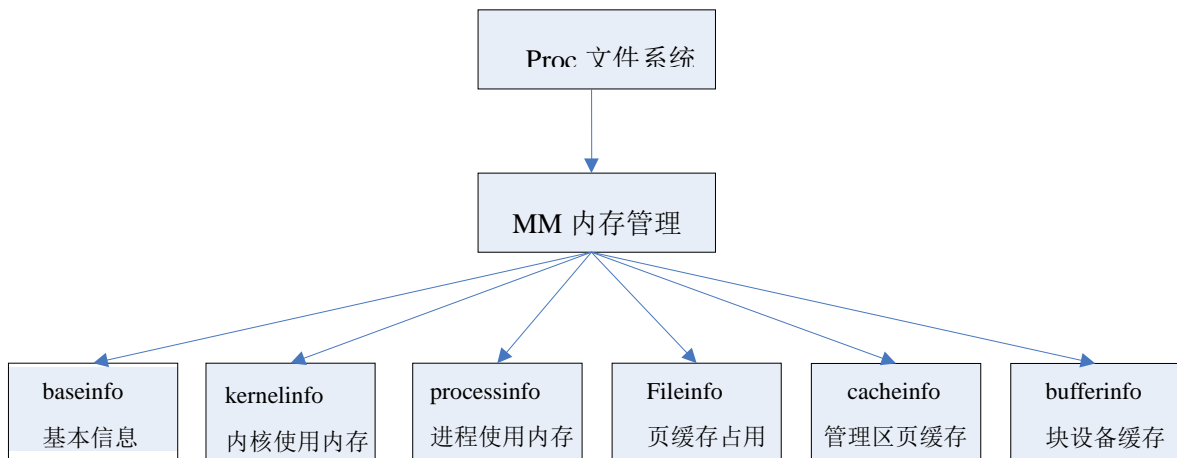


图 9-5 CGEL 新增内存管理功能

嵌入式产品通常需要明确获知系统剩余物理内存的大小，但由于 Linux 内核的内存管理系统非常复杂，且很多模块都采用缓存机制，导致系统当前可用物理内存无法准确统计。为了满足确定性要求，CGEL 进行改造完善，开发了剩余物理内存准确统计功能。

9.6.3.2 应用场景

定位内存相关问题时，需使用内存使用情况的详细统计。此功能模块，从系统、内核、进程等次来显示内存当前使用状况。

9.6.3.3 内核配置

配置：CONFIG_MM_INFO，并且只有 MMU 架构的单板才支持 CONFIG_MMU。

在 cacheinfo 里需要统计 ramfs 信息时需要配置 CONFIG_PAGECACHE_LIMIT。

在 baseinfo 里需要统计可用物理内存时需要配置 CONFIG_EXACT_RECLAIM_ACCOUNT。

可回收物理内存功能配置宏选项 CONFIG_EXACT_RECLAIM_ACCOUNT_V2。

在 menuconfig 中配置如下：

```
#在 cacheinfo 里统计 ramfs 信息
Page cache limit configuration--->
    [*]Enable control the size of page cache
#在 baseinfo 里统计可用物理内存
```

```
reclaim pages--->
    [*]fast account reclaim-able pages<KB>
#可回收物理内存
reclaim pages--->
    [*]fast account reclaim-able pages<KB> version 2
```

9.6.3.4 运行使用

■ 显示内存当前基本信息

包括：物理内存大小；可用物理内存大小；剩余物理内存；高端内存大小；低端内存大小；用户空间占用内存大小；内核空间占用物理内存大小。

■ 显示内核使用内存基本信息

包括：slab 占用内存信息；vmalloc 使用内存情况；buffers 占用内存大小；cache 占用内存大小；swpcached 占用内存大小；保留内存大小。

■ 显示进程内存使用信息

包括：进程占用物理内存大小；进程代码段占用内存大小；进程数据段占用内存大小；进程使用库占用内存大小；进程匿名映射占用内存大小；进程堆栈占用内存大小；进程共享内存大小；进程 IO 映射占用物理内存大小。

■ 显示页缓存内存占用信息（根据所在管理区布局显示）

包括：指定管理区页缓存大小；页缓存中用于交换缓存的大小；被锁定的页缓存大小；处于回写状态的页缓存大小；脏页大小；已与用户进程建立映射关系的页缓存大小；处于活动状态的页缓存大小。

■ 显示页缓存内存占用信息（以文件为单位显示）

包括：占有物理内存的文件名；该文件占用内存实际大小；文件 dentry 使用计数（dentry 已有现成记录）。

■ 显示块设备缓存占用内存信息

包括：块设备名称；块设备占用物理内存大小。

■ 显示 ZONE 信息

包括：管理区被 kswapd 扫描的最近一次开始时间；管理区被 kswapd 扫描的最近一次结束时间；由于管理区内存不足而最近一次唤醒 kswapd 的时间；管理区被 kswapd 扫描的次数；管理区文件映射大小；管理区匿名映射大小。

■ 统计可回收物理内存信息

```
cat /proc/pageinfo
```

显示当前可回收的页缓存数目。

9.6.4 文件系统信息统计

9.6.4.1 功能描述

CGEL 针对 JFFS2 文件系统添加信息统计功能，主要分为以下三个方面：

- 在有 Flash 设备的单板上，CGEL 提供对外接口用于获取 JFFS2 中各种类型的节点数量；
- 统计出 JFFS2 中各种擦除块链表元素个数，从而大致了解 flash 当前垃圾回收的效率
- 分别统计出 very_dirty, dirty, clean 链表中所有擦除块中干净节点（REF_NORMAL）的数量，更准确地了解 flash 当前的回收效率

9.6.4.2 应用场景

经常使用 flash 设备的单板，长时间使用之后，会发现 flash 文件操作变得非常耗时，这可能与 flash 设备中文件系统的节点分布状况有关。所以，在 flash 效率低下时，需要了解 flash 中的节点是否分布比较零乱，以便更好地了解 flash 在各种情况下的操作性能。

9.6.4.3 内核配置

配置宏选项 CONFIG_PROC_FS、CONFIG_JFFS2_FS。在 menuconfig 中配置如下：

```
File system--->
  Pseudo filesystems--->
    [*]/proc file system support
  [*]Miscellaneous filesystems--->
    [*]Journalling Flash File System V2<JFFS2>support
```

9.6.4.4 运行使用

使用 proc 接口查看 jffs2 节点信息：

```
cat /proc/fs/jffs2/mtd0~n/inode_info
```

9.6.5 调度轨迹信息统计

9.6.5.1 功能描述

CGEL 提供记录调度轨迹信息的功能，包括进程切换的各种信息，如切换 PID，切换原因，切换时间，切换调用栈等。参考这些信息，开发人员就可以对单板上电时间、业务性能进行调优。

本功能主要包括两部分，第一部分是调度钩子管理框架，第二部分是非阻塞缓冲区管理。两部分可以分别独立存在。其中，调度钩子框架提供钩子注册、删除、开启、关闭功能，不涉及到具体的钩子函数实现；非阻塞缓冲区提供在苛刻的环境下（如中断上下文），访问管理缓冲区的机制，与调度钩子配合使用时，可以将信息记录到内部区域，并在事后读取。非阻塞缓冲区功能，弥补了现有的保留内存、黑匣子等内存管理区的缺点，即必须在系统启动时划分好区域，相反，非阻塞缓冲区，可以动态的分配，有效的利用了内存空间。

9.6.5.2 应用场景

调度切换钩子，如上节所说，主要用在上电优化，性能调优时；非阻塞缓冲区功能，可以用在任何需要在苛刻环境下记录信息的场景，如中断上下文，进程切换上下文等。

调度切换钩子与非阻塞缓冲区管理这两个功能没有依赖性，可以分别独立存在，提供给外部模块使用。钩子内部实现，不得引发阻塞。具体接口规范见后面的接口说明。

9.6.5.3 内核配置

配置宏选项 CONFIG_SCHED_MONITOR，开启调度切换钩子框架，在 menuconfig 中根目录配置：

```
Schedule Monitor
```

配置宏选项 CONFIG_UNBLOCK_BUF，开启非阻塞缓冲区功能，在 menuconfig 中根目录配置：

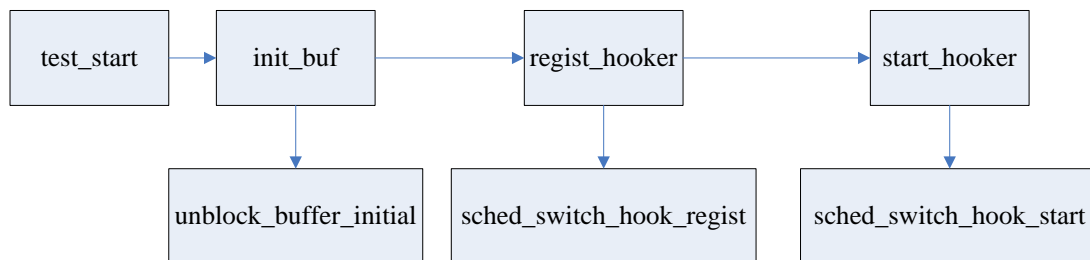
```
Unblock Buffer
```

9.6.5.4 运行使用

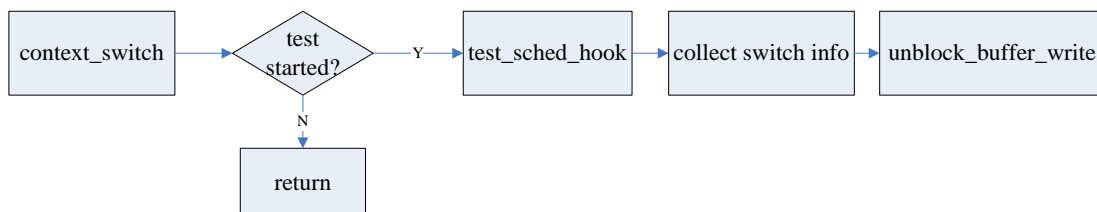
这里以本功能附带的测试 demo 为例说明调度轨迹信息记录的方法，测试 demo 程序位于 CGEL3.x 源码目录的 CGEL3.x\kernel-version\cgel3.0\linux\Documentation\sched-monitor.txt 或 CGEL4.x 源码目录的 CGEL4.x\kernel-version\cgel4.0\linux\Documentation\sched-monitor.txt。

sched-monitor.txt 测试代码流程示意图如下：（横轴为时间轴，竖轴为单个函数调用序列）

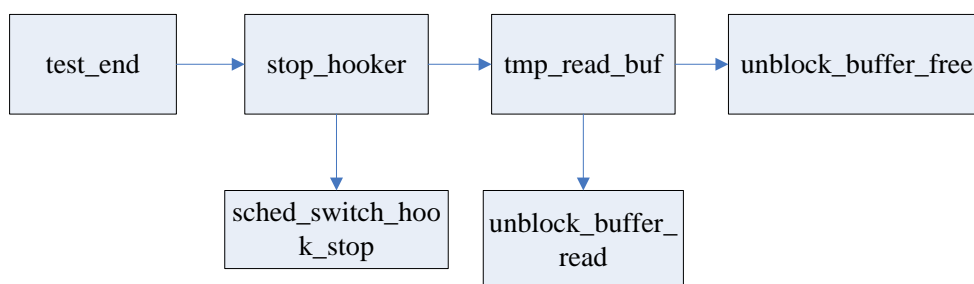
1) 初始化缓冲区，注册回调函数，开启调度监控功能；



2) 进程切换时，根据需要调用注册的回调钩子，在钩子函数里，收集必要信息，并保存到缓冲区内；



3) 结束调度监控功能，将数据从缓冲读出，最后释放缓冲区。



9.6.5.5 外部接口

■ 调度切换钩子框架接口

1) 注册调度切换钩子

结构:

```
typedef void (*sched_hook_func)(struct task_struct* prev,struct task_struct* next);
```

功能: 内核在调度切换时, 外部模块调用本接口注册调度钩子处理函数, 同一时刻系统中只有一个钩子回调生效。

返回值: 无。

2) 删除钩子

```
void sched_switch_hook_delete(void)
```

功能: 外部模块通过此接口, 删除通过 sched_switch_hook_add 注册的回调钩子。

返回值: 无。

3) 开启调度钩子监控

```
typedef struct sched_cmd{  
    cpumask_t cpu_sched_map;  
}sched_cmd
```

功能: 外部模块通过此接口, 开启调度监控功能, 之后内核进程切换时会调用 sched_switch_hook_add 注册的钩子。

输入参数: cpu_sched_map 表示用于监控 CPU 集。

返回值: 无。

4) 停止调度钩子监控

```
void sched_switch_hook_stop(void)
```

功能: 外部模块通过此接口, 关闭调度钩子监控功能。

返回值: 无。

■ 非阻塞缓冲区接口说明:

1) 初始化缓冲区

```
typedef struct unblock_buffer_cmd {  
    cpumask_t      cpu_unblock_map;  
    unsigned long  buf_size;
```

```
int      mode;  
} unblock_buffer_cmd;
```

功能：根据用户传入的参数分配缓冲区和设置缓冲区模式。

输入参数：

- **cpu_unblock_map**：需要分配缓冲区的 CPU 掩码；
- **buf_size**：每 CPU 上分配的缓冲区内存大小（字节）；
- **mode**：缓冲区模式（绕接还是非绕接）；
 - ✧ 当 **mode = 1** 时，绕接方式，用户写入数据超过缓冲区大小，将覆盖最先写入部分；
 - ✧ 当 **mode = 0** 时，非绕接方式，写满缓冲区，再写入失败。

返回值：返回 0 表示成功；返回小于 0 表示失败。具体错误码为：

- **-EINVAL**：传入的 CPU 掩码无效或者缓冲区模式无效；
- **-ENOMEM**：传入的 **buf_size** 大于可分配的内存；
- **-EEXIST**：缓冲区已经初始化。

2) 释放缓冲区

```
void unblock_buffer_free(void)
```

功能：将所有 **cpu** 的缓冲区（页面）释放给内核。

返回值：无

3) 将用户数据写入缓冲区

```
int unblock_buffer_write(char* buf, unsigned long len)
```

功能：外部模块通过此接口，将数据写入本 CPU 缓冲区。为了保证互斥，本函数在写入过程会关闭本 CPU 中断。

输入参数：

- **buf**：包含了用户数据的指针；
- **len**：待写入数据长度。

返回值：成功时返回实际写入数据长度；如果返回 0 代表缓冲区已满，无法写入；返回小于 0 表示失败。具体错误码为：

- **-EINVAL** **buf** 为空指针；

- -EACCES 缓冲区未初始化。

注意：用户提供的内存区域，不能引发缺页异常。

4) 将缓冲区数据读出

```
int unblock_buffer_read(int cpu,char* buf, unsigned long len)
```

功能：外部模块通过此接口，读取缓冲区中的信息。读取方式为先写入先读出。

输入参数：

- cpu：需要读取数据的 CPU 号；
- buf：待存放读出数据的用户内存区域指针；
- len：待读取数据长度。

返回值：返回实际读取的数据长度；返回 0 表示无数据可读；返回小于 0 表示失败。具体错误码为：

- -EINVAL：传入的 CPU 号非法， buf 为空指针；
- -EACCES：缓冲区未初始化。

注意：用户提供的内存区域，不能引发缺页异常。

9.7 高精度定时器

9.7.1.1 功能概述

为提高系统计时精度，支持更为灵活的定时器设置，自 Linux-2.6.16 内核开始，逐步引入高进度定时器（high-resolution kernel timers）。高精度时钟按时间顺序，基于红黑树（rb-tree）实现，摆脱了 tick 的限制，在硬件支持的情况下，精度达到了纳秒（ns）。

9.7.1.2 应用场景

高精度定时器主要应用于以下两大场景：

- 1) 低功耗的嵌入式系统，需要在系统空闲时不做任何事情，周期性的定时也会消耗电能
- 2) 多媒体业务，需要精确的定时，避免误码或回声

9.7.1.3 内核配置

配置宏选项 CONFIG_HIGH_RES_TIMERS。

在 menuconfig 中配置如下：

```
Processor type and features --->
[*]High Resolution Timer Support
```

9.7.1.4 使用方法

完成内核配置后，内核自动开启高精度定时器功能。

9.8 用户态异常处理框架

9.8.1.1 功能概述

异常处理框架作为核心功能层的一部分，主要是处理系统硬件或软件引起的故障，执行应用程序挂接的钩子。另外，异常处理框架作为对 Linux 标准异常处理框架的扩展，可以无缝地融合到标准 Linux 系统框架中。CGEL 的异常处理框架支持内核态和用户态两种模式。内核态异常处理框架在实现时有部分代码是挂接到 Linux 内核异常处理之中的，而在执行的顺序上，是在 Linux 内核的标准异常处理完毕之后再处理的。用户态异常处理框架作为一个单独的库提供，主要架构在标准 Linux 的信号处理机制之上。

CGEL 的异常处理框架提供了结构化的异常处理模式，提供了异常嵌套的处理能力，可以让用户编写更安全的程序。CGEL 异常处理框架主要有如下特性：

- 支持系统级异常回调钩子
- 支持线程级的异常回调钩子，对每个线程都有独立的异常回调钩子链表
- 支持简单的代码块异常处理
- 支持对异常处理的嵌套
- 支持异常信息捕获和多种手段输出（终端，文件，系统黑匣子）
- 支持线程在内核态下异常时的 coredump 功能

9.8.1.2 初始化

在用户态工程中，若用户使用异常处理框架，要先对其进行初始化，方法如下：

第 1 步，获取库文件，所在路径为 <\$ (KIDE_install_path) \target\tools\usrexclib>，其中 \$ (KIDE_install_path) 为 KIDE 2.8 的安装根路径。或是到成研所的对外 FTP (ftp://10.75.8.168/ZX-CGEL/2010 年/CGEL/CGEL VX.X / _CGEL_VX.X/工具/用户态异常处理框架，用户名和密码都为 zteuser) 或 <http://os.zte.com.cn/> 下载获取源代码包 usrexclib.7z。

第 2 步，在应用程序中，必须包含以下头文件，并对头文件路径进行设置：

```
#include "usr_exc.h"
#include "try.h"
```

在 KIDE 中，用户添加头文件搜索路径的方法，主要是通过工程中的【属性】->【工具设置】->【Cygwin C++链接器】->【库】添加以下三个库。



图 9-6 设置 usrexc 库

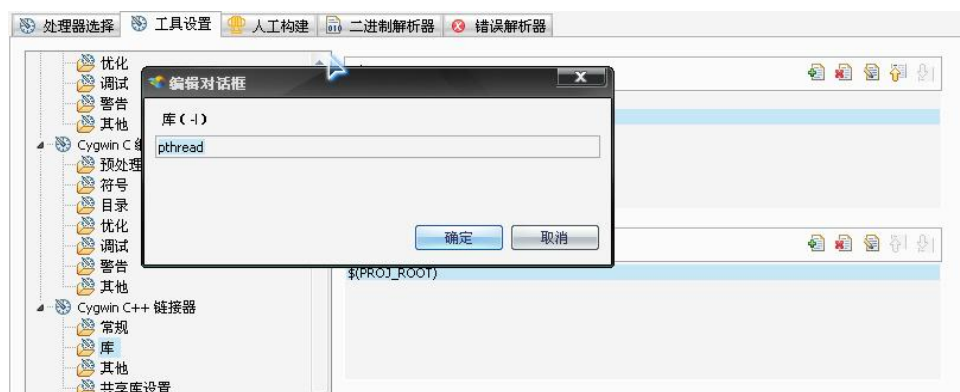


图 9-7 设置 pthread 库

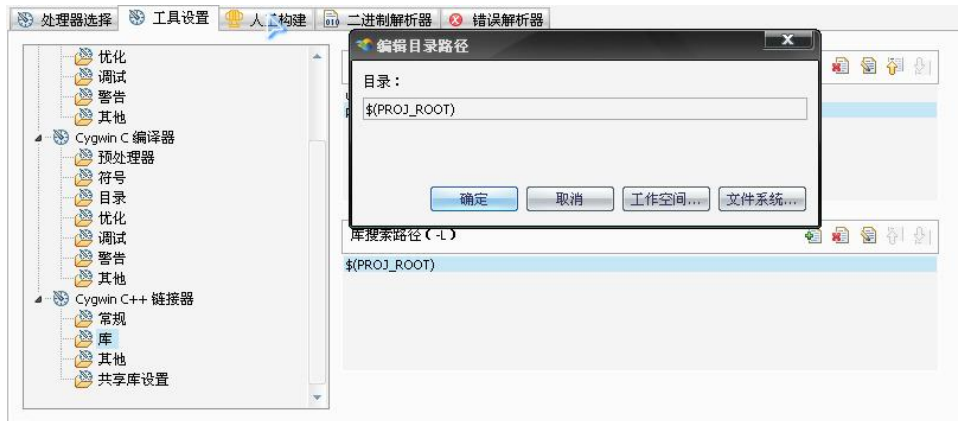


图 9-8 设置\$(PROC_ROOT)库路径

第 3 步，在用户态应用程序需要使用异常处理框架功能的线程入口初始化块使用宏 **USR_EXCLIB_INIT ()** 对异常处理框架进行初始化，初始化成功后系统异常框架正常运行，否则失败。

9.8.1.3 接口使用说明

在使用用户态异常处理框架时，应注意以下事项：

- 使用本库前，必须在调用了用户态异常处理库的初始化函数初始化成功以后才可使用；如果初始化不成功，初始化接口会返回初始化错误的错误代码；如果用户没有初始化就使用，那么本模块提供的结构都会返回模块未初始化的错误；
- 用户开发的接口可以发送信号，不能接受信号，允许被调度。
- 用户开发的异常回调函数的输入、输出以及返回值必须按照异常框架要求设计；
- 异常处理回调函数中使用 **longjmp ()** 和 **siglongjmp ()**，需要使用异常处理框架提供的 **usr_longjmp ()** 和 **usr_siglongjmp ()**；
- 使用异常处理框架后用户程序不能再注册如下信号的处理函数，如果使用异常框架又需要单独的信号处理函数，则可以使用异常处理框架的接口注册。不能再注册信号处理函数的信号有：**SIGBUS**、**SIGFPE**、**SIGSEGV**、**SIGILL**、**SIGABRT**、**SIGIOT**、**SIGXCPU**、**SIGXFSZ**、**SIGSYS**。
- 使用用户态异常处理框架后，第 59 号信号用户慎用。因为异常处理框架中已经使用了此信号。

9.8.1.4 异常抛出机制

在异常处理框架中，提供了异常抛出机制，用户可以在线程中调用相关函数启动异常抛出功能，然

后再通过预先定义好的宏进行系统异常的监控，并获取相应的异常信息。

异常抛出提供 `try-catch` 块的定义语句，一个 `try` 可以有多个 `catch`，每一个 `catch` 指定一个异常范围。一旦 `try` 块抛出某一异常，则 `try` 块后续代码不再执行，程序跳转到匹配的 `catch` 块执行。在 `try` 块抛出异常后由 `catch` 块处理，同时也可以可以在 `catch` 块继续抛出异常到嵌套的上一层 `catch` 块处理。`try` 块包含的任何代码抛出的异常，`catch` 语句检验顺序是按照其在源代码中出现的次序进行，并且匹配成功后此异常就会删除，并不再查找此异常的处理块。另外，`try-catch` 子句允许递归使用，可以嵌套（缺省可以嵌套 32 层）。除了用户程序能够定制异常并主动抛出外，也可以在 `try` 语句块没有显式的 `throw` 子句，而由系统检测异常并且抛出让 `catch` 块处理。要求能够自动抛出 CGEL 3.0 异常处理框架下所能够捕获的异常信号。

■ 使用方法

若用户在不使用异常处理框架的情况下使用抛异常机制，必须按照以下步骤操作才可正常使用。

- 1) 代码中包含头文件 “`try.h`”;
- 2) 线程初始化代码中调用 `try_up ()` 函数使能异常捕获机制;
- 3) 线程级代码中使用异常捕获相应的宏 **TRY**、**CATCH**、...（描述参见后面语义描述）;
- 4) 线程终止代码中调用 `try_down ()` 退出线程并释放资源，但此时并没有退出异常抛出功能。

■ 宏语义描述

本节就线程级代码中可以使用的宏的具体语义进行详细的说明，用户可根据自己需求选择使用，以达到更到地监控捕获异常。

TRY

使用这个宏定义一个 **TRY** 语句块。一个 **TRY** 语句块标识一个可能抛出异常的代码块。这些异常可以在 **CATCH** 和 **AND_CATCH** 代码块中进行处理。允许递归：异常可以使用 **THROW_LAST** 宏抛出到外部更上一级的 **TRY** 块，或者忽略它。**TRY** 块的终止使用宏 **END_CATCH** 或者 **END_CATCH_ALL**。没有 **CATCH** 语句的 **TRY** 块使用宏 **END_TRY**。

CATCH

使用此宏为前置 **TRY** 块定义首个异常异常处理语句块。调用 **THROW_LAST** 宏将异常传递到外层异常处理框架。终止 **TRY** 块使用 **END_CATCH** 宏。

AND_CATCH

为前置 TRY 块定义一个追加的异常信息处理块。使用 CATCH 宏处理一个异常范围，接下来使用 AND_CATCH 宏处理另外一个异常范围。标记 TRY 块结束使用 END_CATCH 宏。

END_CATCH

标记 CATCH 或者 AND_CATCH 块结束。

THROW

抛出指定的异常。THROW 会中断程序的执行，将程序跳转到相应的程序中的 CATCH 块。如果没有提供 CATCH 块，异常将会被删除。

THROW_LAST

抛出异常到外层框架的 CATCH 块。

这个宏允许抛出本次触发的异常。一旦你选择刚刚捕获的异常，跳出该 CATCH 块后该异常将会被删除。使用 THROW_LAST，异常会传递到嵌套的外层 CATCH 处理句柄。

CATCH_ALL

定义处理在前面 TRY 块抛出的余下异常信息的语句块。异常处理块也可以包含一个异常处理返回。调用 THROW_LAST 宏将异常处理抛出到外层的 TRY-CATCH 框架。一旦使用 CATCH_ALL，终止 TRY 块必须使用宏 END_CATCH_ALL。

AND_CATCH_ALL

定义处理在前面 TRY 块抛出的余下异常信息的语句块。使用 CATCH 宏捕获一个异常范围，使用宏 AND_CATCH_ALL 捕获余下的其它异常类型。一旦使用 AND_CATCH_ALL，结束 TRY 块必须使用 END_CATCH_ALL 宏。

END_CATCH_ALL

标记 CATCH_ALL 块或者 AND_CATCH_ALL 块结束。

END_TRY

标记 TRY 块结束。

应用实例

```
int main (int argc, char* argv[])
{
    int retval;
    pthread_t tid;
    retval = pthread_create (&tid, NULL, (void *) testfunc, (void *) NULL);
    if (retval != 0)
    {
        printf ("error: pthread_create failure!\n");
        exit (1);
    }

    int times=0; // 主线程
    for (;times<3;)
    {
        printf ("debug: in main (), %d times.\n", ++times);
        sleep (2);
    }

    return 0;
}

void func_two () // 测试线程 2
{
    TRY
    {
        THROW (1);
    }
    CATCH ( 3, 8 )
    {
        printf ("----- %s ----- %d ----- \n",__FUNCTION__,__LINE__);
    }
}
```

```
}
AND_CATCH_ALL ( )
{
    printf ( " ----- %s ----- %d ----- \n", __FUNCTION__, __LINE__ );
    THROW_LAST ( ) ;
}
END_CATCH
}

void func_one ( ) // 测试线程 1
{
    TRY
    {
        //THROW (5) ;
        func_two ( ) ;
    }
    CATCH ( 1, 10 )
    {
        printf ( " ----- %s ----- %d ----- \n", __FUNCTION__, __LINE__ );
    }
    END_CATCH
}

int testfunc (void) // 测试主函数
{
    try_up ( ) ;

    #if 1 // 无 CATCH 语句块
        TRY
        {
            THROW (1) ;
        }
        END_TRY
    #endif
}
```



```
#if 1 // 基本的 TRY-CATCH 测试

    TRY
    {
        THROW (5);
    }
    CATCH ( 3, 8 )
    {
        printf ("----- %s ----- %d ----- \n",__FUNCTION__,__LINE__);
    }
    END_CATCH
#endif

#if 1 // CATCH 与测试

    TRY
    {
        THROW (1);
    }
    CATCH ( 3, 8 )
    {
        printf ("----- %s ----- %d ----- \n",__FUNCTION__,__LINE__);
    }
    AND_CATCH ( 9, 10 )
    {
        printf ("----- %s ----- %d ----- \n",__FUNCTION__,__LINE__);
    }
    END_CATCH
#endif

#if 1 // CATCH_ALL 测试

    TRY
    {
        THROW (1);
    }
    CATCH_ALL ( )
```

```
{
    printf (" ----- %s ----- %d ----- \n",__FUNCTION__,__LINE__);
}
END_CATCH_ALL
#endif

#if 1 // AND_CATCH_ALL 测试
    TRY
    {
        THROW (2);
    }
    CATCH ( 2, 2 )
    {
        printf (" ----- %s ----- %d ----- \n",__FUNCTION__,__LINE__);
    }
    AND_CATCH_ALL ( )
    {
        printf (" ----- %s ----- %d ----- \n",__FUNCTION__,__LINE__);
    }
    END_CATCH_ALL
#endif

#if 1 // AND_CATCH 与 AND_CATCH_ALL 测试
    TRY
    {
        THROW (2);
    }
    CATCH ( 2, 3 )
    {
        printf (" ----- %s ----- %d ----- \n",__FUNCTION__,__LINE__);
    }
    AND_CATCH ( 4, 4 )
    {
        printf (" ----- %s ----- %d ----- \n",__FUNCTION__,__LINE__);
    }
}
```

```
    }
    AND_CATCH_ALL ( )
    {
        printf ( " ----- %s ----- %d ----- \n", __FUNCTION__, __LINE__ );
    }
    END_CATCH_ALL
#endif

#if 1
    TRY
    {
        func_one ( ) ;
    }
    CATCH_ALL ( )
    {
        printf ( " ----- %s ----- %d ----- \n", __FUNCTION__, __LINE__ );
    }
    END_CATCH_ALL
#endif

    int times=0;
    for (;times<3;)
    {
        printf ( "debug: in testfunc ( ) , %d times.\n", ++times );
        sleep ( 2 );
    }

    try_down ( ) ;

    return 0;
}
```

9.9 多核差异化运行

9.9.1 基于 SMP 系统的多核差异化运行

9.9.1.1 功能描述

基于 SMP 系统的多核差异化运行提供如下功能：

- 支持线程排他性绑定；
- 支持独占式、共享式 CPU 设置；
- 支持中断绑定到指定 cpu。

9.9.1.2 应用场景

多核系统，用户不希望使用两个操作系统，希望多核系统划分出控制面和媒体面。控制面运行内核线程、业务管理进程；媒体面单纯运行高吞吐量的业务进程，通常媒体面得 CPU 占用率极高。此时用户不希望内核线程的运行影响媒体面的业务性能，但是媒体面与控制面同时运行在一个 SMP 系统下。

9.9.1.3 外部接口

关于外部接口的详细说明请参考《广东中兴新支点 CGEL API 参考手册》第 4 章。

■ 排他性绑定

说明：除 idle 任务外，未显式绑定到该 CPU 的线程（包括用户线程和内核线程），不能在该 CPU 上运行。

内核配置：CONFIG_EXCLUSIVE_BIND

在 menuconfig 中配置如下：

```
CPU Exclusive Binding --->
[*]Exclusive binding threads to cpu
```

接口名称：

```
/proc/exclusive_cpus
```

功能：proc 接口，供用户查询和设置排他性 CPU 集合，以及独占 CPU 集合、共享 CPU 集合。

用法:

1) 不配置 MIGRATE_KTHREAD (智能迁移功能) 选项

以下方式可输出当前排他性 CPU 集合, 或者其他读取文件的方法。

```
cat /proc/exclusive_cpus
```

以下方式可输入待设置的排它性 CPU 集合, X 代表输入值, 或者其他写文件的方法。

```
echo X > /proc/exclusive_cpus
```

X 以 16 进制表示, 无需在前面加 0x 前缀, 例如:

设置排它性集合为 cpu2、cpu3, 那么输入 `echo c > /proc/exclusive_cpus`;

设置排它性集合为 cpu2、cpu3、cpu5, 那么输入 `echo 2c > /proc/exclusive_cpus`;

用户对线程的绑定优先于排它性 cpu 集合设置的反向绑定。

用户对线程的绑定会清除反向绑定标志 reverse binding flag。

要取消排它性 cpu 集合, 执行 `echo 0 > /proc/exclusive_cpus`。

2) 配置 MIGRATE_KTHREAD (智能迁移功能) 选项

以下方式可输出当前排他性 CPU 集合、独占 CPU 集合、共享 CPU 集合。(或者其他读取文件的方法)

```
cat /proc/exclusive_cpus
```

以下方式可输入待设置的排它性 CPU 集合、独占 CPU 集合、共享 CPU 集合。X Y Z 分别代表输入的 CPU 掩码, 以 16 进制表示, 中间使用空格隔开, 无需在前面加 0x 前缀。掩码之间的约束规则为: 独占式 CPU 集合不能超过排他性 CPU 集合的大小; 共享式 CPU 集合不能超过非排他性 CPU 集合的大小。(或者其他写文件的方法)

```
echo X Y Z > /proc/exclusive_cpus
```

以下是合法的输入格式:

```
//输入一个参数默认为排他性 CPU 集合, 而独占 CPU 集合默认等于排他 CPU 集合, 共享 CPU 集合默认等于非排他 CPU 集合。
```

```
echo X > /proc/exclusive_cpus
```

```
//三种 CPU 集合同时指定。
```

```
echo X Y Z > /proc/exclusive_cpus
```

以下是非法的输入格式：

```
echo X Y > /proc/exclusive_cpus      //只指定两种 CPU 集合非法输入
echo X Y Z A> /proc/exclusive_cpus    //指定超过三种 CPU 集合输入非法输入
```

■ 独占式、共享式 CPU

说明：独占式集合是指只运行一个或多个高负载进程的 CPU 的集合。共享式集合是指没有运行高负载进程，CPU 占用率适中，用户态进程或者内核线程都可以在其上执行的 CPU 的集合。通过设置这两个集合，可以将影响媒体面性能的内核线程迁移到控制面上。

内核配置：CONFIG_MIGRATE_KTHREAD

在 menuconfig 中配置如下：

```
Uniform System call --->
  Enable uniform system call framework --->
    [*]Migrate kernel thread to shared CPUs<NEW>"
```

接口名称：

```
/proc/migrate_info
```

功能：proc 接口，供用户查询或设置独占式 CPU 集合和共享式 CPU 集合的迁移结果。

用法：查看上一次设置独占式 CPU 集合和共享式 CPU 集合的迁移结果。（或者其他读取文件的方法）

```
cat /proc/migrate_info
```

■ 中断绑定

说明：通过设置掩码，可以让相应中断只在指定的 CPU 集合上出现。

接口：/proc/irq/中断号/smp_affinity

其中 smp_affinity 默认值为 0xffffffff，为 CPU 绑定的位图，例如 0x00000003 为将中断绑定到 CPU0 和 CPU1 上。

内核配置：无需配置，默认存在。

■ 指定 CPU 上 softlockup 开关设置

说明：提供开启和关闭指定 CPU 上 softlockup 功能，允许用户将某些 CPU 上的 softlockup 功能关闭，以减小对该 CPU 的性能影响。

在配置了软件 watchdog 的情况下，每个 CPU 会对应一个 watchdog 内核线程，优先级设置为 99。每秒钟 watchdog 线程被唤醒一次，并更新软件狗的时间戳。每次时钟中断的时候，都会检查当前时间和上次记录的时间戳之间的间隔，如果大于 softlockup_thresh(10s)，则认为系统调度出现了问题。例如在内核或驱动流程中，关闭抢占时间过长，从而造成优先级最高的任务也无法被调度的状况，此时该线程将回溯内核当前执行路径的调用栈信息，以便定位出现问题的环节。

由于媒体面核上跑的应用为一个高优先级死循环，为了避免软件 watchdog 在媒体面核上的死循环告警，可以关闭媒体面核上的 watchdog 功能。

接口：proc/sys/kernel/softlockup_cpumask

内核配置：CONFIG_SOFTLOCKUP_SWITCH

在 menuconfig 中配置如下：

```
Kernel hacking --->
  [*]Kernel debugging
  [*]Detect Soft Lockups
```

设置指定 CPU 上 softlockup 开关用法：

```
echo cpumask > proc/sys/kernel/softlockup_cpumask
```

默认开启所有 CPU 的 softlockup 功能，若要关闭某 cpu 上 softlockup，只需将该 cpu 在 cpumask 中对应位的数值置为 0 即可，比如有 8 个 cpu 的系统，softlockup_cpumask 的值默认是 ff，若要关闭 cpu3 上的 softlockup，需要将 cpumask 修改为 f7。

■ 普通定时器绑定

说明：普通定时器绑定功能是将内核的定时器（timer）绑定到指定的 CPU 上运行。

在 SMP 多核差异化运行的场景下，某些 CPU 上需要持续运行一些单独的任务，而不希望任务被频繁打断，但是默认情况下内核定时器的运行优先级比任务高，并且定时器具体在哪个 CPU 下运行并不能控制。因此，内核定时器可能打断这些需要高效持续并且绑定在某个核上的任务，造成性能降低。为了避免这种情况，可以将内核定时器绑定到指定的 CPU 上运行，避免造成性能损失。

内核配置：打开普通定时器绑定功能宏 CONFIG_BIND_TIMER_TO_CPU 与绑定定时器运行所在的 CPU 号宏 CONFIG_BIND_TIMER_CPU。

在 menuconfig 中配置如下：

```
Bind all kernel timer to one CPU --->
```

```
//开启定时器绑定功能
```

```
(5) Logic CPU number all timer bind to. (0~256)
```

```
//绑定到哪个 CPU，此例为 CPU5
```

完成配置后，编译并启动内核，此时内核定时器自动完成 CPU 的绑定运行。若需要更改定时器运行所在的 CPU，可以通过 sysctl 功能设置：

```
echo 3 > /proc/sys/kernel/ bind_timer_cpu
```

```
//将定时器运行所在的 CPU 更改成 CPU3
```

9.9.2 基于 AMP 系统的多核差异化运行

9.9.2.1 功能描述

多核差异化运行采用 AMP 模式，即多核上运行多个 OS（包括一个 OS 的多个实例），每个 OS 管理一个或多个 CPU 核，实现多核之间的差异化运行。一部分核运行 Linux，进行常规系统管理，处理控制面数据；另一部分核处理用户面数据。这将完全排除两者之间的干扰，大大提升用户面数据处理的能力。AMP 的使用使软件的架构设计变得更加灵活。

主要包括以下功能：

- 提供系统物理内存在多个操作系统间的分配接口；
- 多核异种操作系统之间的内存共享支持；
- 多核异种操作系统之间的自旋锁支持；
- 对其他操作系统映像的版本管理和加载、引导支持；
- 提供虚拟 UART 支持；
- 对运行于其他核上的系统，提供系统级调试和任务级调试支持；
- 提供 P2020 AMP 多核差异化的多核间共享内存支持。

9.9.2.2 应用场景

针对同时处理两种类型数据流时，一种数据流信息量小，对性能要求不高；另一种数据流不仅信息量大，而且对性能要求较高。

Linux 采用的 SMP 模式能使多核 CPU 在内核的管理下最大限度的发挥其性能。但若所有的 CPU 核均工作在 SMP 模式下，难免会相互影响。此时，我们希望更多的 CPU 时间能够用在信息量大，性能要求高的数据流上，而 SMP 的均衡调度无法做到，控制面的某些功能可能会干扰用户面的数据处理。此时，就需要采用多核差异化方式运行。

9.9.2.3 内核配置

配置如下宏选项：

- CONFIG_MULTIOS_MANAGER（多操作系统支持）
- CONFIG_MOSM_SHMEM_TOP_ADDRESS=0x20000000（共享内存最高地址）
- CONFIG_MULTIOS_DEV（多操作系统设备管理支持）
- CONFIG_MULTIOS_VIRT_UART（多操作系统的虚拟串口支持）

在 menuconfig 中配置如下：

```
Device Drivers --->
  Character devices--->
    [*]Multios manager device support
    [*]Multios virtual uart support
```

9.9.2.4 外部接口

■ 多核差异化内核启动参数接口

功能：控制多操作系统管理器的资源划分。

说明：

- mosm_support：为 1 表示支持配置了 mosm 模块，支持多核差异化运行
- linux_cpu_mask，检查每个 bit 是否为 1 确定由 Linux 管理的 cpu 核
- linuxmemsz：单位可为 G/g/M/m/K/k，Linux 操作系统可使用的物理内存总数
- appshmemsz：单位可为 G/g/M/m/K/k，多个操作系统之间的应用共享内存区大小

■ 多核差异化 Loader 加载器接口

功能：通过 Loader 加载器控制除 Linux 操作系统以外的其他各操作系统的物理内存起始地址的布局，对操作系统的启停控制，多核差异化模块运行情况查看。

用法：

```
./loader load -f file -m <cpu mask>
    [ -z <os managable memory size@physical address> ];
    [ -T <virt_uart|ttyS1> ]
    [ --dual-core ]
```

```
./loader stop -m <cpu_mask>  
./loader status
```

命令行参数：

- -f 目标操作系统的 ELF 文件
- -m 目标操作系统被加载的 CPU id
- -z 目标操作系统被加载的内存起始物理地址和占据的物理内存大小
- -T 选择 console(virt_uart/ttyS1)
- --dual-core, 代表 Linux 处于 dual-core 模式，不加代表 Linux 处于 quad-core 模式，仅 CGEL4.X 提供此参数

9.9.2.5 基于共享内存通讯

AMP 多核差异化模块向多个 AMP 内核提供一种基于共享内存的通讯机制，多个内核可以通过此机制进行数据交互，此功能目前只针对 P2020 单板。

■ 基本流程

共享内存通讯机制根据系统中 CPU 个数申请相应个数的共享内存区。P2020 RDB 分为两个核 Core0 和 Core1，所以申请两块共享内存区 A 和 B，共享内存区只能实现单向通信。每块内存通过大小使用宏 CONFIG_MULTIOS_SHMDATA_SIZE 来控制。内存区组织成循环队列方式，可以循环使用。

当 Core0 要与 Core1 通信时：首先，Core0 访问共享内存 A，并在共享内存中获得一块共享区域并存放需要通信的数据，然后 Core0 发送到一个核间中断（IPI）到 Core1，Core1 然后激活睡眠的读线程去访问共享内存中存放的数据。基本流程如下图所示。

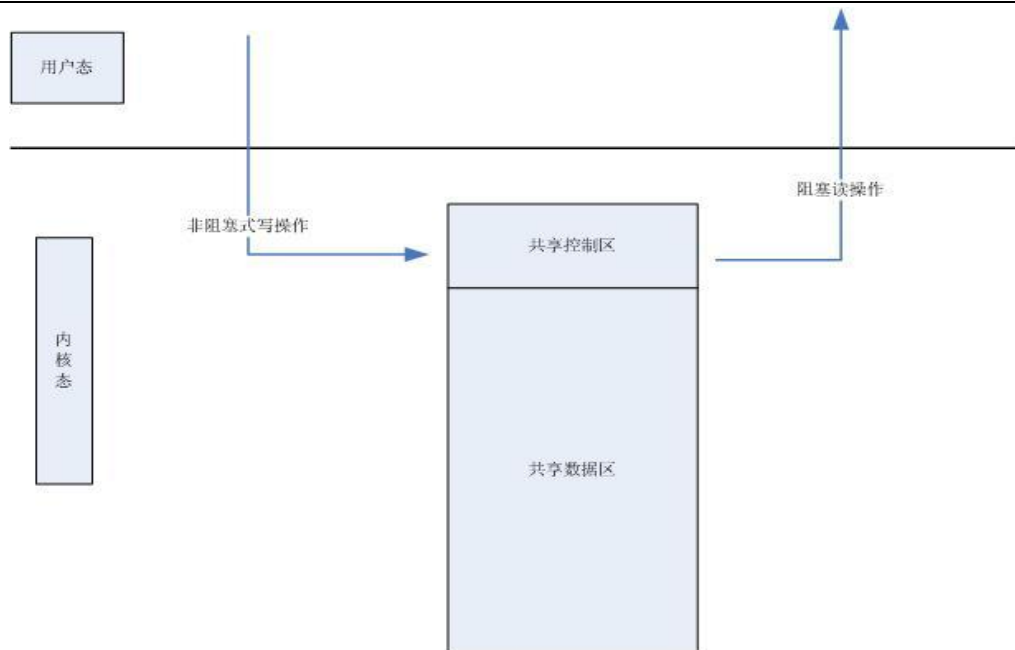


图 9-9 基于共享内存通讯机制

■ 内核配置

配置 P2020 内存共享机制专用宏选项：

- CONFIG_MULTIOS_SHM（共享内存机制支持）
- CONFIG_MULTIOS_SHMDATA_SIZE（一个共享内存的大小）

在 menuconfig 中配置如下，选中 **【Support for shared memory manager】**：

```
Kernel options --->
  Support for shared memory manager
  (0x00008000) max shared memory for one core
```

■ P2020amp 共享内存创建查询接口

功能：通过统一系统调用接口创建或查询共享内存。

结构：

```
syscall(int number, SYSCALL_MOSM_SHMGET, int key, size_t size, int shmflg)
```

参数：number 为统一系统调用的系统调用号。

```
enum {
```

```
SYS_KERN = 1,
SYS_VM,
SYS_NET,
SYS_FS,
SYS_DEV,
SYS_MOSM,
/* adds anything new here */
MAX_NR_SYSCALL = SYS_MOSM,
};
enum {
    /* adds anything new here */
    SYS_MOSM_SHMGET=1,
    SYS_MOSM_SHMCTL,
    SYS_MOSM_SHMDELETE,
    SYS_MOSM_SHMREAD,
    SYS_MOSM_SHMWRITE,
    MAX_NR_VM = SYS_MOSM_SHMWRITE,
};
#define SYS_ID(first, second) ((first << 24) | (second << 16))
#define SYSCALL_MOSM_SHMGET SYS_ID(SYS_MOSM, SYS_MOSM_SHMGET)
#define SYSCALL_MOSM_SHMCTL SYS_ID(SYS_MOSM, SYS_MOSM_SHMCTL)
#define SYSCALL_MOSM_SHMDELETE SYS_ID(SYS_MOSM, SYS_MOSM_SHMDELETE)
#define SYSCALL_MOSM_SHMREAD SYS_ID(SYS_MOSM, SYS_MOSM_SHMREAD)
#define SYSCALL_MOSM_SHMWRITE SYS_ID(SYS_MOSM, SYS_MOSM_SHMWRITE)
```

参数:

- Key: 共享内存的唯一标识, 必须为正整数;
- Size: 申请共享内存的大小;
- Shmflg: 代表查找还是创建, 0 代表创建, 1 代表查找。

返回值: 成功返回传入的 key, 失败返回错误码。

■ P2020amp 共享内存写接口

功能: 通过统一系统调用接口写共享内存。

结构:

```
syscall(int number, SYSCALL_MOSM_SHMWRITE, int key, const void __user* buf, size_t size,int flag)
```

参数:

- number: 统一系统调用的系统调用号;
- Key: 共享内存的唯一标识;
- Buf: 用户缓冲区;
- Size: 需要写入的数据字节数;
- Flag: 暂时不用, 传入 0。

返回值: 成功返回写入的字节数, 否则返回错误码。

说明: 当共享内存中有空间就可以写入, 而不管是否有读操作在进行。没有空间可写, 则等待或者直接返回。写入操作完成后, 会唤醒睡眠的读操作, 进行读操作进行读取数据。写操作之间是互斥, 一个时刻只能有一个写操作在进行。

■ P2020amp 共享内存读接口

功能: 通过统一系统调用接口读共享内存。

结构:

```
syscall(int number, SYSCALL_MOSM_SHMREAD, int key, const void __user* buf, size_t size,int flag)
```

参数:

- number: 统一系统调用的系统调用号;
- Key: 共享内存的唯一标识;
- Buf: 用户缓冲区;
- Size: 需要读出的数据字节数;
- Flag: 阻塞时间, 0 代表不阻塞。

返回值: 成功返回读出的字节数, 否则返回错误码。

说明: 读操作之间是互斥的, 同一时刻只允许一个读操作。读操作完成后可以通过核间中断触发阻塞的写操作。

■ P2020amp 共享内存控制接口

功能: 通过统一系统调用接口唤醒阻塞的读进程。

结构:

```
syscall(int number, SYSCALL_MOSM_SHMCTL, int key,int cpu,int flag)
```

参数:

- number: 统一系统调用的系统调用号;
- Key: 共享内存的唯一标识;
- cpu: 对方 cpu 的 id , 从 0 开始;
- Flag: 暂时不用, 传入 0。

返回值: 成功返回 0, 否则返回错误码。

■ P2020amp 共享内存删除接口

用途: 用于删除共享内存。

结构:

```
syscall(int number, SYSCALL_MOSM_SHMDELETE, int key)
```

说明: 通过统一系统调用接口删除共享内存。删除之前必须保证读写进程已经退出, 并且必须由创建者删除。

参数:

- number: 统一系统调用的系统调用号;
- Key: 共享内存的唯一标识;
- cpu: 对方 cpu 的 id , 从 0 开始;
- Flag: 暂时不用, 传入 0。

返回值: 成功返回 0, 否则返回错误码。

■ P2020amp 共享内存初始化信息 proc 接口

通过/proc/mosm/shminfo 查看共享内存的物理, 虚拟地址, 以及大小。

■ 注意事项

- 每个核只能通过接口创建一个共享内存区, 使用完后必须删除再创建;
- 共享区的大小是在系统初始化时确定的, 所以不是动态改变;
- 共享内存区只能由创建者删除, 并且删除之前必须通知对方读数据, 否则会出现问题。

9.10 安全机制

9.10.1 SMACK 机制

9.10.1.1 功能描述

SMACK (Simplified Mandatory Access Control Kernel) 是基于内核实现的简化强制访问控制。SMACK 安全策略由安全管理员根据安全威胁和安全假设预先设定, 通过比较主、客体的安全标记来决定信息是否可以安全流动。仅 CGEL4.X 内核对此功能提供支持。

9.10.1.2 应用场景

SMACK 具有较少的内存消耗和较高的运行效率, 因而更适用于系统资源需求极少, 且配置简单的嵌入式系统。

9.10.1.3 内核配置

配置宏选项 `CONFIG_SECURITY_SMACK`。

在 `menuconfig` 中配置如下:

```
Security options --->
[*]Simplified Mandatory Access Control Kernel Support
```

9.10.1.4 运行使用

SMACK 构建于 Linux 的 LSM (Linux Security Modules) 框架之上, 它使用了文件的扩展属性, 这与 SELinux 相同。同时也提供了一个伪文件系统 `smackfs`, 用来操纵进程和系统的 SMACK 属性。SMACK 给系统的所有的主体和客体都添加了安全标签, 主客体之间的信息交流权限都由访问控制规则来确定。通过 `smackload` 程序可以将访问控制规则写入 `/smack/load` 中。

用户可以通过 `setxattr` 系统调用来修改文件和目录项的标签。

9.10.2 OCF 框架

9.10.2.1 功能描述

OCF 系统加密框架, 方便应用以统一接口使用硬件的加密功能。OCF 提供两类 API 供其他内核子系

统使用，一类是供消费者（consumers，即其他内核子系统）使用，另一类供生产者（producers，即硬件加密设备驱动）使用。OCF 支持两类算法：对称加密算法和非对称加密算法。对称算法是建立在会话（session）上下文基础上，可以利用加密设备的缓存机制，实现异步处理批量的数据；非对称算法执行相对独立的操作，加密过程的建立和终止是同步的行为。仅 CGEL4.X 对此功能提供支持。

9.10.2.2 应用场景

随着网络应用的发展，越来越多应用交互的网络报文包含了敏感信息，为了保护这些敏感信息，几乎每个应用都采用对称加密算法（symmetric-algorithm）或非对称加密算法（asymmetric-algorithm）进行保护。由于信息量的增加，很多应用采用软件加密已经不能够满足高性能需要，目前提升加密性能的普遍做法是采用硬件加密卡。为保证硬件加密引擎和应用独立，并能最优地发挥硬件加密引擎的性能优势，OpenBSD 操作系统实现了 OCF 加密框架（OpenBSD/FreeBSD Cryptographic Framework）。OCF 加密框架在操作系统内核实现虚拟化服务，通过隐藏硬件加密引擎设备的具体实现细节，提供统一访问的 API。这种抽象设计能够使操作系统很容易支持新的硬件加速器，使应用程序无须关注加速器设备的具体实现而可以使用任何此类硬件加密引擎。

CGEL 作为电信级操作系统，安全性是其中重要特性，因此 CGEL 也提供 OCF 加密框架，方便应用以统一接口使用硬件的加密功能。

9.10.2.3 内核配置

配置宏选项：

CONFIG_OCF_OCF（打开加密框架）；

CONFIG_OCF_CRYPTODEV（支持用户 API 直接访问 crypto 硬件）。

在 menuconfig 中配置如下：

```
OCF Configuration --->
  [*]OCF <Open Cryptographic Framework>
  [*]cryptodev <user space support>
```

9.10.2.4 运行使用

OCF 通过 vfs 暴露/dev/crypto 文件，用户态应用可以通过操作该文件来使用加密服务。内核层的加密卡驱动可以将其加解密服务注册或注销(crypto_register()/crypto_unregister())到 OCF 框架中，包括硬件加密设备驱动软件加密驱动。用户态程序使用 ioctl 操作/dev/crypto，发出初始化，加解密等请求，OCF 通过调度线程，将对应请求发给相应的设备驱动，完成加解密过程。

9.10.3 防 DOS 攻击

9.10.3.1 功能描述

DOS 攻击多是利用网络协议的漏洞，最常见又最容易被利用的攻击方式是 SYN Flood 攻击。SYN Flood 攻击利用 TCP 协议缺陷，通过发送大量的半连接请求，耗费 CPU 和内存资源。SYN Flood 攻击除了能影响主机外，还可以危害路由器、防火墙等网络系统，事实上 SYN 攻击并不管目标是什么系统，只要这些系统打开 TCP 服务就可以实施。

CGEL 的防 DOS 攻击功能主要针对防御 SYN 型 DOS 攻击。

9.10.3.2 应用场景

客户端可能利用网络协议实现的缺陷和网络带宽资源的有限性，向目标服务器或目标网络设备发送大量连接请求或无用的数据包，从而大量占用受害者的系统和带宽资源使其无法再继续响应正常的请求，无法提供正常的服务或资源访问。

9.10.3.3 内核配置

配置宏选项 CONFIG_NET（网络配置，当配置完网络后无需进行其他配置，内核默认启动防 DOS 攻击）。

在 menuconfig 中配置如下：

```
Security options --->
[*]Enable different security models
[*]Socket and Networking Security Hooks
```

9.10.3.4 运行使用

■ 增大队列 SYN 最大半连接数

CGEL 队列默认的 SYN 最大半连接容量为 1024，但这个值对于电信级服务器来说是不够的，一次简单的 SYN 攻击就足以将其完全占用。因此，防御 DOS 攻击最简单的方法就是增大这个默认值，在 CGEL 下输入命令，即可将队列 SYN 最大半连接容量改为 4096，可以容纳更多等待连接的网络连接数：

```
echo 4096 > /proc/sys/net/ipv4/tcp_max_syn_backlog
```

■ 减小超时值

在 Linux 中建立 TCP 连接时，在客户端和服务端之间创建握手过程中，当服务器未收到客户端的确认包时，会重发请求包，直到超时才将此条目从未连接队列删除。即是说半连接存在一定的存活时间，超过这个时间，半连接就会自动断开。在上述 DOS 攻击测试中，当经过较长的时间后，就会发现一些半连接已经自动断开了。半连接存活时间实际上是系统所有重传次数等待的超时时间之和，这个值越大，半连接数占用的 Backlog 队列的时间就越长，系统能处理的 SYN 请求就越少，因此，缩短超时时间就可以有效防御 SYN 攻击，这可以通过缩小重传超时时间和减少重传次数来实现。在 CGEL 中默认的重传次数为 5 次，总超时时间为 3 分钟，在 CGEL 中执行命令：

```
#cat /proc/sys/net/ipv4/tcp_synack_retries
5
#cat /proc/sys/net/ipv4/tcp_syn_retries
5
```

减小重试次数，执行以下命令，将超时次数设置为 1：

```
echo 1 > /proc/sys/net/ipv4/tcp_synack_retries
echo 1 > /proc/sys/net/ipv4/tcp_syn_retries
```

■ 利用 SYN Cookies 来防御 DOS 攻击

SYN Cookie 是通过一个 Cookie 来响应 TCP SYN 请求。在正常的 TCP 连接过程中，当服务器接收一个 SYN 数据包，就会返回一个 SYN-ACK 包来应答，然后进入 TCP-SYN-RECV（半开放连接）状态来等待最后返回的 ACK 包。服务器用一个数据空间来描述所有未决的连接，然而这个数据空间的大小是有限的，所以攻击者将塞满这个空间。在 TCP SYN COOKIE 的执行过程中，当服务器收到一个 SYN 包的时候，他返回一个 SYN-ACK 包，这个数据包的 ACK 序列号是经过加密的，它由 TCP 连接的源地址和端口号，目标地址和端口号，以及一个加密种子经过 HASH 计算得出的，然后服务器释放所有的状态。如果一个 ACK 包从客户端返回后，服务器重新计算 COOKIE 来判断它是不是上个 SYN-ACK 的返回包，若是，则服务器可以直接进入 TCP 连接状态并打开连接。这样服务器就可以避免守候半开放连接了。在 CGEL 中执行如下命令，启动 SYN Cookies 功能：

```
echo 1 > /proc/sys/net/ipv4/tcp_syncookies
```

9.10.4 内核堆栈溢出

9.10.4.1 功能描述

所有进程（包括内核进程和普通进程）都有一个内核栈，如 x86-32 机器上内核栈大小为 4KB，或 8KB，内核栈大小可在编译内核时配置。内核栈有以下两大用途：

- 当进程陷入内核态，即内核代表进程执行系统调用时，系统调用的参数就放在内核栈上，内核栈记录着进程的在内核中的调用链；
- 在内核栈被配置成 8KB 大小的情况下，当中断当前进程时，它将使用当前被中断进程的内核栈。对于用户进程而言，其既有用户地址空间中的栈，也有自身的内核栈。而内核进程就只有内核栈。内核态堆栈如图 9-10 所示：

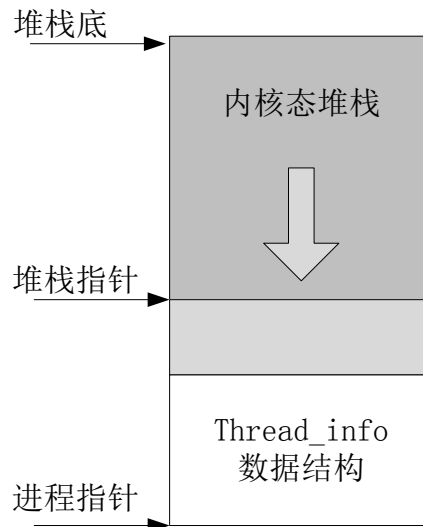


图 9-10 内核态堆栈

如果内核堆栈溢出，即覆盖了任务的数据结构，将会导致内核 panic。

内核堆栈溢出检测将会实时检测内核堆栈的变化情况，若内核堆栈剩余不足 1k 的空间，将会打印出告警和堆栈回溯信息，以供用户及时处理。

9.10.4.2 应用场景

此功能应用于可能发生内核堆栈溢出的情况。

9.10.4.3 内核配置

配置宏选项 CONFIG_DEBUG_STACKOVERFLOW。

在 menuconfig 中配置如下：

```
Kernel hacking --->
[*]Check for stack overflows
```

9.10.4.4 运行使用

完成内核配置后，即可启动该功能。

9.11 启动增强

9.11.1 优雅重启支持

9.11.1.1 功能描述

嵌入式系统设计中目前常用的复位方式主要有2种：看门狗复位以及系统reboot命令复位。

- 看门狗复位是指在具有硬件看门狗设备的嵌入式单板上，采用停止喂狗，从而引起饿狗的方法来达到重启系统的目标。这种复位方式虽然做到了单板级彻底的复位，包括复位单板上的各种硬件设备、芯片，但是，从操作系统来看，这种复位方式相当于异常掉电，其没有对文件系统、系统设备等进行复位时的必要处理，从而可能引起设备状态不确定，文件系统数据异常等问题。
- 系统 reboot 命令复位是指利用系统提供的关机命令，如/sbin/halt、/sbin/reboot 等，来达到重启系统的目的。这种复位方式对系统进行了一些保护，但是依然存在一定的风险，比如在 busybox 下，关机和重启命令没有对文件系统写回及卸载操作，而是直接将 CPU 复位。当然，由于仅仅是复位 CPU，因此，它也不具备以上提到的看门狗复位的复位彻底性。

结合上面两种现有的关机、重启方式，CGEL 提出了一个统一的关机重启方案，来达到优雅重启的目的，最大程度的保护系统的稳定和文件系统的完整，用户可以根据业务的实际需要，对该程序进行定制化修改，以达到对关机时间、关机效率以及看门狗喂狗等其它方面的要求。从下表可以更直观的看出优雅重启与现有复位方式各自的特点：

	功能的使用方式	对系统资源的保护	能否保证系统复位
系统命令	/sbin/reboot	重启前进行部分资源回收	尽可能的保证系统复位
看门狗复位	停止喂狗，看门狗超时引起硬件复位	直接复位，没有资源回收	保证复位
优雅重启	在用户程序中实现本文描述的流程	重启前进行资源回收	保证复位

9.11.1.2 应用场景

当用户需要对单板进行彻底复位，并需要尽可能地保护系统的稳定和文件系统数据的完整性时，即

可采用优雅重启的方式来对系统复位。

9.11.1.3 内核配置

无需内核配置。

9.11.1.4 运行使用

用户可以根据业务的实际需要，按照优雅重启的设计思路编写应用代码，就可以实现系统的优雅重启。考虑到系统复位时需要回收资源，所以本功能不建议在业务进程中直接执行重启流程，而应采用一个独立的进程执行重启流程的方式（如：先 `fork` 或 `vfork` 一个子进程，在该子进程中执行下述流程），并设置该进程为高优先级的实时进程，保证其能在创建后立即运行。

使用 `vfork` 时，子进程共享父进程的地址空间，减少了新资源的分配，适合在对内存资源比较紧张的场景下使用；它的缺点是，父进程必须在子进程执行完之后才能继续执行，像下面的重启进程执行时间可能会比较长，父进程应该把其它事情都做完了再创建下面这个子进程，如果流程上不能设计这么设计，用 `fork` 更好。

下面将详细说明优雅重启的整个流程，其详细流程图如下所示：

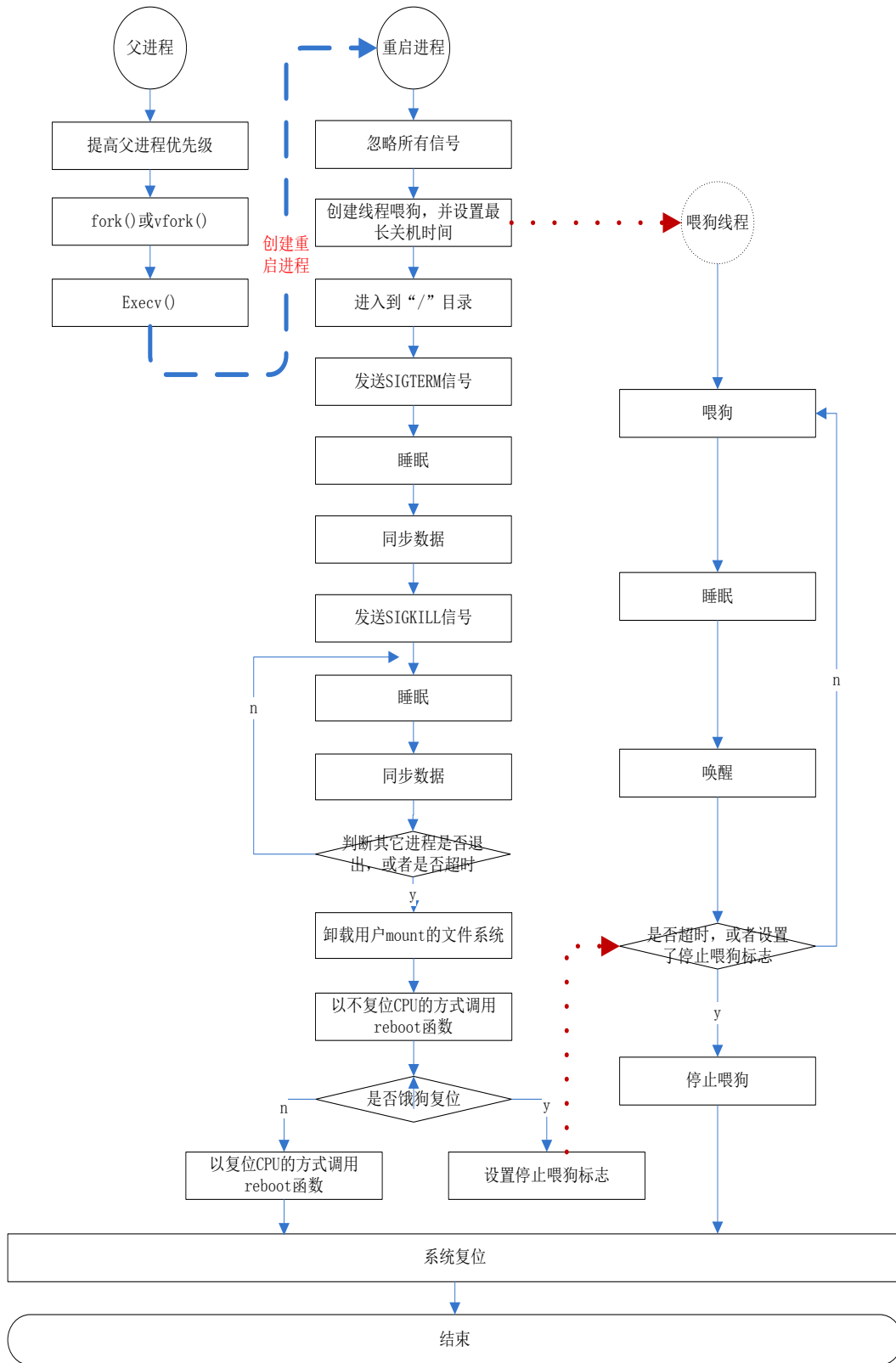


图 9-11 优雅重启运行流程

■ 创建重启进程

为了防止被别的进程抢占，在执行重启流程前，需要把重启进程修改为实时进程，调度策略为 SCHED_FIFO，并设置成最高优先级 99。建议采用先把父进程提升为实时进程，然后调用 fork 创建子进程，最后通过调用 execv 来得到一个重启进程的方式。

■ 忽略所有信号

调用 signal() 函数，忽略所有信号；同时需要注意，由于 SIGKILL、SIGSTOP 信号无法忽略，所以，此时需要注意避免别的进程向此重启进程发送这两个信号，否则流程将被终止。

■ 关闭当前进程所有打开的文件

虽然重启进程本身并没有打开文件，但是 fork 创建的子进程会继承父进程的打开的文件句柄。这里需要通过读 /proc/XXX（当前任务 pid）/fd 目录，获得已打开文件的 fd，然后将其全部关闭。

■ 创建线程喂狗

由于在后面的流程中，业务原来的喂狗进程会被杀死，如果没有其他流程继续喂狗，将会导致单板饿狗复位，因此，在此处需要创建一个线程进行定时喂狗操作。这里不建议采用信号方式进行喂狗，因为进程有可能被阻塞而收不到信号，采用线程的方式可以保证喂狗的及时。

考虑到重启进程可能长时间被阻塞的情况，建议采用系统 reboot 复位的模式，也创建一个超时线程(喂狗线程)，并设置最长关机时间。在超时后，直接调用 reboot() 函数进行 CPU 复位。

创建喂狗线程前，最好调 sync 同步文件数据，防止在设置最长关机时间后需要再同步的脏文件数据过多导致的在系统重启时脏数据还没有同步完的情况。

■ 进入到 “/” 根目录

由于后面涉及到文件系统的卸载，为了避免由于所在路径不恰当导致 umount 失败，所以提前进入到 “/” 根目录。

■ 发送 SIGTERM 信号

向除 1 号进程以外的所有其它进程发送 SIGTERM 信号。SIGTRRM 信号的作用是通知目标进程终止，该信号可以被捕获、阻塞和忽略。建议用户注册该信号的处理函数，以主动进行业务进程退出前的善后处理，例如主动调用 fsync、datasync 完成数据的写回，以保证数据的完整，以及关闭本进程打开的文件等。

■ 睡眠

在发送 SIGTERM 信号以后，关机进程调用 sleep 函数，进入睡眠状态，时间为 1s，以便为其它进程 SIGTERM 信号处理以及退出留出时间。

■ 同步数据

Sync 将高速缓存中的数据写入物理磁盘。

■ 发送 SIGKILL 信号

向除 1 号进程以外的所有其它进程发送 SIGKILL 信号。与 SIGTERM 不同，SIGKILL 信号属于致命信号，不能被捕获、阻塞和忽略。内核收到该信号后，将强制杀死目标进程，回收资源。

该信号就是告诉进程，“不管您在做什么，立刻停止在那里”。当一个进程正在写某个文件时，如果收到 SIGKILL 信号，它会立刻停止写操作，之前已经写到内核缓存中的数据会在后面的步骤中被刷到磁盘，否则数据会丢失。

■ 再次睡眠

在发送 SIGKILL 信号以后，关机进程调用 sleep 函数，进入睡眠状态，时间为 3s，等待内核杀死其它进程。

■ 再次同步数据

进程结束后，会有一些数据残留在内存中，来不及写回磁盘，比如文件数据，因此在内核杀死进程以后，有必要再次进行数据同步。

■ 判断其它进程是否退出

通过读取 /proc 目录内列出的进程数量判断进程是否退出完成。由于内核态进程及部分用户态进程是不能杀死的，如 init 进程。因此，我们判断进程退出完成的标准是比较 1s 前的进程数量和当前的进程数量是否发生变化。如果变化，则认为进程尚未退出完成，继续循环判断，超时时间为 3s；如果没有变化，则认为退出完成，往下执行。

这里通过读 /proc 的方式判断进程数与 busybox 中 ps 命令的实现方法类似，如果要准确判断当前内核中的进程数，需要内核提供相关接口，例如：在内核中可以通过 for_each_process 宏来遍历所有进程。对于一般应用，读 /proc 目录的方式已经足够了。

因为前面发送 kill 信号时，可能正在创建的新子进程没有收到 kill 信号，针对这种情况，比较每次在 /proc 目录下的最大进程号是否变化，若是则再对所有进程发送一次 kill 信号，并且在这种情况下即使 1s 前后的检查到的进程数量相等也不认为其它进程退出，继续下一个循环检查其它进程是否退出。

■ 卸载文件系统

通常在用户进程没有把所有文件关闭完的情况下，调 `umount` 卸载文件系统时会返回 `busy` 失败结果；如果 `umount` 失败，后面的流程 `reboot` 时，很大可能还有进程正在写磁盘，导致系统重启时文件系统的一致性得不到保证。

所以针对应用进程 `mount` 的每个目录，循环调 `umount` 正常卸载文件系统直到卸载成功为止才走后面的流程。有多个挂载目录时，需要注意 `umount` 的顺序，见 5.1.2 中的注意事项。

■ 执行 `reboot` 函数

以不复位 CPU 的方式调用 `reboot()` 函数。如前所述，操作系统 `sys_reboot` 系统调用会调用内核各模块，以及各总线、驱动等注册的复位回调函数，因此我们认为即使在饿狗复位方式下，调用 `reboot` 函数也是必要且有重要意义的。

■ 判断重启方式

对于没有看门狗的系统，则再次以复位 CPU 的方式调用 `reboot` 函数复位系统。对于有看门狗的系统，则以看门狗方式复位。

■ 设置停止喂狗标志

对于采用看门狗复位的系统，设置停止喂狗标志，然后进入死循环，等待看门狗复位系统。

另外，为了保证系统的可靠复位，避免极端情况下，由于关机流程长时间被阻塞而导致系统不能复位的情况，在喂狗线程中设定最长关机时间，一旦超过这个时间阈值，线程就会停止喂狗操作，等待看门狗超时，复位系统。

9.11.1.5 注意事项

1) 在 5.1.1.12 中，卸载文件系统时，不能越级卸载文件系统。先卸载掉子文件系统，才能去卸载父文件系统；在卸载 NFS 服务器端已经输出的本地文件系统时，先卸载掉 NFS 客户端，再用 `exportfs` 命令关闭 NFS 对应的输出目录，才能卸载 NFS 服务器端成功。

2) 必须等待所有的 I/O 都同步完成，优雅重启时才不会破坏文件系统的一致性，但是因为进程不能及时被调度到、需要同步的脏数据量多 I/O 没有全部同步完成、最长关机时间设置太短、存储设备端不响应等原因，最长关机时间到时，优雅重进程的程可能还没有走完，所以优雅重启在某些特殊情况下还是存在重启时文件系统一致性被破坏的风险，重启后文件系统自身或借助 `fsck` 工具将文件系统修复好后才能正常使用文件系统。

3) 优雅重启的时间长短的合理设置对此方案的实施效果影响很大，建议一般情况下设置在 5 分钟以

上，实际设置时请根据下面的原则进行设置：

- ✧ 根据业务硬性要求系统重启的最长时间上限，关机时间应该尽量长些；
- ✧ 优雅重启前内存中的脏数据量大的话，关机时间应该长些；在 2.2 节方案描述中先做 sync 的目的就是在设置最长关机时间前把之前在内存中已经存在的脏数据全部同步到内存中，减少后面同步脏数据的数量，从而减少同步脏数据的时间；
- ✧ 存储设备的访问速度慢的，关机时间应该长些；
- ✧ 应用进程多的情况下，关机时间应该长些。

9.11.1.6 编程范例

以下是优雅重启的 demo 代码，仅供参考。

1) Execv.c: 新建优先级为 99 的实时进程，并创建 graceful_reboot 子进程（后面的示例）来完成优雅重启流程。

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <sched.h>
#include <unistd.h>

char *clean_env[] = {
    "HOME=/",
    "PATH=/bin:/usr/bin:/sbin:/usr/sbin",
    "TERM=dumb",
    NULL,
};

extern char **environ;

/* **argv 由用户指定要卸载的各个目录*/
int main(int argc, char **argv)
{
    extern int      getopt();
    int is_reboot = 0;
    int c, i;
    int last = 0, now = 0;
```

```
int pid;
struct sched_param param;

param.sched_priority = 99;
if (sched_setscheduler(0, SCHED_FIFO, &param) == -1) {
    fprintf(stderr, "pid = %d,sched_setscheduler() failed and error is :%s.\n",getpid(),
    strerror(errno));
}

if ((pid = fork()) < 0) {
    printf("create process failed.\n\n");
} else if (pid == 0) {
    execv("./graceful_reboot", argv);
    printf("child process fail. errno %d", errno);
}

i = 0;
while (1) {
    printf("this is parent process.%d\n", ++i);
    sleep(10);
}

return 0;
}
```

2) graceful_reboot.c: 优雅重启程序。

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <time.h>
#include <sys/reboot.h>
```

```
#include <sys/wait.h>
#include <stdarg.h>
#include <string.h>
#include <dirent.h>
#include <sched.h>
#include <time.h>
#include <sys/time.h>
#include <pthread.h>
#include <mntent.h>
#include <inttypes.h>
#include <netdb.h>
#include <setjmp.h>
#include <stddef.h>
#include <sys/poll.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <termios.h>
#include <utime.h>
#include <limits.h>
#include <sys/param.h>
#include <sys/mount.h>
#ifdef RB_HALT_SYSTEM
#   define BMAGIC_HALT          RB_HALT_SYSTEM
#else
#   define BMAGIC_HALT          RB_HALT
#endif
#define BMAGIC_REBOOT          RB_AUTOBOOT
#define LINUX_REBOOT_CMD_PREPARE          0xABCD9876
#define LINUX_REBOOT_CMD_DO_RESTART       0xA AFF9900
#define WATCHDOG_TIMEOUT_SEC              60 * 5
#define WATCHDOG_INTERVAL_SEC             1
#define ENABLE_FEATURE_MOUNT_LOOP         1
#define LOOP_CLR_FD                        0x4C01
#define ENABLE_FEATURE_CLEAN_UP            1
#define ENABLE_WATCHDOG_RESTART            1
```

```
char *clean_env[] = {
    "HOME=/",
    "PATH=/bin:/usr/bin:/sbin:/usr/sbin",
    "TERM=dumb",
    NULL,
};

extern char **environ;

static int stop_feed_watchdog = 0;

const char * bb_path_mtab_file = "/proc/ mounts";
/*dir:ĪÇжÔµκ÷Ķ¼*/

int umount_dir(int num,char *dir[])
{
    struct mntent *me;
    int count = 0;
    FILE *fp;
    struct mtab_list {
        char *dir;
        struct mtab_list *next;
    }*m,*temp;
    m = NULL;
    fp = setmntent(bb_path_mtab_file, "r");
    if (!fp) {
        printf("open file error.\r\n");
    } else {
        while ((me = getmntent(fp))) {
            for(count = 0; count < num; count++)
                if(strcmp(dir[count],me->mnt_dir) == 0){
                    m = malloc(sizeof(struct mtab_list));
                    m->next = m;
                    m->dir = strdup(me->mnt_dir);
                    break;
                }
        }
        endmntent(fp);
    }
    while (m) {
        if (umount(m->dir))
```

```
        continue;

        temp = m;
        m = m->next;
        free(temp);

        sleep(1);
    }
    return EXIT_SUCCESS;
}

/*
 *    Show usage message.
 */
void usage(void)
{
    fprintf(stderr,
        "Usage:\t shutdown [-rh]\n"
        "\t\t -r:      reboot\n"
        "\t\t -h:      halt.\n");
    exit(1);
}

int new_process = 0;
int process_statistic(void)
{
    DIR *d;
    struct dirent *file;
    struct stat sb;
    int pid = 0;
    int count_pid = 0;
    static int max_pid = 0;

    if((d = opendir("/proc")) == NULL) {
        printf("error\n");
        return 0;
    }
}
```

```
new_process = 0;
while((file = readdir(d)) != NULL) {
    if(file->d_name[0] == '.')
        continue;
    if(isdigit(file->d_name[0]) == 0)
        continue;
    sscanf(file->d_name, "%d", &pid);

    if(pid > max_pid){
        max_pid = pid;
        new_process = 1;
    }
    //printf("%s\t %d %d\r\n", file->d_name, pid, count_pid);
    if (pid > 0) {
        count_pid++;
        pid = 0;
    }
}

closedir(d);
printf("count_pid:%d\r\n", count_pid);

return count_pid;
}

static void loop_forever(void)
{
    while (1)
        sleep(1);
}

void feed_watchdog(void)
{
    static int count = WATCHDOG_TIMEOUT_SEC;
    int i = 0;
    while (1) {
        //feed watchdog
        sleep(WATCHDOG_INTERVAL_SEC);
        if (stop_feed_watchdog == 1) {
```

```
        //printf("stop feeding watchdog. %d\n", i++);
        continue;
    }
    //printf("This is a pthread. %d\n", i++);
}
}

/*
 *
 */
static int close_current_fds(void)
{
    struct dirent *entry;
    DIR *dir;
    char dirname[32];
    int i=0,fd;
    pid_t pid = getpid();
    sprintf(dirname, "/proc/%d/fd", (int)pid);
    dir = opendir(dirname);
    if (!dir)
        return -1;
    while ((entry=readdir(dir))!=NULL)
    {
        if (!strcmp(entry->d_name, ".") || !strcmp(entry->d_name, ".."))
            continue;
        fd = strtol(entry->d_name, NULL, 10);
        if (fd > 2)
            close(fd);
    }
    closedir(dir);
    printf("close_current_fds complete!\n");
    return 0;
}

/*
 *      API for application
 */
```



```
int main(int argc, char *argv[])
{
    int c, i, ret;
    int last = 0, now = 0;
    pthread_t id;
    signal(SIGQUIT, SIG_IGN);
    signal(SIGCHLD, SIG_IGN);
    signal(SIGHUP, SIG_IGN);
    signal(SIGTSTP, SIG_IGN);
    signal(SIGTTIN, SIG_IGN);
    signal(SIGTTOU, SIG_IGN);
    ret = close_current_fds();
    if (ret != 0)
        printf("close_current_fds failed!\n");
    sync();

    ret = pthread_create(&id, NULL, (void *) feed_watchdog, NULL);
    if (ret != 0) {
        printf("Create pthread error!\n");
    }
    chdir("/");

    fprintf(stderr, "sending all processes the TERM signal...\n");
    (void) kill(-1, SIGTERM);
    sleep(1);
    sync();
    fprintf(stderr, "sending all processes the KILL signal...\n");
    (void) kill(-1, SIGKILL);
    now = process_statistic();
    if (new_process)
        (void) kill(-1, SIGKILL);
    for (i = 3; i > 0; i--) {
        last = now;
        sleep(1); /* Give init the chance to collect zombies. */
        now = process_statistic();
        sync();
        if (last == now && !new_process)
```

```
        break;
    }
    fprintf(stderr, "unmounting all file systems\r\n");
    umount_dir(argc-1,argv);
    reboot(LINUX_REBOOT_CMD_PREPARE);
    if (!ENABLE_WATCHDOG_RESTART) {
        reboot(LINUX_REBOOT_CMD_DO_RESTART);
    }
    stop_feed_watchdog = 1;
    loop_forever();

    return 0;
}
```

9.12 协议栈增强

9.12.1 原始套接字接收外发报文（LOP）

9.12.1.1 功能描述

CGEL 提供对发出去的各种协议网络数据包进行回环接收的功能，并且针对各协议单独生效。因此，CGEL 内核在对外发送各种协议数据包时，将感兴趣的网络协议的数据包向本地复制发送一份，并且利用 PF_PACKET 域在二层对这些数据包进行回环接收。

9.12.1.2 应用场景

CGEL 提供对发出去的各种协议网络数据包进行回环接收的功能，并且对于各种协议能单独实现该回环功能。

9.12.1.3 内核配置

配置宏选项 CONFIG_PACKET_LOP。

在 menuconfig 中配置如下：

Networking ---->

```
[*]Networking support
Networking options--->
    [*]Packet socket
    [*]Packet socket: Loopback the Outgoing Packets for Non ETH_ALL so
```

9.12.1.4 使用方法

➤ 设置选项

在内核配置开启 LOP 功能后，需要在代码中加入如下内容来使用 LOP。

```
optval = 1; /* optval = 0 when disable lop */
setsockopt(fd, SOL_PACKET, PACKET_LOP, &optval, sizeof(int));
```

➤ 获取选项

如需要获取 LOP 功能是否开启可使用如下代码：

```
getsockopt(fd, SOL_PACKET, PACKET_LOP, &optval, &optlen);
```

其中 PACKET_LOP 暂时是放在用户程序中定义的。

```
#define PACKET_LOP    249 /* Loopback the Outgoing Packets */
```

9.12.2 RPS/RFS

目前单个 CPU 的处理能力已经难以适应高性能网卡接收/发送报文的速度，因此提高多核处理器数据报文并行处理能力则成为 SMP 系统下提高网络吞吐量的必要手段。

RPS：全称是 Receive Packet Steering，这是 Google 工程师 Tom Herbert 提交的内核补丁。这个 patch 采用软件模拟的方式，实现了多队列网卡所提供的功能，把不同流的数据包分发给不同的 CPU 核来处理，分散了在多 CPU 系统上数据接收时的负载，把软中断分到各个 CPU 处理，而不需要硬件支持，大大提高了网络性能。

RFS：全称是 Receive Flow Steering，这也是 Tom 提交的内核补丁，它是用来配合 RPS 补丁使用的，是 RPS 补丁的改进扩展补丁。它不仅能够把同一流的数据包分发给同一个 CPU 核来处理，还能够分发给其“被期望”的 CPU 核来处理，提高 cache 的命中率。

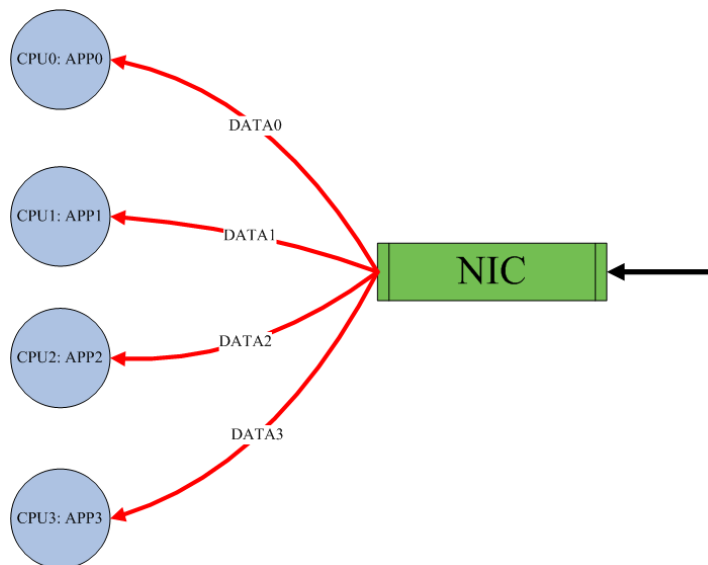
9.12.2.1 功能描述

内核现在网卡的驱动支持两种模式，一种是 NAPI，一种是非 NAPI 模式：

- 在 NAPI 中，中断收到数据包后调用 `__napi_schedule` 调度软中断，然后软中断处理函数中会调用注册的 `poll` 回调函数中调用 `netif_receive_skb` 将数据包发送到 3 层，没有进行任何的软中断负载均衡；
- 在非 NAPI 中，中断收到数据包后调用 `netif_rx`，这个函数会将数据包保存到 `input_pkt_queue`，然后调度软中断，这里为了兼容 NAPI 的驱动，他的 `poll` 方法默认是 `process_backlog`，最终这个函数会从 `input_pkt_queue` 中取得数据包然后发送到 3 层。

可见，在目前的处理机制中，不管是 NAPI 还是非 NAPI 的话都无法做到软中断的负载均衡，因为软中断此时都是运行在在硬件中断相应的 CPU 上。也就是说如果始终是 CPU0 相应网卡的硬件中断，那么始终都是 CPU0 在处理软中断，而此时 CPU1 就被浪费了，因为无法并行的执行多个软中断。

针对此问题，RPS/RFS 补丁解决了在多核情况下网络协议栈的软中断的负载均衡。这里的负载均衡也就是指能够将软中断均衡的放在不同的 CPU 核心上运行。RPS/RFS 的基本原理是根据数据包的源地址，目的地址以及目的和源端口（这里它是将两个端口组合成一个 4 字节的无符号数进行计算的）计算出一个 hash 值，然后根据这个 hash 值来选择软中断运行的 CPU。从上层来看，也就是说将每个连接和 CPU 绑定，并通过这个 hash 值，来均衡软中断在多个 CPU 上。



9.12.2.2 应用场景

网卡可以接收/发送包的速度已经开始让宿主机的 CPU 难以跟上。而由于单个 CPU 的处理能力也慢慢变得难以提升，就需要把数据包分布到多个 CPU 上处理。

9.12.2.3 内核配置

配置宏选项 CONFIG_RPS。

在 menuconfig 中配置如下：

```
Networking --->
  [*]Networking support
  [*]Net RPS feature
```

9.12.2.4 使用方法

内核自动实现网络的优化。用户也可通过/sys/class/net/<device>/queues/rx-<n>接口设置把数据放到哪个 cpu 队列执行。

■ /sys/class/net/<device>/queues/rx-<n>/rps_cpus 接口

用法：

```
echo "ff" >/sys/class/net/<device>/queues/rx-<n>/rps_cpus
```

功能：通过给此接口设置一个 mask 指定特定网卡 device 或者网卡上的某个接收队列 rx-<n>（如果支持多接受队列的话）可以包交给特定的几个 CPU 处理。

说明：包的 hash 值和 rps_cpus 的值共同决定了该数据包会放入哪个 CPU 的 backlog 队列。

注意：

- 一个数据包到达后，中断所到达的 CPU 负责分配，这时会通过 SKB 内容计算 hash，由于是第一次读这个 SKB，所以发生 cache miss 正常。但是之后这个数据包可能会被分配到其他 CPU 上处理，对于另一个 CPU 来说，SKB 仍然是一个未知内容，所以 cache miss 又会发生，这次的 cache miss 就是 RPS 带来的新开销。不过幸好，很多网卡都天生支持计算这样的 hash，驱动只需要把该值存入 SKB，就可以提高 cache hit，以及计算机整体性能。
- rps_cpus 可以在运行时被动态改变，但是这样的改动可能会导致包的乱序。因为改动前某个 transaction 的数据包都是由某个 CPU 处理，而 rps_cpus 改变，可能导致该 transaction 的包被发往另一个 CPU 上处理，如果后到的包在第二个 CPU 上被处理的速度快于先到的包在第一个 CPU 上被处理的速度，就会发生乱序。所以尽可能不要频繁改变 rps_cpus 的值。

■ /sys/class/net/<device>/queues/rx-<n>/rps_flow_cnt 接口

用法：

```
echo 1024>/sys/class/net/<device>/queues/rx-<n>/rps_flow_cnt
```

功能：设置和获取每个队列 rxqueue 的数据流表 rps_dev_flow_table 的 rps_dev_flow 总数。

说明：相对于 RPS 来说就是添加了一个 CPU 的选择，也就是说我们需要根据应用程序的 CPU 来选择软中断需要被处理的 CPU。这里做法是当调用 recvmmsg 的时候，应用程序的 CPU 会被存储在一个 hash table 中，而索引是根据 socket 的 rxhash 进行计算的。而这个 rxhash 就是 RPS 中计算得出的那个 skb 的 hash 值。

可是这里会有一个问题，那就是当多个线程或者进程读取相同的 socket 的时候，此时就会导致 cpu id 不停的变化，从而导致大量的 OOO 的数据包（这是因为 cpu id 变化，导致下面软中断不停的切换到不同的 CPU，此时就会导致大量的乱序的包）。

而 RFS 是如何解决这个问题的呢，它做了两个表 rps_sock_flow_table 和 rps_dev_flow_table，其中第一个 rps_sock_flow_table 是一个全局的 hash 表，针对 socket 的，映射了 socket 对应的 CPU，这里的 CPU 就是应用层期待软中断所在的 CPU。rps_sock_flow_table 的总数可通过 /proc/sys/net/core/rps_sock_flow_entries 设置。

```
struct rps_sock_flow_table {
    unsigned int mask;
    //hash 表
    u16 ents[0];
};
```

可以看到它有两个域，其中第一个是掩码，用于来计算 hash 表的索引，而 ents 就是保存了对应 socket 的 cpu。

然后是 rps_dev_flow_table，这个是针对设备的，每个设备队列都含有一个 rps_dev_flow_table（这个表主要是保存了上次处理相同链接上的 skb 所在的 cpu），这个 hash 表中每一个元素包含了一个 cpu id，一个 tail queue 的计数器，这个值是一个很关键的值，它主要就是用来解决上面大量 OOO 的数据包的问题的，它保存了当前的 dev flow table 需要处理的数据包的尾部计数。

```
struct netdev_rx_queue {
    struct rps_map *rps_map;
    //每个设备的队列保存了一个 rps_dev_flow_table
    struct rps_dev_flow_table *rps_flow_table;
    struct kobject kobj;
    struct netdev_rx_queue *first;
    atomic_t count;
} ____cacheline_aligned_in_smp;
```

```
struct rps_dev_flow_table {
    unsigned int mask;
    struct rcu_head rcu;
    struct work_struct free_work;
    //hash 表
    struct rps_dev_flow flows[0];
};

struct rps_dev_flow {
    u16 cpu;
    u16 fill;
    //tail 计数。
    unsigned int last_qtail;
};
```

由于大量的 OOO 的数据包的引起是因为多个进程同时请求相同的 socket，而此时会导致这个 socket 对应的 cpu id 不停的切换，然后软中断如果不做处理，只是简单的调度软中断到不同的 cpu，就会导致顺序的数据包被分发到不同的 cpu，由于是 smp，因此会导致大量的 OOO 的数据包。而在 RFS 中是这样解决这个问题，在 soft_net 中添加了 2 个值 input_queue_head 和 input_queue_tail，然后在设备队列中添加了 rps_flow_table，而 rps_flow_table 中的元素 rps_dev_flow 包含有一个 last_qtail，RFS 就通过这 3 个值来控制乱序的数据包。

这里为什么需要 3 个值呢，这是因为每个 cpu 上的队列的个数 input_queue_tail 是一直增加的，而设备每一个队列中的 flow table 对应的 skb 则是有可能被调度到另外的 cpu，而 dev flow table 的 last_qtail 表示当前的 flow table 所需要处理的数据包队列（backlog queue）的尾部队列计数，也就是说当 input_queue_head 大于等于它的时候说明当前的 flow table 可以切换了，否则的话不能切换到进程期待的 cpu。

每个接收队列若要启用 RFS，需要设置下面两个参数，参数的值会被进位到最近的 2 的幂次方值。（例如，参数的值是 7，则实际有效值是 8；参数是值 32，则实际值就是 32。）在设置时注意，流计数依赖于期待的有效的连接数，这个值明显小于连接总数。经测试，rps_sock_flow_entries 设置成 32768，在中等负载的服务器上工作效率明显提升。对于单队列设备，单队列的 rps_flow_cnt 值建议被配置成与 rps_sock_flow_entries 相同。对于一个多队列设备，每个队列的 rps_flow_cnt 建议被配置成 rps_sock_flow_entries/N，N 是队列总数。例如，如果 rps_sock_flow_entries 设置成 32768，并且有 16 个接收队列，每个队列的 rps_flow_cnt 最好被配置成 2048。

```
/proc/sys/net/core/rps_sock_flow_entries
```

```
/sys/class/net/<dev>/queues/rx-<n>/rps_flow_cnt
```

■ 示例（每队列绑定到所有 cpu 核上）

```
/sys/class/net/eth0/queues/rx-0/rps_cpus 000000ff
/sys/class/net/eth0/queues/rx-1/rps_cpus 000000ff
/sys/class/net/eth0/queues/rx-2/rps_cpus 000000ff
/sys/class/net/eth0/queues/rx-3/rps_cpus 000000ff
/sys/class/net/eth0/queues/rx-4/rps_cpus 000000ff
/sys/class/net/eth0/queues/rx-5/rps_cpus 000000ff
/sys/class/net/eth0/queues/rx-6/rps_cpus 000000ff
/sys/class/net/eth0/queues/rx-7/rps_cpus 000000ff

/sys/class/net/eth0/queues/rx-0/rps_flow_cnt 4096
/sys/class/net/eth0/queues/rx-1/rps_flow_cnt 4096
/sys/class/net/eth0/queues/rx-2/rps_flow_cnt 4096
/sys/class/net/eth0/queues/rx-3/rps_flow_cnt 4096
/sys/class/net/eth0/queues/rx-4/rps_flow_cnt 4096
/sys/class/net/eth0/queues/rx-5/rps_flow_cnt 4096
/sys/class/net/eth0/queues/rx-6/rps_flow_cnt 4096
/sys/class/net/eth0/queues/rx-7/rps_flow_cnt 4096

/proc/sys/net/core/rps_sock_flow_entries 32768
```

此例中，软中断负载已经能分散到各个 CPU 核上，有效利用了多 CPU 的特性，大大提高了系统的网络性能。

9.12.3 IP 地址冲突被动检测

9.12.3.1 功能描述

CGEL 支持 IP 地址冲突被动检测功能：内核协议栈收到 ARP 时，检查其源 IP 地址是否和自己配置的某个 IP 地址相同，如果相同则使用 `printk` 打印出现 IP 地址冲突。

9.12.3.2 应用场景

可能出现 IP 地址冲突的开发、测试、工程中。

9.12.3.3 内核配置

配置宏选项 CONFIG_DUPLICATE_IP_DETECTION。

在 menuconfig 中配置如下：

```
Networking --->
  [*]Networking support
  Networking options--->
    [ ]Use INTERPEAK TCP/IP networking
    [*]Use native TCP/IP networking
    [*]Duplicate IP and MAC address detection
```

9.12.3.4 使用方法

配置内核后，当检测到 IP 冲突，自动打印冲突信息。

9.12.4 IPV6 地址保存

9.12.4.1 功能描述

标准 Linux 系统中，IPv6 端口在系统 DOWN 掉过程中，会删除所有地址，而在 UP 过程中只自动生成 linklocal 地址，从而导致端口状态变化后，用户配置的地址丢失。IPV6 地址保存功能实现当用户将网卡 DOWN 掉后，可以将该网卡上原所有的 IPV6 地址保存至保留地址链表，网卡端口 UP 后，该地址依然可用。

9.12.4.2 应用场景

当用户使用 IPV6 时，想要在网卡 DOWN 掉后，再 UP 时，恢复之前对该网卡配置的所有 IPV6 的地址信息。

9.12.4.3 内核配置

配置宏选项 CONFIG_IPV6，同时，需要 busybox 支持 IPV6。

在 menuconfig 中配置如下：

```
Networking --->
  [*]Networking support
```

```
Networking options--->
  [ ]Use INTERPEAK TCP/IP networking
  [*]Use native TCP/IP networking
  [*]IPv6:Multiple Routing Tables
```

9.12.4.4 使用方法

正常的 IPV6 操作，无特殊操作。

9.12.5 arp 老化时间设置参数立即生效

9.12.5.1 功能描述

目前，IPV4 使用 arp 协议作为邻居协议，并在 proc 文件系统下提供 `proc/sys/net/ipv4/neigh/default/base_reachable_time` 文件，用户可通过修改该文件来调整 arp 条目的老化时间，默认时间为 30 秒。但在使用本功能前，若设置了超时时间，则系统无法立即生效，需要等待大约 5 分钟时间。使用该功能后，arp 条目的老化时间可以立即生效。

9.12.5.2 应用场景

对于使用内核 IPV4 协议栈，并且需要调整 arp 老化时间的情况。

9.12.5.3 内核配置

配置宏选项 `CONFIG_REACHABLE_TIME_SETSYNC`。

在 menuconfig 中配置如下：

```
Networking --->
  [*]Networking support
  Networking options--->
    [ ]Use INTERPEAK TCP/IP networking
    [*]Use native TCP/IP networking
    [*]Set base_reachable_time sync to reachable_time
```

9.12.5.4 使用方法

通过 `echo xxx > /proc/sys/net/ipv4/neigh/[dev]/base_reachable_time` 或 `sysctl` 接口设置

base_reachable_time。

9.12.6 多网卡绑定

9.12.6.1 功能描述

网卡绑定技术，把多个网络接口（network interface）组合成一个逻辑的“bonded”接口，具有如下优点：

- 提供了一种廉价、有效、透明的方法扩展网络设备，在利用原有资源的情况下，通过网卡 bonding 技术，使得虚拟网卡几乎拥有所有网卡的带宽，提高带宽的同时提高了服务器的可用性；
- 可有效的避免传输时出现的单点故障，网卡 bonding 设备驱动会检测当前活动的网卡，以及到用户指定节点的网络路径，当检测到其中的一块物理网卡出现故障，数据包会自动发送到下一个可用的网卡上，而不会中断现有的用户连接，网卡恢复后会自动返回到链路聚集服务中；
- 可提高数据传输时的负载均衡，在网卡 bonding 设备驱动的传输算法中实现了网络传输负载在设备内所有网卡中共享，提高了利用效率。

9.12.6.2 内核配置

配置宏选项 CONFIG_BONDING。

在 menuconfig 中配置如下：

```
Device Drivers --->
  Network device support--->
    [*]Network device support
    [*]Bonding driver support
```

9.12.6.3 驱动配置

提供四种方式来配置 bonding 驱动支持。

1) 通过 Sysconfig 配置 bonding

适用于发行包里使用 sysconfig 的用户，且 sysconfig 必须能够支持 bonding，如，SuSE Linux Enterprise Server 9。

2) 通过 Initscripts 配置 bonding

适用于使用支持 bonding 的 initscripts 的发行包，比如 Red Hat Linux 9 或 Red Hat Enterprise Linux


version 3 或 4。

3) 通过 Ifenslave 手工配置 bonding

适用于某些发行包，它们的网络初始化脚本（sysconfig 或 initscripts 包）没有 bonding 相关的知识。SuSE Linux Enterprise Server 版本 8 就是这样的发行包。

4) 通过 Sysfs 手工配置 bonding

sysfs 接口允许在不卸载模块的情况下动态配置所有 bonds，它也可以在运行时增加和移除 bonds。Ifenslave 已经不再需要了，尽管它还被支持。

 **提示：**目前 CGEL 只支持通过 Sysfs 或 Ifenslave 方式手工配置 bonding。

9.12.6.4 使用方法

■ 增加/移除一个新的 bond:

```
echo +bond0 > /sys/class/net/bonding_masters
echo -bond0 > /sys/class/net/bonding_masters

echo +bond1 > /sys/class/net/bonding_masters
echo -bond1 > /sys/class/net/bonding_masters
```


■ 显示所有存在的 bonds:

```
cat /sys/class/net/bonding_masters
bond0 bond1
```

■ 增加和移除 Slaves

```
ifconfig bond0 ip up
echo +eth0 > /sys/class/net/bond0/bonding/slaves           //把 eth0 加入 bond(bond0)
# echo -eth0 > /sys/class/net/bond0/bonding/slaves         //把 eth0 从 bond(bond0)卸载
ifconfig bond1 ip up
```

```
echo +eth1 > /sys/class/net/bond1/bonding/slaves          //把 eth1 加入 bond(bond1)
# echo -eth1 > /sys/class/net/bond0/bonding/slaves        //把 eth1 从 bond(bond1)卸载
```

 提示：bond 必须在 slave 加入之前启动，所有 slave 必须在 bond 接口断开前移除。不要再使用

ifconfig 去启动 eth0 ,eth1 ,即不要有类似的操作 ,如 ifconfig eth0 192.168.2.111 up 和 ifconfig eth1 192.168.2.112 up 等等。


■ 改变 Bond 的配置

把 bond0 配置为 balance-alb 模式：

```
ifconfig bond0 down
echo 6 > /sys/class/net/bond0/bonding/mode
```

或者

```
# echo balance-alb > /sys/class/net/bond0/bonding/mode
```

 提示：在修改模式前，请先断开 bond 接口。

■ 其它配置

在 bond0 上启用 MII 监控，使用 100ms 的时间间隔：

```
echo 100 > /sys/class/net/bond0/bonding/miimon
```

在 bond1 上启用 MII 监控，使用 100ms 的时间间隔：

```
echo 100 > /sys/class/net/bond1/bonding/miimon
```

在 bond0 上启用 ARP 监控，使用 100ms 的时间间隔：

```
echo 100 > /sys/class/net/bond0/bonding/arp_interval
```

在 bond1 上启用 ARP 监控，使用 100ms 的时间间隔：

```
echo 100 > /sys/class/net/bond1/bonding/arp_interval
```

 提示：如果 ARP 监控被启用，当 MII 监控启用时它会被禁止，反之亦然。

■ 查看到 bond0 的工作状态

```
cat /proc/net/bonding/bond0
```

■ 使用 ifenslave

➤ 添加 bond

```
echo +bond0 > /sys/class/net/bonding_masters  
echo -bond0 > /sys/class/net/bonding_masters
```

➤ 查看是否添加成功

```
cat /sys/class/net/bonding_masters  
bond0
```

➤ 设置模式

```
echo 100 > /sys/class/net/bond0/bonding/miimon
```

➤ 开启 bond


```
ifconfig bond0 ip up
```

➤ 绑定 eth0, eth1

```
ifenslave bond0 eth0 eth1
```

➤ 解除绑定

```
ifenslave -d bond0 eth0 eth1
```

 提示：不要在此过程中设置 eth0, eth1 的 IP。

9.12.7 网卡报文截获

9.12.7.1 功能描述

CGEL 内核支持网卡接收数据包的截获功能，可根据用户定义的过滤规则对数据包进行过滤，将用户感兴趣的数据发送给更上层的应用程序进行分析。该功能不支持 RPS 及网卡多队列。

应用程序可使用该功能来截获指定网卡收发的指定二层数据包。上层应用在用户态设置指定物理网卡的接收数据包 filter 规则，以及在内核态挂接收数据包的回调处理函数。在网卡接收到报文后，如果数据包满足 filter 规则的过滤条件就会调用相应的回调处理函数，根据返回值判断数据包是否已被回调处理函数截获。如果数据包已被截获，内核就直接丢弃，不再处理，否则继续将数据包交由上层协议栈处理。

9.12.7.2 应用场景

需要对网卡接收数据包进行筛选过滤的开发、测试、工程中。

9.12.7.3 内核配置

在 menuconfig 中配置选中【Net Packet Intercept】即可。

```
Networking support --->
  Net Packet Intercept
```

9.12.7.4 接口介绍

■ filter 规则的设置查询接口

接口：

```
syscall(int __NR_unify_syscall, unsigned int cmd, int ifindex, unsigned int opt, void *buf, unsigned int size)
```

功能：网卡数据包截获功能 filter 规则的设置/取消/查询接口（用户态）。

输入参数：

- __NR_unify_syscall：为当前架构的统一系统调用号；
- cmd：代表具体的子功能号，此处对应 SYSCALL_PACKET_INTERCEPT；
- ifindex，为要设置的物理网卡索引，不支持网卡别名、bond、网桥等，由上层保证有效性；

- opt: 选项如下;
 - ✧ --SET_INTERCEPT_RECVFILTER, 设置网卡接收报文时的截获 filter;
 - ✧ --SET_INTERCEPT_SENDFILTER, 设置网卡发送报文时的截获 filter;
 - ✧ --GET_INTERCEPT_RECVFILTER, 获取网卡接收报文时的截获 filter;
 - ✧ --GET_INTERCEPT_SENDFILTER, 获取网卡发送报文时的截获 filter。
- buf: 如果是设置 filter, 则为用户态传入的 filter 规则的地址, filter 定义为 struct sock_fprog, 使用同 setsockopt 的 SO_ATTACH_FILTER; 为 NULL 则表示取消 filter 设置。如果是获取 filter, 则用来保存内核返回的 filter;
- size, 为 buf 的大小。

返回值: 成功, 0; 失败, -1, 并显示错误码 (ENOSYS 无此系统调用; ENODEV 无此设备; EINVAL 非法参数; EPERM 获取 filter, 但对应的 filter 并未设置; EFAULTbuf 指针不可访问)。

说明: 若重复设置, 将覆盖前一次设置。

■ 回调注册接口 netdev_register_intercept_func

接口:

```
int netdev_register_intercept_func(char *ifname, intercept_func recv_func, intercept_func send_func)
```

功能: 注册网卡数据包截获功能的回调函数。

intercept_func 结构: 回调函数类型定义在 linux/netdevice.h, 引用时需包含该头文件。

```
typedef int (*intercept_func)(struct sk_buff *);
```

其中, 参数 sk_buff 为网络报文 buffer。返回值 PACKET_UNINTERCEPTED 为报文未被截获, 内核继续处理; PACKET_INTERCEPTED 为报文被截获, 内核将其丢弃。

输入参数:

- ifname: 为要设置的物理网卡名, 不支持网卡别名、bond、网桥等, 由上层保证有效性;
- recv_func: 接收报文的截获处理函数, 为 NULL 即不注册这个处理;
- send_func: 发送报文的截获处理函数, 为 NULL 即不注册这个处理。

返回值: 成功, 0; 失败, -ENODEV, 无此设备。

说明: 重复注册, 将覆盖前一次注册的处理函数。

■ 回调注销接口 `netdev_unregister_intercept_func`

接口：

```
int netdev_unregister_intercept_func(char *ifname, int which)
```

功能：注销网卡数据包截获功能的回调函数。

输入参数：

- `ifname`：为要设置的物理网卡名，不支持网卡别名、bond、网桥等，由上层保证有效性；
- `which`：如果设置了 `RECV_INTERCEPT_FUNC` 位，就注销接收报文的截获处理函数；如果设置了 `SEND_INTERCEPT_FUNC` 位，就注销发送报文的截获处理函数。

返回值：成功，0；失败，-ENODEV，无此设备。

9.12.7.5 使用方法

■ 报文被截获条件

- 1) 数据包的格式不满足用户指定的数据包过滤通过规则（`filter`），此条件满足后该数据包会传递给应用的回调处理函数；
- 2) 应用的回调函数决定把此报文截获。

■ 使用报文截获功能

用户使用该功能需进行 `filter` 规则的设置和回调处理函数的注册。数据包被应用截获后内核的协议层就不再对其可见，反之内核协议栈继续对其进行正常处理。

■ 停止报文截获功能

用户只需取消 `filter` 规则或注销回调处理函数即可停止报文截获。

9.12.8 按网口禁止 LINUX 协议栈报文收发

9.12.8.1 功能描述

提供在某网口上禁止 Linux 协议栈报文接收和发送的功能，在该网口上只允许 `ETH_P_ALL` 类型的链路层原始套接字接收和发送。此外还提供以下功能：

- 提供按网口禁止 LINUX 协议栈报文收发功能的设置接口；

- 提供按网口禁止 LINUX 协议栈报文收发功能的清除接口；
- 提供按网口禁止 LINUX 协议栈报文收发功能的查询接口。

9.12.8.2 应用场景

有些用户需要在虚拟化的单板环境上，需要同时运行项目私有协议栈和 LINUX 协议栈。其中单板的某些网口通过 ETH_P_ALL 类型的链路层原始套接字接收和发送项目私有协议栈的报文，这些网口只供项目私有协议栈使用。由于项目私有协议栈和 LINUX 协议栈共同管理一个网口，会造成报文上送到 LINUX 协议栈，这样会对业务产生影响。用户希望在某物理网口上实现禁止 Linux 协议栈本身的接收和发送报文功能，即只允许 ETH_P_ALL 类型的链路层原始套接字接收和发送报文，专门供项目私有协议栈使用。

用户项目通过如下命令创建链路层原始套接字来接收和发送数据，用以实现自定义协议栈的功能。这样网口上既有 Linux 协议栈，也有项目自定义协议栈。为了避免两者产生冲突，用户项目希望内核提供禁止某个网口发送和接收 linux 协议栈报文的功能，即报文不走协议栈，只走原始套接字。

```
socket(PF_PACKET, type, htons(ETH_P_ALL))
```

9.12.8.3 使用约束

1) 在使用此功能的网口上，只有 ETH_P_ALL 类型的链路层原始套接字可以接收和发送报文。其它类型的原始套接字不能接收和发送；

2) 该功能只针对真实物理网口和 bonding 口有效（如 eth0, eth3, bond0 等），不能在基于此物理网口的虚接口（如别名、VLAN 子接口等）上设置此功能。

9.12.8.4 内核配置

配置宏 CONFIG_PACKET_FOR_RAWSOCK，在 menuconfig 中配置如下：

```
Networking --->
[*] Netdevice packet only for raw socket
```

9.12.8.5 使用方法

用内核提供的统一系统调用方式对网口进行设置、清除和查询的功能。

1) 首先通过 /proc/nr_unify_syscall 接口查询该单板对应的统一系统调用号；

2) 再通过/proc/cmd_unify_syscall 接口查 SYSCALL_DEVICE_ONLYFOR_RAWSOCK 对应的子功能号;

3) 参考后面的接口说明提供的接口, 对网口设置、清除和查询此功能。

设置此功能后, 该网口只能用于原始套接字进行接收和发送。

9.12.8.6 接口介绍

■ 设置功能

接口:

```
ret = syscall(nr_syscall, id, ifindex, SET_DEVICE_RAWSOCK);
```

输入参数:

- nr_syscall: 为系统调用号, 可通过/proc/nr_unify_syscall 获取;
- id: 为子功能号, 可通过/proc/cmd_unify_syscall 查询 SYSCALL_DEVICE_ONLYFOR_;
- RAWSOCK: 对应的子功能号;
- ifindex: 表示网卡的编号, 可通过 ioctl 获取;
- SET_DEVICE_RAWSOCK: 表示设置网口的 IFF_ONLY_FOR_RAWSOCK 标志。

返回值:

- 0 表示操作成功;
- -1 表示失败, errno 为 ENODEV 表示网络设备不存在, errno 为 EINVAL 表示参数错误。

■ 清除功能

接口:

```
ret = syscall(nr_syscall, id, ifindex, CLEAR_DEVICE_RAWSOCK);
```

输入参数:

- nr_syscall: 为系统调用号, 可通过/proc/nr_unify_syscall 获取;
- id: 为子功能号, 可通过/proc/cmd_unify_syscall 查询 SYSCALL_DEVICE_ONLYFOR_;
- RAWSOCK: 对应的子功能号;
- ifindex: 表示网卡的编号, 可通过 ioctl 获取;
- GET_DEVICE_RAWSOCK: 表示清除网口的 IFF_ONLY_FOR_RAWSOCK 标志。

返回值:

- 0 表示操作成功;
- -1 表示失败, `errno` 为 `ENODEV` 表示网络设备不存在, `errno` 为 `EINVAL` 表示参数错误。

■ 查询功能

接口:

```
ret = syscall(nr_syscall, id, ifindex, GET_DEVICE_RAWSOCK);
```

输入参数:

- `nr_syscall`: 为系统调用号, 可通过 `/proc/nr_unify_syscall` 获取;
- `id`: 为子功能号, 可通过 `/proc/cmd_unify_syscall` 查询 `SYSCALL_DEVICE_ONLYFOR_`;
- `RAWSOCK`: 对应的子功能号;
- `ifindex`: 表示网卡的编号, 可通过 `ioctl` 获取;
- `GET_DEVICE_RAWSOCK`: 表示查询网口的 `IFF_ONLY_FOR_RAWSOCK` 标志。

返回值:

- 0 表示该网口未设置 `IFF_ONLY_FOR_RAWSOCK` 标志;
- 1 表示该网口设置了 `IFF_ONLY_FOR_RAWSOCK` 标志;
- -1 表示失败, `errno` 为 `ENODEV` 表示网络设备不存在, `errno` 为 `EINVAL` 表示参数错误。

第 10 章

开源 LICENSE 说明

10.1 GPL 简介

GPL 是 GNU 通用公共许可证的简称，为大多数的 GNU 程序和超过半数的自由软件所采用，Linux 内核就是基于 GPL 协议而开源。GPL 许可协议对在 GPL 条款下发布的程序以及基于程序的衍生著作的复制与发布行为提出了保留著作权标识及无担保声明、附上完整、相对应的机器可判读源码等较为严格的要求。GPL 规定，如果将程序做为独立的、个别的著作加以发布，可以不要求提供源码。但如果作为基于源程序所生著作的一部分而发布，就要求提供源码。

成都研究所 OS 平台基于开源社区研发提供适用于各产品线的嵌入式 Linux 产品及相关工具，从总体而言，主要应该遵循 GPL 许可协议的条款。Linux 是基于 GPL2.0 发布的开源软件，CGEL 是基于 Linux 内核进行修改完成的新作品，因此，根据 GPL 协议第 2 条的规定，CGEL 属于修改后的衍生作品，并且不属于例外情况。因此，CGEL 在发布时需开放源代码，具体形式需要符合 GPL 协议第 3 条的规定。

10.2 开发指导

为尽可能地保护公司在 Linux 方面的自有研发成果，特别是产品线的应用程序，同时顺从于 GPL 协议条款的规定，这里为大家提供一些开发指导原则供参考，以便有效地规避由于顺从 GPL 条款所带来的风险。

■ 应用程序尽可能运行于用户态

应用程序尽量放到用户态运行，应用程序对于内核的访问、功能使用主要通过信号、数据、文件系统、系统调用，基本属于松耦合，而且绝大部分系统调用都是通过 C 库封装进行，能比较充分地证明应用的独立性、可移植性，避免开源。

■ 以动态链接方式链接开源库

以 GLIBC 库为例，应用程序对于库的链接方式应尽量采用动态链接，这样可遵循 LGPL，避免公开源码。如果需要静态链接，则可以设计一个封装容器，与开源库静态链接在一起，这样只需开放封装容器源码。其余应用程序以动态链接方式与封装容器链接，只需要提供二进制文件。

■ 内核应用采用模块加载方式

内核应用可以采用静态链接、模块加载方式加入内。

静态链接方式是将所有的模块（包括应用）都编译到内核中，形成一个整体的内核映像文件。这种情况下，比较难以鉴别应用程序与内核的独立性、可移植性，按照 GPL 的规定，就很可能被要求开发源代码。

模块加载方式是指将上层应用独立地编译连接成标准 linux 所支持的模块文件 (*.mod)。运行时，首先启动 Linux 内核，然后通过 insmod 命令将各模块按照彼此间的依赖关系插入到内核。该方式下应用程序相对独立于 Linux 内核，有可能被证明为独立的作品，与 OS 无关，认为是纯粹的聚集行为，从而可以不用开放源码。

因此，对于运行于内核态的驱动和应用软件应尽量采用模块加载方式进行独立编译，通过模块的可动态装载、卸载，以及跨 OS 的可移植性来证明应用与 OS 的独立性。

10.3 源码获取方式

广东中兴新支点技术有限公司所发布的 CGEL 操作系统是基于开源软件 Linux 2.6 版本开发的，遵从 GPL 协议开放源代码。如果你需要源码，请发送源码申请邮件到 cgel@gd-linux.com 联系获取源码。