要实现这个项目，您必须稍微了解一下xv6是如何实现系统调用的。

系统调用是从应用程序（以用户模式运行）到操作系统（以内核模式运行）的受保护控制传输。

通用方法使内核能够保持对机器的控制，同时通常允许用户应用程序高效地运行，而无需内核干预。

为了理解系统调用，我们将特别跟踪代码中发生的事情。

系统调用允许操作系统代表用户请求以受保护的方式运行代码，既可以跳入内核

（以一种非常特定和受限的方式），也可以同时提高硬件的特权级别，以便操作系统可以执行某些受限操作。

## System Call Overview

在深入研究细节之前，我们首先概述整个过程。我们试图解决的问题很简单：我们如何构建一个系统，使操作系统能够访问机器的所有资源（包括访问特殊指令、物理内存和任何设备），而用户程序只能以受限的方式这样做？

我们实现这一目标的方法是通过硬件支持。硬件必须明确地具有特权的概念，从而能够区分操作系统何时运行和典型的用户应用程序。

## Getting Into The Kernel: A Trap

系统调用的第一步从应用程序的用户级开始。希望进行系统调用的应用程序（如read（））调用相关的库例程。然而，系统调用的库版本所做的只是在相关寄存器中放置适当的参数，并发出某种trap指令，

正如我们在usys.S的扩展版本中看到的那样（一些宏用于定义这些函数，以便使内核开发人员的生活更轻松；

该示例显示宏（已扩展到实际的汇编代码）。

在这里，我们可以看到read（）库函数实际上没有什么作用；它将值5移到寄存器%eax中，

并发出x86 trap指令，该指令被混淆地称为int（interrupt的缩写）。

内核将使用%eax中的值来向量化正确的系统调用，即，它确定调用哪个系统调用。

int指令接受一个参数（这里是64），它告诉硬件这是哪种陷阱类型。

在xv6中，trap 64用于处理系统调用。传递给系统调用的任何其他参数都在堆栈上传递。

一旦int指令被执行，硬件将接管并代表调用者执行一系列工作。

硬件所做的一件重要的事情是将CPU的特权级别提升到内核模式；

在x86上，这通常意味着从CPL（当前特权级别）3（用户应用程序运行的级别）移动到cpl0（内核运行的级别）。

是的，在特权级别之间有几个，但是大多数系统不使用这些。

Main.c

硬件所做的第二件重要的事情是将控制转移到系统的陷阱向量。

要使硬件知道在发生特定陷阱时要运行什么代码，

操作系统在引导时必须确保在发生此类陷阱时通知硬件要运行的代码的位置。这在main.c中完成，如下所示：

SETGATE（）宏是此处的相关代码。它用于设置idt数组，以便在发生各种陷阱和中断时指向要执行的正确代码。

对于系统调用，我们感兴趣的是单个SETGATE（）调用（在循环之后）。

下面是宏的作用（以及它设置的门描述符）：

mmu.h

从代码中可以看到，SETGATE（）宏所做的一切就是设置内存中数据结构的值。

最重要的是off参数，它告诉硬件陷阱处理代码的位置。在初始化代码中，值向量[T\u SYSCALL]被传入；因此，无论向量数组指向什么，都将是发生系统调用时要运行的代码。还有其他细节（也很重要）；请参阅x86硬件体系结构手册（特别是第3a和3b章）以获取更多信息。

但是请注意，我们仍然没有将此信息通知硬件，而是填充了一个数据结构。

实际的硬件通知发生在引导序列的稍后一点；在xv6中，它发生在文件main.c中的例程mpmain（）中，

该例程在trap.c中调用idtinit，该例程在include文件x86.h中调用lidt（）：

在这里，您可以看到如何（最终）调用单个汇编指令来告诉硬件在内存中的何处找到中断描述符表（IDT）。

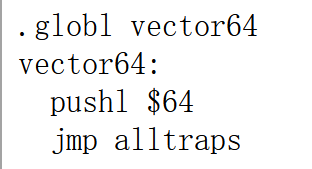
注意，这是在mpmain（）中完成的，因为系统中的每个处理器都必须有这样一个表（当然它们都使用相同的表）。

最后，在执行这个指令（只有在内核以特权模式运行时才可能）之后，我们可以考虑当用户应用程序调用系统调用时会发生什么。

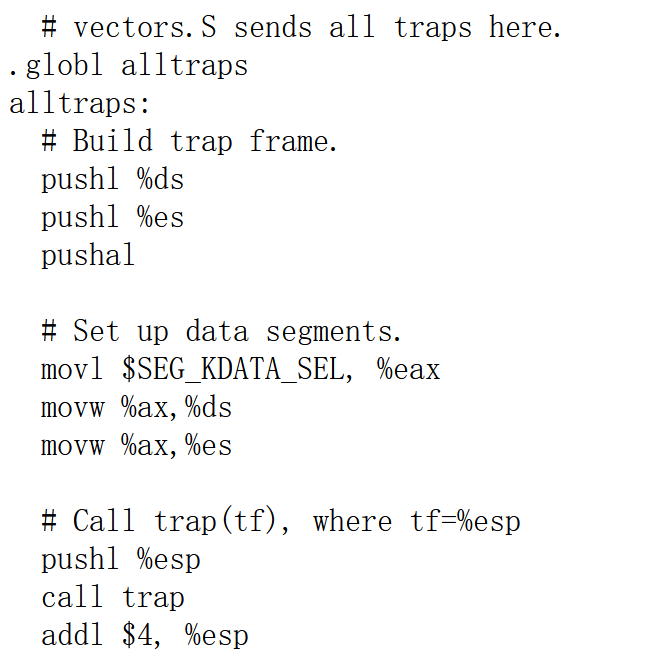
**从低级到C陷阱处理程序**

操作系统已经仔细地设置了它的陷阱处理程序，因此，一旦应用程序通过int指令发出系统调用，我们就可以看到操作系统端发生了什么。在运行任何代码之前，硬件必须执行许多任务。它要做的第一件事是软件本身难以/不可能完成的任务，包括将当前PC（英特尔术语中的IP或EIP）保存到堆栈上，以及许多其他寄存器，如eflags寄存器（包含程序运行时CPU的当前状态）、堆栈指针等第四。通过查看x86.h中定义的trapframe结构，可以看出硬件需要节省什么。

从trapframe结构的底部可以看到，trapframe的一些部分由硬件填充（直到err字段）；其余部分将由OS保存。运行的第一个代码是vector64（），如vectors.S中所示（由脚本自动生成）向量.pl).



此代码将trap编号推送到堆栈上（填充trap帧的trapno字段），然后调用alltraps（）将大部分上下文保存到trap帧中。

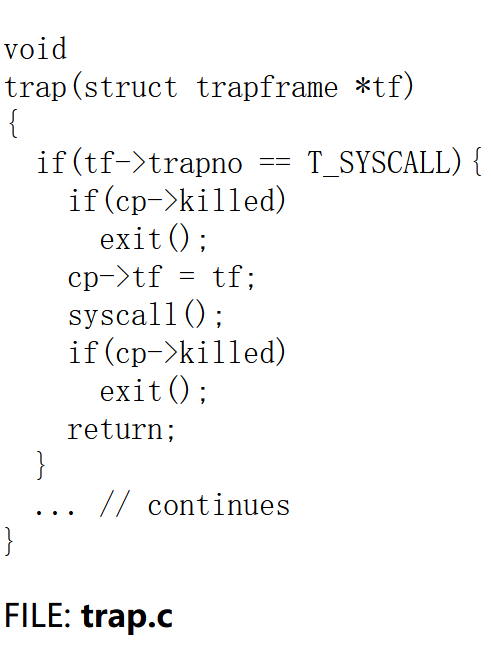


alltraps（）中的代码在通过pushal指令将剩余的通用寄存器推送到trap帧之前，将更多的段寄存器（这里还没有描述）推送到堆栈上。

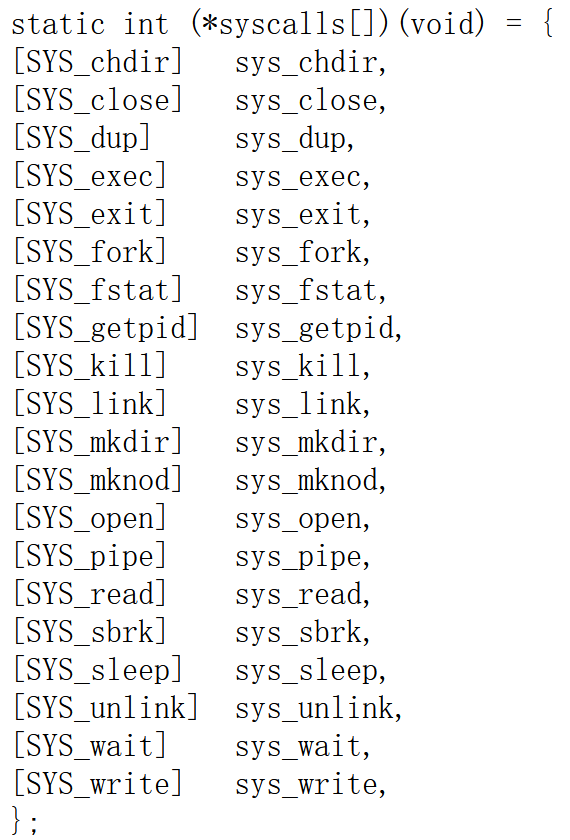
然后，操作系统更改描述符段和额外段寄存器，以便它可以访问自己的（内核）内存。最后调用C trap处理程序。

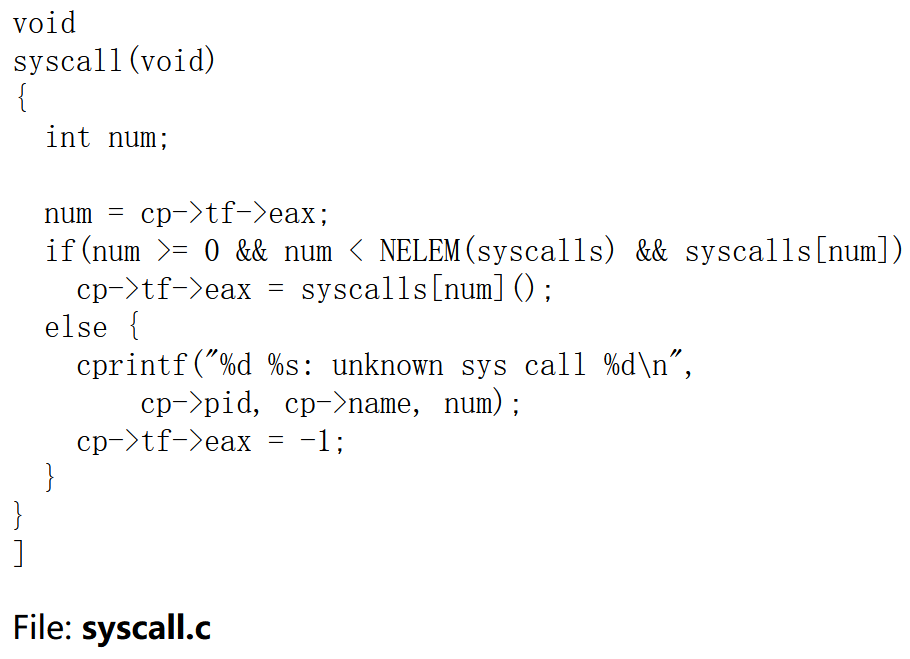
**C陷阱处理程序**

一旦完成了设置陷阱框架的底层细节，底层汇编代码就会调用一个名为trap（）的通用C陷阱处理程序，并向其传递一个指向陷阱框架的指针。在所有类型的中断和陷阱上调用这个陷阱处理程序，从而检查陷阱帧（trapno）的trap number字段以确定要做什么。第一个检查是系统调用陷阱号（T\u SYSCALL，或traps.h中任意定义的64），它随后处理系统调用，如下所示：



代码不太复杂。它检查当前进程（发出系统调用的进程）是否已终止；如果已终止，它将简单地退出并清除该进程（因此不会继续进行系统调用）。然后它调用syscall（）来实际执行系统调用；下面将详细介绍这一点。最后，它检查进程是否在返回之前再次被终止。请注意，我们将更详细地遵循下面的返回路径。



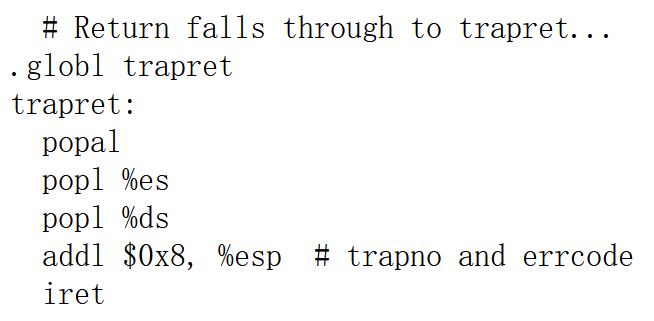


**引导到系统调用**

一旦我们最终进入syscall.c中的syscall（）例程，就没有多少工作要做了（见上文）。**系统调用号**已在寄存器%eax中传递给我们，现在我们从trap帧中解包该号码，并使用它调用系统调用表syscalls[]中定义的相应例程。几乎所有操作系统都有一个类似的表来定义它们支持的各种系统调用。在仔细检查系统调用号是否在界限内之后，将调用pointed to例程来处理调用。例如，如果系统调用read（）是由用户调用的，那么例程sys\u read（）将在此处调用。您可能会注意到，返回值存储在%eax中以传回给用户。

**返回路径**

返回路径非常简单。首先，系统调用返回一个整数值，syscall（）中的代码获取该整数值并将其放入陷阱帧的%eax字段中。然后代码返回到trap（），trap（）只返回到程序集陷阱处理程序中调用它的位置。



File: **trapasm.S**

这个返回代码不会做太多的工作，只是确保从堆栈中弹出相关值以恢复正在运行的进程的上下文。最后，还有一条特殊的指令被称为：iret，或trap返回指令。此指令类似于过程调用的返回，但同时将权限级别降低回用户模式，并跳回调用的int指令之后的指令以调用系统调用，从而恢复已保存到陷阱帧中的所有状态。此时，read（）的用户存根（如usys.S代码中所示）将再次运行，它只使用过程调用指令（ret）的正常返回来返回调用方。

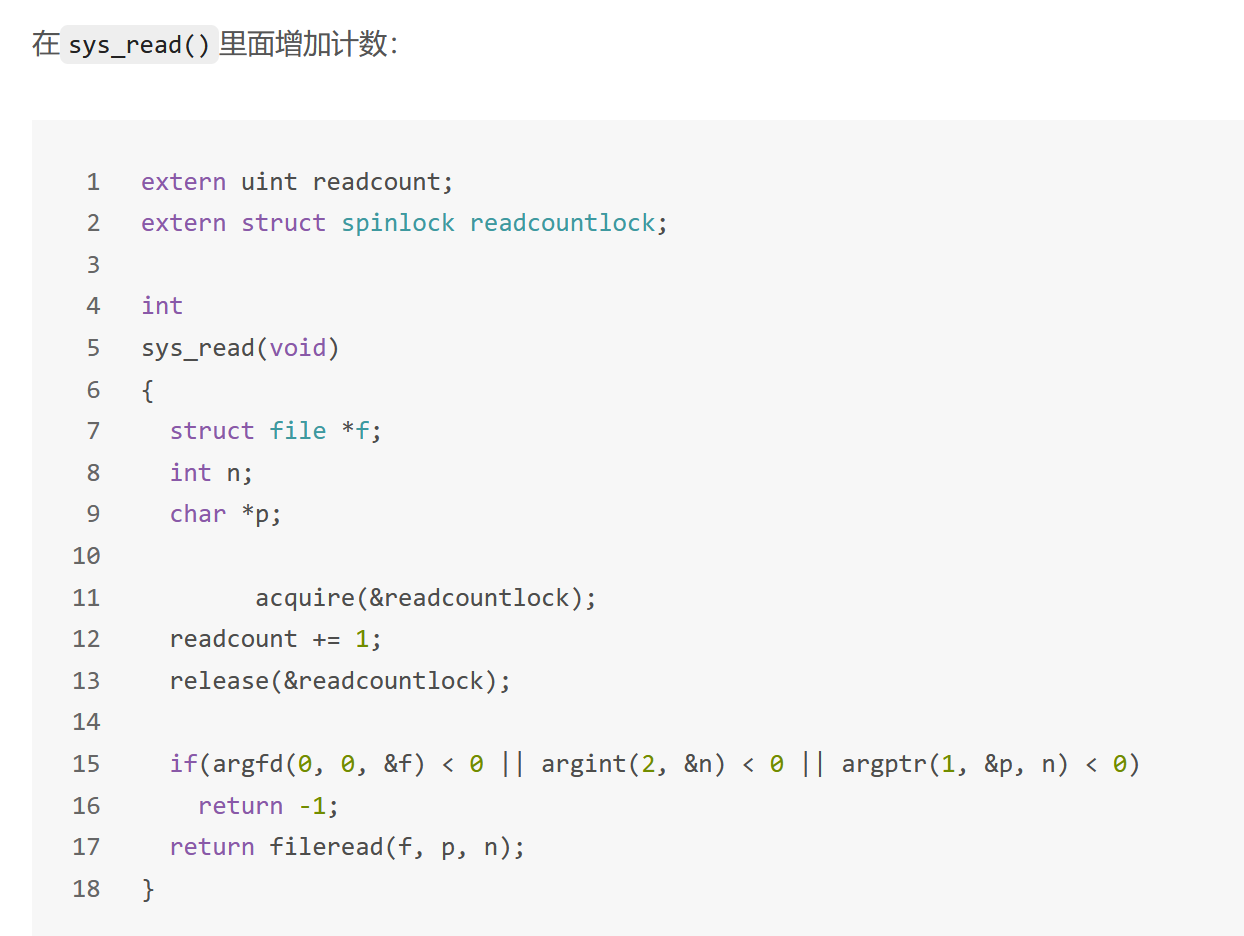
**summary**

我们已经在系统调用中看到了进出内核的路径。可以看出，它比简单的过程调用要复杂得多，并且需要一个代表操作系统和硬件的谨慎协议，以确保在输入和返回时正确保存和恢复应用程序状态。和往常一样，这个概念很简单：对于操作系统，魔鬼总是在细节中。









**概述**

在这个任务中，您将实现一个命令行解释器（CLI），或者更常见的shell。shell应该以这种基本方式操作：当您键入命令（响应其提示）时，shell将创建一个子进程，执行您输入的命令，然后在完成后提示更多用户输入。

您实现的shell与您在Unix中每天运行的shell类似，但比后者更简单。如果你不知道你在运行什么shell，可能是bash。你自己应该做的一件事就是通过阅读手册页或其他在线资料来更多地了解你的shell。

**程序规范**

**基本外壳**：seush

您的基本shell称为seush（seush，自然是东南大学shell的缩写），它基本上是一个交互式循环：它反复打印一个提示seush> （注意大于号后面的空格），解析输入，执行在该输入行上指定的命令，并等待命令完成。重复此操作，直到用户类型退出。最终可执行文件的名称应为seush。

shell可以在没有参数或只有一个参数的情况下调用；其他的都是错误。以下是无参数方法：

提示> ./seush

seush>

此时，seush正在运行，并准备接受命令。别打了！

上述模式称为交互模式，允许用户直接键入命令。shell还支持批处理模式，它从批处理文件中读取输入并从中执行命令。下面是如何使用名为batch.txt的批处理文件运行shell：

提示符>/seush batch.txt

批处理模式和交互模式之间的一个区别是：在交互模式下，会打印一个提示（seush>）。在批处理模式下，不应打印任何提示。

您应该构造shell，以便它为每个新命令创建一个进程（以下讨论的内置命令除外）。基本shell应该能够解析命令并运行与该命令对应的程序。例如，如果用户键入ls-la/tmp，那么shell应该使用给定的参数-la和/tmp运行程序/bin/ls（shell如何知道运行/bin/ls？这就是所谓的shell path；更多信息见下文）。

**结构**

**基本外壳**

shell非常简单（概念上）：它在while循环中运行，反复请求输入来告诉它要执行什么命令。然后执行该命令。循环将无限期地继续，直到用户键入内置的命令exit，此时它将退出。就这样！

要读取输入行，应该使用getline（）。这允许您轻松地获得任意长的输入行。通常，shell将在交互模式下运行，用户键入一个命令（一次一个），shell对其进行操作。但是，您的shell也将支持批处理模式，在这种模式下，shell将获得一个命令输入文件；在这种情况下，shell不应该读取用户输入（来自stdin），而是从这个文件中获取要执行的命令。

在这两种模式中，如果您点击了文件结尾标记（EOF），您应该调用exit（0）并优雅地退出。

要将输入行解析为组成部分，您可能需要使用strsep（）。详细阅读手册页。

要执行命令，请查看fork（）、exec（）和wait（）/waitpid（）。有关这些功能，请参见手册页。

您将注意到exec家族中有各种各样的命令；对于此项目，必须使用execv。不应使用system（）库函数调用来运行命令。记住，如果execv（）成功，它将不会返回；如果它确实返回，则表示有错误（例如，命令不存在）。最具挑战性的部分是正确指定参数。

**路径**

在上面的示例中，用户键入ls，但shell知道执行程序/bin/ls。你的shell怎么知道的？

结果是，用户必须指定一个path变量来描述要搜索可执行文件的目录集；组成路径的目录集有时称为shell的搜索路径。path变量包含用户键入命令时要按顺序搜索的所有目录的列表。

重要提示：注意shell本身不实现ls或其他命令（除了内置命令）。它所做的只是在path指定的一个目录中查找这些可执行文件，并创建一个新进程来运行它们。

要检查某个特定文件是否存在于目录中并且是可执行的，请考虑access（）系统调用。例如，当用户键入ls，并且path设置为同时包含/bin和/usr/bin时，请尝access（“/bin/ls”，X\_OK）。如果失败，请尝试“/usr/bin/ls”。如果这也失败了，那就是一个错误。

初始shell路径应包含一个目录：`/bin'

注意：大多数shell允许您使用绝对路径或相对路径来指定二进制文件，而无需使用搜索路径。例如，用户可以键入绝对路径/bin/ls并执行ls二进制文件，而不需要搜索路径。用户还可以指定一个相对路径，该路径从当前工作目录开始，并直接指定可执行文件，例如，./main。在这个项目中，您不必担心这些特性。

**内置命令**

每当shell接受命令时，它都应该检查该命令是否为内置命令。如果是，它不应该像其他程序一样执行。相反，shell将调用内置命令的实现。例如，要实现exit内置命令，只需调用exit（0）；在你的愿望源代码，然后将退出shell。

在这个项目中，您应该实现exit、cd和path作为内置命令。

exit：当用户类型exit时，shell只需调用exit系统调用，并将0作为参数。传递任何参数以退出是错误的。

cd：cd始终接受一个参数（0或大于1的参数应作为错误发出信号）。要更改目录，请使用chdir（）系统调用和用户提供的参数；如果chdir失败，那也是一个错误。

path：path命令接受0个或多个参数，每个参数之间用空格隔开。一个典型的用法是：wish>path/bin/usr/bin，它将/bin和/usr/bin添加到shell的搜索路径中。如果用户将path设置为空，那么shell应该不能运行任何程序（除了内置命令）。path命令总是用新指定的路径覆盖旧路径。Putenv

**重定向**

很多时候，shell用户更喜欢将程序的输出发送到文件而不是屏幕。通常，shell通过>字符提供了这个很好的特性。在形式上，这被称为标准输出的重定向。为了让您的shell用户满意，您的shell还应该包括这个特性，但要稍微有点变化（下面解释）。

例如，如果用户键入ls-la/tmp>output，则屏幕上不应打印任何内容。相反，ls程序的标准输出应该被重新路由到文件输出。此外，程序的标准错误输出应该被重新路由到文件输出（关键是这与标准重定向略有不同）。

如果在运行程序之前输出文件已经存在，您应该简单地覆盖它（在截断它之后）。

重定向的确切格式是一个命令（可能还有一些参数），后跟重定向符号和文件名。多个重定向运算符或重定向符号右侧的多个文件是错误的。

注意：不要担心内置命令的重定向（例如，当您键入path/bin>file时，我们不会测试会发生什么）。

**并行命令**

shell还允许用户启动并行命令。这是通过与号运算符实现的，如下所示：

Wish>cmd1 & cmd2 args1 args2 & cmd3 args1

在这种情况下，在等待cmd1、cmd2和cmd3中的任何一个完成之前，shell应该并行运行cmd1、cmd2和cmd3（每个都带有用户传递给它的任何参数），而不是运行cmd1然后等待它完成。

然后，在启动所有这些进程之后，必须确保使用wait（）（或waitpid）等待它们完成。所有过程完成后，像往常一样将控制权返回给用户（或者，如果处于批处理模式，则转到下一行）。

程序错误

唯一的错误消息。每当遇到任何类型的错误时，都应打印此错误消息：

char error\_message[30]=“发生错误\n”；

Write(STDERR\_FILENO，error\_message，strlen(error\u message))；

错误消息应该打印到stderr（标准错误），如上所示。

在出现大多数错误后，shell只需在打印出唯一的错误消息后继续处理。但是，如果使用多个文件调用shell，或者如果shell被传递了一个错误的批处理文件，那么它应该通过调用exit（1）退出。

shell捕获的错误与程序捕获的错误之间存在差异。shell应该捕获此项目页中指定的所有语法错误。如果命令的语法看起来很完美，只需运行指定的程序。如果存在任何与程序相关的错误（例如，运行ls时ls的参数无效），shell就不必担心这个问题（相反，程序将打印自己的错误消息并退出）。

**其他提示**

在担心所有的错误情况和最终情况之前，请记住让shell的基本功能正常工作。例如，首先运行一个命令（可能首先运行一个没有参数的命令，如ls）。

接下来，添加内置命令。然后，尝试重定向。最后，考虑并行命令。其中每一项都需要在解析上付出更多的努力，但是每一项都不应该太难实现。

在某种程度上，您应该确保您的代码对各种类型的空白（包括空格和制表符）是健壮的。一般来说，用户应该能够在命令、参数和各种运算符前后放置不同数量的空白；但是，操作符（重定向和并行命令）不需要空格。

从工作一开始就检查所有系统调用的返回代码。这通常会捕获调用这些新系统调用时的错误。这也是很好的编程意识。

打你自己的代码！您是这段代码的最佳（在本例中也是唯一）测试人员。向它抛出许多不同的输入，确保shell运行良好。好的代码来自测试；您必须运行许多不同的测试以确保事情按预期运行。别客气——其他用户肯定不会。

最后，保留代码的版本。更高级的程序员将使用诸如git之类的源代码控制系统。最起码，当一个功能正常工作时，复制一个.c文件（可能是一个有版本号的子目录，如v1、v2等）。通过保留旧的、可用的版本，您可以轻松地添加新的功能，安全的是，如果需要，您可以随时返回到旧的、可用的版本。

测试点1：cd没有参数

测试点2：cd两个参数

测试点3：ls with a bad directory name

测试点4：input to run misc. Commands cd tests/pa2-test

测试点5：exit后面接了参数

测试点6：try running a shell script without setting path //path命令测试

测试点7：set path, run a shell script overwrite path and then try running the script again

测试点8：redirection with no output file specified

测试点9：重定向过多output参数

测试点10：重定向过多>

测试点11：正常的重定向测试

测试点12：上来就>前面都没有指令

测试点13：批处理模式 文件打开无效：in里啥也没有

测试点14：批处理模式 文件bad？坏的？

测试点15：好多空格 错误

测试点16：只有一个&

测试点17：最后有个&

测试点18：正常并行指令测试

测试点19：&前后没有空格

测试点20：重定向与并行结合

测试点21：empty command

测试点22：try to check that commands are not executed serially