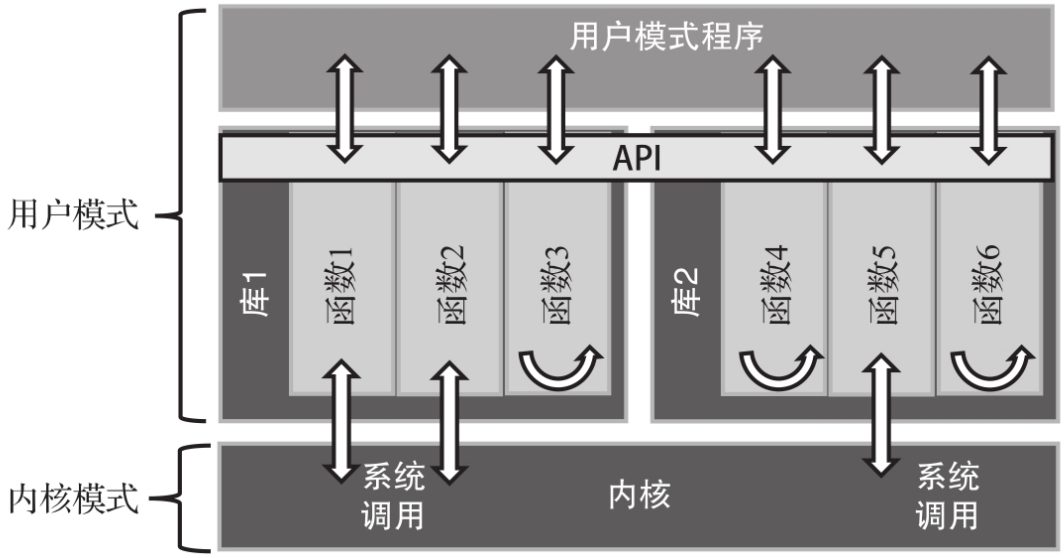

图10-14 跨库实现的操作系统API。这些库中的一些函数对内核进行系统调用，其他的则没有。用户模式程序与API交互

在典型的Linux发行版中，许多可用的Linux内核系统调用都是通过GNU C库（或glibc）提供的。这个库也包括了C语言的标准库，含不需系统调用的函数。主glibc文件通常按libc.so.6这种形式来命名，其中so的意思是共享对象（shared object），6表示版本。使用这个库，用C或C++的软件开发人员能轻松利用由Linux内核和C运行时库提供的功能。考虑到这个库在大多数Linux发行版中无处不在，把glibc中的函数看作标准Linux API的一部分是合理的。

注意

请参阅设计25尝试使用GNU C库。

Microsoft Windows API相当宽泛，多年来，它已经发展到含有多个库。kernel32.dll、user32.dll和gdi32.dll是3个基本的Windows API库文件。从NT内核导出的系统调用通过kernel32.dll提供给用户模式程序使用。从win32k（窗口和图形系统）导出的系统调用通过user32.dll和gdi32.dll提供给用户模式程序使用。

这些文件的dll扩展名表示它们是动态链接库（dynamic link library），类似于Linux中的共享对象（.so）文件。也就是说，dll文件扩展名表示该文件包含了进程可以加载并运行的共享库代码。文件名中的后缀32是作为16位到32位Windows过渡的一部分而添加的。现在，为了兼容，64位版本的Windows仍然保留着这些文件名的后缀32。实际上，64位版本的Windows包含了这些文件的两个版本（相同的名称，不同的目录），一个用于32位应用程序，一个用于64位应用程序。

注意

程序可能不通过软件库来进行系统调用。通过设置处理器寄存器中的值并发出特定于处理器的指令，比如ARM中的SVC或x86中的SYSCALL，程序可以直接进行系统调用。但是，这需要用汇编语言编程，从而导致源代码不能在不同的处理器架构上运行。此外，操作系统的API可以包括不使用系统调用实现的函数，所以直接进行系统调用不能取代操作系统软件库。

Windows Subsystem for Linux

Linux内核和Windows NT内核公开了不同的系统调用，它们的可执行文件以不同的格式存储，这使得为其中一个操作系统编译的软件不能兼容另一个操作系统。但是，微软在2016年宣布了Windows Subsystem for Linux（WSL），即Windows 10的一个功能，它允许许多64位Linux程序无须修改就能在Windows上运行。在第一版WSL中，其实现方法是：拦截Linux可执行文件发出的系统调用，并在NT内核中处理它们。第二版WSL依靠一个真实的Linux内核来处理系统调用。这个Linux内核与NT内核一起在虚拟机中运行。我们将在第13章中详细介绍虚拟机。

10.12 应用程序二进制接口

现在我们已经介绍了应用程序编程接口（API）的概念以及它与系统调用和库的关系，接下来我们来看一个相关概念——ABI。应用程序二进制接口（Application Binary Interface，ABI）定义了软件库的机器码接口。这与定义源代码接口的API形成对比。一般来说，API在不同的处理器系列之间

是一致的，而ABI在不同的处理器系列之间是不同的。开发人员可以编写使用操作系统API的代码，然后针对多个处理器类型进行编译。源代码的目标是通用API，而编译后的代码的目标是特定于架构的ABI。

编译后，生成的机器码将遵循目标架构的ABI。这就意味着，在执行时，定义已编译程序与软件库之间交互的实际上是ABI，而不是API。重要的是，操作系统库公开的ABI始终保持一致。这种一致性使得旧程序无须重新编译，就能继续在新版本的操作系统上运行。

10.13 设备驱动程序

现在的计算机支持各种各样的硬件设备，比如显示器、键盘、相机等。这些设备每个都实现了输入/输出接口，使得设备能与系统其他部分通信。不同类型的设备使用不同的I/O方式：Wi-Fi适配器与游戏控制器的需求有很大不同。即使是相同通用类型的设备也可能实现不同的I/O方式。例如，两种不同型号的显卡与系统其他部分的通信方式可能就有很大的差异。直接与硬件的交互仅限于运行于内核模式的代码，但希望操作系统内核知道如何与每个设备进行通信是不合理的。这就是设备驱动程序的用武之地了。设备驱动程序是与硬件设备交互并为该硬件提供编程接口的软件。

通常，设备驱动程序实现为内核模块，即包含内核可以加载并在内核模式下执行的代码的文件。这是允许驱动程序访问硬件所必需的。为此，设备驱动程序具有广泛的访问权限，类似于内核本身，所以只应安装可信的驱动程序。内核与设备驱动程序协同工作，代表在用户模式下运行的代码与硬件交互。这允许在没有操作系统或应用程序不了解使用特定硬件的详细信息的情况下进行硬件交互。这是封装的一种形式。在某些情况下，驱动程序可以在用户模式下执行（比如那些使用微软的用户模式驱动程序框架的驱动程序），但这种方式仍然需要内核模式下的一些组件来处理与硬件的交互，一般这些组件由操作系统提供。

注意

请参阅设计26查看Raspberry Pi操作系统上加载的内核模块，包括设备驱动程序。

10.14 文件系统

几乎所有的计算机都有某种类型的辅存，通常是硬盘驱动器（HHD）或固态驱动器（SSD）。这些设备实际上是可读写位的容器，即使系统断电，数据也能在其上保留。存储设备按区域划分，称为分区。操作系统实现文件系统，把存储设备上的数据组织成文件和目录。在操作系统使用分区之前，必须用特定的文件系统把其格式化。不同的操作系统使用不同的文件系统。Linux通常使用ext（扩展）系列的文件系统（ext2、ext3、ext4），而Windows使用FAT（File Allocation Table，文件分配表）和NTFS（NT File System，NT文件系统）。有些操作系统把存储表示为卷（Volume），即建立在一个或多个分区上的逻辑抽象。在这样的系统中，文件系统驻留在卷上，而不是分区上。

文件是数据的容器，目录（也称为文件夹）是文件或其他目录的容器。文件的内容可以是任何东西，存储在文件中的数据的结构由把该文件写入存储设备的程序决定。类UNIX系统把它们的目录结构组织成统一的目录层次结构。这个层次结构从根目录开始，用一个正斜杠号（/）指明，其他的目录都是根目录的后代。例如，库文件存储在/usr/lib里，其中usr是根目录的子目录，lib是usr的子目录。即使系统中有多个存储设备，这种统一层次结构也适用。其他存储设备被映射到目录结构中的某个位置，称为安装（mounting）设备。例如，一个USB驱动器可以安装到/mnt/usb1。

与之相比，Microsoft Windows为每个卷都分配了一个驱动器号（A～Z）。因此，每个驱动器都有自己的根目录和目录层次结构，而不是一个统一的目录结构。Windows在其目录路径中使用反斜杠号（\），在驱动器号后面使用冒号（:）。例如，存储在驱动器C上的Windows系统文件通常定位在C:\windows\system32目录下。这种惯例可以回溯到DOS（以及更早），当时驱动器A和B预留给软盘，而驱动器C代表内部硬盘。时至今日，驱动器C常常用作安装Windows的卷的驱动器号。

注意

请参阅设计27查看Raspberry Pi操作系统上存储设备和文件的详细信息。

10.15 服务和守护进程

操作系统使得进程能在后台自动运行，无须与用户交互。这样的进程在Windows中称为服务，在类UNIX系统中称为守护进程（daemon）。典型的操作系统中包含了许多默认运行的此类服务，比如配置网络设置的服务或者按计划运行任务的服务。服务用于提供不与特定用户绑定的功能，不需要在内核模式下运行，但需要按需可用。

操作系统一般包含一个负责管理服务的组件。一些服务需要在操作系统启动时启动，另一些则需要在响应特定事件时运行。在出现意外故障时，通常应重启服务。在Windows中，服务控制管理器（Service Control Manager，SCM）执行这些类型的功能。SCM的可执行文件是services.exe，它在Windows引导过程的早期启动，只要Windows自身运行，它就会持续运行。许多现代Linux发行版已经采用systemd作为管理守护进程的标准组件，尽管在Linux中也可以用其他机制启动和管理守护进程。如前所述，systemd还充当init进程，所以它很早就在Linux引导过程中启动了，并且在系统启动时继续运行。

UNIX和Linux术语"守护进程"来源于Maxwell的daemon（恶魔），一个在物理思维实验中描述的假想生物。这个"生物"在后台工作，很像一个计算机守护进程。在计算机之外，daemon通常发音类似于"demon"，但在提到后台进程时，"DAY- mon"同样是一个可以接受的发音。从历史上看，服务是特定于Windows的术语，但现在它也用于Linux，通常是指由systemd启动的守护进程。

注意

请参阅设计28查看Raspberry Pi操作系统上的服务。

10.16 安全

操作系统为在其上运行的代码提供安全模型。在这种情况下，安全的意思是软件以及该软件的用户应该只被授予访问系统合适的部分的权限。对于像笔记本计算机或智能手机这样的个人设备来说，这似乎没什么大不了的。如果只有一个用户登录到系统中，那么他不应该有权访问所有的内容吗？不，至少默认情况下不是。用户会犯错误，包括运行不可信的代码。如果用户无意中在其设备上运行了恶意软件，操作系统可以通过限制该用户的访问帮忙限制受到的损害。在多个用户登录的共享系统上，一个用户不应能读取或修改其他用户的数据，至少在默认情况下不能。

操作系统使用多种技术来提供安全性。这里我们只介绍其中的几个。对于确保软件不会有意或无意地与其他应用程序或内核搞混来说，简单地把应用程序放入一个用户模式气泡是很有帮助的。操作系统还实现文件系统安全性，确保存储在文件中的数据只能被合适的用户和进程访问。虚存本身是安全的——内存区域可以标记为只读或可执行，这有助于限制内存滥用。为用户提供登录系统使得操作系统能根据用户身份来进行安全管理。这些都是对现代操作系统的基本期望。可惜的是，在操作系统中经常能发现安全漏洞，让恶意行为者绕过操作系统的防御。保持现代互联网连接的操作系统更新到最新状态，对维护安全性至关重要。

10.17 总结

本章讨论了操作系统，一种与计算机硬件通信并为程序执行提供环境的软件。你了解了操作系统内核、非内核组件以及内核模式与用户模式的分离。我们回顾了两个主要的操作系统系列：类UNIX操作系统和Microsoft Windows。你知道了程序在被称为进程的容器中运行，在进程中能并行执行多个线程。我们研究了以编程方式与操作系统交互的各个方面：API、系统调用、软件库、ABI。第11章将超越单设备计算，探讨互联网，研究使互联网成为可能的各种层和协议。

设计20：查看运行中的进程

前提条件：一个运行Raspberry Pi操作系统的Raspberry Pi。

在本设计中，你将查看在Raspberry Pi上运行的进程。ps工具提供了运行中进程的各种视图。让我们从下面的命令开始，该命令提供了进程的树视图。

```
$ ps -eH
```

输出应类似于下面的文本，这里，我只展示了其中的一部分：

```
  1 ?        00:00:10 systemd
 93 ?        00:00:09   systemd-journal
133 ?        00:00:01   systemd-udevd
233 ?        00:00:01   systemd-timesyn
274 ?        00:00:02   thd
275 ?        00:00:01   cron
276 ?        00:00:00   dbus-daemon
286 ?        00:00:03   rsyslogd
287 ?        00:00:01   systemd-logind
291 ?        00:00:08   avahi-daemon
296 ?        00:00:00     avahi-daemon
297 ?        00:00:01   dhcpcd
351 tty1     00:00:00   agetty
352 ?        00:00:00   agetty
358 ?        00:00:00   sshd
5016 ?       00:00:00     sshd
5033 ?       00:00:00       sshd
5036 pts/0   00:00:00         bash
5178 pts/0   00:00:00           ps
```

缩进级别表示父/子关系。例如，在上面的输出中，我们看到systemd是systemd-journal、systemd-udevd等的父节点。反过来，我们可以看到ps（当前运行的命令）是bash的子节点，bash是sshd的子节点，以此类推。

显示的列分别为：

❑**PID**：进程ID。

❑**TTY**：相关的终端。

❑**TIME**：累计CPU时间。

❑**CMD**：可执行文件名。

以这种方式运行ps时，运行中进程的数量可能会令人惊讶！操作系统处理许多事情，所以在任何给定时间运行大量进程都是很正常的。一般你会看到第一个列出的进程是PID 2，即kthreadd。这是内核线程的父级，你看到的在kthreadd下面列出的线程运行在内核模式下。另一个要注意的进程是PID 1，即init进程，启动的第一个用户模式进程。在前面的输出中，init进程是systemd。Linux内核按顺序启动init进程和kthreadd，这能保证它们分别被分配PID 1和PID 2。

让我们来看看init进程。这是启动的第一个用户模式进程，运行的具体可执行文件在不同的Linux版本中可能有所不同。你可以用ps查找用于启动PID 1的命令：

```
$ ps 1
```

你应该会看到如下输出：

```
PID TTY      STAT   TIME COMMAND
  1 ?        Ss     0:03 /sbin/init
```

这告诉你用于启动init进程的命令是/sbin/init。那么，运行/sbin/init是如何导致systemd执行的呢，就如同你在前面的ps输出中看到的那样？之所以发生这种情况是因为/sbin/init实际上是指向systemd的符号链接。符号链接指向另一个文件或目录。你可以用如下命令查看它：

```
$ stat /sbin/init

  File: /sbin/init -> /lib/systemd/systemd
  Size: 20              Blocks: 0          IO Block: 4096   symbolic link
```

在这个输出中，你可以看到/sbin/init是指向/lib/systemd/systemd的符号链接。

进程树的另一个方便视图可以用pstree工具生成，就如本章之前所述。运行pstree会展示一个从init进程开始的、格式良好的用户模式进程树。试一下：

```
$ pstree
```

或者，如果Raspberry Pi被配置成引导到桌面环境，你可能还想尝试Raspberry Pi操作系统自带的任务管理器应用程序。它提供了运行中进程的图形视图，如图10-15所示。

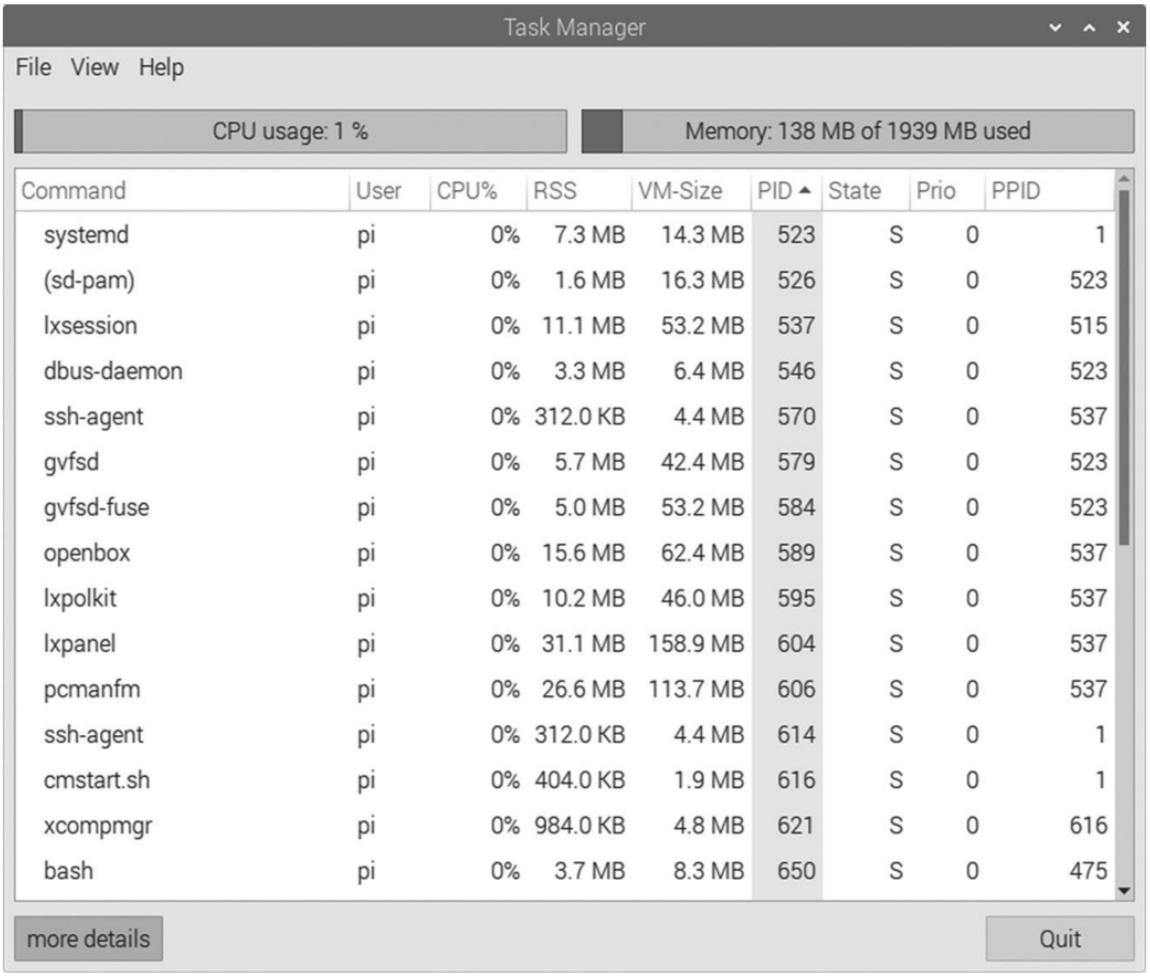

图10-15 Raspberry Pi操作系统中的任务管理器

设计21：创建并观察线程

前提条件：一个运行Raspberry Pi操作系统的Raspberry Pi。

在本设计中，你将编写程序创建一个线程，然后观察该线程的运行情况。使用文本编辑器在主文件夹根目录中创建一个名为threader.c的新文件。在文本编辑器中输入如下C代码（不需要保留缩进和空行，但需要保留换行符）：

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/syscall.h>

void * mythread(void* arg)❶
{
  while(1)❷
  {
    printf("mythread PID: %d\n", (int)getpid());❸
    printf("mythread TID: %d\n", (int)syscall(SYS_gettid));
    sleep(5);❹
  }
}
int main()❺
{
  pthread_t thread;

  pthread_create(&thread, NULL, &mythread, NULL);❻

  while(1)❼
  {
    printf("main     PID: %d\n", (int)getpid());❽
    printf("main     TID: %d\n", (int)syscall(SYS_gettid));
    sleep(10);❾
  }

  return 0;
}
```

让我们来查看一下这段源代码。我不会在这里介绍所有的细节，但总的来说，程序开始于main函数❺，它创建了运行函数mythread❶的线程❻。这意味着有两个线程：main线程和mythread线程。这两个线程都在一个无限循环❷❼中运行，它们每隔一段时间就会输出当前线程的PID和TID❸❽。根据各自的特性，mythread大约每隔5s输出一次❹，而main大约每隔10s输出一次❾。这有助于说明线程实际上是并行运行并按它们自己的计划工作的。自己试一下吧！

保存文件后，用GNU C编译器（gcc）把代码编译成可执行文件。下面的命令把threader.c作为输入，然后输出名为threader的可执行文件：

```
$ gcc -pthread -o threader threader.c
```

现在，试着用下面的命令运行代码：

```
$ ./threader
```

运行的程序应该输出如下内容（虽然PID和TID编号会有所不同）：

```
main     PID: 2300
main     TID: 2300
mythread PID: 2300
mythread TID: 2301
```

程序运行时，预计这两个线程会持续输出它们的PID和TID信息。对于程序的这个实例，PID和TID编号不会变化，因为整个过程中运行的是同样的进程和线程。你应该看到mythread输出次数是main的两倍。

让程序保持运行，看看运行中的进程和线程的列表。为此，你需要打开第二个终端窗口，并运行如下命令：

```
$ ps -e -T | grep threader
 2300  2300 pts/0    00:00:00 threader
 2300  2301 pts/0    00:00:00 threader
```

在ps命令中添加T选项，显示进程和线程。grep实用程序可以过滤输出，只查看threader进程信息。在这个输出中，第一列是PID，第二列是TID。因此，你可以看到ps的输出和程序的输出是一致的。这两个线程共享一个PID，但有不同的TID。另外还要注意，main线程的TID与其PID是一样的。对于进程中的第一个线程，这是预期的。

要停止threader程序的执行，你可以在运行它的终端窗口按下<CTRL+C>。或者，你可以使用kill实用程序，在第二个终端窗口中指定main线程的PID，如下所示：

```
$ kill 2300
```

设计22：查看虚存

前提条件：一个运行Raspberry Pi操作系统的Raspberry Pi。

在本设计中，你将查看Raspberry Pi操作系统上虚存的使用情况。我们先来看看如何在内核模式和用户模式之间划分地址空间。本设计假设你使用的是32位版本的Raspberry Pi操作系统，这表示有4GB虚拟地址空间。Linux允许把4GB划分成2GB用户模式虚拟地址和2GB内核模式虚拟地址，或者3GB用户模式虚拟地址和1GB内核模式虚拟地址。较低地址用于用户模式，较高地址用于内核模式。这意味着在2:2划分中，内核模式地址从0x80000000开始，在3:1划分中，内核模式地址从0xC0000000开始。你可以用下面的命令查看内核模式地址的起点：

```
$ dmesg | grep lowmem
```

如果dmesg命令没有产生任何输出，只需重启Raspberry Pi，然后再次运行dmesg命令即可。这个命令应产生如下输出：

```
lowmem  : 0x80000000 - 0xbb400000   ( 948 MB)
```

如果你想知道为什么当这个命令没有输出时需要重启Raspberry Pi，这里有一些背景信息。Linux内核把诊断消息记录在所谓的内核环形缓冲区中，dmesg工具将显示这个缓冲区。该缓冲区中的消息用于让用户了解内核的工作方式。这里只存储有限数量的消息，当增加新消息时，旧消息就会被移除。我们想要查看的特定消息（关于lowmem）是在系统启动时被写入的，所以，如果系统已经运行了一段时间，那么它可能已经被覆盖了。重启系统可以保证该消息再次被写入。

如你所见，在我的系统中，内核lowmem从0x80000000开始，表示是2:2划分。这意味着用户模式进程可以使用从0x00000000到0x7fffffff的地址。该地址范围可以引用2GB的内存，虽然每个进程都可以使用整个地址范围，但一般的进程实际上只需要使用这个范围内的一部分地址。某些地址映射到物理内存，但其他地址没有映射。

如果对于lowmem的起点返回的值是0xc0000000，那么系统就是按3:1划分的。这为用户模式进程提供了3GB的虚拟地址空间，从0x00000000到0xbfffffff。

我们选一个进程，然后查看它的虚存使用情况。Raspberry Pi操作系统使用Bash作为它的默认壳进程，所以，如果你在Raspberry Pi操作系统中使用命令行，那么至少应该运行一个bash实例。让我们找到bash实例的PID：

```
$ ps | grep bash
```

上面的命令应该输出与下面类似的文本：

```
 2670 pts/0    00:00:00 bash
```

在我的例子中，bash的PID是2670。现在，运行下面的命令，看看bash进程中的虚存映射。当输入该命令时，请务必把<***pid***>替换成系统返回的PID：

```
$ pmap <pid>
```

输出将与下面的内容类似，其中的每一行都代表进程地址空间中的一个虚存区域：

```
2670:   -bash
00010000    872K r-x-- bash
000f9000      4K r---- bash
000fa000     20K rw--- bash
000ff000     36K rw---   [ anon ]
00ee7000   1044K rw---   [ anon ]
76b30000     36K r-x-- libnss_files-2.24.so
76b39000     60K ----- libnss_files-2.24.so
76b48000      4K r---- libnss_files-2.24.so


76b49000      4K rw--- libnss_files-2.24.so
...
7ec2c000    132K rw---   [ stack ]
7ec74000      4K r-x--   [ anon ]
7ec75000      4K r----   [ anon ]
7ec76000      4K r-x--   [ anon ]
ffff0000      4K r-x--   [ anon ]
 total     6052K
```

第一列是区域的起始地址，第二列是区域大小，第三列表示区域权限（r=读，w=写，x=执行，p=私有，s=共享），最后一列是区域名。区域名可以是文件名，如果内存区域不是从文件映射的话，区域名也可以是标识该内存区域的名称。

你可以看到几乎每个输出的区域都在预期的用户模式地址范围（0x00000000到0x7fffffff）之内。最后一项是个例外，它对应的是ARM CPU向量页，表示一种特殊情况，因为它在标准用户模式地址范围之外。正如你在前面输出中看到的，这个特殊的bash实例在可能的2GB中总共只有6052KB（约6MB）的虚存映射，占大约0.3%。

设计23：尝试操作系统API

前提条件：一个运行Raspberry Pi操作系统的Raspberry Pi。

在本设计中，你将尝试用各种方法调用操作系统API。你将关注文件创建以及向文件中写入文本。使用文本编辑器，在主文件夹根目录中创建一个名为newfile.c的新文件。在文本编辑器中输入如下C代码：

```
#include <fcntl.h>
#include <unistd.h>

#define msg "Hello, file!\n"❶

int main()❷
{
  int fd;❸
  fd = open("file1.txt", O_WRONLY|O_CREAT|O_TRUNC, 0644);❹
  write(fd, msg, sizeof(msg) - 1);❺
  close(fd);❻
  return 0;❼
}
```

让我们检查一下源代码，以准确了解它的功能。简而言之，这个程序使用三个API函数open、write和close来创建一个新的文件、向这个文件中写入一些文本、关闭该文件。这里重点查看操作系统的API是如何允许程序与计算机硬件（特别是存储设备）交互的。让我们仔细地看看这个程序。

在必需的include语句之后，下一行把msg定义为稍后将要写入新创建文件中的文本字符串❶。然后，定义main，即程序的入口点❷。在main中，声明一个名为fd的整数❸。接下来，调用操作系统API的open函数来创建一个名为file1.txt的新文件❹。传递给open函数的其他参数指定了打开文件所用方法的详细信息。为了简单起见，这里不会涉及这些细节，你可以随意研究这些参数的含义。open函数返回一个文件描述符，它保存在fd变量中。

然后，使用write函数把msg文本写入file1.txt（由保存在fd中的文件描述符标识）❺。write函数需要输入要写的数据（msg）和写入的字节数（由sizeof（msg）-1决定）。“-1”是因为C语言用null字符终止字符串，这个字节是不需要写入输出文件的。程序现在已经完成了对文件的处理，并用文件描述符调用close函数以表示该文件不再使用❻。最后，程序退出，返回代码为0❼，表示成功。

保存文件后，使用GNU C编译器（gcc）把代码编译成可执行文件。下面的命令将newfile.c作为输入，生成一个名为newfile的可执行文件：

```
$ gcc -o newfile newfile.c
```

现在，尝试用如下命令运行该代码。你不会看到任何输出，因为文本被写入文件而不是终端：

```
$ ./newfile
```

要确定程序是否运行成功，你需要查看文件是否已经被创建。该文件应该被命名为file1.txt，且存在于当前目录下。你可以使用ls命令列出当前目录中的内容并查找该文件。若file1.txt存在，就可以用cat命令查看其内容：

```
$ ls
$ cat file1.txt
```

该命令应该把Hello,file！输出到终端，因为这是程序写入文件的文本。你可以在Raspberry Pi操作系统桌面的文件管理器应用程序中查看该文件的