

运行时虚拟机系统、系统运行方法和装置

本专利申请要求于 2023 年 7 月 28 日提交的、申请号为
5 202310943023.5、发明名称为“运行时虚拟机系统、系统运行方法和装置”的中国专利申请的优先权，该申请的全文以引用的方式并入本申请中。

技术领域

10 本公开涉及计算机技术领域，具体涉及运行时虚拟机系统、系统运行方法和装置、电子设备以及计算机可读介质。

背景技术

现有运行时虚拟机技术，一般基于 C 运行库开发，各类操作系统
15 都是通过 C 运行库提供对系统调用的封装，各类运行时虚拟机环境下执行的应用程序，实际都需要通过调用宿主操作系统的系统调用，实现底层 IO 操作，然而相对于用户态代码，执行系统调用的代价极高。

发明内容

20 本公开的实施例提出了线程虚拟化方法和装置、运行时虚拟机系统、电子设备、计算机可读介质。

第一方面，本公开的实施例提供了一种运行时虚拟机系统，该系统包括：用户态应用程序；异步 IO 模型，用于使用户态应用程序与内核态之间共享 IO 请求内存队列，以实现对物理设备的异步驱动；可在
25 用户态应用程序启动时加载的动态运行库，所述动态运行库用于被加载后生成替换用户态应用程序的运行时环境的运行时虚拟环境；其中，动态运行库包括 IO 操作库函数钩子和监控线程函数；IO 操作库函数钩子基于用户态应用程序的原生库函数构建得到，IO 操作库函数用于在运行时虚拟环境下，适配原生库函数的应用程序接口和异步 IO 模
30 型，将用户态应用程序所在操作系统的 IO 操作转化为由异步 IO 模型处理；监控线程函数用于在运行时虚拟环境下，监控异步 IO 模型的 IO

操作是否完成，并在该 IO 操作完成时，唤醒因该 IO 操作处于等待状态的任务。

在一些实施例中，上述 IO 操作库函数钩子包括：文件描述符管理钩子函数、文件描述符读写钩子函数以及 POLL 状态钩子函数；文件描述符管理钩子函数用于在运行时虚拟环境下，创建文件描述符，将文件描述符放入文件描述符表中，并将文件描述符注册至异步 IO 模型中；文件描述符管理钩子函数还用于在运行时虚拟环境下且关闭文件描述符时，将文件描述符从异步 IO 模型中进行反注册；文件描述符读写钩子函数用于在运行时虚拟环境下，接收对文件描述符的读请求或写请求，并将读请求或写请求放入异步 IO 模型的提交队列；POLL 状态钩子函数用于在运行时虚拟环境下，接收 POLL 等待请求或 POLL 唤醒请求，并将 POLL 等待请求或 POLL 唤醒请求放入异步 IO 模型的提交队列。

在一些实施例中，上述动态运行库采用协程模拟原生库函数中实现的线程，IO 操作库函数钩子包括：文件 IO 钩子函数以及文件 IO 回调函数，网络 IO 钩子函数以及网络 IO 回调函数；文件 IO 钩子函数是采用挂钩（HOOK）机制构建的、与原生库函数中文件 IO 库函数相关的函数；网络 IO 钩子函数是采用 HOOK 机制构建的、与原生库函数中网络 IO 库函数相关的函数；文件 IO 钩子函数与网络 IO 钩子函数用于在运行时虚拟环境下的挂起或等待 IO 操作完成时判断操作主体是否为协程，若为协程，采用协程切换或挂起机制挂起协程；文件 IO 回调函数用于在运行时虚拟环境下，将文件 IO 钩子函数中与 IO 相关操作改为与异步 IO 模型同步；网络 IO 回调函数用于在运行时虚拟环境下，将网络 IO 钩子函数中与 IO 相关操作改为与异步 IO 模型同步。

在一些实施例中，上述动态运行库还包括：线程调度库函数钩子，线程调度库函数钩子是采用 HOOK 机制构建的与原生库函数中线程创建、回收以及挂起库函数相关的函数，线程创建、回收以及挂起库函数为应用程序的原生运行库中能够创建、回收以及挂起线程的函数，线程调度库函数钩子用于在运行时虚拟环境下，将线程的创建、回收

替换为协程的创建、回收，并且在应用程序运行后，生成协程调度线程，以通过协程调度线程调度至少一个协程。

5 在一些实施例中，上述动态运行库还包括：内存管理库函数钩子，内存管理库函数钩子是采用 HOOK 机制构建的、与原生库函数中内存管理相关的函数，内存管理库函数钩子用于在运行时虚拟环境下，使一个线程能够启动一个或多个内存分配区，归属于当前协程调度线程的协程按内存分配区分组，并且归属于同一个内存分配区的协程共享同一个协程调度线程缓存。

10 在一些实施例中，监控线程函数在用户态应用程序启动之后且在运行时虚拟环境下，为异步 IO 模型中每个上下文创建监控线程，监控线程用于轮询异步 IO 模型的完成队列，当收到完成队列中新的就绪任务时，唤醒 IO 操作库函数钩子对应的等待队列中的协程。

第二方面，本公开的实施例提供了一种系统运行方法，该方法包括：加载异步 IO 模型的库文件，异步 IO 模型用于使用户态应用程序
15 与内核态之间共享 IO 请求内存队列，以实现对物理设备的异步驱动；在启动用户态应用程序运行时，加载动态运行库，生成运行时虚拟化环境，运行时虚拟化环境为替换用户态应用程序的运行环境的环境；在运行时虚拟环境下，响应于检测到用户态应用程序对用户态应用程序的原生库函数的调用信息，通过 IO 操作库函数将用户态应用程序
20 所在操作系统的 IO 操作转化为由异步 IO 模型处理；

在运行时虚拟化环境下，通过监控线程函数监控异步 IO 模型的 IO 操作是否完成，并在该 IO 操作完成时，通过监控线程函数唤醒因该 IO 操作处于等待状态的任务。

第三方面，本公开的实施例提供了一种系统运行装置，该装置包
25 括：文件加载单元，被配置成加载异步 IO 模型的库文件，异步 IO 模型用于使用户态应用程序与内核态之间共享 IO 请求内存队列，以实现对物理设备的异步驱动；库加载单元，被配置成在启动用户态应用程序运行时，加载动态运行库，生成运行时虚拟化环境，运行时虚拟化环境为替换用户态应用程序的运行环境的环境；转化单元，被配
30 置成在运行时虚拟环境下且检测到用户态应用程序对用户态应用程序

5 的原生库函数的调用信息时，通过 IO 操作库函数将用户态应用程序所在操作系统的 IO 操作转化为由异步 IO 模型处理；唤醒单元，被配置成在运行时虚拟环境下，通过监控线程函数监控异步 IO 模型的 IO 操作是否完成，并在该 IO 操作完成时，通过监控线程函数唤醒因该 IO 操作处于等待状态的任务。

第四方面，本公开的实施例提供了一种电子设备，该电子设备包括：一个或多个处理器；存储装置，其上存储有一个或多个程序；当一个或多个程序被一个或多个处理器执行，使得一个或多个处理器实现如第二方面中任一实施例描述的方法。

10 第五方面，本公开的实施例提供了一种计算机可读介质，其上存储有计算机程序，该程序被处理器执行时实现如第二方面中任一实施例描述的方法。

附图说明

15 通过阅读参照以下附图所作的对非限制性实施例所作的详细描述，本公开的其它特征、目的和优点将会变得更明显：

图 1 是根据本公开的运行时虚拟机系统的一个实施例的结构示意图；

20 图 2 是根据本公开的运行时虚拟机系统的另一个实施例的结构示意图；

图 3 是根据本公开的系统运行方法的一个实施例的流程图；

图 4 是根据本公开的系统运行装置的一个实施例的结构示意图；

图 5 是适于用来实现本公开的实施例的电子设备的结构示意图。

25 具体实施方式

下面结合附图和实施例对本公开作进一步的详细说明。可以理解的是，此处所描述的具体实施例仅仅用于解释相关发明，而非对该发明的限定。另外还需要说明的是，为了便于描述，附图中仅示出了与有关发明相关的部分。

30 需要说明的是，在不冲突的情况下，本公开中的实施例及实施例

中的特征可以相互组合。下面将参考附图并结合实施例来详细说明本公开。

运行时虚拟机系统往往用来提供跨平台支持或提供虚拟化运行环境，而动态库注入则是一种可被用于实现运行时虚拟机的技术方案。

- 5 向应用程序中注入一个动态库，这个动态库以其函数钩子覆盖原始应用中的函数、以及原始应用所链接的动态库中的函数。

各类运行时虚拟机环境下执行的应用程序都需要通过调用宿主操作系统的系统调用，然而执行系统调用的代价极高，这是因为：一方面，陷入系统调用和从系统调用返回，意味着一次用户态向内核态的
10 切换、以及一次内核态向用户态的切换；另一方面，用户态虚拟地址空间与内核态地址空间是相互隔离的，因而用户态和内核态之间不能直接进行内存互访，需要进行内存拷贝。鉴于系统调用的高代价，一些高性能纯用户态编程模型被提出，通过尽可能少的执行系统调用(让应用程序绝大多数时间都运行于用户态，不做任何用户态/内核态切
15 换)，提升应用程序性能以及实现更高的 IO 吞吐能力。

鉴于现有的纯用户态编程模型有的没有完备的网络协议栈和文件系统生态，有的没有提供对应用接口完整的支持，应用程序接入时需要进行代码改造，实现复杂，操作困难，人力成本较大。

为了更好地理解传统技术，下面对一些名词进行介绍：

- 20 动态库 (Shared library)：又称“动态链接库”，是操作系统中实现共享函数或对象的一种方式。编译好的函数存储于动态库中，应用程序可以动态加载此类库文件，并通过函数名字引用相应函数；同一个动态库可以被不同的应用程序加载。

- 25 C 运行库 (C runtime library)：一类动态链接库的统称，提供对系统调用和各类操作系统管理接口的封装，为应用程序在运行时与操作系统间的交互提供基于 C 语言的编程接口。应用程序可以直接基于 C 运行库进行构建，或者运行于某种基于 C 运行库构建的运行时虚拟机 (例如 Java 虚拟机——JVM)。在类 Linux 系统中，不同的发行版会提供各自的 C 运行库 (一般遵循 POSIX (Portable Operating System
30 Interface, 可移植操作系统接口) 协议或基于 POSIX 的某种扩展)；例

如，GNU 发行版中的 C 运行库就称为“glibc”，而 Android 发行版中的 C 运行库则叫做“bionic”。在 Windows 系统中，C 运行库则被称为 CRT 库。

运行时虚拟机 (Runtime virtual machine): 一种为其他应用程序提供开发接口和运行时环境的程序和库的统称。这类程序一般基于 C 运行库开发，旨在提供一种跨平台的程序运行环境。相较于“虚拟机” (Virtual machine)，运行时虚拟机并不会完整的模拟出一台虚拟主机，而只是提供运行时环境，因而更加轻量。

动态库注入 (Dynamic library injection): 逆向修改第三方应用，使其执行被注入的代码，这种行为被称为“代码注入”(Code injection); 而动态库注入，就是实现代码注入的一种方式。所谓“动态库注入”，就是通过向应用程序中加载动态库的方式，将动态库中的代码注入到目标应用，使得这部分代码被该应用执行，或者用注入的代码动态替换该应用中的原始代码。

内核态和用户态: 为保证系统运行的安全性，一般现代 CPU 都有几种不同的指令级别。在高执行级别下，代码可以执行特权指令，访问任意物理地址，这种运行级别对应着操作系统的内核态。而在低执行级别下，代码的掌控范围会受到限制，只能在对应级别允许的范围内活动，这种运行级别就对应用户态。当用户态程序需要运行操作系统内核提供的功能时，可以通过执行相应的系统调用，从用户态切换至内核态，在内核态中运行高执行级别的代码。

用户态驱动程序: 设备驱动通常运行在内核空间，但随着硬件技术的不断发展，当试图提供高并发连接服务器能力时，内核态驱动的弊端就开始逐渐显露。比如，系统调用的巨大开销 (应用程序的每次设备访问都必须通过系统调用)，对驱动程序稳定性的极高要求 (因为内核态错误会直接导致系统崩溃)，对开发者而言陡峭的学习曲线和开发接口的不稳定性 (内核版本升级时，内核驱动程序接口一旦发生变化，就需要驱动程序重新适配新版本的内核)，等等。因而，诸如英特尔、德州仪器和飞思卡尔这样的硬件公司随即开始研究这一问题，为他们的硬件提供用户态驱动的解决方案。

DPDK 和 SPDK: DPDK (Data Plane Development Kit, 数据平面开发套件) 是用户态网卡驱动解决方案, DPDK 定义了一套运行环境, 该环境为每一个 CPU 内核都定义了一个内核线程。为了保证网络吞吐最大化, 每个内核线程都会绑定其对应的 CPU 内核, 在已经绑定了内核的 CPU 内核上就不会再运行任何其他线程。因而, 内核线程会占用 CPU 算力, 启动的内核线程越多, 网络设备的并行吞吐能力就越高, 但算力占用也越多。而 SPDK (Storage Performance Development Kit, 存储性能开发套件) 则是基于 DPDK 开发的一套用户态存储设备驱动套件, 旨在为其高速存储设备提供高性能驱动程序解决方案。

10 Io_uring 技术: 一种新型异步输入/输出 (I/O) 模型, io_uring 主要为了解决系统调用和内存拷贝开销大, 仅支持直接 IO, 对数据有大小对齐限制等问题。其利用内存映射技术在用户态和内核态之间创建可共享的环状内存队列, 通过共享内存通信达到减少系统调用和内存拷贝的目的。理想情况下, 可以实现零系统调用的 IO 访问。

15 协程 (co-routine): 又称为“纤程” (Fiber thread) 或“微”线程 (Micro thread), 一种产生非抢占式多任务子程序的计算机组件。协程一般依附于某个线程, 允许不同的子程序入口在同一位置暂停或开始; 简单地说, 协程就是可以暂停和开始的函数。不同的协程可以被看作是不同的函数, 被宿主线程调用执行; 在执行过程中, 协程可以通过主动让出 (即 “yield”) 的方式返回到宿主线程调用该协程的位置; 而当宿主线程再次调用该协程函数时 (即 “resume”), 协程会从之前的让出点位置继续执行。

25 线程本地存储 (TLS, Thread local storage): 又称为线程局部存储, 是特定场景下线程安全问题的一种解决方案。简单地说, 就是一个作用域为进程全局的变量, 当其带有线程本地约束时, 这个变量在不同线程中可以有不同的取值 (即, 线程本地值)。TLS 按实现方法, 又分为两种类型, 三种情况。第一种类型, 就是编码时声明的带有线程本地约束的变量, 例如:

```
static __thread int global_tls_value = 0;
```

30 上面的变量声明, 即表示一个初始值为 0 的全局整型变量。其所

在程序的所有线程中，都可以访问此变量；但不同线程对这个变量值的变更，只会对本线程可见。这种类型的 TLS 变量，在编译阶段就可以确定其在程序数据段中的位置，以及其初始值。因而，其相关信息和初始值会被存入编译好的目标程序中，并在程序被加载时，就为其
5 锁定其在线程本地上下文中的位置。还有一种情况，就是该程序在运行中加载某个动态库的时候，如果该动态库中也存在 TLS 变量，则需在加载动态库时，再动态确定这些 TLS 在线程本地上下文中的位置。除了编码时就已经声明的 TLS 变量，还有第二种类型，就是动态创建的 TLS。

10 本公开的实施例提供的运行时虚拟机系统，包括：用户态应用程序；异步 IO 模型，用于使用户态应用程序与内核态之间共享 IO 请求内存队列，以实现对物理设备的异步驱动；可在用户态应用程序启动时加载的动态运行库，所述动态运行库用于被加载后生成替换用户态应用程序的运行时环境的运行时虚拟环境；其中，动态运行库包括 IO
15 操作库函数钩子和监控线程函数；IO 操作库函数钩子基于用户态应用程序的原生库函数构建得到，IO 操作库函数用于在运行时虚拟环境下，适配原生库函数的应用程序接口和异步 IO 模型，将用户态应用程序所在操作系统的 IO 操作转化为由异步 IO 模型处理；监控线程函数用于在运行时虚拟环境下，监控异步 IO 模型的 IO 操作是否完成，并在
20 该 IO 操作完成时，唤醒因该 IO 操作处于等待状态的任务。由此，本公开的运行动态库在用户态应用程序启动后，加载到用户态应用程序的内存，生成运行时虚拟环境，通过运行时虚拟环境可以直接替换用户态应用程序的原生运行库的运行时环境，接管用户态应用程序所在操作系统的 IO 操作，并将 IO 操作转化为由异步 IO 模型处理，通过
25 异步 IO 模型处理 IO 操作，减少了系统调用的次数，提升了 IO 性能；通过动态注入技术，在不对应用程序进行任何修改的前提下，使用户态应用程序具有了纯用户态的 IO 能力，节省了人力成本，提高了用户态应用程序操作的便利性。

30 本公开提供了一种运行时虚拟机系统，如图 1，示出了根据本公开的运行时虚拟机系统的一个实施例的结构示意图，该运行时虚拟机

系统 100 包括：用户态应用程序 101，异步 IO 模型 102 以及动态运行库 103。其中，异步 IO 模型 102 用于使用户态应用程序 101 与内核态之间共享 IO 请求内存队列，以实现对物理设备的异步驱动。

动态运行库 103 在用户态应用程序启动时加载到用户态应用程序 101 的内存，在加载动态运行库之后，生成替换用户态应用程序的运行时环境的运行时虚拟环境；动态运行库 103 包括 IO 操作库函数钩子 1031 和监控线程函数 1032；IO 操作库函数钩子 1031 基于用户态应用程序的原生库函数构建得到，IO 操作库函数 1031 用于在运行时虚拟环境下，适配原生库函数的应用程序接口和异步 IO 模型 102，将用户态应用程序所在操作系统的 IO 操作转化为由异步 IO 模型处理。

监控线程函数 1032 用于在运行时虚拟环境下，监控异步 IO 模型 102 的 IO 操作是否完成，并在该 IO 操作完成时，唤醒因该 IO 操作处于等待状态的任务。

本实施例中，用户态应用程序 101 是运行于用户态的应用程序，用户态是提供应用程序运行的空间，一般地，所有应用程序一般只能运行于用户态。用户态应用程序的原生库函数是用户态应用程序的原生 C 运行库中编译完成的函数，原生运行库是用户应用程序实现其功能需要调用的动态链接库，相对于本公开的动态运行库，应用程序需要通过多次调用宿主操作系统的系统调用，系统调用的代价较高。

本实施例中，动态运行库 103 在用户态应用程序运行时加载，生成有别于用户态应用程序的运行时环境的运行时虚拟环境，动态运行库中库函数的执行逻辑可以直接替换原生运行库中库函数的执行逻辑，或者，在一些条件下替换原生运行库中库函数的执行逻辑，在另一些条件下保留原生运行库中库函数的执行逻辑。

本实施例中，在用户态应用程序所在操作系统具有 IO 操作时，动态运行库 103 运行时的库函数可以直接将该 IO 操作转化为由异步 IO 模型处理。

本实施例中，异步 IO 模型 102 是一种基于内存映射技术实现的模型，异步 IO 模型可以使应用程序在内存分配或释放时，不会陷入任何系统调用，从而提升了内存分配或释放效率。

本实施例中，异步 IO 模型 102 可以采用 io_uring，io_uring 也采用内存映射技术在用户态和内核态之间共享 IO 缓存。顾名思义，io_uring 的内存缓存是一种“环形队列”，应用程序亦可通过循环往复的环形队列中内存缓存的状态，实现与设备之间的数据交互。不同于 DPDK 直接驱动物理设备，io_uring 可以基于文件句柄进行设备或文件访问（即支持内核协议栈和文件系统）。io_uring 把访问内核协议栈和文件系统的操作全部异步封装在了内核内部，应用程序通过 io_uring 发起的 IO 操作完全不会陷入系统调用，也无需与内核之间进行内存拷贝。

10 本实施例中，异步 IO 模型包括：提交队列（Submission Queue，SQ）和完成队列（Completion Queue，CQ）；其中，提交队列是一整块连续的内存空间存储的环形队列，用于存放将执行 I/O 操作的数据，数据是指向提交队列项数组（Submission Queue Entry，SQE）的索引，提交队列项数据是提交队列中的一项。完成队列：一整块连续的内存
15 空间存储的环形队列，用于存放 I/O 操作完成后返回的结果。

可选地，异步 IO 模型还可以采用完备了内核协议栈和文件系统的 DPDK 或者 SPDK，DPDK/SPDK 是一种完全用户态实现的异步模型，但是两者的文件系统和协议栈生态不完备，通过程序开发者对 DPDK 或 SPDK 进行代码开发可以基于文件句柄进行设备或文件访问（即在
20 用户态实现与内核协议栈和内核文件系统等效的系统），具体的代码开发过程此处不再赘述。

本实施例中，监控线程函数可以是一种在运行时虚拟环境下生成监控线程的函数，通过监控线程函数生成的监控线程可以实时监控异步 IO 模型执行 IO 操作的情况，在该 IO 操作完成时，唤醒因该 IO 操作处于等待状态的任务，保证了任务的有效执行。
25

本公开的实施例提供的运行时虚拟机系统，包括：用户态应用程序；异步 IO 模型，用于使用户态应用程序与内核态之间共享 IO 请求内存队列，以实现对物理设备的异步驱动；可在用户态应用程序启动时加载的动态运行库，动态运行库包括 IO 操作库函数钩子和监控线程函数；IO 操作库函数钩子基于用户态应用程序的原生库函数构建得
30

到，IO 操作库函数用于在运行时虚拟环境下，适配原生库函数的应用程序接口和异步 IO 模型，将用户态应用程序所在操作系统的 IO 操作转化为由异步 IO 模型处理；监控线程函数用于在运行时虚拟环境下，监控异步 IO 模型的 IO 操作是否完成，并在该 IO 操作完成时，唤醒
5 因该 IO 操作处于等待状态的任务。由此，本公开的运行动态库在用户态应用程序启动后，加载到用户态应用程序的内存，可以直接替换用户态应用程序的原生运行库，接管用户态应用程序所在操作系统的 IO 操作，并将 IO 操作转化为由异步 IO 模型处理，通过异步 IO 模型处理 IO 操作，减少了系统调用的次数，提升了 IO 性能；通过动态注入
10 技术，在不对应用程序进行任何修改的前提下，使用户态应用程序具有了纯用户态的 IO 能力，节省了人力成本，提高了用户态应用程序操作的便利性。

在本公开的一些可选实现方式中，上述 IO 操作库函数钩子包括：
15 文件描述符管理钩子函数、文件描述符读写钩子函数以及投票(POLL)状态钩子函数。

其中，上述文件描述符管理钩子函数用于在运行时虚拟环境下，创建文件描述符，将文件描述符放入文件描述符表中，并将文件描述符注册至异步 IO 模型中；文件描述符管理钩子函数还用于在运行时
20 虚拟环境下，关闭文件描述符时，将文件描述符从异步 IO 模型中进行反注册，并释放文件描述符表中与该文件描述符相应的内容。

上述文件描述符读写钩子函数用于在运行时虚拟环境下，接收对文件描述符的读请求或写请求，并将读请求或写请求放入异步 IO 模型的提交队列。

25 上述 POLL 状态钩子函数用于在运行时虚拟环境下，接收 POLL 等待请求或 POLL 唤醒请求，并将 POLL 等待请求或 POLL 唤醒请求放入异步 IO 模型的提交队列。

本实施例提供的 IO 操作库函数钩子，包括：文件描述符管理钩子函数、文件描述符读写钩子函数以及 POLL 状态钩子函数，文件描述
30 符管理钩子函数相对于原生库函数中文件描述符管理函数，具有与异

步 IO 模型进行数据通信的能力, 可以将文件描述符注册或者反注册至异步 IO 模型; 文件描述符读写钩子函数相对于原生库函数中文件描述符读写函数, 可以将文件描述符的读请求和写请求放入异步 IO 模型的提交队列, 使文件描述符的读写均处于用户态, 进一步减少了系统调用, 提高了 IO 性能; POLL 状态钩子函数相对于原生库函数中 POLL 状态函数, 可以将 POLL 等待请求或 POLL 唤醒请求放入异步 IO 模型的提交队列, 进一步减少了系统调用, 提高了 IO 性能。

在本公开的一些可选实现方式中, 动态运行库采用协程模拟原生库函数中实现的线程, IO 操作库函数钩子包括: 文件 IO 钩子函数以及文件 IO 回调函数, 网络 IO 钩子函数以及网络 IO 回调函数。

本实施例中, 文件 IO 钩子函数与网络 IO 钩子函数用于在运行时虚拟环境下的挂起或等待 IO 操作完成时判断操作主体是否为协程, 若为协程, 采用协程切换或挂起机制挂起协程。

其中, 上述文件 IO 钩子函数是采用 HOOK 机制构建的、与原生库函数中文件 IO 库函数相关的函数; 在原生库函数是 C 运行库的函数时, 与原生库函数中文件 IO 库函数相关的函数包括: lseek、read/write、readv/writev 等。文件 IO 钩子函数会判断当前上下文是否处于线程状态, 如果是线程, 则调用原生系统调用或库函数; 如果是协程, 才需要使用文件 IO 钩子函数自身的逻辑。

网络 IO 钩子函数是采用 HOOK 机制构建的、与原生库函数中网络 IO 库函数相关的函数; 在原生库函数是 C 运行库的函数时, 与原生库函数中网络 IO 库函数相关的函数包括: socket/close、listen/accept、send/recv、sendto/recvfrom、sendmsg/recvmsg、read/write、readv/writev、poll/select/epoll_wait 等。网络 IO 钩子函数在一个新的 socket 文件被创建 (调用 socket 或者 accept) 时, 将新创建的 socket 文件描述符加入到结构管理文件(UrFd)表中, 并通过异步 IO 模型处理与该 socket 文件描述符相关的所有网络 IO 操作 (如通过异步 IO 模型处理 poll_wait 操作)。

对系统原生线程 (包括主线程), 阻塞式等待并挂起当前线程是没

有问题的，因而在网络 IO 钩子函数中都会在挂起或唤醒操作时判断当前上下文是否处于线程状态，如果是线程，则采用线程的挂起或唤醒机制，即线程挂起或唤醒；如果是协程，则采用协程的挂起或唤醒机制，即协程的 yield/resume。

- 5 网络 IO 钩子函数用于在协程调用 send/recv 之类的库函数，试图进入阻塞式等待时，通过 UrFd 结构管理文件描述符，以及文件描述符对应的 POLL 状态，如果对应接收或发送缓存没有就绪，则标记当前协程上下文在这个 socket 上进入 IO 等待，并以 yield 方式让出。

- 类似的，当协程调用 poll 之类的库函数，试图等待某些 socket 状态就绪时，网络 IO 钩子函数会从 UrFd 表中判断这些 socket 中是否有已经处于就绪状态的。如有，则立即返回；如没有，则在这次 poll 操作监控的所有 socket 上标记当前协程上下文的 IO 等待状态，并以 yield 让出。
- 10

- 后台 epoll 线程收到一批 socket 就绪信号时，会查看是否有协程上下文在这些 socket 上标记 IO 等待状态，如有，则批量唤醒这些协程。
- 15

- 在动态运行库中具有生成协程调度线程的钩子函数，而在协程调度线程生成之后，每个协程调度线程都维护一个无锁的唤醒队列（wakeup queue）；上述唤醒某个协程是指：将此协程的协程上下文放入其所在调度线程的唤醒队列中。
- 20

- 本实施例中，本实施例中，执行主体使用自己的 UrFd 结构和 UrFd 表（也是一个无锁哈希表）管理已打开的文件描述符。UrFd 结构至少包含以下 3 个字段：第一个是文件描述符（也就是 UrFd 表的主键）；第二个是该文件的 poll 状态；第三个是无锁等待队列。与文件 IO 钩子函数、网络 IO 钩子函数生成的虚拟机系统不同之处在于，该虚拟机系统的 IO 钩子函数中，只有在协程上下文状态下才会接管库函数；在线程上下文状态下，仍然通过执行系统调用实现。但本实施例的核心目标在于构建“纯用户态”运行时环境，因而即使在线程上下文状态下，也需要使用纯用户态方式实现，即通过将 IO 操作转为由异步 IO 模型实现，尽可能使用纯用户态方案提升整体性能。本公开的 UrFd 结
- 25
- 30

构中，无锁等待队列用于维护多个协程（或线程）在同一个文件描述符上陷入 poll 等待或 IO 等待的情形。

本实施例中，文件 IO 回调函数用于在运行时虚拟环境下，将文件 IO 钩子函数中与 IO 相关操作改为与异步 IO 模型同步；通过文件 IO 回调函数可以接管文件 IO 库函数，并且对接异步 IO 模型，达到与异步 IO 模型同步的目的。具体地，文件 IO 回调函数包括：文件描述符读相关的第一回调函数，第一回调函数用于接收到文件描述符相关的读请求之后，将该读请求提交至异步 IO 模型的提交队列，通过将当前协程上下文放入对应的文件描述符的等待队列，并使用 yield（或线程挂起）方式让出。在得到异步 IO 模型就绪的消息时，设置文件描述符的状态，并唤醒文件描述符的等待队列中的协程（或线程），其中，文件描述符的等待队列是一种无锁的队列结构，支持带超时/无超时等待，并同时支持协程换出和线程挂起。

文件 IO 回调函数还包括：与文件描述符写相关的第二回调函数，第二回调函数用于接收到文件描述符相关的写请求之后，将该写请求提交至异步 IO 模型的提交队列，通过将当前协程上下文放入对应的文件描述符的等待队列，并使用 yield（或线程挂起）方式让出。在得到异步 IO 模型就绪的消息时，设置文件描述符的状态，并唤醒文件描述符的等待队列中的协程（或线程）。

网络 IO 回调函数用于在运行时虚拟环境下，将网络 IO 钩子函数中与 IO 相关操作改为与异步 IO 模型同步；通过网络 IO 回调函数可以接管网络 IO 库函数，并且对接异步 IO 模型，达到与异步 IO 模型同步的目的。

本实施例中，网络 IO 回调函数包括：与文件描述符管理（创建或关闭）相关的第二回调函数、与 POLL 等待相关的第三回调函数。其中，第二回调函数在创建或关闭文件描述符时启动，为新创建的文件描述符创建 UrFd 结构，并放入 UrFd 表，并将文件描述符注册到异步 IO 模型中；在关闭文件描述符时，将该文件描述符从异步 IO 模型中反注册，并释放 UrFd 表对应中的 UrFd 结构。

第三回调函数在 POLL 状态未就绪时，将 POLL 状态请求提交至

异步 IO 模型的提交队列,将当前协程上下文放入 UrFd 的等待队列中,并使用 yield (或线程挂起) 方式让出。带超时的等待操作通过无锁超时队列实现。

本实施例提供的动态运行库,通过文件 IO 钩子函数以及网络 IO 钩子函数实现文件 IO 和网络 IO 中协程模拟原生库函数中实现的线程,通过文件 IO 回调函数接管文件 IO 钩子函数的功能并实现与异步 IO 模型的对接,通过网络 IO 回调函数接管网络 IO 钩子函数并实现与异步 IO 模型的对接,加大了运行时虚拟机系统的运行时环境的用户态效果,减少了系统调用,提升了 IO 性能。

10

在本实施例的一些可选实现方式中,如图 2 所示,动态运行库还包括:线程调度库函数钩子,线程调度库函数钩子是采用 HOOK 机制构建的与原生库函数中线程创建、回收以及挂起库函数相关的函数,线程创建、回收以及挂起库函数为应用程序的原生运行库中能够创建、回收以及挂起线程的函数,线程调度库函数钩子用于在运行时虚拟环境下,将线程的创建、回收替换为协程的创建、回收,并且在应用程序运行后,生成协程调度线程,以通过协程调度线程调度至少一个协程。

本实施例中,在启动用户态应用程序时加载动态运行库,可以生成协程调度线程。

其中,协程调度线程包括:调度器以及用于放置至少一个协程的无锁队列,协程调度线程调度至少一个协程的具体步骤包括:调度器从无锁队列中获取至少一个协程中处于就绪状态的协程;同步该协程的协程上下文与线程控制块中的线程上下文;在该协程主动让出之后,调度器从无锁队列中获取下一个处于就绪状态的协程,并同步下一个协程的协程上下文与线程控制块中的线程上下文;在下一个协程主动让出之后,调度器继续从无锁队列中获取协程并同步线程控制块,直至无锁队列中无任何协程为止。

本实施例中,线程控制块是原生运行库中线程的数据结构,叫做“线程控制块”(Thread Control Block,简称 TCB),也就是线程的上

30

下文结构。dtv 和 specific array，就是 TCB 里面存储 TLS 数据的数据结构。dtv：用于存储编码阶段就定义好的 TLS 变量。这个数据会随着应用程序加载新的动态库而产生增量，因为动态库中也有可能存在 TLS 变量。

5 specific：动态创建的 TLS（其中，创建和删除 TLS key 的库函数分别为：pthread_key_create 和 pthread_key_delete；而读取和修改 TLS 值的库函数分别为：pthread_getspecific 和 pthread_setspecific）。这类 TLS 用一组指针数组维护（specific arrays），因而个数存在上限（Linux 最多支持创建 1024 个动态 TLS）。

10 本实施例中，调度器会在 TCB 和协程上下文之间做 TLS 数据的增量同步。

在线程调度库函数钩子中，采用协程上下文结构替换线程描述符，并通过带有线程本地约束的全局变量标记当前运行线程或协程，通过同步管理模块管理协程上下文结构中的线程本地存储信息，同步管理
15 模块用于在协程调度线程调度协程时，同步该协程的线程本地存储。

协程上下文结构（以下称为 COCTX 结构）是放置协程上下文（以下称为 COCTX）的结构，用于维护协程的上下，其类似于线程描述符（或线程句柄）的作用。为了提供与线程行为的兼容性，需要 hook 原生运行库中创建线程、以及获取线程描述符的库函数，用 COCTX 替代线程描述符。以 Linux 为例，其线程描述符被称为“TCB”，其实现
20 是一个名为“pthread”的结构体。Linux 中创建线程和获取线程描述符的调用分别是：

pthread_create：创建线程并返回线程描述符，这个线程描述符的值，就是 TCB 结构的地址。本实施例中，创建线程函数的钩子就以返回 COCTX 的指针取而代之。
25

pthread_self：获取当前线程的描述符，同样以 COCTX 的指针取代。

本实施例中，可以采用一个全局 TLS 变量（带有线程本地约束的全局变量）获取当前协程的上下文，其声明如下：

30 extern __thread COCTX* _current_coctx;

本实施例中，COCTX 有以下 4 种类型：

协程 COCTX：对应一个实际的协程；

Global COCTX：主线程的 `_current_coctx` 会被置为 global COCTX，这个 COCTX 在运行动态库刚刚被加载时就会被创建；

5 Scheduler COCTX：每个协程调度线程（就是运行协程的线程）都会有一个默认的 COCTX，没有协程处于就绪状态，或者两个协程切换的中间状态，当前线程的 `_current_coctx` 就会被置为 scheduler COCTX；

Single COCTX：创建系统原生线程时，会为其创建一个虚拟的 COCTX。

10 如上，`pthread_self` 的 hook 函数，实际上就是返回 `_current_coctx` 的值。程序中，通过判断当前 COCTX 的类型，就可以得知当前处于哪种线程、哪种运行状态。

本实施例中，COCTX 结构中存储的内容包括：协程栈以及 TLS。运行动态库为每个协程分配一个独立的栈，因而可以完整的模拟一个
15 线程的行为（包括在栈中分配的内存大小和调用栈深度）。

如果协程不能完备的模拟线程的 TLS 机制，将会导致协程不能安全的访问带有 TLS 约束的变量。本实施例，通过同步管理模块（以下称为 `co_tls`）实现了对 TLS 管理的封装，`co_tls` 提供的能力对于“透明”模拟线程至关重要。以 Linux 为例，其 TCB 中存储以下两类 TLS：
20 `dtv` 和 `specific`。

本公开的 `co_tls` 对 `specific` 的支持：`co_tls` 中需要维护一个与 TCB 中 `specific arrays` 类似的数据结构，当一个新创建的协程第一次被换入执行时，会复制其所属调度线程 TCB 中的 `specific`。此外，还需要 hook 管理 `specific` 的库函数，前文列举了 Linux 的库函数，Windows 中也有
25 类似函数，分别是：`TlsAlloc`、`TlsFree`、`TlsGetValue`、和 `TlsSetValue`。

本公开的 `co_tls` 对 `dtv` 的支持：同样，新创建的协程第一次被调度执行时，也需要复制 TCB 中的 `dtv`。然而与 `specific` 不同的是，`co_tls` 中 `specific` 的内容无需再与 TCB 同步；但由于运行时可能动态加载其他动态库，因而 `co_tls` 的 `dtv` 需要与 TCB 中的 `dtv` 进行增量同步。我们
30 在 scheduler COCTX 中存储一个对 TCB `dtv` 的增量备份，用以描述

每次加载动态库时，dtv 产生的增量部分。同时，我们需要对加载动态库的库函数调用进行 hook(Linux 中该库函数叫做 dlopen, 而 Windows 中则对应于 LoadLibrary)。以 Linux 为例，一个叫做 “link_map” 的数据结构描述了所有动态库信息，结合 link_map 和 dtv 增量备份，就可以解析出当前协程与 dtv 增量之间的差异。对于 dtv 增量的合并，采取“懒”同步方案，即在协程即将换入时再判断是否需要同步 dtv。如果 dlopen 是在某个协程中调用的，则需要在 dlopen 的 hook 中识别出 _current_coctx 为协程类型，并立即为当前协程同步 dtv。

本实施例中，通过线程调度库函数钩子，即可模拟出与线程能力完全一致的协程。协程的切换过程如下：首先，协程调度线程在初始状态下，_current_coctx 为 scheduler COCTX; 而后，当需要换入(resume)一个协程时，检查该协程与 dtv 增量备份之间是否存在差异(若存在则需同步)，完成差异同步后将 _current_coctx 置为当前协程的上下文；最后，当该协程要求换出时，再把 _current_coctx 还原成 scheduler COCTX。整个协程切换过程完全在用户态执行，无需与操作系统发生任何形式的交互，从而保持了协程的轻量化特征。

可选地，线程调度库函数钩子还包括：锁库函数钩子，在锁库函数钩子中，通过带有线程本地约束的全局变量，判断当前运行的主体，基于当前运行主体，给出当前运行主体相应的锁子机制。

本实施例提供的动态运行库，可以将线程创建、回收以协程创建、回收来模拟；同时，提供了对协程换入换出(即，对线程挂起和唤醒的模拟) 支持和对 TLS (线程本地存储) 的妥善处理(即，协程间的 TLS 并发安全访问)；并且，基于上述能力提供了对协程化锁和信号量(mutex、rwlock、条件变量、semaphore 等)，以及超时唤醒(定时器) 机制的支持。所以，无论是线程管理，还是线程挂起/唤醒(包括：锁挂起、定时器挂起等) 操作，都无需陷入任何系统调用，完全在用户态实现，在将用户态应用程序的运行环境“透明”替换为纯用户态环境过程，采用本实施例的可选实现方式提供的方案，保证了线程调度整体“透明”替换为用户态。

在本实施例的一些可选实现方式中，如图 2 所示，动态运行库还包括：内存管理库函数钩子，内存管理库函数钩子是采用 HOOK 机制构建的、与原生库函数中内存管理相关的函数，内存管理库函数钩子用于在运行时虚拟环境下，使一个线程能够启动一个或多个内存分配区，归属于当前协程调度线程的协程按内存分配区分组，并且归属于同一个内存分配区的协程共享同一个协程调度线程缓存。

通过使用 mmap（memory map，内存映射）预分配大块内存的方式，可实现纯用户态内存管理。

具体地，mmap 可以映射一段虚拟地址连续的内存地址空间；并且，只要起始地址对齐，就可以实现无论在用户态还是内核态，都能够通过内存映射的起始地址直接访问各自地址空间中的内存页面（即，相同页面在用户态和内核态，相对于起始地址的偏移都是一样的）。因而，用户态程序访问 mmap 映射的内存时，与一般内存访问没有区别，不用显性的使用系统调用分配物理内存。当应用程序访问一段尚未分配物理内存页面的内存地址时，会隐式触发一个“缺页中断”（page fault），由中断处理程序分配物理内存，并建立映射关系，应用程序本身则不会陷入内核态。但是，缺页中断机制分配的物理内存，有可能因可用物理内存不足，而被内核释放或换出到交换分区，再次回到内存页“缺位”状态；在下次访问此内存页时，再通过缺页中断重新为其分配物理内存，或者将交换分区上的页面重新“换入”物理内存。为了保证内存页始终“在位”，同时减少缺页中断次数（页面越大页数越少，Linux 一般使用 2MB 巨页），纯用户态编程框架中往往采取“巨页”内存映射的方式管理用户态和内核态共享的内存映射。巨页内存不会被内核换出，因而保证了其内存页被“锚定”在对应的物理内存页面上。

针对传统的虚拟机中默认的内存分配器，在分配小尺寸内存时使用系统调用实现，只有在分配大尺寸内存时才使用 mmap，因此还是会在分配内存时经常性陷入系统调用的情况。本系统中的内存管理系统，使其内存的实现完全基于 mmap 分配大块内存，实现了纯用户态内存管理。在将用户态应用程序的运行时环境“透明”替换为纯用

户态环境过程，采用本实施例的可选实现方式提供的方案，保证了内存管理整体“透明”替换为用户态。

在本实施例的一些可选实现方式中，监控线程函数在用户态应用程序启动之后且在运行时虚拟环境下，为异步 IO 模型中每个上下文创建监控线程，监控线程用于轮询异步 IO 模型的完成队列，当收到完成队列中新的就绪任务时，唤醒 IO 操作库函数钩子对应的等待队列中的协程。

本实施例提供的动态运行库，可以为异步 IO 模型的上下文创建一个后台线程—监控线程，其中，监控线程可以是一个，也可以是多个通过创建的多个监控线程，可以实现 IO 请求负载均衡。后台的监控线程周而复始的对异步 IO 模型的完成队列进行轮询，当收到新的就绪任务时，就将无锁等待队列中的协程上下文唤醒，以达到唤醒执行相应任务的协程的目的。

本实施例提供的动态运行库，采用监控线程函数生成监控线程，通过监控线程对异步 IO 模型的完成队列进行轮询，提高信息得到的全面性，保证了用户态应用程序的纯用户态效果。

通过图 2 所示的运行时虚拟机系统，本发明提出一种“透明”的纯用户态运行时虚拟机系统，应用程序只要通过在运行时预先加载动态库的方式，即可在不对应用程序进行任何修改的前提下，运行于纯用户态的虚拟化环境，从而提升应用性能；即本公开将协程化运行时虚拟机系统与纯用户态编程技术完美融合，从而提供一种对应用程序完全“透明”的纯用户态运行时环境；

本发明方案将协程（即，纯用户态虚拟线程调度）、基于 mmap 的纯用户态内存管理器，以及基于 io_uring 的纯用户态 IO 子系统相互结合，构建出一套在绝大多数情况下可以避开（bypass）一切系统调用的纯用户态运行环境；

本发明方案可以提供对 POSIX 协议接口的完美兼容，开发者可以使用基于本发明实现的开发库，“透明”实现纯用户态应用程序的开发。即本公开了提供了融合原生网络协议栈和文件系统生态，并完全兼容

POSIX 协议的开发接口。

进一步参考图 3，作为对上述各图所示运行时虚拟机系统在运行时的实现，本公开提供了系统运行方法的一个实施例。如图 1，示出了
5 根据本公开的系统运行方法的一个实施例的流程 300，该系统运行方法包括以下步骤：

步骤 301，加载异步 IO 模型的库文件。

本实施例中，异步 IO 模型用于使用户态应用程序与内核态之间共享 IO 请求内存队列，以实现对物理设备的异步驱动。

10 本实施例中，异步 IO 模型的库文件实现异步 IO 模型的文件，当在用户态应用程序的内存加载了该库文件之后，运行用户态应用程序时可以实现对异步模型的调用。

本实施例中，通过异步 IO 模型可以实现三种库函数调用，设置异步 IO 模型的上下文；提交和获取完成任务；通过内存映射技术使注册
15 内核用户共享缓冲区。其中，提交和获取任务完成状态的调用，都是在用户态直接和内存映射队列交互的，不需要系统调用。只有异步 IO 上下文设置和文件描述符的注册/反注册，才需要实现为系统调用。

步骤 302，在启动用户态应用程序运行时，加载动态运行库，生成运行时虚拟化环境。

20 本实施例中，动态运行库是一种动态库，启动用户态应用程序时加载动态运行库，直接生成与运动动态库对应的运行时虚拟化环境，并且运行时虚拟化环境为替换用户态应用程序的运行时的环境的环境。

本实施例中，编译好的函数存储于动态运行库中，动态运行库基于库函数向应用程序提供 API（Application Program Interface，应用程序接口）接口，用户态应用程序通过调用 API 向动态运行库发出任务
25 请求。可选地，在原生运行库的基础上，通过对原生运行库中与线程调度相关的库函数进行 hook，可以得到动态运行库，并且得到动态运行库在加载之后，可以具有与线程等价的调度能力。

本实施例中，可以全面对 C 运行库进行 hook，得到动态运行库，
30 通过动态运行库提供运行时虚拟化环境，应用程序在该运行时虚拟化

环境下运行，实现以协程方式模拟原生运行库中实现的线程，达到协程模拟线程以及通过协程调度线程调度协程的目的。

步骤 303，在运行时虚拟环境下，响应于检测到用户态应用程序对用户态应用程序的原生库函数的调用信息，通过 IO 操作库函数将用户态应用程序所在操作系统的 IO 操作转化为由异步 IO 模型处理。

步骤 304，在运行时虚拟环境下，通过监控线程函数监控异步 IO 模型的 IO 操作是否完成，并在该 IO 操作完成时，通过监控线程函数唤醒因该 IO 操作处于等待状态的任务。

本公开的实施例提供的系统运行方法，首先，加载异步 IO 模型的库文件；其次，在启动用户态应用程序时加载动态运行库，生成运行时虚拟化环境；再次，在运行时虚拟环境下，响应于检测到用户态应用程序对用户态应用程序的原生库函数的调用信息，通过 IO 操作库函数将用户态应用程序所在操作系统的 IO 操作转化为由异步 IO 模型处理；最后，在运行时虚拟环境下，通过监控线程函数监控异步 IO 模型的 IO 操作是否完成，并在该 IO 操作完成时，通过监控线程函数唤醒因该 IO 操作处于等待状态的任务。由此，通过动态注入技术与异步 IO 模型相结合，将运行库中与 IO 操作相关的系统调用的封装都改用纯用户态技术实现，并且宿主机上原有的用户态应用程序完全“透明”的运行于纯用户态的运行环境，从而实现了在不对应用程序做任何修改的前提下获得更高的性能，可以有效控制系统中的线程资源占用率，降低系统资源消耗和资源调度压力，从而提升系统整体的吞吐量和可用性。

进一步参考图 4，作为对上述各图所示系统运行方法的实现，本公开提供了系统运行装置的一个实施例，该装置实施例与图 3 所示的方法实施例相对应，该装置具体可以应用于各种电子设备中。

如图 4 所示，本公开的实施例提供了一种系统运行装置 400，该装置 400 包括：文件加载单元 401、库加载单元 402、转化单元 403、唤醒单元 404。其中，上述文件加载单元 401，可以被配置成加载异步 IO 模型的库文件，异步 IO 模型用于使用户态应用程序与内核态之间

共享 IO 请求内存队列，以实现对物理设备的异步驱动。上述库加载单元 402，可以被配置成在启动用户态应用程序运行时，加载动态运行库，生成运行时虚拟化环境，运行时虚拟化环境为替换用户态应用程序的运行时的环境的环境。上述转化单元 403，可以被配置成在运行时虚拟环境下且检测到用户态应用程序对用户态应用程序的原生库函数的调用信息时，通过 IO 操作库函数将用户态应用程序所在操作系统的 IO 操作转化为由异步 IO 模型处理。上述唤醒单元 404，可以被配置成在运行时虚拟环境下，通过监控线程函数监控异步 IO 模型的 IO 操作是否完成，并在该 IO 操作完成时，通过监控线程函数唤醒因该 IO 操作处于等待状态的任务。

在本实施例中，装置 400 中，文件加载单元 401、库加载单元 402、转化单元 403 的具体处理及其所带来的技术效果可分别参考图 1 对应实施例中的步骤 301、步骤 302、步骤 303。

本公开的实施例提供的系统运行装置，首先，文件加载单元 401 加载异步 IO 模型的库文件；其次，库加载单元 402 在启动用户态应用程序时加载动态运行库，生成运行时虚拟化环境，运行时虚拟化环境为替换用户态应用程序的运行时的环境的环境；从次，转化单元 403 在运行时虚拟环境下且检测到用户态应用程序对用户态应用程序的原生库函数的调用信息，通过 IO 操作库函数将用户态应用程序所在操作系统的 IO 操作转化为由异步 IO 模型处理。最后，唤醒单元在运行时虚拟环境下，通过监控线程函数监控异步 IO 模型的 IO 操作是否完成，并在该 IO 操作完成时，通过监控线程函数唤醒因该 IO 操作处于等待状态的任务。由此，通过动态注入技术与异步 IO 模型相结合，将运行库中与 IO 操作相关的系统调用的封装都改用纯用户态技术实现，并且宿主机上原有的用户态应用程序完全“透明”的运行于纯用户态的运行时的环境，从而实现了在不对应用程序做任何修改的前提下获得更高的性能，可以有效控制系统中的线程资源占用率，降低系统资源消耗和资源调度压力，从而提升系统整体的吞吐量和可用性。

下面参考图 5，其示出了适于用来实现本公开的实施例的电子设

备 500 的结构示意图。

如图 5 所示,电子设备 500 可以包括处理装置(例如中央处理器、图形处理器等) 501,其可以根据存储在只读存储器 (ROM) 502 中的程序或者从存储装置 508 加载到随机访问存储器 (RAM) 503 中的程序而执行各种适当的动作和处理。在 RAM503 中,还存储有电子设备 500 操作所需的各种程序和数据。处理装置 501、ROM 502 以及 RAM 503 通过总线 504 彼此相连。输入/输出 (I/O) 接口 505 也连接至总线 504。

通常,以下装置可以连接至 I/O 接口 505: 包括例如触摸屏、触摸板、键盘、鼠标、等的输入装置 506; 包括例如液晶显示器 (LCD, Liquid Crystal Display)、扬声器、振动器等的输出装置 507; 包括例如磁带、硬盘等的存储装置 508; 以及通信装置 509。通信装置 509 可以允许电子设备 500 与其他设备进行无线或有线通信以交换数据。虽然图 5 示出了具有各种装置的电子设备 500,但是应理解的是,并不要求实施或具备所有示出的装置。可以替代地实施或具备更多或更少的装置。图 5 中示出的每个方框可以代表一个装置,也可以根据需要代表多个装置。

特别地,根据本公开的实施例,上文参考流程图描述的过程可以被实现为计算机软件程序。例如,本公开的实施例包括一种计算机程序产品,其包括承载在计算机可读介质上的计算机程序,该计算机程序包含用于执行流程图所示的方法的程序代码。在这样的实施例中,该计算机程序可以通过通信装置 509 从网络上被下载和安装,或者从存储装置 508 被安装,或者从 ROM 502 被安装。在该计算机程序被处理装置 501 执行时,执行本公开的实施例的方法中限定的上述功能。

需要说明的是,本公开的实施例的计算机可读介质可以是计算机可读信号介质或者计算机可读存储介质或者是上述两者的任意组合。计算机可读存储介质例如可以是——但不限于——电、磁、光、电磁、红外线、或半导体的系统、装置或器件,或者任意以上的组合。计算机可读存储介质的更具体的例子可以包括但不限于: 具有一个或多个导线的电连接、便携式计算机磁盘、硬盘、随机访问存储器 (RAM)、

只读存储器 (ROM)、可擦式可编程只读存储器 (EPROM 或闪存)、光纤、便携式紧凑磁盘只读存储器 (CD-ROM)、光存储器件、磁存储器件、或者上述的任意合适的组合。在本公开的实施例中, 计算机可读存储介质可以是任何包含或存储程序的有形介质, 该程序可以被指令执行系统、装置或者器件使用或者与其结合使用。而在本公开的实施例中, 计算机可读信号介质可以包括在基带中或者作为载波一部分传播的数据信号, 其中承载了计算机可读的程序代码。这种传播的数据信号可以采用多种形式, 包括但不限于电磁信号、光信号或上述的任意合适的组合。计算机可读信号介质还可以是计算机可读存储介质以外的任何计算机可读介质, 该计算机可读信号介质可以发送、传播或者传输用于由指令执行系统、装置或者器件使用或者与其结合使用的程序。计算机可读介质上包含的程序代码可以用任何适当的介质传输, 包括但不限于: 电线、光缆、RF (Radio Frequency, 射频) 等等, 或者上述的任意合适的组合。

上述计算机可读介质可以是服务器中所包含的; 也可以是单独存在, 而未装配入该服务器中。上述计算机可读介质承载有一个或者多个程序, 当上述一个或者多个程序被该服务器执行时, 使得该服务器: 加载异步 IO 模型的库文件, 异步 IO 模型用于使用户态应用程序与内核态之间共享 IO 请求内存队列, 以实现对物理设备的异步驱动; 在启动用户态应用程序运行时, 加载动态运行库, 生成运行时虚拟化环境, 运行时虚拟化环境为替换用户态应用程序的运行时的环境; 在运行时虚拟环境下, 响应于检测到动态运行库中与异步 IO 模型相适配的 IO 操作库函数钩子调用信息, 通过 IO 操作库函数将用户态应用程序所在操作系统的 IO 操作转化为由异步 IO 模型处理。

可以以一种或多种程序设计语言或其组合来编写用于执行本公开的实施例的操作的计算机程序代码, 程序设计语言包括面向对象的程序设计语言—诸如 Java、Smalltalk、C++, 还包括常规的过程式程序设计语言—诸如 “C” 语言或类似的程序设计语言。程序代码可以完全地在用户计算机上执行、部分地在用户计算机上执行、作为一个独立的软件包执行、部分在用户计算机上部分在远程计算机上执行、或者完

全在远程计算机或服务器上执行。在涉及远程计算机的情形中，远程计算机可以通过任意种类的网络——包括局域网(LAN)或广域网(WAN)——连接到用户计算机，或者，可以连接到外部计算机（例如利用因特网服务提供商来通过因特网连接）。

5 附图中的流程图和框图，图示了按照本公开的各种实施例的系统、方法和计算机程序产品的可能实现的体系架构、功能和操作。在这点上，流程图或框图中的每个方框可以代表一个模块、程序段、或代码的一部分，该模块、程序段、或代码的一部分包含一个或多个用于实现规定的逻辑功能的可执行指令。也应当注意，在有些作为替换的实现中，方框中所标注的功能也可以以不同于附图中所标注的顺序发生。
10 例如，两个接连地表示的方框实际上可以基本并行地执行，它们有时也可以按相反的顺序执行，这依所涉及的功能而定。也要注意的，框图和/或流程图中的每个方框、以及框图和/或流程图中的方框的组合，可以用执行规定的功能或操作的专用的基于硬件的系统来实现，或者
15 可以用专用硬件与计算机指令的组合来实现。

描述于本公开的实施例中所涉及到的单元可以通过软件的方式实现，也可以通过硬件的方式来实现。所描述的单元也可以设置在处理器中，例如，可以描述为：一种处理器，包括文件加载单元、库加载单元、转化单元。其中，这些单元的名称在某种情况下并不构成对该
20 单元本身的限定，例如，文件加载单元还可以被描述为“被配置成加载异步 IO 模型的库文件”的单元。

以上描述仅为本公开的较佳实施例以及对所运用技术原理的说明。本领域技术人员应当理解，本公开的实施例中所涉及的发明范围，并不限于上述技术特征的特定组合而成的技术方案，同时也应涵盖在不
25 脱离上述发明构思的情况下，由上述技术特征或其等同特征进行任意组合而形成的其它技术方案。例如上述特征与本公开的实施例中公开的(但不限于)具有类似功能的技术特征进行互相替换而形成的技术方案。

权 利 要 求

- 5 1、一种运行时虚拟机系统，所述系统包括：
 用户态应用程序；
 异步输入输出（IO）模型，用于使所述用户态应用程序与内核态
 之间共享 IO 请求内存队列，以实现对物理设备的异步驱动；以及
 可在所述用户态应用程序启动时加载的动态运行库，所述动态运
10 行库用于被加载后生成替换所述用户态应用程序的运行时环境的运行
 时虚拟环境；其中，所述动态运行库包括 IO 操作库函数钩子和监控线
 程函数；
 所述 IO 操作库函数钩子基于所述用户态应用程序的原生库函数
 构建得到，所述 IO 操作库函数用于在所述运行时虚拟环境下，适配所
15 述原生库函数的应用程序接口和所述异步 IO 模型，将所述用户态应
 用程序所在操作系统的 IO 操作转化为由所述异步 IO 模型处理；
 所述监控线程函数用于在所述运行时虚拟环境下，监控所述异步
 IO 模型的 IO 操作是否完成，并在该 IO 操作完成时，唤醒因该 IO 操
 作处于等待状态的任务。
- 20 2、根据权利要求 1 所述的系统，其中，所述 IO 操作库函数钩子
 包括：文件描述符管理钩子函数、文件描述符读写钩子函数以及投票
 （POLL）状态钩子函数；
 所述文件描述符管理钩子函数用于在所述运行时虚拟环境下，创
25 建文件描述符，将所述文件描述符放入文件描述符表中，并将所述文
 件描述符注册至所述异步 IO 模型中；
 所述文件描述符管理钩子函数还用于在所述运行时虚拟环境下且
 关闭所述文件描述符时，将所述文件描述符从所述异步 IO 模型中进
 行反注册；
30 所述文件描述符读写钩子函数用于在所述运行时虚拟环境下，接
 收对所述文件描述符的读请求或写请求，并将所述读请求或写请求放

入所述异步 IO 模型的提交队列；

所述 POLL 状态钩子函数用于在所述运行时虚拟环境下，接收 POLL 等待请求或 POLL 唤醒请求，并将所述 POLL 等待请求或所述 POLL 唤醒请求放入所述异步 IO 模型的提交队列。

5

3、根据权利要求 1 所述的系统，其中，所述动态运行库采用协程模拟原生库函数中实现的线程，所述 IO 操作库函数钩子包括：文件 IO 钩子函数以及文件 IO 回调函数，网络 IO 钩子函数以及网络 IO 回调函数；

10 所述文件 IO 钩子函数是采用挂钩（HOOK）机制构建的、与所述原生库函数中文件 IO 库函数相关的函数；

所述网络 IO 钩子函数是采用 HOOK 机制构建的、与所述原生库函数中网络 IO 库函数相关的函数；所述文件 IO 钩子函数与所述网络 IO 钩子函数用于在所述运行时虚拟环境下的挂起或等待 IO 操作完成
15 时判断操作主体是否为协程，若为协程，采用协程切换或挂起机制挂起所述协程；

所述文件 IO 回调函数用于在所述运行时虚拟环境下，将所述文件 IO 钩子函数中与 IO 相关操作改为与所述异步 IO 模型同步；

所述网络 IO 回调函数用于在所述运行时虚拟环境下，将所述网络
20 IO 钩子函数中与 IO 相关操作改为与所述异步 IO 模型同步。

4、根据权利要求 1 所述的系统，所述动态运行库还包括：线程调度库函数钩子，所述线程调度库函数钩子是采用 HOOK 机制构建的与
所述原生库函数中线程创建、回收以及挂起库函数相关的函数，所述
25 线程创建、回收以及挂起库函数为所述应用程序的原生运行库中能够创建、回收以及挂起线程的函数，所述线程调度库函数钩子用于在所述运行时虚拟环境下，将线程的创建、回收替换为协程的创建、回收，并且在所述应用程序运行后，生成协程调度线程，以通过协程调度线程调度至少一个协程。

30

5、根据权利要求3所述的系统，所述动态运行库还包括：内存管理库函数钩子，所述内存管理库函数钩子是采用HOOK机制构建的、与所述原生库函数中内存管理相关的函数，所述内存管理库函数钩子用于在所述运行时虚拟环境下，使一个线程能够启动一个或多个内存分配区，归属于当前协程调度线程的协程按内存分配区分组，并且归属于同一个内存分配区的协程共享同一个协程调度线程缓存。

6、根据权利要求3-5之一所述的系统，所述监控线程函数在所述用户态应用程序启动之后且在所述运行时虚拟环境下，为所述异步IO模型中每个上下文创建监控线程，所述监控线程用于轮询所述异步IO模型的完成队列，当收到所述完成队列中新的就绪任务时，唤醒IO操作库函数钩子对应的等待队列中的协程。

7、一种系统运行方法，所述方法包括：

15 加载异步IO模型的库文件，所述异步IO模型用于使用户态应用程序与内核态之间共享IO请求内存队列，以实现物理设备的异步驱动；

在启动所述用户态应用程序运行时，加载动态运行库，生成运行时虚拟化环境，所述运行时虚拟化环境为替换所述用户态应用程序的运行时环境的环境；

在所述运行时虚拟环境下，响应于检测到所述用户态应用程序对所述用户态应用程序的原生库函数的调用信息，通过IO操作库函数将所述用户态应用程序所在操作系统的IO操作转化为由所述异步IO模型处理；

25 在所述运行时虚拟环境下，通过监控线程函数监控所述异步IO模型的IO操作是否完成，并在该IO操作完成时，通过所述监控线程函数唤醒因该IO操作处于等待状态的任务。

8、一种系统运行装置，所述装置包括：

30 文件加载单元，被配置成加载异步IO模型的库文件，所述异步IO

模型用于使用户态应用程序与内核态之间共享 IO 请求内存队列，以实现对物理设备的异步驱动；

库加载单元，被配置成在启动所述用户态应用程序运行时，加载动态运行库，生成运行时虚拟化环境，所述运行时虚拟化环境为替换
5 所述用户态应用程序的运行时环境的环境；

转化单元，被配置成在所述运行时虚拟环境下且检测到所述用户态应用程序对所述用户态应用程序的原生库函数的调用信息时，通过 IO 操作库函数将所述用户态应用程序所在操作系统的 IO 操作转化为由所述异步 IO 模型处理；

10 唤醒单元，被配置成在所述运行时虚拟环境下，通过监控线程函数监控所述异步 IO 模型的 IO 操作是否完成，并在该 IO 操作完成时，通过所述监控线程函数唤醒因该 IO 操作处于等待状态的任务。

9、一种电子设备，包括：

15 一个或多个处理器；

存储装置，其上存储有一个或多个程序；

当所述一个或多个程序被所述一个或多个处理器执行，使得所述一个或多个处理器实现如权利要求 7 所述的方法。

20 10、一种计算机可读介质，其上存储有计算机程序，其中，该程序被处理器执行时实现如权利要求 7 所述的方法。