

Linux 环境编程 从应用到内核

高峰 李彬 著

Programming in Linux Environment
from Userspace to Kernel

- Linux领域第一本将应用编程与内核实现相结合的图书
- Linux环境编程的进阶指导，解析Linux接口的工作原理，帮助应用开发人员快速深入内核，掌握Linux系统运行机制



Linux 环境编程 从应用到内核

Programming in Linux Environment
from Userspace to Kernel

高峰 李彬 著



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

Linux 环境编程: 从应用到内核 / 高峰, 李彬著. —北京: 机械工业出版社, 2016.5
(Linux/Unix 技术丛书)

ISBN 978-7-111-53610-9

I. L… II. ①高… ②李… III. Linux 操作系统 IV. TP316.89

中国版本图书馆 CIP 数据核字 (2016) 第 086961 号

Linux 环境编程: 从应用到内核

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 余 洁

责任校对: 殷 虹

印 刷: 三河市宏图印务有限公司

版 次: 2016 年 6 月第 1 版第 1 次印刷

开 本: 186mm×240mm 1/16

印 张: 38

书 号: ISBN 978-7-111-53610-9

定 价: 99.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzit@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东

为什么要写这本书

我从事 Linux 环境的开发工作已有近十年的时间，但我一直认为工作时间并不等于经验，更不等于能力。如何才能把工作时间转换为自己的经验和能力呢？我认为无非是多阅读、多思考、多实践、多分享。这也是我在 ChinaUnix 上的博客座右铭，目前我的博客一共有 247 篇博文，记录的大都是 Linux 内核网络部分的源码分析，以及相关的应用编程。机械工业出版社华章公司的 Lisa 正是通过我的博客找到我的，而这也促成了本书的出版。

其实在 Lisa 之前，就有另外一位编辑与我聊过，但当时我没有下好决心，认为自己无论是在技术水平，还是时间安排上，都不足以完成一本技术图书的创作。等到与 Lisa 洽谈的时候，我感觉自己的技术已经有了一些沉淀，同时时间也相对比较充裕，因此决定开始撰写自己技术生涯的第一本书。

对于 Linux 环境的开发人员，《Unix 环境高级编程》（后文均简称为 APUE）无疑是最为经典的入门书籍。其作者 Stevens 是我从业以来最崇拜的技术专家。他的 Advanced Programming in the Unix Environment、Unix Network Programming 系列及 TCP/IP Illustrated 系列著作，字字珠玑，本本经典。在我从业的最初几年，这几本书每本都阅读了好几遍，而这也为我进行 Linux 用户空间的开发奠定了坚实的基础。在掌握了这些知识以后，如何继续提高自己的技能呢？经过一番思考，我选择了阅读 Linux 内核源码，并尝试将内核与应用融会贯通。在阅读了一定量的内核源码之后，我才真正理解了 Linux 专家的这句话“Read the fucking codes”。只有阅读了内核源码，才能真正理解 Linux 内核的原理和运行机制，而此时，我也发现了 Stevens 著作的一个局限——APUE 和 UNP 毕竟是针对 Unix 环境而写的，Linux 虽然大部分与 Unix 兼容，但是在很多行为上与 Unix 还是完全不同的。这就导致了书中的一些内容与 Linux 环境中的实际效果是相互矛盾的。

现在有机会来写一本技术图书，我就想在向 Stevens 致敬的同时，写一本类似于 APUE

风格的技术图书，同时还要在 Linux 环境下，对 APUE 进行突破。大言不惭地说，我期待这本书可以作为 APUE 的补充，还可以作为 Linux 开发人员的进阶读物。事实上，本书的写作布局正是以 APUE 的章节作为参考，针对 Linux 环境，不仅对用户空间的接口进行阐述，同时还引导读者分析该接口在内核的源码实现，使得读者不仅可以知道接口怎么用，同时还可以理解接口是怎么工作的。对于 Linux 的系统调用，做到知其然，知其所以然。

读者对象

根据本书的内容，我觉得适合以下几类读者：

- 在 Linux 应用层方面有一定开发经验的程序员。
- 对 Linux 内核有兴趣的程序员。
- 热爱 Linux 内核和开源项目的技术人员。

如何阅读本书

本书定位为 APUE 的补充或进阶读物，所以假设读者已具备了一定的编程基础，对 Linux 环境也有所了解，因此在涉及一些基本概念和知识时，只是蜻蜓点水，简单略过。因为笔者希望把更多的笔墨放在更为重要的部分，而不是各种相关图书均有讲解的基本概念上。所以如果你是初学者，建议还是先学习 APUE、C 语言编程，并且在具有一定的操作系统知识后再来阅读本书。

Linux 环境编程涉及的领域太多，很难有某个人可以在 Linux 的各个领域均有比较深刻的认识，尤其是已有 APUE 这本经典图书在前，所以本书是由高峰、李彬两个人共同完成的。

高峰负责第 0、1、2、3、4、12、13、14、15 章，李彬负责第 5~11 章。两位不同的作者，在写作风格上很难保证一致，如果给各位读者带来了不便，在此给各位先道个歉。尽管是由两个人共同写作，并且负责的还是我们各自相对擅长的领域，可是在写作的过程中我们仍然感觉到很吃力，用了将近三年的时间才算完成本书。对比 APUE，本书一方面在深度上还是有所不及，另一方面在广度上还是没有涵盖 APUE 涉及的所有领域，这也让我们对 Stevens 大师更加敬佩。

本书使用的 Linux 内核源代码版本为 3.2.44，glibc 的源码版本为 2.17。

勘误和支持

由于作者的水平有限，主题又过于宏大，书中难免会出现一些错误或不准确的地方，

如有不妥之处，恳请读者批评指正。如果你发现有什么问题，或者有什么疑问，都可以发邮件至我的邮箱 gfree.wind@gmail.com，期待您的指导！

致谢

首先要感谢伟大的 Linux 内核创始人 Linus，他开创了一个影响世界的操作系统。

其次要感谢机械工业出版社华章公司的编辑杨绣国老师 (Lisa)，感谢你的魄力，敢于找新人来写作，并敢于信任新人，让其完成这么大的一个项目。感谢你的耐心，正常的一年半的写作时间，被我们生生地延长到了将近三年的时间，感谢你在写作过程中对我们的鼓励和帮助。

然后要感谢我的搭档李彬，在我加入当前的创业公司后，只有很少的空闲时间和精力来投入写作。这时，是李彬在更紧张的时间内，承担了本书的一半内容。并且其写作态度极其认真，对质量精益求精。没有李彬的加入，本书很可能就半途而废了。再次感谢李彬，我的好搭档。

最后我要感谢我的亲人。感谢我的父母，没有你们的培养，绝没有我的今天；感谢我的妻子，没有你的支持，就没有我事业上的进步；感谢我的岳父岳母对我女儿的照顾，使我没有后顾之忧；最后要感谢的是我可爱的女儿高一涵小天使，你的诞生为我带来了无尽的欢乐和动力！

谨以此书，献给我最亲爱的家人，以及众多热爱 Linux 的朋友们。

高峰

中国北京

2016年3月

目 录 Contents

前 言	1.2.4 如何选择文件描述符	17
第 0 章 基础知识	1.2.5 文件描述符 fd 与文件管理结构	18
0.1 一个 Linux 程序的诞生记	file	18
0.2 程序的构成	1.3 creat 简介	19
0.3 程序是如何“跑”的	1.4 关闭文件	19
0.4 背景概念介绍	1.4.1 close 介绍	19
0.4.1 系统调用	1.4.2 close 源码跟踪	19
0.4.2 C 库函数	1.4.3 自定义 files_operations	21
0.4.3 线程安全	1.4.4 遗忘 close 造成的问题	22
0.4.4 原子性	1.4.5 如何查找文件资源泄漏	25
0.4.5 可重入函数	1.5 文件偏移	26
0.4.6 阻塞与非阻塞	1.5.1 lseek 简介	26
0.4.7 同步与非同步	1.5.2 小心 lseek 的返回值	26
第 1 章 文件 I/O	1.5.3 lseek 源码分析	27
1.1 Linux 中的文件	1.6 读取文件	29
1.1.1 文件、文件描述符和文件表	1.6.1 read 源码跟踪	29
1.1.2 内核文件表的实现	1.6.2 部分读取	30
1.2 打开文件	1.7 写入文件	31
1.2.1 open 介绍	1.7.1 write 源码跟踪	31
1.2.2 更多选项	1.7.2 追加写的实现	33
1.2.3 open 源码跟踪	1.8 文件的原子读写	33
	1.9 文件描述符的复制	34
	1.10 文件数据的同步	38

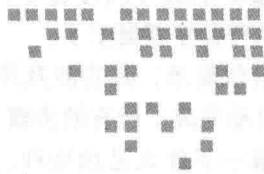
1.11 文件的元数据.....	41	3.5.2 编译生成和使用动态库.....	80
1.11.1 获取文件的元数据.....	41	3.5.3 程序的“平滑无缝”升级.....	82
1.11.2 内核如何维护文件的元数据.....	42	3.6 避免内存问题.....	84
1.11.3 权限位解析.....	43	3.6.1 尴尬的 realloc.....	84
1.12 文件截断.....	45	3.6.2 如何防止内存越界.....	85
1.12.1 truncate 与 ftruncate 的简单 介绍.....	45	3.6.3 如何定位内存问题.....	86
1.12.2 文件截断的内核实现.....	45	3.7 “长跳转” longjmp.....	90
1.12.3 为什么需要文件截断.....	48	3.7.1 setjmp 与 longjmp 的使用.....	90
第 2 章 标准 I/O 库.....	50	3.7.2 “长跳转”的实现机制.....	91
2.1 stdin、stdout 和 stderr.....	50	3.7.3 “长跳转”的陷阱.....	93
2.2 I/O 缓存引出的趣题.....	51	第 4 章 进程控制：进程的一生.....	96
2.3 fopen 和 open 标志位对比.....	52	4.1 进程 ID.....	96
2.4 fdopen 与 fileno.....	55	4.2 进程的层次.....	98
2.5 同时读写的痛苦.....	56	4.2.1 进程组.....	99
2.6 ferror 的返回值.....	57	4.2.2 会话.....	102
2.7 clearerr 的用途.....	57	4.3 进程的创建之 fork().....	103
2.8 小心 fgetc 和 getc.....	60	4.3.1 fork 之后父子进程的内存关系.....	104
2.9 注意 fread 和 fwrite 的返回值.....	60	4.3.2 fork 之后父子进程与文件的 关系.....	107
2.10 创建临时文件.....	61	4.3.3 文件描述符复制的内核实现.....	110
第 3 章 进程环境.....	66	4.4 进程的创建之 vfork().....	115
3.1 main 是 C 程序的开始吗.....	66	4.5 daemon 进程的创建.....	117
3.2 “活雷锋” exit.....	70	4.6 进程的终止.....	119
3.3 atexit 介绍.....	75	4.6.1 _exit 函数.....	119
3.3.1 使用 atexit.....	75	4.6.2 exit 函数.....	120
3.3.2 atexit 的局限性.....	76	4.6.3 return 退出.....	122
3.3.3 atexit 的实现机制.....	77	4.7 等待子进程.....	122
3.4 小心使用环境变量.....	78	4.7.1 僵尸进程.....	122
3.5 使用动态库.....	80	4.7.2 等待子进程之 wait().....	124
3.5.1 动态库与静态库.....	80	4.7.3 等待子进程之 waitpid().....	126
		4.7.4 等待子进程之等待状态值.....	129

4.7.5	等待子进程之 waitid().....	131	5.7	CPU 的亲合力.....	214	
4.7.6	进程退出和等待的内核实现.....	133	第 6 章 信号		219	
4.8	exec 家族.....	141	6.1	信号的完整生命周期.....	219	
4.8.1	execve 函数.....	141	6.2	信号的产生.....	220	
4.8.2	exec 家族.....	142	6.2.1	硬件异常.....	220	
4.8.3	execve 系统调用的内核实现.....	144	6.2.2	终端相关的信号.....	221	
4.8.4	exec 与信号.....	151	6.2.3	软件事件相关的信号.....	223	
4.8.5	执行 exec 之后进程继承的 属性.....	152	6.3	信号的默认处理函数.....	224	
4.9	system 函数.....	152	6.4	信号的分类.....	227	
4.9.1	system 函数接口.....	153	6.5	传统信号的特点.....	228	
4.9.2	system 函数与信号.....	156	6.5.1	信号的 ONESHOT 特性.....	230	
4.10	总结.....	157	6.5.2	信号执行时屏蔽自身的特性.....	232	
第 5 章 进程控制：状态、调度和 优先级			158	6.5.3	信号中断系统调用的重启特性.....	233
5.1	进程的状态.....	158	6.6	信号的可靠性.....	236	
5.1.1	进程状态概述.....	159	6.6.1	信号的可靠性实验.....	236	
5.1.2	观察进程状态.....	171	6.6.2	信号可靠性差异的根源.....	240	
5.2	进程调度概述.....	173	6.7	信号的安装.....	243	
5.3	普通进程的优先级.....	181	6.8	信号的发送.....	246	
5.4	完全公平调度的实现.....	186	6.8.1	kill、tkill 和 tkill.....	246	
5.4.1	时间片和虚拟运行时间.....	186	6.8.2	raise 函数.....	247	
5.4.2	周期性调度任务.....	190	6.8.3	sigqueue 函数.....	247	
5.4.3	新进程的加入.....	192	6.9	信号与线程的关系.....	253	
5.4.4	睡眠进程醒来.....	198	6.9.1	线程之间共享信号处理函数.....	254	
5.4.5	唤醒抢占.....	202	6.9.2	线程有独立的阻塞信号掩码.....	255	
5.5	普通进程的组调度.....	204	6.9.3	私有挂起信号和共享挂起 信号.....	257	
5.6	实时进程.....	207	6.9.4	致命信号下，进程组全体 退出.....	260	
5.6.1	实时调度策略和优先级.....	207	6.10	等待信号.....	260	
5.6.2	实时调度相关 API.....	211	6.10.1	pause 函数.....	261	
5.6.3	限制实时进程运行时间.....	213	6.10.2	sigsuspend 函数.....	262	

6.10.3	sigwait 函数和 sigwaitinfo 函数	263	7.9	性能杀手：伪共享	323
6.11	通过文件描述符来获取信号	265	7.10	条件等待	328
6.12	信号递送的顺序	267	7.10.1	条件变量的创建和销毁	328
6.13	异步信号安全	272	7.10.2	条件变量的使用	329
6.14	总结	275	第 8 章 理解 Linux 线程 (2)		333
第 7 章 理解 Linux 线程 (1)			8.1	线程取消	333
7.1	线程与进程	276	8.1.1	函数取消接口	333
7.2	进程 ID 和线程 ID	281	8.1.2	线程清理函数	335
7.3	pthread 库接口介绍	284	8.2	线程局部存储	339
7.4	线程的创建和标识	285	8.2.1	使用 NPTL 库函数实现线程局部存储	340
7.4.1	pthread_create 函数	285	8.2.2	使用 __thread 关键字实现线程局部存储	342
7.4.2	线程 ID 及进程地址空间布局	286	8.3	线程与信号	343
7.4.3	线程创建的默认属性	291	8.3.1	设置线程的信号掩码	344
7.5	线程的退出	292	8.3.2	向线程发送信号	344
7.6	线程的连接与分离	293	8.3.3	多线程程序对信号的处理	345
7.6.1	线程的连接	293	8.4	多线程与 fork()	345
7.6.2	为什么要连接退出的线程	295	第 9 章 进程间通信：管道		349
7.6.3	线程的分离	299	9.1	管道	351
7.7	互斥量	300	9.1.1	管道概述	351
7.7.1	为什么需要互斥量	300	9.1.2	管道接口	352
7.7.2	互斥量的接口	304	9.1.3	关闭未使用的管道文件描述符	356
7.7.3	临界区的大小	305	9.1.4	管道对应的内存区大小	361
7.7.4	互斥量的性能	306	9.1.5	shell 管道的实现	361
7.7.5	互斥锁的公平性	310	9.1.6	与 shell 命令进行通信 (popen)	362
7.7.6	互斥锁的类型	311	9.2	命名管道 FIFO	365
7.7.7	死锁和活锁	314	9.2.1	创建 FIFO 文件	365
7.8	读写锁	316	9.2.2	打开 FIFO 文件	366
7.8.1	读写锁的接口	317			
7.8.2	读写锁的竞争策略	318			
7.8.3	读写锁总结	323			

9.3	读写管道文件	367	11.2	POSIX 消息队列	415
9.4	使用管道通信的示例	372	11.2.1	消息队列的创建、打开、关闭及删除	415
第 10 章 进程间通信: System V			11.2.2	消息队列的属性	418
	IPC	375	11.2.3	消息的发送和接收	422
10.1	System V IPC 概述	375	11.2.4	消息的通知	423
10.1.1	标识符与 IPC Key	376	11.2.5	I/O 多路复用监控消息队列	427
10.1.2	IPC 的公共数据结构	379	11.3	POSIX 信号量	428
10.2	System V 消息队列	383	11.3.1	创建、打开、关闭和删除有名信号量	430
10.2.1	创建或打开一个消息队列	383	11.3.2	信号量的使用	431
10.2.2	发送消息	385	11.3.3	无名信号量的创建和销毁	432
10.2.3	接收消息	388	11.3.4	信号量与 futex	433
10.2.4	控制消息队列	390	11.4	内存映射 mmap	436
10.3	System V 信号量	391	11.4.1	内存映射概述	436
10.3.1	信号量概述	391	11.4.2	内存映射的相关接口	438
10.3.2	创建或打开信号量	393	11.4.3	共享文件映射	439
10.3.3	操作信号量	395	11.4.4	私有文件映射	455
10.3.4	信号量撤销值	399	11.4.5	共享匿名映射	455
10.3.5	控制信号量	400	11.4.6	私有匿名映射	456
10.4	System V 共享内存	402	11.5	POSIX 共享内存	456
10.4.1	共享内存概述	402	11.5.1	共享内存的创建、使用和删除	457
10.4.2	创建或打开共享内存	403	11.5.2	共享内存与 tmpfs	458
10.4.3	使用共享内存	405	第 12 章 网络通信: 连接的建立		462
10.4.4	分离共享内存	407	12.1	socket 文件描述符	462
10.4.5	控制共享内存	408	12.2	绑定 IP 地址	463
第 11 章 进程间通信: POSIX IPC		410	12.2.1	bind 的使用	464
11.1	POSIX IPC 概述	411	12.2.2	bind 的源码分析	465
11.1.1	IPC 对象的名字	411	12.3	客户端连接过程	468
11.1.2	创建或打开 IPC 对象	413	12.3.1	connect 的使用	468
11.1.3	关闭和删除 IPC 对象	414			
11.1.4	其他	414			

12.3.2	connect 的源码分析	469	14.2	数据包从内核空间到用户空间的 流程	537
12.4	服务器端连接过程	477	14.3	UDP 数据包的接收流程	540
12.4.1	listen 的使用	477	14.4	TCP 数据包的接收流程	544
12.4.2	listen 的源码分析	478	14.5	TCP 套接字的三个接收队列	553
12.4.3	accept 的使用	480	14.6	从网卡到套接字	556
12.4.4	accept 的源码分析	480	14.6.1	从硬中断到软中断	556
12.5	TCP 三次握手的实现分析	483	14.6.2	软中断处理	557
12.5.1	SYN 包的发送	483	14.6.3	传递给协议栈流程	559
12.5.2	接收 SYN 包, 发送 SYN+ ACK 包	485	14.6.4	IP 协议处理流程	564
12.5.3	接收 SYN+ACK 数据包	494	14.6.5	大师的错误? 原始套接字的 接收	568
12.5.4	接收 ACK 数据包, 完成三次 握手	499	14.6.6	注册传输层协议	571
第 13 章 网络通信: 数据报文的 发送			14.6.7	确定 UDP 套接字	571
13.1	发送相关接口	505	14.6.8	确定 TCP 套接字	576
13.2	数据包从用户空间到内核空间的 流程	506	第 15 章 编写安全无错代码		
13.3	UDP 数据包的发送流程	510	15.1	不要用 memcmp 比较结构体	582
13.4	TCP 数据包的发送流程	517	15.2	有符号数和无符号数的移位 区别	583
13.5	IP 数据包的发送流程	527	15.3	数组和指针	584
13.5.1	ip_send_skb 源码分析	528	15.4	再论数组首地址	587
13.5.2	ip_queue_xmit 源码分析	531	15.5	“神奇”的整数类型转换	588
13.6	底层模块数据包的发送流程	532	15.6	小心 volatile 的原子性误解	589
第 14 章 网络通信: 数据报文的 接收			15.7	有趣的问题: “x == x” 何时 为假?	591
14.1	系统调用接口	536	15.8	小心浮点陷阱	593
			15.8.1	浮点数的精度限制	593
			15.8.2	两个特殊的浮点值	593
			15.9	Intel 移位指令陷阱	595



基础知识

基础知识是构建技术大厦不可或缺的稳定基石，因此，本书首先来介绍一下书中所涉及的一些基础知识。这里以第 0 章命名，表明我们要注重基础，从 0 开始，同时也是向伟大的 C 语言致敬。

基础知识看似简单，但是想要真正理解它们，是需要花一番功夫的。除了需要积累经验以外，更需要对它们进行不断的思考和理解，这样，才能写出高可靠性的程序。这些基础知识很多都可以独立成文，限于篇幅，这里只能是简单的介绍，都是笔者根据自己的经验和理解进行的总结和概括，相信对读者会有所帮助。感兴趣的朋友可以自己查找更多的资料，以得到更准确、更细致的介绍。

**注意**

本书中的示例代码为了简洁明了，没有考虑代码的健壮性，例如不检查函数的返回值、使用全局变量等。

0.1 一个 Linux 程序的诞生记

一本编程书籍如果开篇不写一个“hello world”，就违背了“自古以来”的传统了。因此本节也将以 hello world 为例来说明一个 Linux 程序的诞生过程，示例代码如下：

```
#include <stdio.h>

int main(void)
{
    printf("Hello world!\n");
    return 0;
}
```

下面使用 gcc 生成可执行程序：`gcc -g -Wall 0_1_hello_world.c -o hello_world`。这样，一个 Linux 可执行程序就诞生了。

整个过程看似简单，其实涉及预处理、编译、汇编和链接等多个步骤。只不过 gcc 作为一个工具集自动完成了所有的步骤。下面就分别来看看其中所涉及各个步骤。

首先来了解一下什么是预处理。预处理用于处理预处理命令。对于上面的代码来说，唯一的预处理命令就是 `#include`。它的作用是将头文件的内容包含到本文件中。注意，这里的“包含”指的是该头文件中的所有代码都会在 `#include` 处展开。可以通过“`gcc -E 0_1_hello_world.c`”在预处理后自动停止后面的操作，并把预处理的结果输出到标准输出。因此使用“`gcc -E 0_1_hello_world.c > 0_1_hello_world.i`”，可得到预处理后的文件。

理解了预处理，在出现一些常见的错误时，才能明白其中的原因。比如，为什么不能在头文件中定义全局变量？这是因为定义全局变量的代码会存在于所有以 `#include` 包含该头文件的文件中，也就是说所有的这些文件，都会定义一个同样的全局变量，这样就不可避免地造成了冲突。

编译环节是指对源代码进行语法分析，并优化产生对应的汇编代码的过程。同样，可以使用 gcc 得到汇编代码，而非最终的二进制文件，即“`gcc -S 0_1_hello_world.c -o 0_1_hello_world.s`”。gcc 的 `-S` 选项会让 gcc 在编译完成后停止后面的工作，这样只会产生对应的汇编文件。

汇编的过程比较简单，就是将源代码翻译成可执行的指令，并生成目标文件。对应的 gcc 命令为“`gcc -c 0_1_hello_world.c -o 0_1_hello_world.o`”。

链接是生成最终可执行程序的最后一个步骤，也是比较复杂的一步。它的工作就是将各个目标文件——包括库文件（库文件也是一种目标文件）链接成一个可执行程序。在这个过程中，涉及的概念比较多，如地址和空间的分配、符号解析、重定位等。在 Linux 环节下，该工作是由 GNU 的链接器 ld 完成的。

实际上我们可以使用 `-v` 选项来查看完整和详细的 gcc 编译过程，命令如下。

```
gcc -g -Wall -v 0_1_hello_world.c -o hello_world.
```

由于输出过多，此处就不粘贴结果了。感兴趣的朋友可以自行执行命令，查看输出。通过 `-v` 选项，可以看到 gcc 在背后做了哪些具体的工作。

0.2 程序的构成

Linux 下二进制可执行程序的格式一般为 ELF 格式。以 0.1 节的 hello world 为例，使用 `readelf` 查看其 ELF 格式，内容如下：

```
ELF Header:
Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00
Class: ELF32
Data: 2's complement, little endian
Version: 1 (current)
```

```

OS/ABI: UNIX - System V
ABI Version: 0
Type: EXEC (Executable file)
Machine: Intel 80386
Version: 0x1
Entry point address: 0x8048320
Start of program headers: 52 (bytes into file)
Start of section headers: 5148 (bytes into file)
Flags: 0x0
Size of this header: 52 (bytes)
Size of program headers: 32 (bytes)
Number of program headers: 9
Size of section headers: 40 (bytes)
Number of section headers: 36
Section header string table index: 33
Section Headers:
[Nr] Name Type Addr Off Size ES Flg Lk Inf Al
[ 0] NULL 00000000 000000 000000 00 0 0 0
[ 1] .interp PROGBITS 08048154 000154 000013 00 A 0 0 1
[ 2] .note.ABI-tag NOTE 08048168 000168 000020 00 A 0 0 4
[ 3] .note.gnu.build-id NOTE 08048188 000188 000024 00 A 0 0 4
[ 4] .gnu.hash GNU_HASH 080481ac 0001ac 000020 04 A 5 0 4
[ 5] .dynsym DYNSYM 080481cc 0001cc 000050 10 A 6 1 4
[ 6] .dynstr STRTAB 0804821c 00021c 00004a 00 A 0 0 1
[ 7] .gnu.version VERSYM 08048266 000266 00000a 02 A 5 0 2
[ 8] .gnu.version_r VERNEED 08048270 000270 000020 00 A 6 1 4
[ 9] .rel.dyn REL 08048290 000290 000008 08 A 5 0 4
[10] .rel.plt REL 08048298 000298 000018 08 A 5 12 4
[11] .init PROGBITS 080482b0 0002b0 000024 00 AX 0 0 4
[12] .plt PROGBITS 080482e0 0002e0 000040 04 AX 0 0 16
[13] .text PROGBITS 08048320 000320 000188 00 AX 0 0 16
[14] .fini PROGBITS 080484a8 0004a8 000015 00 AX 0 0 4
[15] .rodata PROGBITS 080484c0 0004c0 000015 00 A 0 0 4
[16] .eh_frame_hdr PROGBITS 080484d8 0004d8 000034 00 A 0 0 4
[17] .eh_frame PROGBITS 0804850c 00050c 0000c4 00 A 0 0 4
[18] .init_array INIT_ARRAY 08049f08 000f08 000004 00 WA 0 0 4
[19] .fini_array FINI_ARRAY 08049f0c 000f0c 000004 00 WA 0 0 4
[20] .jcr PROGBITS 08049f10 000f10 000004 00 WA 0 0 4
[21] .dynamic DYNAMIC 08049f14 000f14 0000e8 08 WA 6 0 4
[22] .got PROGBITS 08049ffc 000ffc 000004 04 WA 0 0 4
[23] .got.plt PROGBITS 0804a000 001000 000018 04 WA 0 0 4
[24] .data PROGBITS 0804a018 001018 000008 00 WA 0 0 4
[25] .bss NOBITS 0804a020 001020 000004 00 WA 0 0 4
[26] .comment PROGBITS 00000000 001020 00006b 01 MS 0 0 1
[27] .debug_aranges PROGBITS 00000000 00108b 000020 00 0 0 1
[28] .debug_info PROGBITS 00000000 0010ab 000094 00 0 0 1
[29] .debug_abbrev PROGBITS 00000000 00113f 000044 00 0 0 1
[30] .debug_line PROGBITS 00000000 001183 000043 00 0 0 1
[31] .debug_str PROGBITS 00000000 0011c6 0000cb 01 MS 0 0 1
[32] .debug_loc PROGBITS 00000000 001291 000038 00 0 0 1
[33] .shstrtab STRTAB 00000000 0012c9 000151 00 0 0 1
[34] .symtab SYMTAB 00000000 0019bc 000490 10 35 51 4
[35] .strtab STRTAB 00000000 001e4c 00025a 00 0 0 1

```

由于输出过多，后面的结果并没有完全展示出来。ELF文件的主要内容就是由各个

section 及 symbol 表组成的。在上面的 section 列表中，大家最熟悉的应该是 text 段、data 段和 bss 段。text 段为代码段，用于保存可执行指令。data 段为数据段，用于保存有非 0 初始值的全局变量和静态变量。bss 段用于保存没有初始值或初值为 0 的全局变量和静态变量，当程序加载时，bss 段中的变量会被初始化为 0。这个段并不占用物理空间——因为完全没有必要，这些变量的值固定初始化为 0，因此何必占用宝贵的物理空间？

其他段没有这三个段有名，下面来介绍一下其中一些比较常见的段：

- ❑ debug 段：顾名思义，用于保存调试信息。
- ❑ dynamic 段：用于保存动态链接信息。
- ❑ fini 段：用于保存进程退出时的执行程序。当进程结束时，系统会自动执行这部分代码。
- ❑ init 段：用于保存进程启动时的执行程序。当进程启动时，系统会自动执行这部分代码。
- ❑ rodata 段：用于保存只读数据，如 const 修饰的全局变量、字符串常量。
- ❑ symtab 段：用于保存符号表。

其中，对于与调试相关的段，如果不使用 -g 选项，则不会生成，但是与符号相关的段仍将会存在，这时可以使用 strip 去掉符号信息，感兴趣的朋友可以自己参考 strip 的说明进行实验。一般在嵌入式的产品中，为了减少程序占用的空间，都会使用 strip 去掉非必要的段。

0.3 程序是如何“跑”的

在日常工作中，我们经常会说“程序‘跑’起来了”，那么它到底是怎么“跑”的呢？在 Linux 环境下，可以使用 strace 跟踪系统调用，从而帮助自己研究系统程序加载、运行和退出的过程。此处仍然以 hello_world 为例。

```
strace ./hello_world
execve("./hello_world", [ "./hello_world" ], [ /* 59 vars */ ]) = 0
brk(0) = 0x872a000
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7778000
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=80063, ...}) = 0
mmap2(NULL, 80063, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7764000
close(3) = 0
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
open("/lib/i386-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0000\226\1\0004\0\0\0"...
    512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1730024, ...}) = 0
mmap2(NULL, 1743580, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb75ba000
mprotect(0xb775d000, 4096, PROT_NONE) = 0
mmap2(0xb775e000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_
```

```

DENYWRITE, 3, 0x1a3) = 0xb775e000
mmap2(0xb7761000, 10972, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_
ANONYMOUS, -1, 0) = 0xb7761000
close(3) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
= 0xb75b9000
set_thread_area({entry_number:-1 -> 6, base_addr:0xb75b9900, limit:1048575,
seg_32bit:1, contents:0, read_exec_only:0, limit_in_pages:1, seg_not_present:0,
useable:1}) = 0
mprotect(0xb775e000, 8192, PROT_READ) = 0
mprotect(0x8049000, 4096, PROT_READ) = 0
mprotect(0xb779b000, 4096, PROT_READ) = 0
munmap(0xb7764000, 80063) = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 3), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb7777000
write(1, "Hello world!\n", 13Hello world!
) = 13
exit_group(0) = ?

```

下面就针对 `strace` 输出说明其含义。在 Linux 环境中，执行一个命令时，首先是由 shell 调用 `fork`，然后在子进程中来真正执行这个命令（这一过程在 `strace` 输出中无法体现）。`strace` 是 `hello_world` 开始执行后的输出。首先是调用 `execve` 来加载 `hello_world`，然后 `ld` 会分别检查 `ld.so.nohwcap` 和 `ld.so.preload`。其中，如果 `ld.so.nohwcap` 存在，则 `ld` 会加载其中未优化版本的库。如果 `ld.so.preload` 存在，则 `ld` 会加载其中的库——在一些项目中，我们需要拦截或替换系统调用或 C 库，此时就会利用这个机制，使用 `LD_PRELOAD` 来实现。之后利用 `mmap` 将 `ld.so.cache` 映射到内存中，`ld.so.cache` 中保存了库的路径，这样就完成了所有的准备工作。接着 `ld` 加载 c 库——`libc.so.6`，利用 `mmap` 及 `mprotect` 设置程序的各个内存区域，到这里，程序运行的环境已经完成。后面的 `write` 会向文件描述符 1（即标准输出）输出 "Hello world!\n"，返回值为 13，它表示 `write` 成功的字符个数。最后调用 `exit_group` 退出程序，此时参数为 0，表示程序退出的状态——此例中 `hello-world` 程序返回 0。

0.4 背景概念介绍

0.4.1 系统调用

系统调用是操作系统提供的服务，是应用程序与内核通信的接口。在 x86 平台上，有多种陷入内核的途径，最早是通过 `int 0x80` 指令来实现的，后来 Intel 增加了一个新的指令 `sysenter` 来代替 `int 0x80`——其他 CPU 厂商也增加了类似的指令。新指令 `sysenter` 的性能消耗大约是 `int 0x80` 的一半左右。即使是这样，相对于普通的函数调用来说，系统调用的性能消耗也是巨大的。所以在追求极致性能的程序中，都在尽力避免系统调用，譬如 C 库的 `gettimeofday` 就避免了系统调用。

用户空间的程序默认是通过栈来传递参数的。对于系统调用来说，内核态和用户态使用的是不同的栈，这使得系统调用的参数只能通过寄存器的方式进行传递。