TURING 图灵程序设计丛书

30日でできる! OS自作入門

30天自制操作系统

【日】川合秀实 著　　周自恒 李黎明 曾祥江 张文旭 译

代码光盘

只需30天
从零开始编写一个五脏俱全的
图形操作系统

39.1K迷你系统
实现多任务、汉字显示、文件压缩，
还能听歌看图玩游戏

日本编程天才
揭开CPU、内存、磁盘以及操作系统
底层工作模式的神秘面纱

人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵程序设计丛书

30日でできる! OS自作入門

30天自制操作系统

【日】川合秀实 著　　周自恒 李黎明 曾祥江 张文旭 译

人民邮电出版社
POSTS & TELECOM PRESS

图书在版编目（CIP）数据

30天自制操作系统 /（日）川合秀实著 ；周自恒等译. -- 北京 ：人民邮电出版社，2012.8
（图灵程序设计丛书）
ISBN 978-7-115-28796-0

Ⅰ. ①3… Ⅱ. ①川… ②周… Ⅲ. ①操作系统 Ⅳ. ①TP316

中国版本图书馆CIP数据核字(2012)第147654号

内 容 提 要

这是一本兼具趣味性、实用性与学习性的操作系统图书。作者从计算机的构造、汇编语言、C 语言开始解说，让读者在实践中掌握算法。在这本书的指导下，从零编写所有代码，30 天后就可以制作出一个具有窗口系统的 32 位多任务操作系统。

本书适合操作系统爱好者和程序设计人员阅读。

图灵程序设计丛书

30天自制操作系统

◆ 著 [日] 川合秀实
译 周自恒 李黎明 曾祥江 张文旭
责任编辑 傅志红
执行编辑 乐 馨

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京 印刷

◆ 开本：800×1000 1/16
印张：45
字数：1063千字 2012年 8 月第 1 版
印数：1－4 000 册 2012年 8 月北京第 1 次印刷

著作权合同登记号 图字：01-2011-6036号

ISBN 978-7-115-28796-0

定价：99.00元

读者服务热线：(010)51095186转604 印装质量热线：(010)67129223
反盗版热线：(010)67171154

版 权 声 明

前　言

“好想编写一个操作系统呀!”笔者的朋友曾说这是所有程序员都曾经怀揣的一个梦想。说“所有的程序员”可能有点夸张了，不过作为程序员的梦想，它至少也应该能排进前十名吧。

也许很多人觉得编写操作系统是个天方夜谭，这一定是操作系统业界的一个阴谋（笑）。他们故意让大家相信编写操作系统是一件非常困难的事情，这样就可以高价兜售自己开发的操作系统，而且操作系统的作者还会被顶礼膜拜。那么实际情况又怎么样呢？和别的程序相比，其实编写操作系统并没有那么难，至少笔者的感觉是这样。

在各位读者之中，也许有人曾经挑战过操作系统的编写，但因为太难而放弃了。拥有这样经历的人也许不会认同笔者的观点。其实你错了，你的失败并不是因为编写操作系统太难，而是因为没有人告诉你那其实是一件很简单的事而已。

不仅是编写操作系统，任何事都是一样的。如果讲解的人认为它很难，那就不可能把它讲述得通俗易懂，即便是同样的内容，也会讲得无比复杂。这样的讲解，肯定是很难懂的。

那么，你想不想和笔者一起再挑战一次呢？如果你曾经梦想过编写自己的操作系统，一定会觉得乐在其中的。

可能有人会说，这本书足足有700多页，怎么会“有趣”和“简单”呢？唔，这么一说笔者也觉得挺心虚的，不过其实也只是长了那么一点点啦。平均下来的话，每天只有大约23页的内容，你看，也没有那么长吧？

这本书的文风非常轻松，也许你不知不觉中就会读得很快。但是这样的话可能印象不会很深，最好还是能静下心来慢慢地读。书中所展示的程序代码和文字的说明同样重要，因此也希望大家仔细阅读。只要注意这些，理解本书的内容就应该没有问题了。

在本书中，我们使用C语言和汇编语言来编写操作系统，不过不必担心，你可以在阅读本书的同时来逐步学习关于这些编程语言的知识。本书在这方面写得非常仔细，如果能有人通过本书终于把C语言中的指针给搞懂了，那笔者的目的也就达到了。即便是从这样的水平开始，30天后你也能够编写出一个很棒的操作系统，请大家拭目以待吧!

目　录

第 0 天

着手开发之前

- 前言
- 何谓操作系统
- 开发操作系统的各种方法
- 无知则无畏
- 如何开发操作系统
- 操作系统开发中的困难
- 学习本书时的注意事项（重要！）
- 各章内容摘要

1 前言

现在，挑选自己喜欢的配件来组装一台世界上独一无二的、个性化的PC（个人电脑）对我们来说已不再困难。不仅如此，只要使用合适的编译器[①]，我们就可以自己编写游戏、制作自己的工具软件；使用网页制作工具，我们还可以轻而易举地制作主页；如果看过名著《CPU制作法》[②]的话，就连自制CPU也不在话下。

然而，在“自制领域”里至今还有一个无人涉足的课题——自己制作操作系统（OS）[③]，它看起来太难以至于初学者不敢轻易挑战。电脑组装也好，游戏、工具软件制作也好，主页也好，CPU也好，这些都已经成为初学者能够尝试的项目，而唯独操作系统被冷落在一边，实在有些遗憾。“既然还没有这样的书，那我就来写一本。”这就是笔者撰写本书的初衷。

也许是因为面向初学者的书太少的缘故吧，一说起操作系统，大家就会觉着那东西复杂得不得了，简直是高深莫测。特别是像Windows和Linux这些操作系统，庞大得一张光盘都快装不下了，要是一个人凭着兴趣来开发的话，不知道需要历经多么漫长的过程才能完成。笔者也认为，像这么复杂的操作系统，单凭一个人来做，一辈子都做不出来。

---

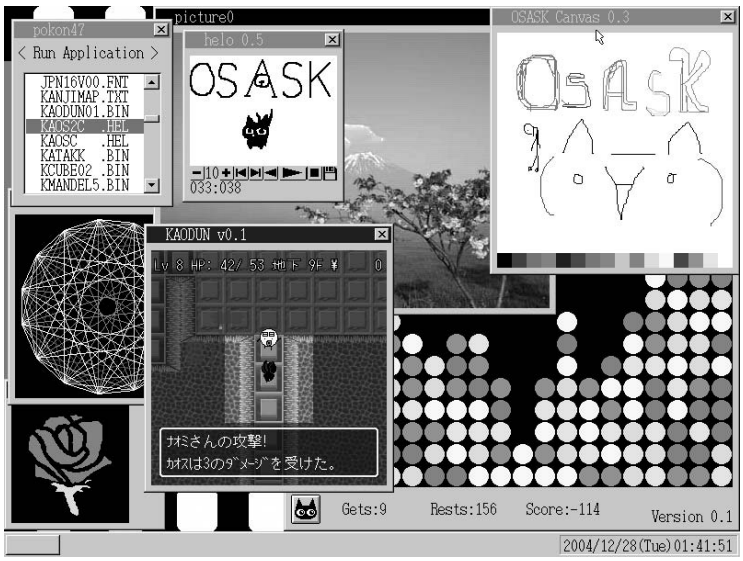
① 英文为compiler，指能够将源代码编译成机器码的软件。

②《CPU制作法》，渡波郁著，每日Communications出版公司，ISBN 4-8399-0986-5。

③ Operating System的缩写，汉语译作“操作系统”。Windows、Linux、MacOS、MS-DOS等软件的总称。

不过大家也不必担心太多。笔者就成功地开发过一个小型操作系统，其大小还不到80KB[①]。麻雀虽小，五脏俱全，这个操作系统的功能还是很完整的。有人也许会怀疑：“这么小的操作系统，是不是只有命令行窗口[②]啊？要不就是没有多任务[③]？”不，这些功能都有。

怎么样，只有80KB的操作系统，大家不觉得稍作努力就可以开发出来吗？即使是初学者，恐怕也会觉得这不是件难事吧？没错，我们用一个月的时间就能写出自己的操作系统！所以大家不用想得太难，我们轻轻松松地一起来写写看吧。



以本书作者为主角开发的操作系统OSASK[④]

大家一听到编译后的文件大小为80KB可能会觉得它作为程序来讲已经很小了，不过曾经编过程序的人可以查一查自己编的程序（.exe文件）的大小，这样就能体会到80KB到底是难是易了。

---

① kilobyte，程序及数据大小的度量单位，1字节（byte）的1024倍。一张软盘的容量是1440KB。顺便提一下，1024KB等于1MB（兆字节）。1字节是8个比特，正好能记录8位0和1的信息。B到底是指字节（byte），还是指比特（bit），有时容易混淆。这里根据一般的规则，用大写B表示字节，小写b表示比特。

② console，通过键盘输入命令的一种方式，基本上只用文字进行计算机操作，是MS-DOS等老式操作系统的主流操作方式。

③ 在操作系统的世界里，运行中的程序叫做“任务”，而同时执行多个任务的方式就被称为“多任务”（multitask）。

④ 笔者与他人一起合作开发的操作系统（趁机宣传一下）。虽然只有小小的78KB，不过为了做它也花了好几年的时间。而这次能在短时间内开发完成操作系统，是因为我们较好地总结了开发操作系统所必要的知识。也就是说，如果笔者在年轻时可以看到现在这本书的话，可能在短时间内就能开发出OSASK了，所以笔者很羡慕大家呀。

没编过程序的人也可以下载一个看上去不是很复杂的自由软件，看看它的可执行文件有多大。Windows 2000的计算器程序大约是90KB，大家也可以根据这个想象一下。

本书对于不打算自己写操作系统，甚至连想都没想过这个问题的人来说也会大有裨益。举个例子，读本自己组装PC的书就能知道PC是由哪些组件构成的，PC的性能是由哪些部分决定的；读本如何编写游戏的书，就能明白游戏是怎样运行的；同理，读了本书，了解了操作系统的开发过程，就能掌握操作系统的原理。所以说，对操作系统有兴趣的人，哪怕并不想自己做一个出来，也可以看看这本书。

阅读本书几乎不需要相关储备知识，这一点稍后还会详述。不管是用什么编程语言，只要是曾经写过简单的程序，对编程有一些感觉，就已经足够了（即使没有任何编程经验，应该也能看懂），因为这本书主要就是面向初学者的。书中虽然有很多C语言程序，但实际上并没有用到很高深的C语言知识，所以就算是曾经因为C语言太难而中途放弃的人也不用担心看不懂。当然，如果具备相关知识的话，理解起来会相对容易一些，不过即使没有相关知识也没关系，书中的说明都很仔细，大家可以放心。

本书以IBM PC/AT兼容机（也就是所谓的Windows个人电脑）为对象进行说明。至于其他机型[①]，比如Macintosh（苹果机）或者PC-9821等，虽然本书也参考了其中某些部分，但基本上无法开发出在这些机型上运行的操作系统，这一点还请见谅。严格地说，不是所有能称为AT兼容机的机型都可以开发我们这个操作系统，我们对机器的配置要求是CPU高于386（因为我们要开发32位操作系统）。换句话说，只要是能运行Windows 95以上操作系统的机器就没有问题，况且现在市面上（包括二手市场）恐怕都很难找到Windows 95以下的机器了，所以我们现在用的机型一般都没问题。

另外，大家也不用担心内存容量和硬盘剩余空间，我们需要使用的空间并不大。只要满足以上条件，就算机器又老又慢，也能用来开发我们的操作系统。

2 何谓操作系统

说老实话，其实笔者也不是很清楚。估计有人会说：“连这个都不懂，还写什么书？”不好意思……笔者见过很多种操作系统，有的功能非常多，而有的功能特别少。在比较了各种操作系统之后，笔者还是没有找到它们功能的共同点，无法下定义。结果就是，软件作者坚持说自己做的就是操作系统，而周围的人也不深究，就那样默认了，以至于什么软件都可以算是操作系统。笔者现在就是这么认为的。

既然就操作系统而言各有各的说法，那笔者也可以反过来利用这一点，一开始就根据自己的需要来定义操作系统，然后开发出一个满足自己定义条件的软件就可以了。这当然也算是开发操

① 本书所讲的操作系统内容仅用Macintosh是开发不了的，并且开发出的操作系统也不能直接在Macintosh上运行。但是在PC上开发的操作系统，可以通过模拟器在Macintosh上运行。

作系统了。哪怕做一个MS-DOS那样的，在一片漆黑的画面上显示出白字，输入个命令就能执行的操作系统也可以，这对笔者来说很简单。

但这样肯定会让一些读者大失所望。现在初学者也都见多识广，一提到操作系统，大家就会联想到Windows、Linux之类的庞然大物，所以肯定期待自制操作系统至少能任意显示窗口、实现鼠标光标控制、同时运行几个应用程序，等等。所以为了满足读者的期待，我们这次就来开发一个具有上述功能的操作系统。

3 开发操作系统的各种方法

开发操作系统的方法也是各种各样的。

笔者认为，最好的方法就是从既存操作系统中找一个跟自己想做的操作系统最接近的，然后在此基础上加以改造。这个方法是最节省时间的。

但本书却故意舍近求远，一切从零开始，完完全全是自己从头做起，这是因为笔者想向各位读者介绍从头到尾开发操作系统的全过程。如果我们找一个现成的操作系统，然后在此基础上删删改改的话，那这本书就不能涉及操作系统全盘的知识了，这样肯定无法让读者朋友满意。不过由于是全部从零做起，所以篇幅长些，还请读者朋友们耐下心来慢慢看。

要开发操作系统，首先遇到的问题就是使用什么编程语言，这次我们想以C语言为主。“啊，C语言啊？”笔者仿佛已经听到大家抱怨的声音了（苦笑）。“这都什么年代了，用C语言多土啊”、“用C++多好呀”、“还是Java好”、“不，我就喜欢Delphi”、“我还是觉得Visual Basic最好”……大家个人喜好习惯各不相同。这种心情笔者都能理解，但为了讲解时能简单一些，笔者还是想用C语言，请大家见谅。C语言功能虽不多，但用起来方便，所以用来开发操作系统刚好合适。要是用其他语言的话，仅讲解语言本身就要花很长时间，大家恐怕就没兴趣看下去了。

在这里先向大家传授一个从零开始开发操作系统的诀窍，那就是不要一开始就一心想着要开发操作系统，先做一个有点操作系统样子的东西就行了。如果我们一上来就要开发一个完整的操作系统的话，要做的东西太多，想想脑袋都大了，到时恐怕连着手的勇气也没有了。笔者就是因为这个，几年间遇到了很多挫折。所以在这本书里，我们不去大张旗鼓地想着要开发一个操作系统，而是编写几个像操作系统的演示程序[①]就行了。其实在开发演示程序的过程中大家就会逐步发现，演示程序不再是简单的演示程序，而是越来越像一个操作系统了。

4 无知则无畏

当我们打算开发操作系统时，总会有人从旁边跳出来，罗列出一大堆专业术语，问这问那，像内核怎么做啦，外壳怎么做啦，是不是单片啦，是不是微内核啦，等等。虽然有时候提这些问

① 演示程序的英文是demonstration。指不是为了使用，而是为了演示给人看的软件。

题也是有益的，但一上来就问这些，当然会让人无从回答。

要想给他们一个满意答复，让他们不再从旁指手画脚的话，还真得多学习，拿出点像模像样的见解才行。但我们是初学者，没有必要去学那些麻烦的东西，费时费力且不说，当我们知道现有操作系统在各方面都考虑得如此周密的时候，就会发现自己的想法太过简单而备受打击没了干劲。如果被前人的成果吓倒，只用这些现有的技术来做些拼拼凑凑的工作，岂不是太没意思了。

所以我们这次不去学习那些复杂的东西，直接着手开发。就算知道一大堆专业术语、专业理论，又有什么意思呢？还不如动手去做，就算做出来的东西再简单，起码也是自己的成果。而且自己先实际操作一次，通过实践找到其中的问题，再来看看是不是已经有了这些问题的解决方案，这样下来更能深刻地理解那些复杂理论。不管怎么说，反正目前我们也无法回答那些五花八门的问题，倒不如直接告诉在一旁指手画脚的人们：我们就是想用自己的方法做自己喜欢的事情，如果要讨论高深的问题，就另请高明吧。

■■■■■

其实反过来看，什么都不知道有时倒是好事。正是因为什么都不知道，我们才可能会认真地去做那些专家们嗤之以鼻的没意义的“傻事”。也许我们大多时候做的都没什么意义，但有时也可能会发掘出专家们千虑一失的问题呢。专家们在很多方面往往会先入为主，甚至根本不去尝试就断定这也不行那也不行，要么就浅尝辄止。因此能够挑战这些问题的，就只有我们这种什么都不知道的门外汉。任何人都能通过学习成为专家，但是一旦成为专家，就再也找不回门外汉的挑战精神了。所以从零开始，在没有各种条条框框限制的情况下，能做到什么程度就做到什么程度，碰壁以后再回头来学习相关知识，也为时未晚。

实际上笔者也正是这样一路磕磕绊绊地走过来，才有了今天。笔者没去过教授编程的学校，也几乎没学什么复杂的理论就开始开发操作系统了。但也正是因为这样，笔者做出的操作系统与其他的操作系统大不相同，非常有个性，所以得到了专家们的一致好评，而且现在还能有机会写这本书，向初学者介绍经验。总地说来，笔者从着手开发直到现在，每天都是乐在其中的。

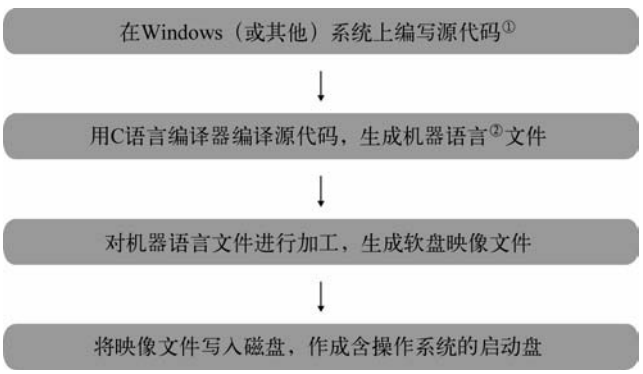
正是像笔者这样自己摸着石头过河，一路磕磕绊绊走过来的人，讲出的东西才简单易懂。不过在讲解过程中会涉及失败的经验，以及如何重新修正最终取得成功，所以已经懂了的人看着可能会着急。不好意思，如果碰到这种情况请忍耐一下吧。

读了这部分内容或许有人会觉得“是不是什么都不学习才是最好的啊”，其实那倒不是。比如工作上需要编写某些程序，或者一年之内要完成某些任务，这时没有时间去故意绕远路，所以为了避免不必要的失败，当然是先学习再着手开发比较好。但这次我们是因为自己的兴趣而学习操作系统的开发的，既然是兴趣，那就是按自己喜欢的方式慢慢来，这样就挺好的。

5 如何开发操作系统

操作系统（OS）一般打开电源开关就会自动执行。这是怎么实现的呢？一般在Windows上开发的可执行文件（～.exe），都要在操作系统启动以后，双击一下才能运行。我们这次想要做的可不是这种可执行程序，而是希望能够做到把含有操作系统的CD-ROM或软盘插入电脑，或者将操作系统装入硬盘后，只要打开电源开关就能自动运行。

为了开发这样的操作系统，我们准备按照如下的步骤来进行。



也就是说，所谓开发操作系统，就是想办法制作一张“含有操作系统的，能够自动启动的磁盘”。

这里出现的“映像文件”一词，简单地说就是软盘的备份数据。我们想要把特定的内容写入磁盘可不是拿块磁铁来在磁盘上晃晃就可以的。所以我们要先做出备份数据，然后将这些备份数据写入磁盘，这样才能做出符合我们要求的磁盘。

软盘的总容量是1440KB，所以作为备份数据的映像文件也恰好是1440KB。一旦我们掌握了制作磁盘映像的方法，就可以按自己的想法制作任意内容的磁盘了。

这里希望大家注意的是，开发操作系统时需要利用Windows等其他的操作系统。这是因为我们要使用文本编辑器或者C编译器，就必须使用操作系统。既然是这样，那么世界上第一个操作系统又是怎么做出来的呢？在开发世界上第一个操作系统时，当然还没有任何现成的操作系统可供利用，因此那时候人们不得不对照着CPU的命令代码表，自己将0和1排列起来，然后再把这些数据写入磁盘（估计那个时候还没有磁盘，用的是其他存储设备）。这是一项非常艰巨的工作。所以恐怕最初的操作系统功能非常有限，做好之后人们再利用它来开发一个稍微像点样的操作系统，然后再用这个来开发更实用的操作系统……操作系统应该就是这样一步一步发展过来的。

---

① source program，为了生成机器码所写的程序代码。可通过编译器编译成机器语言。

② CPU能够直接理解的语言，由二进制的0和1构成。其实源代码也是由 0和1构成的（后述）。

由于这次大部分初学者都是Windows用户，所以决定使用Windows这个现成的操作系统，Windows95/98/Me/2000/XP中任意一个版本都可以。肯定也有人会说还是Linux好用，所以笔者也总结了一下Linux上的做法，具体内容写在了帮助与支持[①]里，有需要的人请一定看一看。

另外，如果C编译器和映像文件制作工具等不一样的话，开发过程中就会产生一些细微的差别，这很难一一解释，所以笔者就直接把所有的工具都放到附带光盘里了。这些几乎都是笔者所发布的免费软件，它们大都是笔者为了开发后面的OSASK操作系统而根据需要自己编写的。这些工具的源代码也是公开的。除此之外，我们还会用到其他一些免费软件，所有这些软件的功能我们会在使用的时候详细介绍。

6 操作系统开发中的困难

现在市面上众多的C编译器都是以开发Windows或Linux上的应用程序为前提而设计的，几乎从来没有人想过要用它们来开发其他的软件，比如自己的操作系统。笔者所提供的编译器，也是以Windows版的gcc[②]为基础稍加改造而做成的，与gcc几乎没什么不同。或许也有为开发操作系统而设计的C编译器，不过就算有，恐怕也只有开发操作系统的公司才会买，所以当然会很贵。这次我们用不了这么高价的软件。

因为这些原因，我们只能靠开发应用程序用的C编译器想方设法编写出一个操作系统来。这实际上是在硬来，所以当中就会有很多不方便的地方。

就比如说printf(“hello\n”);吧，这个函数总是出现在C语言教科书的第一章，但我们现在就连它也无法使用。为什么呢？因为printf这个函数是以操作系统提供的功能为前提编写的，而我们最开始的操作系统可是什么功能都没有。因此，如果我们硬要执行这个函数的话，CPU会发生一般保护性异常[③]，直接罢工。刚开始的时候不仅是printf，几乎所有的函数都无法使用。

> 关于这次开发语言的选择，如果非要说出个所以然的话，其实也是因为C语言还算是很少依赖操作系统功能的语言，基本上只要不用函数就可以了。如果用C++的话，像new/delete这种基本而重要的运算符都不能用了，另外对于类的做法也会有很多要求，这样就无法发挥C++语言的优势了。当然，为了使用这些函数去开发操作系统，只要我们想办法，还是能够克服种种困难的。但是如果做到这个份上，我们不禁会想，到底是在用C++做操作系统呢，

① http://hrb.osask.jp。

② GNU项目组开发的免费C编译器，GNU C Compiler的简称。有时也指GUN开发的各种编译器的集合(GNU Compiler Collection)。

③ 电脑的CPU非常优秀，如果接到无视OS保护的指令或不可能执行的指令时，首先会保存当前状态，中断正在执行的程序，然后调用事先设定的函数。这种机制称为异常保护功能，比如除法异常、未定义指令异常、栈异常等。不能归类到任何异常类型中去的异常事态被称为一般保护异常。这种异常保护功能或许会让老Windows用户想起那噩梦般的蓝屏画面，但是如果经历过操作系统开发以后，大家就会觉得这种机制实在是太有用了。

还是在为了C++而做操作系统呢。对别的语言而言这个问题会更加突出，所以这次还是决定使用C语言，希望大家予以理解。

顺便插一句，在开发操作系统时不会受到限制的语言大概就只有汇编语言[①]了。还是汇编语言最厉害[②]（笑）。但是如果本书仅用汇编来编写操作系统的话，恐怕没几个人会看，所以就算是做事管前不顾后的笔者也不得不想想后果。

另外，在开发操作系统时，需要用到CPU上的许多控制操作系统的寄存器[③]。一般的C编译器都是用于开发应用程序的，所以根本没有任何操作这些寄存器的命令。另外，C编译器还具有非常优秀的自动优化功能，但有时候这反而会给我们带来麻烦。

归根到底，为了克服以上这些困难，有些没法用C语言来编写的部分，我们就只好用汇编语言来写了。这个时候，我们就必须要知道C编译器到底是怎样把程序编译成机器语言的。如果不能够与C编译器保持一致的话，就不能将汇编语言编写的部分与C语言编写的部分很好地衔接起来。这可是在编写普通的C语言程序时所体会不到哦！不过相比之下，今后的麻烦可比这种好处多得多啊（苦笑）。

同样，如果用C++来编写操作系统，也必须知道C++是如何把程序编译成机器语言的。当然，C++比C功能更多更强，编译规则也更复杂，所以解释起来也更麻烦，我们选用C语言也有这一层理由。总之，如果不理解自己所使用的语言是如何进行编译的，就没法用这种语言来编写操作系统。

书店里有不少C语言、C++的书，当然也还有Delphi、Java等其他各种编程语言的书，但这么多书里没有一本提到过“这些源代码编译过后生成的机器语言到底是什么样的”。不仅如此，虽然我们是在通过程序向CPU发指令的，但连CPU的基本结构都没有人肯给我们讲一讲。作为一个研究操作系统的人，真觉得心里不是滋味。为了弥补这一空缺，我们这本书就从这些基础讲起（但也仅限于此次开发操作系统所必备的基础知识）。

我们具备了这样的知识以后，说不定还会改变对程序设计的看法。以前也许只想着怎么写出漂亮的源代码来，以后也许就会更注重编译出来的是怎样的机器语言。源代码写得再漂亮，如果不能编译成自己希望的机器语言，不能正常运行的话，也是毫无意义的。反过来说，即便源代码写得难看点儿，即便只有特定的C编译器才能编译，但只要能够得到自己想要的机器语言就没有问题了。虽然不至于说“只要编译出了想要的机器语言，源代码就成了一张废纸”，但从某种意

① Assembler，与机器语言最接近的一种编程语言。过去掌握这种语言的人会备受尊敬，而现在这种人恐怕要被当作怪人了，真是可悲啊。原本汇编语言的正式名称应该是Assembly语言，而Assembler一般指的是编译程序。不过像笔者这样的老程序员，往往不对这两个词进行区分，统称为Assembler。

② 读到这里，大家可能还不理解为什么这么说，越往后看就越能慢慢体会到了。

③ Register，有些类似机器语言中的变量。对CPU而言，内存是外部存储装置，在CPU内核之中，存储装置只有寄存器。全部寄存器的容量加起来也不到1KB。

义上说还真就是这样。

对于开发操作系统的人而言，源程序无非是用来得到机器语言的“手段”，而不是目的。浪费太多时间在手段上就是本末倒置了。

对了，还有一点或许会有人担心，所以在这里事先说明一下：虽然操作系统是用C语言和汇编语言编写的，但并不是用C++编写的应用程序就无法在这个操作系统上运行。编写应用程序所用的语言，与开发操作系统所使用的语言是没有任何关系的，大家大可不必担心。

7 学习本书时的注意事项（重要!）

本书从第1章开始，写的是每一天实际开发的内容，虽然一共分成了30天，但这些都是根据笔者现在的能力和讲解的长度来大概切分的，并不是说读者也必须得一天完成一章。每个人觉得难的地方各不相同，有时学习一章可能要花上一星期的时间，也有时可能一天就能学会三章的内容。

当然，学习过程中可能会遇到看不太懂的章节，这种时候不要停下来，先接着往下读上个一两章也许会突然明白过来。如果往后看还是不明白的话，就先确认一下自己已经理解到哪一部分了，然后回过头来再从不懂的地方重新看就是了。千万别着急，看第二遍时，没准就会豁然开朗了。

如果已经弄清了哪里没理解，而且没理解的部分看了很多遍还是不明白的话，大家可以参阅我们的帮助与支持页面[①]，或许“问题与解答”（Q&A）页里会有解说。

■■■■■

本书对C语言的指针和结构体的说明与其他书籍有很大区别。这是因为本书先讲CPU的基本结构，然后讲汇编，最后再讲C语言，而其他的书都不讲这些基础知识，刚一提到指针，马上就转到变量地址如何如何了。所以就算大家“觉得”已经明白了那些书里讲的指针，也不要把本书的指针部分跳过去，相信这次大家能真正地理解指针。当然，如果真的已经弄明白了的话，大概看看就可以了。

■■■■■

从现在开始我们来一点一点地开发操作系统，我们会将每个阶段的进展情况总结出来，这些中间成果都刻在附带光盘里了，只要简单地复制一下就能马上运行。关于这些程序，有些需要注意的地方，我们在这里简单说明一下。

比如最初出现的程序是“helloos0”，下一个出现的程序是“helloos1”。 即使我们以helloos0

① http://hrb.osask.jp。

为基础，把书中讲解的内容一个不漏地全部做上一遍，也不能保证肯定可以得到后面的helloos1。书中可能偶尔有讲解得很完整的地方，但其实大多部分都讲得不够明确，这主要是因为笔者觉得这些地方不讲那么仔细大家肯定也能明白。

笔者说这些主要就是想要告诉大家，不仅要看书里的内容，更要好好看程序。有时候书上写得很含糊，读起来晦涩难懂，但一看程序马上就明白了。本书的主角不是正文内容，而是附录中的程序。正文仅仅是介绍程序是如何做出来的。

所以说从这个意义上讲，与其说这是“一本附带光盘的书”，倒不如说这是“一张附带一本大厚书的光盘”（笑）。

■■■■■

关于程序还有一点要说明的——这里收录的程序的版权全部归笔者所有。可是，读了这本书后打算开发自己的操作系统的话，可能有不少地方要仿照着附带程序来做；也有人可能想把程序的前期部分全盘照搬过来用；还有人可能想接着本书最后的部分继续开发自己的操作系统。

这是一本关于操作系统的教材，如果大家有上面这些想法却不能自由使用附录程序的话，这教材也就没什么意义了，所以大家可以随意使用这些程序，也不用事先提出任何申请。尽管大家最后做出来的操作系统中可能会包含笔者编写的程序，不过也不用在版权声明中署上笔者的名字。大家可以把它当作自己独立开发的操作系统，也可以卖了它去赚钱。就算大家靠这个系统成了亿万富翁，笔者也不会要分毫的分成，大家大可放心[①]。

而且这不只是买了本书的人才能享受的特权，从图书馆或朋友那儿借书看的人，甚至在书店里站着只看不买的人，也都享有以上权利。当然，大家要是买了这本书，对笔者、对出版社都是一个帮助。（笑）

在引用本书程序时，只有一点需要注意，那就是大家开发的操作系统的名字。因为它已经不是笔者所开发的操作系统了，所以请适当地改个名字，以免让人误解，仅此一点请务必留意。不管程序的内部是多么相像，它都是大家自己负责发布的另外一个不同的操作系统。给它起个响亮的名字吧。

以上声明仅适用于书中的程序，以及附带光盘中收录的用作操作系统教材的程序。本书正文和附带光盘中的其他工具软件不在此列。复制或修改都受到著作权法的保护。请在法律允许范围内使用这些内容。与光盘中的工具软件相关的许可权会放在本书最后一章予以说明。

---

① 在版权署名时，如果有人执意要署上笔者的名字，笔者也不反对。另外，要是大家一不小心发了大财，一定要给笔者分红的话，笔者当然也会心存感激地接受下来（笑）。

8 各章内容摘要

估计看过目录大家就能大概了解各章内容了，但因为目录里项目太多，所以在这里概括总结一下。如果有人想要保留一份神秘感，想边看边猜“后面的内容会是什么”，那么可以跳过本节不读（笑）。这一部分可以说是全书的灯塔，当大家在阅读本书的过程中感觉有什么不放心的时候，就回过头来重新看看本节内容吧。

第一周（第1天 ～ 第7天）

一开始首先要考虑怎么来写一个“只要一通电就能运行的程序”。这部分用C语言写起来有些困难，所以主要还是用汇编语言来写。

这步完成之后，下一步就要写一个从磁盘读取操作系统的程序。这时即便打开电脑电源，它也不会自动地将操作系统全部都读进来，它只能读取磁盘上最开始的512字节的内容，所以我们要编写剩余部分的载入程序。这个程序也要用汇编语言编写。

一旦完成了这一步，以后的程序就可以用C语言来编写了。我们就尽快使用C语言来学习开发显示画面的程序。同时，我们也能慢慢熟悉C语言语法。这个时候我们好像在做自己想做的事，但事实上我们还没有自由操纵C语言。

接下来，为了实现“移动鼠标”这一雄心，我们要对CPU进行细致的设定，并掌握中断处理程序的写法。从全书总体看来，这一部分是水平相当高的部分，笔者也觉得放在这里有些不妥，但从本书条理上讲，这些内容必须放在这里，所以只好请大家忍耐一下了。在这里，CPU的规格以及电脑复杂的规格都会给我们带来各种各样的麻烦。而且开发语言既有C语言，又有汇编语言，这又给我们造成了更大的混乱。这个时候我们一点儿也不会觉得这是在做自己想做的事，怎么看都像是在“受人摆布”。

渡过这个痛苦的时期，第一周就该结束了。

第二周（第8天 ～ 第14天）

一周的苦战还是很有意义的，回头一看，我们就会发现自己还是斩获颇丰的。这时我们已经基本掌握了C语言的语法，连汇编语言的水平也能达到本书的要求了。

所以现在我们就可以着手开发像样的操作系统了。但是这一次我们又要为算法头痛了。即使掌握了编程语言的语法，如果不懂得好的算法的话，也还是不能开发出来自己想要的操作系统。所以这一周我们就边学习算法边慢慢地开发操作系统。不过到了这一阶段，我们就能感觉到基本上不会再受技术问题限制了。

第三周（第15天 ～ 第21天）

现在我们的技术已经相当厉害了，可以随心所欲地开发自己的操作系统了。首先是要支持多任务，然后是开发命令行窗口，之后就可以着手开发应用程序了。到本周结束时，就算还不够完备，我们也能拿出一个可以称之为操作系统的软件了。

第四周（第22天 ～ 第28天）

在这个阶段，我们可以尽情地给操作系统增加各种各样的功能，同时还可以开发出大量像模像样的应用程序来。这个阶段我们已经能做得很好了，这可能也是我们最高兴的时期。这部分要讲解的内容很少，笔者也不用再煞费苦心地去写那些文字说明了，可以把精力都集中在编程上（笑）。对了，说起文字才想起来，正好在这个时期可以让我们的操作系统显示文字了。

免费赠送两天（第29天 ～ 第30天）

剩下的两天用来润色加工。这两天我们来做一些之前没来得及做，但做起来既简单又有趣的内容。

■■■■■

以上就是从第1天到第30天的内容摘要，越到后面介绍越短，这也说明最开始的内容是最复杂的。那么，就让我们做好准备，开始第一天的学习吧。啊，大家不用紧张，放松！放松！

第 15 天

多任务（1）

- 挑战任务切换（harib12a）
- 任务切换进阶（harib12b）
- 做个简单的多任务（1）（harib12c）
- 做个简单的多任务（2）（harib12d）
- 提高运行速度（harib12e）
- 测试运行速度（harib12f）
- 多任务进阶（harib12g）

1 挑战任务切换（harib12a）

“话说，多任务到底是啥呢？”我们今天的内容，就从这个问题开始吧。

多任务，在英语中叫做“multitask”，顾名思义就是“多个任务”的意思。简单地说，在Windows等操作系统中，多个应用程序同时运行的状态（也就是同时打开好几个窗口的状态）就叫做多任务。

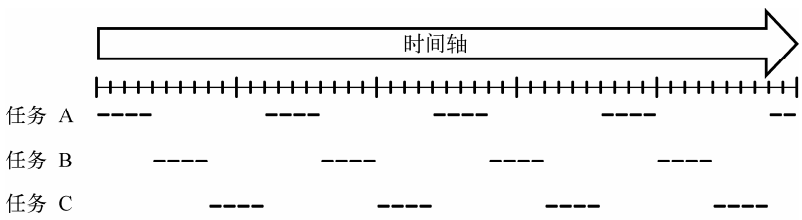
对于生活在现代社会的各位来说，这种多任务简直是理所当然的事情。比如你会一边用音乐播放软件听音乐一边写邮件，邮件写到一半忽然有点东西要查，便打开Web浏览器上网搜索。这对于大家来说这些都是家常便饭了吧。可如果没有多任务的话会怎么样呢？想写邮件的时候就必须关掉正在播放的音乐，要查东西的时候就必须先保存写到一半的邮件，然后才能打开Web浏览器……光想象一下就会觉得太不方便了。

然而在从前，没有多任务反倒是普遍的情形（那个时候大家不用电脑听音乐，也没有互联网）。在那个年代，电脑一次只能运行一个程序，如果要同时运行多个程序的话，就得买好几台电脑才行。

就在那个时候，诞生了最初的多任务操作系统，大家都觉得太了不起了。从现在开始，我们也要准备给“纸娃娃系统”添加执行多任务的能力了。连这样一个小不点儿操作系统都能够实现多任务，真是让人不由地感叹它生逢其时呀。

稍稍思考一下我们就会发现，多任务这个东西还真是奇妙，它究竟是怎样做到让多个程序同时运行的呢？如果我们的电脑里面装了好多个CPU的话，同时运行多个程序倒也顺理成章，但实际上就算我们只有一个CPU，照样可以实现多任务。

其实说穿了，这些程序根本没有在同时运行，只不过看上去好像是在同时运行一样：程序A运行一会儿，接下来程序B运行一会儿，再接下来轮到程序C，然后再回到程序A……如此反复，有点像日本忍者的“分身术”呢（笑）。



※ 1个CPU通过反复切换来执行3个任务
※ 由于切换速度很快，看上去好像在同时执行3个任务一样

为了让这种分身术看上去更完美，需要让操作系统尽可能快地切换任务。如果10秒才切换一次，那就连人眼都能察觉出来了，同时运行多个程序的戏码也就穿帮了。再有，如果我们给程序C发出一个按键指令，正巧这个瞬间系统切换到了程序A的话，我们就不得不等上20秒，才能重新轮到程序C对按键指令作出反应。这实在是让人抓狂啊（哭）。

在一般的操作系统中，这个切换的动作每0.01 ~ 0.03秒就会进行一次。当然，切换的速度越快，让人觉得程序是在同时运行的效果也就越好。不过，CPU进行程序切换（我们称为“任务切换”）这个动作本身就需要消耗一定的时间，这个时间大约为0.0001秒左右，不同的CPU及操作系统所需的时间也有所不同。如果CPU每0.0002秒切换一次任务的话，该CPU处理能力的50%都要被任务切换本身所消耗掉。这意味着，如果同时运行2个程序，每个程序的速度就只有单独运行时的1/4，这样你会觉得开心吗？如果变成这种结果，那还不如干脆别搞多任务呢。

相比之下，即便是每0.001秒切换一次任务，单单在任务切换上面也要消耗CPU处理能力的10%。大概有人会想，10%也没什么大不了的吧？可如果你看看速度快10%的CPU卖多少钱，说不定就会恍然大悟，“对啊，只要优化一下任务切换间隔，就相当于一分钱也不花，便换上了比现在更快的CPU嘛……”（笑），你也就明白了浪费10%也是很不值得的。正是因为这个原因，任务切换的间隔最短也得0.01秒左右，这样一来只有1%的处理能力消耗在任务切换上，基本上就可以忽略不计了。

关于多任务是什么的问题，已经大致讲得差不多了，接下来我们来看看如何让CPU来处理多任务。

当你向CPU发出任务切换的指令时，CPU会先把寄存器中的值全部写入内存中，这样做是为了当以后切换回这个程序的时候，可以从中断的地方继续运行。接下来，为了运行下一个程序，CPU会把所有寄存器中的值从内存中读取出来（当然，这个读取的地址和刚刚写入的地址一定是不同的，不然就相当于什么都没变嘛），这样就完成了一次切换。我们前面所说的任务切换所需要的时间，正是对内存进行写入和读取操作所消耗的时间。

■■■■■

接下来我们来看看寄存器中的内容是怎样写入内存里去的。下面这个结构叫做“任务状态段”（task status segment），简称TSS。TSS有16位和32位两个版本，这里我们使用32位版。顾名思义，TSS也是内存段的一种，需要在GDT中进行定义后使用。

```
struct TSS32 {
    int backlink, esp0, ss0, esp1, ss1, esp2, ss2, cr3;
    int eip, eflags, eax, ecx, edx, ebx, esp, ebp, esi, edi;
    int es, cs, ss, ds, fs, gs;
    int ldtr, iomap;
};
```

参考上面的结构定义，TSS共包含26个int成员，总计104字节（摘自CPU的技术资料），我特意把它们分成4行来写。从开头的backlink起，到cr3为止的几个成员，保存的不是寄存器的数据，而是与任务设置相关的信息，在执行任务切换的时候这些成员不会被写入（backlink除外，某些情况下是会被写入的）。后面的部分中我们会用到这里的设定，不过现在你完全可以先忽略它。

第2行的成员是32位寄存器，第3行是16位寄存器，应该没必要解释了吧……不对，eip好像到现在还没讲过呢。EIP的全称是“extended instruction pointer”，也就是“扩展指令指针寄存器”的意思。这里的“扩展”代表它是一个32位寄存器，也就是说其对应的16位版本叫做IP，类比一下的话，跟EAX与AX之间的关系是一样的。

EIP是CPU用来记录下一条需要执行的指令位于内存中哪个地址的寄存器，因此它才被称为“指令指针”。如果没有这个寄存器，记性不好的CPU就会忘记自己正在运行哪里的程序，于是程序就没办法正常运行了。每执行一条指令，EIP寄存器中的值就会自动累加，从而保证一直指向下一条指令所在的内存地址。

> 说点题外话，JMP 指令实际上是一个向 EIP 寄存器赋值的指令。JMP 0x1234 这种写法，CPU 会解释为 MOV EIP,0x1234，并向 EIP 赋值。也就是说，这条指令其实是篡改了 CPU 记忆中下一条该执行的指令的地址，蒙了 CPU 一把。这样一来，CPU 在读取下一条指令时，

就会去读取0x1234这个地址中的指令。你看，这不就相当于是做了一个跳转吗？

对了，如果你在汇编语言里用MOV EIP,0x1234这种写法是会出错的，还是不要尝试的好。在汇编语言中，应该使用JMP 0x1234来代替MOV EIP,0x1234。

如果在TSS中将EIP寄存器的值记录下来，那么当下次再返回这个任务的时候，CPU就可以明白应该从哪里读取程序来运行了。

按照常识，段寄存器应该是16位的才对，可是在TSS数据结构中却定义成了int（也就是DWORD）类型。我们可以大胆想象一下，说不定英特尔公司的人将来会把段寄存器变成32位的，这样想想也挺有意思的呢（笑）。

第4行的ldtr和iomap也和第1行的成员一样，是有关任务设置的部分，因此在任务切换时不会被CPU写入。也许你会想，那就和第1行一样，暂时先忽略好了——但那可是绝对不行的！如果胡乱赋值的话，任务就无法正常切换了，在这里我们先将ldtr置为0，将iomap置为0x40000000就好了。

■■■■■

关于TSS的话题暂且先告一段落，我们回来继续讲任务切换的方法。要进行任务切换，其实还得用JMP指令。JMP指令分为两种，只改写EIP的称为near模式，同时改写EIP和CS的称为far模式，在此之前我们使用的JMP指令基本上都是near模式的。不记得CS是什么了？CS就是代码段（code segment）寄存器啦。

说起来我们其实用过一次far模式的JMP指令，就在asmhead.nas的“bootpack启动”的最后一句（见8.5节）。

```
JMP     DWORD 2*8:0x0000001b
```

这条指令在向EIP存入0x1b的同时，将CS置为2*8（=16）。像这样在JMP目标地址中带冒号（:）的，就是far模式的JMP指令。

如果一条JMP指令所指定的目标地址段不是可执行的代码，而是TSS的话，CPU就不会执行通常的改写EIP和CS的操作，而是将这条指令理解为任务切换。也就是说，CPU会切换到目标TSS所指定的任务，说白了，就是JMP到一个任务那里去了。

CPU每次执行带有段地址的指令时，都会去确认一下GDT中的设置，以便判断接下来要执行的JMP指令到底是普通的far-JMP，还是任务切换。也就是说，从汇编程序翻译出来的机器语言来看，普通的far-JMP和任务切换的far-JMP，指令本身是没有任何区别的。

■■■■■

好了，枯燥的讲解就到这里，让我们实际做一次任务切换吧。我们准备两个任务：任务A和任务B，尝试从A切换到B。

首先，我们需要创建两个TSS：任务A的TSS和任务B的TSS。

本次的HariMain节选

```
struct TSS32 tss_a, tss_b;
```

向它们的ldtr和iomap分别存入合适的值。

本次的HariMain节选

```
tss_a.ldtr = 0;
tss_a.iomap = 0x40000000;
tss_b.ldtr = 0;
tss_b.iomap = 0x40000000;
```

接着将它们两个在GDT中进行定义。

本次的HariMain节选

```
struct SEGMENT_DESCRIPTOR *gdt = (struct SEGMENT_DESCRIPTOR *) ADR_GDT;

set_segmdesc(gdt + 3, 103, (int) &tss_a, AR_TSS32);
set_segmdesc(gdt + 4, 103, (int) &tss_b, AR_TSS32);
```

将tss_a定义在gdt的3号，段长限制为103字节，tss_b也采用类似的定义。

■■■■■

现在两个TSS都创建好了，该进行实际的切换了。

我们向TR寄存器存入3 * 8这个值，这是因为我们刚才把当前运行的任务定义为GDT的3号。TR寄存器以前没有提到过，它的作用是让CPU记住当前正在运行哪一个任务。当进行任务切换的时候，TR寄存器的值也会自动变化，它的名字也就是“task register”（任务寄存器）的缩写。我们每次给TR寄存器赋值的时候，必须把GDT的编号乘以8，因为英特尔公司就是这样规定的。如果你有意见的话，可以打电话找英特尔的大叔投诉哦（笑）。

给TR寄存器赋值需要使用LTR指令，不过用C语言做不到。唉，各位是不是都已经见怪不怪了啊？啥？你早就料到了？（笑）所以说，正如你所料，我们只能把它写进naskfunc.nas里面。

本次的HariMain节选

```
load_tr(3 * 8);
```

本次的naskfunc.nas节选

```
_load_tr:       ; void load_tr(int tr);
        LTR     [ESP+4]         ; tr
        RET
```

对了，LTR指令的作用只是改变TR寄存器的值，因此执行了LTR指令并不会发生任务切换。

要进行任务切换，我们必须执行far模式的跳转指令，可惜far跳转这事C语言还是无能为力，这种语言还真是不方便啊。没办法，这个函数我们也得在naskfunc.nas里创建。

本次的naskfunc.nas节选

```
_taskswitch4:   ; void taskswitch4(void);
        JMP     4*8:0
        RET
```

也许有人会问，在JMP指令后面写个RET有意义吗？也对，通常情况下确实没意义，因为已经跳转到别的地方了嘛，后面再写什么指令也不会被执行了。不过，用作任务切换的JMP指令却不太一样，在切换任务之后，再返回这个任务的时候，程序会从这条JMP指令之后恢复运行，也就是执行JMP后面的RET，从汇编语言函数返回，继续运行C语言主程序。

另外，如果far-JMP指令是用作任务切换的话，地址段（冒号前面的4*8的部分）要指向TSS这一点比较重要，而偏移量（冒号后面的0的部分）并没有什么实际作用，会被忽略掉，一般来说像这样写0就可以了。

现在我们需要在HariMain的某个地方来调用taskswitch()，可到底该写在哪里呢？唔，有了，就放在显示“10[sec]”的语句后面好了。也就是说，程序启动10秒以后进行任务切换。

本次的HariMain节选

```
    } else if (i == 10) { /* 10秒计时器*/
        putfonts8_asc_sht(sht_back, 0, 64, COL8_FFFFFF, COL8_008484, "10[sec]", 7);
        taskswitch4();  /*这里！  */
    } else if (i == 3) { /* 3秒计时器  */
```

■■■■■

大功告成了？不对，我们还没准备好tss_b呢。在任务切换的时候需要读取tss_b的内容，因此我们得在TSS中定义好寄存器的初始值才行。

本次的HariMain节选

```
    tss_b.eip = (int) &task_b_main;
    tss_b.eflags = 0x00000202; /* IF = 1; */
    tss_b.eax = 0;
```

```
    tss_b.ecx = 0;
    tss_b.edx = 0;
    tss_b.ebx = 0;
    tss_b.esp = task_b_esp;
    tss_b.ebp = 0;
    tss_b.esi = 0;
    tss_b.edi = 0;
    tss_b.es = 1 * 8;
    tss_b.cs = 2 * 8;
    tss_b.ss = 1 * 8;
    tss_b.ds = 1 * 8;
    tss_b.fs = 1 * 8;
    tss_b.gs = 1 * 8;
```

乍看之下，貌似会有很多看不懂的地方吧，我们从后半段对寄存器赋值的地方开始看。这里我们给cs置为GDT的2号，其他寄存器都置为GDT的1号，asmhead.nas的时候也是一样的。也就是说，我们这次使用了和bootpack.c相同的地址段。当然，如果你用别的地址段也没问题，不过这次我们只是想随便做个任务切换的实验而已，这种麻烦的事情还是以后再说吧。

继续看剩下的部分，关于eflags的赋值，如果把STI后的EFLAGS的值通过io_load_eflags赋给变量的话，该变量的值就显示为0x00000202，因此在这里就直接使用了这个值，仅此而已。如果还有看不懂的地方，大概就是eip和esp的部分了吧。

■■■■■

在eip中，我们需要定义在切换到这个任务的时候，要从哪里开始运行。在这里我们先把task_b_main这个函数的内存地址赋值给它。

本次的bootpack.c节选

```
void task_b_main(void)
{
    for (;;) { io_hlt(); }
}
```

这个函数只执行了一个HLT，没有任何实际作用，后面我们会对它进行各种改造，现在就先这样吧。

■■■■■

task_b_esp是专门为任务B所定义的栈。有人可能会说，直接用任务A的栈不就好了吗？那可不行，如果真这么做的话，栈就会混成一团，程序也无法正常运行。

本次的HariMain节选

```
int task_b_esp;

task_b_esp = memman_alloc_4k(memman, 64 * 1024) + 64 * 1024;
```

总之先写成这个样子了。我们为任务B的栈分配了64KB的内存，并计算出栈底的内存地址。请各位回忆一下向栈PUSH数据（入栈）的动作，ESP中存入的应该栈末尾的地址，而不是栈开头的地址。

好了，我们已经讲解得够多了，现在总算是万事俱备啦，马上“make run”一下吧。这个程序如果运行正常的话应该是什么样子呢？嗯，启动之后的10秒内，还是跟以前一样的，10秒一到便执行任务切换，task_b_main开始运行。因为task_b_main只有一句HLT，所以接下来程序就全部停止了，鼠标和键盘也应该都没有反应了。

唔……这样看起来好像很无聊啊，算了，总之我们先来“make run”吧。10秒钟的等待还真是漫长……哇！停了停了！

看来我们的首次任务切换获得了圆满成功。

输入到一半就停住了哦！

2　任务切换进阶（harib12b）

刚才我们只是实现了一次性从任务A切换到任务B，现在我们要尝试再切换回任务A。好，那我们就在切换到任务B的5秒后，让它再切换回任务A吧。

这其实很容易，只要稍微改写一下task_b_main就可以了。

本次的bootpack.c节选

```
void task_b_main(void)
{
    struct FIFO32 fifo;
    struct TIMER *timer;
    int i, fifobuf[128];

    fifo32_init(&fifo, 128, fifobuf);
    timer = timer_alloc();
    timer_init(timer, &fifo, 1);
    timer_settime(timer, 500);

    for (;;) {
        io_cli();
        if (fifo32_status(&fifo) == 0) {
            io_stihlt();
        } else {
            i = fifo32_get(&fifo);
            io_sti();
            if (i == 1) { /*超时时间为5秒  */
                taskswitch3(); /*返回任务A */
            }
        }
    }
}
```

你看，这样就搞定了。在这里所使用的变量名，比如fifo、timer等，和HariMain里面是一样的，不过别担心，计算机会把它们当成不同的变量来处理。无论我们对这里的变量如何赋值，都不会影响到HariMain中的对应变量。这并不是因为它们处于不同的任务，而是因为它们名字虽然一样，但实际上根本是不同的变量（之前一直没有机会解释这一点，现在稍微晚了点，不过还是在这里讲一下吧）。

对了，taskswitch3还没有创建，我们需要创建它。

本次的naskfunc.nas节选

```
_taskswitch3:   ; void taskswitch3(void);
        JMP     3*8:0
        RET
```

好了，准备完毕！

■■■■■

我们来“make run”一下。哇，经过10秒之后光标停止闪烁，鼠标没有反应，键盘也无法输入文字了。然而又过了5秒，光标又重新开始闪烁，刚才键盘没反应的时候打进去的字一口气全都冒了出来，鼠标也又能动了。

这就说明我们已经成功回到了任务A并继续运行了，真顺利呀。

3　做个简单的多任务（1）（harib12c）

接下来，我们要实现更快速的，而且是来回交替的任务切换。这样一来，我们就可以告别光标停住、鼠标卡死、键盘打不了字等情况，从而让两个任务看上去好像在同时运行一样。

在开始动手之前，笔者认为像taskswitch3、taskswitch4这种写法实在不好。假设我们有100个任务，难道就要创建100个任务切换函数不成？这样肯定不行，最好是写成一个函数，比如像taskswitch(3);这样。

为了解决这个问题，我们先创建这样一个函数。

本次的naskfunc.nas节选

```
_farjmp:        ; void farjmp(int eip, int cs);
        JMP     FAR [ESP+4]             ; eip, cs
        RET
```

“JMP FAR”指令的功能是执行far跳转。在JMP FAR指令中，可以指定一个内存地址，CPU会从指定的内存地址中读取4个字节的数据，并将其存入EIP寄存器，再继续读取2个字节的数据，并将其存入CS寄存器。当我们调用这个函数，比如farjmp(eip,cs);，在[ESP+4]这个位置就存放了eip的值，而[ESP+8]则存放了cs的值，这样就可以实现far跳转了。

因此我们需要将调用的部分改写如下：

```
taskswitch3(); → farjmp(0, 3 * 8);
taskswitch4(); → farjmp(0, 4 * 8);
```

■■■■■

现在我们来缩短切换的间隔。在任务A和任务B中，分别准备一个timer_ts变量，以便每隔0.02秒执行一次任务切换。这个变量名中的ts就是“task switch”的缩写，代表“任务切换计时器”的意思。

本次的bootpack.c节选

```
void HariMain(void)
{
    （中略）

    timer_ts = timer_alloc();
    timer_init(timer_ts, &fifo, 2);
    timer_settime(timer_ts, 2);

    （中略）

    for (;;) {
        io_cli();
```

```
        if (fifo32_status(&fifo) == 0) {
            io_stihlt();
        } else {
            i = fifo32_get(&fifo);
            io_sti();
            if (i == 2) {
                farjmp(0, 4 * 8);
                timer_settime(timer_ts, 2);
            } else if (256 <= i && i <= 511) { /*键盘数据*/
                (中略)
            } else if (512 <= i && i <= 767) { /*鼠标数据*/
                (中略)
            } else if (i == 10) { /* 10秒计时器*/
                putfonts8_asc_sht(sht_back, 0, 64, COL8_FFFFFF, COL8_008484, "10[sec]", 7);
            } else if (i == 3) { /* 3秒计时器*/
                putfonts8_asc_sht(sht_back, 0, 80, COL8_FFFFFF, COL8_008484, "3[sec]", 6);
            } else if (i <= 1) { /*光标用计时器*/
                (中略)
            }
        }
    }
}

void task_b_main(void)
{
    struct FIFO32 fifo;
    struct TIMER *timer_ts;
    int i, fifobuf[128];

    fifo32_init(&fifo, 128, fifobuf);
    timer_ts = timer_alloc();
    timer_init(timer_ts, &fifo, 1);
    timer_settime(timer_ts, 2);

    for (;;) {
        io_cli();
        if (fifo32_status(&fifo) == 0) {
            io_stihlt();
        } else {
            i = fifo32_get(&fifo);
            io_sti();
            if (i == 1) { /*任务切换*/
                farjmp(0, 3 * 8);
                timer_settime(timer_ts, 2);
            }
        }
    }
}
```

上面的代码应该没有什么难点，不过还是稍微解释一下吧。在每个任务中，当从farjmp返回的时候，我们都将计时器重新设定到0.02秒之后，以便让程序在返回0.02秒之后，再次执行任务切换。

好了，这样做是不是能像我们所设想的那样，让键盘和鼠标持续响应呢？我们来“make run”……不错，键盘打字、鼠标操作、光标闪烁，全都运行正常，完全没有卡住。我们成功了。

不过，我们真的成功了吗？感觉不是很靠谱啊，task_b_main到底有没有运行啊？嗯，下面我们想办法来确认一下。

4　做个简单的多任务（2）（harib12d）

为了确认task_b_main到底有没有运行，我们需要让task_b_main显示点什么东西出来，最好是显示点会动的东西，要不还是让它数数吧……喂喂，是谁在下面叫“又来了啊”？（笑）

本次的bootpack.c节选

```
void task_b_main(void)
{
    struct FIFO32 fifo;
    struct TIMER *timer_ts;
    int i, fifobuf[128], count = 0;
    char s[11];
    struct SHEET *sht_back;

    (中略)

    for (;;) {
        count++;
        sprintf(s, "%10d", count);
        putfonts8_asc_sht(sht_back, 0, 144, COL8_FFFFFF, COL8_008484, s, 10);
        io_cli();
        if (fifo32_status(&fifo) == 0) {
            io_sti();
        } else {
            (中略)
        }
    }
}
```

写到这里，我们遇到了一个问题，那就是sht_back。HariMain知道这个变量的值，但task_b_main可不知道。怎么办呢？怎样才能把这个变量的值从任务A传递给任务B呢？随便找一个内存地址存进去，然后再从那里读出来，这样应该可以吧。好，就用0x0fec这个地址，这个地址是BOOTINFO−4。

本次的HariMain节选

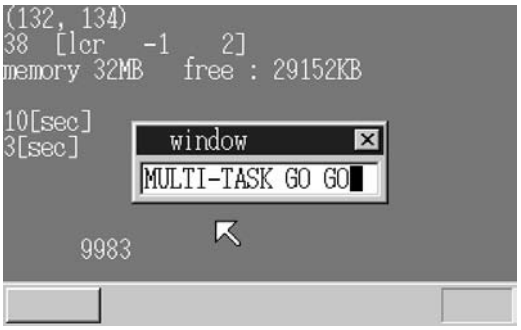
```
    *((int *) 0x0fec) = (int) sht_back;
```

本次的task_b_main节选

```
    sht_back = (struct SHEET *) *((int *) 0x0fec);
```

这里用了很多强制数据类型转换操作，代码比较难读，不过就先这样吧。

现在让我们来运行一下。不知道结果如何，心里好紧张啊。“make run”，哇，动了动了！task_b_main和HariMain在同时运行！当然，实际上只是因为切换速度很快，所以造成了在同时运行的假象。无论如何，我们的多任务取得了圆满成功！



多任务成功

（其实我们在harib12c的时候就已经成功实现了多任务，只不过当时还没有加入显示功能，所以无法实际感受到而已。）

5 提高运行速度（harib12e）

刚开始看到harib12d的成果还觉得挺感动的，过段时间头脑冷静下来以后再看的话，发现task_b_main数数的速度即便在真机环境下运行还是非常慢，我们得想办法提高它的运行速度。Harib10i在7秒钟的时间内可以数到0099969264，相比之下，harib12d也太慢了。任务A和任务B交替运行的情况下，性能下降到原来的一半还可以理解，如果比这个还慢的话就让人无法忍受了。

那运行速度为什么会这么慢呢？因为我们的程序每计1个数就在画面上显示一次，但1秒钟之内刷新100次以上的话，人眼根本就分辨不出来，所以我们不需要计1个数就刷新一次，只要每隔0.01秒刷新一次就足够了。

本次的bootpack.c节选

```
void task_b_main(struct SHEET *sht_back)
{
```

```
    struct FIFO32 fifo;
    struct TIMER *timer_ts, *timer_put;
    int i, fifobuf[128], count = 0;
    char s[12];

    fifo32_init(&fifo, 128, fifobuf);
    timer_ts = timer_alloc();
    timer_init(timer_ts, &fifo, 2);
    timer_settime(timer_ts, 2);
    timer_put = timer_alloc();
    timer_init(timer_put, &fifo, 1);
    timer_settime(timer_put, 1);

    for (;;) {
        count++;
        io_cli();
        if (fifo32_status(&fifo) == 0) {
            io_sti();
        } else {
            i = fifo32_get(&fifo);
            io_sti();
            if (i == 1) {
                sprintf(s, "%11d", count);
                putfonts8_asc_sht(sht_back, 0, 144, COL8_FFFFFF, COL8_008484, s, 11);
                timer_settime(timer_put, 1);
            } else if (i == 2) {
                farjmp(0, 3 * 8);
                timer_settime(timer_ts, 2);
            }
        }
    }
}
```

基本上就是这个样子。对了，代码开头的sht_back我们改为作为函数的参数来传递了，关于这一点我们以后会讲到，大家不必担心。

另外，上面的代码还把任务切换计时器超时的时候向FIFO写入的值改为了2。其实不改也没什么问题，只不过因为这个计时器定了0.02秒这个数，所以就顺手改成2了。

还有，count数值的显示格式改成了11位数字，因为运行速度变快了的话，说不定数字位数会不够用呢（笑）。

■■■■■

关于将sht_back的值从HariMain传递过来的方法，*((int *) 0x0fec)这样的写法感觉实在是不好看，于是果断废弃了，我们用栈来替代它。

举个例子，load_tr(123);这样的函数调用，如果从汇编语言的角度来考虑的话，参数指定的数值（123）就放在内存中，地址为ESP+4，这是C语言的一个既定机制。

既然有这种机制，那么我们可以反过来利用一下，也就是说，在HariMain里面这样写：

本次的HariMain节选

```
task_b_esp = memman_alloc_4k(memman, 64 * 1024) + 64 * 1024 - 8;

*((int *) (task_b_esp + 4)) = (int) sht_back;
```

这样一来，在任务B启动的时候，[ESP+4]这个地址里面就已经存入了sht_back的值，因此我们就欺骗了task_b_main，让它以为自己所接收到的sht_back是作为一个参数传递过来的。

可能有人不明白为什么我们要把task_b_esp的地址减8，减4不就可以了吗？我们当然不能减4，只要仔细思考一下就能搞清楚这里的奥妙。

假设memman_alloc_4k分配出来的内存地址为0x01234000，由于我们申请分配了64KB的内存空间，那么我们可以自由使用的内存地址就是从0x01234000到0x01243fff为止的这一块。如果在这里我们既不减4也不减8，而是直接加上64 * 1024的话，task_b_esp即为0x01244000。如果我们减去4，task_b_esp即为0x01243ffc，但我们写入sht_back的地址是task_b_esp＋4，算下来就变成了0x01244000，如果把4字节的sht_back值写入这个地址的话，就超出了分配给我们的内存范围（0x01234000～0x01243fff），这样不行。

而如果我们减去8，task_b_esp即为0x01243ff8，写入sht_back的地址是task_b_esp + 4，即0x01243ffc，从这个地址向后写入4字节的sht_back值，则正好在分配出来的内存范围（0x01234000～0x01243fff）内完成操作，这样就不会出问题了。

好，我们来运行一下，看看是不是变快了？还有，task_b_main有没有被我们欺骗而顺利接收到sht_back的值呢？如果这一招不成功的话，sht_back的值就会出现异常，画面上也就应该显示不出数字了。“make run”，哇，成功了，而且速度飞快（请注意，右图显示的是程序还没有运行到10秒的状态，这时就已经数到这么大的数字了！）。

即便是模拟器环境下运行速度也已经相当快了

COLUMN-9 千万不能 return？

在这一节中，task_b_main 已经变得像一个普通函数一样了，但是在这个函数中千万不能使用 return。

return 的功能，说到底其实是返回函数被调用位置的一个 JMP 指令，但这个 task_b_main 并不是由某段程序直接调用的，因此不能使用 return。如果强行 return 的话，就会像“执行数据”一样发生问题，程序无法正常运行。

HariMain 的情况也是一样的，也禁止使用 return。

我们在 15.1 节中讲过，为了记住现在正在执行的指令所在的内存地址，需要使用 EIP 寄存器，那么 return 的时候要返回的地址又记录在哪里呢？对于记性不好的 CPU 来说，肯定会把这个地址保存在某个地方，没错，它就保存在栈中，地址是[ESP]。

因此，我们不仅可以利用[ESP+4]，还可以利用[ESP]来欺骗 CPU，其实只要向[ESP]写入一个合适的值，告诉 CPU 应该返回到哪个地址，task_b_main 中就可以使用 return 了。

6 测试运行速度（harib12f）

我们的程序运行得很快，可是到底有多快呢？我们得想个办法测一下。可能有人会说，别搞这种节外生枝的玩意儿了，赶快继续往下讲吧！嗯，要说也是，这的确是有点不务正业了，不过该玩的时候也要玩一玩嘛！

我们向task_b_main添加了一些代码。

本次的bootpack.c节选

```
void task_b_main(struct SHEET *sht_back)
{
    struct FIFO32 fifo;
    struct TIMER *timer_ts, *timer_put, *timer_1s;
    int i, fifobuf[128], count = 0, count0 = 0;
    char s[12];

    （中略）
    timer_1s = timer_alloc();
    timer_init(timer_1s, &fifo, 100);
    timer_settime(timer_1s, 100);

    for (;;) {

        count++;
        io_cli();
        if (fifo32_status(&fifo) == 0) {
            io_sti();
        } else {
            i = fifo32_get(&fifo);
```

```
            io_sti();
            if (i == 1) {
                (中略)
            } else if (i == 2) {
                (中略)
            } else if (i == 100) {
                sprintf(s, "%11d", count - count0);
                putfonts8_asc_sht(sht_back, 0, 128, COL8_FFFFFF, COL8_008484, s, 11);
                count0 = count;
                timer_settime(timer_1s, 100);
            }
        }
    }
}
```

我们来运行一下，先得看看它是不是能正常工作，“make run”。不错，在模拟器环境下运行成功。

上面的数字显示的是速度哦

现在我们到真机环境运行一下。哇，好快！果然真机环境就是快，速度已经达到大约4638200[①]了。

我们把这个速度和harib10i做个对比。harib10i在7秒内计数到0099969264，即速度为每秒14281323，相比之下性能是现在的3倍。咦，怎么会这样？如果是2倍的话还可以理解，3倍就有点过分了。

看到这个结果心里当然会很不爽，我们来找找原因。嗯，每隔0.01秒刷新显示是不是不太好呢？如果显示计数是导致速度慢的原因，那干脆就别显示了吧。我们把开头的timer_settime

① 最后两位数字的值经常变动，因此用00代替。

（timer_put, 1）;一句删掉，这样一来由于计时器没有被设定，就不会引起超时中断，也就不会触发显示了。

现在仅显示速度值了

那么在真机环境下运行情况如何呢？哇，速度真的快了不少，现在的成绩是6774100，和14281323相比，性能差距为2.1倍，这样已经很令人满意了。大概JMP的地址也会影响计数的速度，另外，如果把速度显示改成每隔5秒刷新一次，任务切换间隔再改成0.03秒的话，估计性能差距可以更加接近理想的2.0倍，不过现在这个阶段我们就不去一一尝试了。

7　多任务进阶（harib12g）

到现在为止，我们所做的多任务都是依靠在HariMain和task_b_main中写入负责任务切换的代码来实现的。有人会说，这种多任务方式“不是真正的多任务”（即便如此，应该也不至于被说成是“假的”多任务）。

那么真正的多任务又是什么样的呢？真正的多任务，是要做到在程序本身不知道的情况下进行任务切换。既然如此，我们就来为“纸娃娃系统”添加真正的多任务吧。

首先我们来创建这样一个函数。

本次的mtask.c节选

```
struct TIMER *mt_timer;
int mt_tr;

void mt_init(void)
{

    mt_timer = timer_alloc();
    /*这里没有必要使用timer_init */
    timer_settime(mt_timer, 2);
    mt_tr = 3 * 8;
    return;
}
```

```
void mt_taskswitch(void)
{
    if (mt_tr == 3 * 8) {
        mt_tr = 4 * 8;
    } else {
        mt_tr = 3 * 8;
    }
    timer_settime(mt_timer, 2);
    farjmp(0, mt_tr);
    return;
}
```

mt_init函数的功能是初始化mt_timer和mt_tr的值，并将计时器设置为0.02秒之后，仅此而已。在这里，变量mt_tr实际上代表了TR寄存器，而不需要使用timer_init是因为在发生超时的时候不需要向FIFO缓冲区写入数据。具体内容请继续往下看。

接下来，mt_taskswitch函数的功能是按照当前的mt_tr变量的值计算出下一个mt_tr的值，将计时器重新设置为0.02秒之后，并进行任务切换，很简单吧。

■■■■■

下面我们来改造一下timer.c的inthandler20。

本次的timer.c节选

```
void inthandler20(int *esp)
{
    char ts = 0;
    （中略）
    for (;;) {
        /* timers的计时器全部在工作中，因此不用确认flags */
        if (timer->timeout > timerctl.count) {
            break;
        }
        /*超时*/
        timer->flags = TIMER_FLAGS_ALLOC;
        if (timer != mt_timer) {
            fifo32_put(timer->fifo, timer->data);
        } else {
            ts = 1; /* mt_timer超时*/
        }
        timer = timer->next; /*将下一个计时器的地址赋给timer */
    }
    timerctl.t0 = timer;
    timerctl.next = timer->timeout;
    if (ts != 0) {
        mt_taskswitch();
    }
    return;
}
```

在这里，如果产生超时的计时器是mt_timer的话，不向FIFO写入数据，而是将ts置为1。最后

判断如果ts的值不为0，就调用mt_taskswitch进行任务切换。

看了上面这段代码，你可能会问，为什么要用ts这个变量呢？在 /* 超时 */ 的地方直接调用mt_taskswitch不就好了吗？也就是下面这样：

出问题的例子

```
void inthandler20(int *esp)
{
     （中略）
    for (;;) {
        /* timers的计时器全部在工作中，因此不用确认flags */
        if (timer->timeout > timerctl.count) {
            break;
        }
        /*超时*/
        timer->flags = TIMER_FLAGS_ALLOC;
        if (timer != mt_timer) {
            fifo32_put(timer->fifo, timer->data);
        } else {
            mt_taskswitch();
        }
        timer = timer->next; /*将下一个计时器的地址赋给timer */
    }
    timerctl.t0 = timer;
    timerctl.next = timer->timeout;
    return;
}
```

为什么不这样写呢？这样写的确可以让代码更简短，但是会出问题。

出问题的原因在于，调用mt_taskswitch进行任务切换的时候，即便中断处理还没完成，IF（中断允许标志）的值也可能会被重设回1（因为任务切换的时候会同时切换EFLAGS）。这样可不行，在中断处理还没完成的时候，可能会产生下一个中断请求，这会导致程序出错。

因此我们需要采用这样的设计——等中断处理全部完成之后，再在必要时调用mt_taskswitch。

■■■■■

接下来我们只需要将HariMain和task_b_main里面有关任务切换的代码删掉即可。删代码没什么难度，而且HariMain又很长，为了节约纸张我们就省略了，只把task_b_main写在下面吧。

本次的bootpack.c节选

```
void task_b_main(struct SHEET *sht_back)
{
    struct FIFO32 fifo;
    struct TIMER *timer_put, *timer_1s;
```

```
    int i, fifobuf[128], count = 0, count0 = 0;
    char s[12];

    fifo32_init(&fifo, 128, fifobuf);
    timer_put = timer_alloc();
    timer_init(timer_put, &fifo, 1);
    timer_settime(timer_put, 1);
    timer_1s = timer_alloc();
    timer_init(timer_1s, &fifo, 100);
    timer_settime(timer_1s, 100);

    for (;;) {
        count++;
        io_cli();
        if (fifo32_status(&fifo) == 0) {
            io_sti();
        } else {
            i = fifo32_get(&fifo);
            io_sti();
            if (i == 1) {
                sprintf(s, "%11d", count);
                putfonts8_asc_sht(sht_back, 0, 144, COL8_FFFFFF, COL8_008484, s, 11);
                timer_settime(timer_put, 1);
            } else if (i == 100) {
                sprintf(s, "%11d", count - count0);
                putfonts8_asc_sht(sht_back, 0, 128, COL8_FFFFFF, COL8_008484, s, 11);
                count0 = count;
                timer_settime(timer_1s, 100);
            }
        }
    }
}
```

像这样，把有关任务切换的部分全部删掉就可以了。

好，我们来试试看能不能正常工作吧。“make run”，成功了，真开心！不过看上去和之前没什么区别。

和上一节相比，为什么现在的设计可以称为“真正的多任务”呢？因为如果使用这样的设计，即便在程序中不进行任务切换的处理（比如忘记写了，或者因为bug没能正常切换之类的），也一定会正常完成切换。之前那种多任务的话，如果任务B因为发生bug而无法进行切换，那么当切换到任务B以后，其他的任务就再也无法运行了，这样会造成无论是按键盘还是动鼠标都毫无反应的悲剧。

真正的多任务也成功了！

真正的多任务不会发生这样的问题，因此这种方式更好……话虽如此，但其实即便是harib12g，在任务B发生bug的情况下，也有可能出现键盘输入失去响应的问题。例如，明明写了io_cli();却忘记写io_sti();的话，中断就会一直处于禁止状态，即使产生了计时器中断请求，也不会被传递给中断处理程序。这样一来，mt_taskswitch当然也就不会被调用，这意味着任务切换也就不会被执行。

其实CPU已经为大家准备了解决这个问题的方法，因此我们稍后再考虑这个问题吧。

好，我们在真机环境下运行一下，看看速度会不会变慢。咦？速度非但没有变慢，反而变快了？运行结果是6493300，和之前的14281323相比，性能的差距是2.2倍。harib12f的时候还是差3倍来着，这次也太快了吧。我们再把timer_settime(timer_put,1);删掉，看看如果不显示计数只显示速度会怎样？说不定速度会变得更快呢？哇！结果出来了，6890930，居然达到了2.07倍，离理想值2.0倍又近了一步呢。

现在想想看，为什么速度反而会变快呢？我想这是因为在任务切换的时候，我们不再使用FIFO缓冲区的缘故。之前我们向FIFO中写入超时的编号，然后从中读取这个编号来判断是否执行任务切换，相比之下，现在的做法貌似对于CPU来说负担更小些，一定是这个原因吧。

哎呀，不知不觉就已经很晚了。今天就先到这里吧，我们明天继续。

30天自制操作系统

自己编写一个操作系统，是许多程序员的梦想。也许有人曾经挑战过，但因为太难而放弃了。其实你错了，你的失败并不是因为编写操作系统太难，而是因为没有人告诉你那其实是一件很简单的事。那么，你想不想再挑战一次呢？

这是一本兼具趣味性、实用性与学习性的书籍。作者从计算机的构造、汇编语言、C语言开始解说，让你在实践中掌握算法。在这本书的指导下，从零编写所有代码，30天后就可以制作出一个具有窗口系统的32位多任务操作系统。

本书以课题为主导，边做边玩，抛开晦涩难懂的语言，行文风格十分随性，还充满了各种欢乐的吐槽，适合操作系统爱好者和程序设计人员阅读。

图灵社区：www.ituring.com.cn
新浪微博：@图灵教育 @图灵社区
反馈/投稿/推荐信箱：contact@turingbook.com
热线：(010)51095186转604

分类建议 计算机/操作系统

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-28796-0
定价：99.00元