

运行时虚拟化方法和装置、运行时虚拟机系统

相关申请的交叉引用

本专利申请要求于 2023 年 7 月 28 日提交的、申请号为 202310942815.0、发明名称为“运行时虚拟化方法和装置、运行时虚拟机系统”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

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

背景技术

协程是一种分时复用宿主线程的运行时片的轻量并行子程序，由于协程复用了宿主线程的上下文，协程的换入换出过程只相当于一次函数调用，省去了额外的上下文切换代价。相对于普通线程，协程一次换入换出的代价一般低一个数量级左右。

发明内容

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

在一个或多个实施例中，本公开提供了一种运行时虚拟化方法，该方法包括：基于应用程序的原生运行库，构建动态运行库，动态运行库以协程方式模拟原生运行库中实现的线程；在启动应用程序时加载动态运行库，生成运行时虚拟化环境以及运行于运行时虚拟化环境中的协程调度线程，协程调度线程用于调度至少一个协程，运行时虚拟化环境为替换原生运行库所生成的运行时环境后的环境；在运行时虚拟化环境下，接收应用程序的任务请求；基于任务请求和动态运行库，采用协程调度线程调度协程，以使协程完成任务请求对应的任务。

在一些实施例中，上述基于任务请求和动态运行库，采用协程调度线程调度协程，以使协程完成任务请求对应的任务包括：基于任务请求和动态运行库，创建至少一个协程；将至少一个协程发送给协程调度线程，以采用协程调度线程调

度至少一个协程。

在一些实施例中，上述协程调度线程包括：调度器以及用于放置至少一个协程的无锁队列，采用协程调度线程调度至少一个协程包括：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。

在一些实施例中，上述方法还包括：接收应用程序的原生请求，基于原生请求和动态运行库中的原生运行库，创建至少一个线程，通过至少一个线程执行原生请求对应的任务。

在一些实施例中，上述基于应用程序的原生运行库，构建动态运行库包括：采用 hook 机制构建与第一库函数相关的线程库函数钩子，第一库函数为原生运行库中创建线程、获取线程描述符的库函数；在线程库函数钩子中，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。

在一些实施例中，上述基于应用程序的原生运行库，构建动态运行库还包括：采用 hook 机制构建与第二库函数相关的锁库函数钩子，第二库函数为原生运行库中分别与锁、条件变量、信号量相关的库函数；在锁库函数钩子中，通过全局变量，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。

在一些实施例中，上述基于应用程序的原生运行库，构建动态运行库还包括：采用 hook 机制构建与第三库函数相关的切换库函数钩子，第三库函数为原生运行库中能够导致线程切换或挂起操作的库函数；切换库函数钩子用于采用协程切换或挂起机制代替线程切换或挂起机制。

在一个或多个实施例中，本公开提供了一种运行时虚拟化装置，该装置包括：构建单元，被配置成基于应用程序的原生运行库，构建动态运行库，动态运行库以协程方式模拟原生运行库中实现的线程；生成单元，被配置成在启动应用程序时加载动态运行库，生成运行时虚拟化环境以及运行于运行时虚拟化环境中的协

程调度线程，协程调度线程用于调度至少一个协程，运行时虚拟化环境为替换原生运行库所生成的运行时环境后的环境；任务接收单元，被配置成在运行时虚拟化环境下，接收应用程序的任务请求；协程调度单元，被配置成基于任务请求和动态运行库，采用协程调度线程调度协程，以使协程完成任务请求对应的任务。

在一些实施例中，上述调度单元被配置成：基于任务请求和动态运行库，创建至少一个协程；将至少一个协程发送给协程调度线程，以采用协程调度线程调度至少一个协程。

在一些实施例中，上述协程调度线程包括：调度器以及用于放置至少一个协程的无锁队列，调度单元进一步被配置成：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。

在一些实施例中，上述装置还包括：原生接收单元，被配置成接收应用程序的原生请求；线程调度单元，被配置成基于原生请求和动态运行库，创建至少一个线程，通过至少一个线程执行原生请求对应的任务。

在一些实施例中，上述构建单元被配置成：采用 hook 机制构建与第一库函数相关的线程库函数钩子，第一库函数为原生运行库中创建线程、获取线程描述符的库函数；在线程库函数钩子中，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。

在一些实施例中，上述构建单元被配置成：采用 hook 机制构建与第二库函数相关的锁库函数钩子，第二库函数为原生运行库中与锁相关的库函数；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。

在一些实施例中，上述构建单元被配置成：采用 hook 机制构建与第三库函数相关的切换库函数钩子，第三库函数为原生运行库中能够导致线程切换或挂起操作的库函数；切换库函数钩子用于采用协程切换或挂起机制代替线程切换或挂

起机制。

在一个或多个实施例中，本公开提供了一种运行时虚拟机系统，系统包括：运行在运行时虚拟化环境下的协程调度线程、主线程、事件驱动（EPOLL）线程以及异步输入输出（AIO）和定时器线程，其中，运行时虚拟化环境通过加载动态运行库生成，且动态运行库基于应用程序的原生运行库构建得到，动态运行库以协程方式模拟原生运行库中实现的线程；协程调度线程用于调度至少一个协程，以使协程完成应用程序的任务请求；主线程在应用程序启动时创建；事件驱动线程用于维护文件句柄的状态，并在读或写缓存就绪、且存在阻塞在文件句柄上的读写任务时，唤醒相应的协程；异步输入输出和定时器线程，获取异步输入输出任务结果，唤醒处于 IO 等待的协程；在系统调用超时返回时，弹出超时耗尽的任务并唤醒；在系统调用异步 IO 任务结果时，唤醒处于 IO 等待的协程。

在一个或多个实施例中，本公开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计算机程序；当一个或多个计算机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上述方法实施例中任一实施例描述的方法。

在一个或多个实施例中，本公开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上述方法实施例中任一实施例描述的方法。

附图说明

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，本公开的其他特征、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化方法的一个实施例的流程图；

图 2 是根据本公开的运行时虚拟化装置的一个实施例的结构示意图；

图 3 是根据本公开的运行时虚拟机系统的一个结构示意图；

图 4 是根据本公开的运行时虚拟机系统的另一个结构示意图；

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。

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。

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。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。

目前已经公开的此类方案中，基于 Android bionic 的 libhybris 就是一个典型例子。Android 动态库都是基于 bionic 编译的，然而 GNU 版本的 Linux 发行版中并没有 bionic 的运行环境。为了在 GNU 发行版系统中运行 Android 动态库，就需要一个兼容 glibc 的运行环境虚拟机，也就是 libhybris 库基于 glibc 的系统加载 Android 动态库，并使用 libhybris 中的钩子函数覆盖 bionic 的库函数；而 libhybris 的钩子函数，则通过动态加载和调用 glibc 中对应的函数，模拟对应于 bionic 库函数中的类似实现。也就是说，libhybris 解决了两种 C 运行库的兼容性问题，实现了跨平台运行 Android 程序的运行时环境。

Libhybris 可以被看作是一个完整的运行时虚拟机，在 GNU 环境下实现了一整套 bionic 模拟运行环境。此外，很多协程库也实现了部分 C 运行库钩子，用以解决协程的锁等待以及 IO 等待的异步化问题，例如开源的 libco 库、libgo 库等等。

一个完备的协程库都会 hook 那些需要阻塞式等待的 IO 库函数（IO 很可能不会立刻完成，因而相应的库函数调用就需要等待 IO 操作结束才能返回），以及协程专用的锁。

为什么协程需要 hook IO 库函数呢？因为协程的运行依附于其宿主机线程，一旦这个协程试图调用一个可能产生阻塞式等待的库函数，那么在等待过程中，宿主线程就会被挂起，直到该函数返回。在这个挂起阶段，宿主线程无法执行任何其他代码。

协程是通过分时复用宿主线程的时间片进行调度的，其性能取决于宿主线程时间片的有效利用率。而一旦宿主线程被某个协程挂起，而在此期间还无法调用其他协程，那么这个挂起阶段的时间片就会被浪费掉，从而导致整体性能的下降。协程挂起宿主线程，不仅会影响性能，还会引发一种调度性“死锁”的问题。试

想一种情形，当所有宿主线程都陷入了长时间的挂起等待，而此时有一个新的协程希望被调度执行，那么这个系统就陷入了一种类似“死锁”的状态，新的协程只能等待其中一个宿主线程从挂起的库函数中返回。而一旦新协程存在某种“超时”机制，此时就可能出现新协程永远超时失败的极端情况。此外，如果协程使用“线程锁”，就会加重“调度性死锁”。假如，协程 1 在持有线程锁 A 的情况下被换出（yield），而此时被当前宿主线程换入（resume）的协程 2 也要对线程锁 A 加锁，那么协程 2 就会挂起宿主线程等待线程锁 A 的释放；但宿主线程由于被挂起，就永远没有机会再换入可以将线程锁 A 解锁的协程 1，从而导致彻底的死锁。

一个完备的协程库，需要把所有可能引发阻塞式等待的库函数（包括 IO 等待和锁等待）都用非阻塞式的异步调用进行模拟。也就是说，当协程可能陷入 IO 等待或锁等待时，应该利用换出（yield）机制返回宿主线程，使宿主线程可以继续调用其他的协程；后续，当之前因等待而换出的协程可以继续执行的时候（即，其所等待的资源就绪——如 IO 完成、解锁等），就可以在宿主线程后续的空闲时间片再次被调度（resume）。协程库中提供 C 运行库的 IO 库函数钩子，就是一种比较完美的解决方案。例如，使用协程库的应用程序调用 recv 函数接收网络包时，实际上就会调用被协程库 hook 的 recv 函数；如果此时网络包还没有到达，这个 hook 函数就会以 yield 机制换出；后续，当网络包到达时，此前换出的协程就会在某个合适的时机再次被 resume 换入，并从之前换出的位置继续执行，完成网络包接收并从 hook 函数中返回。这样，在开发应用程序时，就可以直接使用 C 运行库中的函数；直到运行时，再通过动态库注入的方式将原生库函数动态替换为协程版 hook 函数。这样，通过上述方案，就实现了对应用近乎“透明”的协程化。

一个应用程序中可能同时存在协程和线程，协程锁的等待以 yield 行为模拟，而线程锁等待则需要真的挂起线程，二者的行为模式往往不能兼容。因而，使用协程库的应用如果要引入锁机制，就需要谨慎的选择使用协程锁还是线程锁，也就不能做到彻底的对应用的“透明”。

进一步地，现有协程库，一般都没有妥善处理线程本地存储的跨携程安全问题。也就是说，被同一个宿主线程调度的协程，会共享宿主线程的线程本地存储，这就导致协程之间会彼此“污染”线程本地存储，无法做到对线程本地存储的跨协程安全访问，也就不具备与线程等价的能力。

综上，多数协程库并不是一种完整的运行时虚拟机，而是仅仅利用动态库注入方式 hook 部分 C 运行库的库函数，以解决并发性能问题和调度性死锁问题。也就是说，协程库一般就只是一种对线程的运行时虚拟化方案（用协程模拟线程），而不会对 C 运行库中的底层接口进行完整的模拟。

为了更好地理解传统技术，下面对一些名词进行介绍：

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。

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。

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（例如“JAVA 虚拟机”，其屏蔽了不同操作系统的 C 运行库之间的差异，提供系统无关的开发接口和运行环境）；或者提供针对目标操作系统的虚拟化运行环境（例如“容器”，可以在一个宿主系统中同时运行多个容器系统）。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（后者又被称为代码“钩子”——hook）。

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（也就是前文中所谓“同一位置”）继续执行。

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;
```

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（glibc 中称之为“dtv”），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。不同的操作系统中，都会提供类似的 API，以提供对此类 TLS 的支持。

本公开提供了一种运行时虚拟化方法，可以灵活的采用协程实现了对线程的模拟以及调度，提升了并发性能。如图 1，示出了根据本公开的运行时虚拟化方法的一个实施例的流程 100，该运行时虚拟化方法包括以下步骤：

步骤 101，基于应用程序的原生运行库，构建动态运行库。

本实施例中，原生运行库是支持应用程序运行的运行库，为应用程序在运行时与操作系统间的交互提供编程接口，原生运行库可以是基于 C 语言构建得到。

本实施例中，动态运行库是一种动态库，编译好的函数存储于动态运行库中，动态运行库向应用程序提供支持任务请求的 API（Application Program Interface，应用程序接口）接口，该 API 接口为动态运行库的库函数，在未加载动态运行库时，应用程序需要调用原生运行库的 API（原生运行库的库函数），在加载动态运

行库之后，原生运行库的 API 被动态运行库的库函数劫持，为此，在应用程序有任务需求时，实际调用的是动态运行库的 API，并向动态运行库的 API 发出任务请求。在原生运行库的基础，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，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 102，在启动应用程序时加载动态运行库，生成运行时虚拟化环境以及运行于运行时虚拟化环境中的协程调度线程。

本实施例中，协程调度线程用于调度至少一个协程，运行时虚拟化环境为替换原生运行库所生成的运行时环境后的环境。

本实施例中，加载动态运行库是在操作系统启动应用程序时，向该应用程序动态注入动态运行库，而动态运行库的运行时注入，就是在操作系统启动一个应用程序，并将该程序向内存中进行加载之前，先将某个动态库加载进来，并运行这个动态库中的初始化代码。通过这种程序加载方式，这个被预先载入的动态库就有机会提前构建出一个运行时的虚拟化环境，该运行时虚拟化环境可以替换原生运行库所生成的运行时环境，在该运行时的虚拟化环境下，应用程序或者操作系统可以实时调用动态运行库中的函数，本实施例中，应用程序启动之后，相应可以启动协程调度线程。

本实施例中，通过加载动态运行库可以在不对应用程序做任何修改（既不需要改应用程序的代码，也不需要应用程序进行重新编译）的前提下实现运行时虚拟化环境，此时应用程序无任何感知的运行在运行时虚拟化环境中（在该运行时虚拟化环境下，应用程序需要调取原生运行库的库函数时，实际调用的是动态运行库的库函数）。本公开提供的运行时虚拟化方法，不仅提升了性能、降低资源消耗，而且使得基于原生运行库开发的应用程序，在无修改、无感知的前提下，就可以获得性能的提升和资源消耗的降低。

本实施例中，协程调度线程是一种调度协程的线程，协程调度线程可以是一个，也可以是多个，协程调度线程的数量可以基于内核的数量确定，例如，64 核

的操作系统，协程调度线程为 64 个。

可选地，本实施例中的协程可以与线程能力相当，并且协程还可以完整的模拟线程的行为，为此，协程调度线程也可以认为是一种调度线程的线程。

步骤 103，在运行时虚拟化环境下，接收应用程序的任务请求。

本实施例中，应用程序的任务请求对应的任务可以是一个，也可以是多个，针对各个不同的任务，应用程序通过调用动态运行库中的函数可以建立相应的线程或协程。其中，每个线程执行相应的一个任务，每个协程执行相应的一个任务。

本实施例中，应用程序是一种基于原生运行库开发的程序，在应用程序无修改、无感知的前提下，使该应用程序无感知的运行在运行时虚拟化环境下，应用程序调用动态运行库中库函数实现相应功能。

步骤 104，基于任务请求和动态运行库，采用协程调度线程调度协程，以使协程完成任务请求对应的任务。

本实施例中，任务请求是一种通过协程处理任务的请求，执行主体在得到应用程序的任务请求之后，调用动态运行库中函数生成对应任务请求的至少一个协程，或者调用动态运行库中函数采集任务请求对应的现有的协程，基于协程的协程上下文区分不同的协程调度线程，执行主体将协程发送给相应的协程调度线程，协程调度线程调度得到的至少一个协程，使各个协程工作，完成任务请求对应的任务。

本公开的实施例提供的运行时虚拟化方法，首先，基于应用程序的原生运行库，构建动态运行库；其次，在启动应用程序时加载动态运行库，生成运行时虚拟化环境以及运行于运行时虚拟化环境中的协程调度线程，协程调度线程用于调度至少一个协程，运行时虚拟化环境为替换原生运行库所生成的运行时环境后的环境；再次，在运行时虚拟化环境下，接收应用程序的任务请求；最后，基于任务请求和动态运行库，采用协程调度线程调度协程，以使协程完成任务请求对应的任务。由此，本公开通过动态运行库生成运行时虚拟化环境，透明替换应用程序的运行时环境，在不对应用程序进行任何修改或重新编译的前提下，提升了应用程序的并发性能；通过动态运行库生成的具有线程能力的协程支持轻量化的协程调度，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。

在本实施例的一些可选实现方式中，基于任务请求和动态运行库，采用协程调度线程调度协程，以使协程完成任务请求对应的任务包括：基于任务请求和动态运行库，创建至少一个协程；将至少一个协程发送给协程调度线程，以采用协程调度线程调度至少一个协程。

本实施例中，上述基于任务请求和动态运行库，创建至少一个协程包括：基于任务请求确定任务；调度动态运行库中线程库函数钩子，通过线程库函数钩子创建与任务对应的协程。其中，线程库函数钩子是 hook 与第一库函数得到，第一库函数是原生运行库中创建线程、获取线程描述符的库函数。

本实施例提供的采用协程调度线程调度协程的方法，首先采用构建线程的方式构建与任务请求对应的协程，可以使得协程具有与线程相同的能力；其次，采用协程调度线程调度构建得到的协程，提高了任务执行效率。

在本实施例的一些可选实现方式中，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，采用协程调度线程调度至少一个协程包括：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。

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）。

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。

本实施例提供的协程调度线程，具有调度器和无锁队列，通过调度器调度无

锁队列中的协程实现了应用程序的任务请求的任务，并且由于协程调度线程不会被任何操作挂起，保证了任务执行的可靠性。

在本公开的另一个实施例中，上述方法还包括：接收应用程序的原生请求；基于原生请求和动态运行库中的原生运行库，创建至少一个线程，通过至少一个线程执行原生请求对应的任务。

本实施例中，原生请求是应用程序调取原生运行库中库函数，生成线程的请求，上述基于原生请求和动态运行库中的原生运行库，创建至少一个线程包括：基于原生请求，确定创建线程任务；基于创建线程任务从原生运行库中调取创建原生线程的函数，创建至少一个执行原生请求对应任务的线程。

本实施例提供的运行时虚拟化方法，通过动态运行库保留原生运行库的能力，在应用程序需要调度原生运行库的构建线程函数时，创建至少一个线程，通过至少一个线程执行原生请求对应的任务，提高了请求执行的多样性，保证了线程与协程的兼容效果。

鉴于传统技术协程库没有实现对线程本地存储的协程化模拟，在协程中使用线程本地存储不安全的问题，在公开的一些可选实现方式中，上述基于应用程序的原生运行库，构建动态运行库包括：

采用 hook 机制构建与第一库函数相关的线程库函数钩子，第一库函数为原生运行库中创建线程、获取线程描述符的库函数；在线程库函数钩子中，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。

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：

```
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 在动态运行库刚刚被加载时就会被创建；

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。

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。动态运行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: 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 需与 TCB 中的 dtv 进行增量同步。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，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。

本实施例实现了纯用户态的协程 TLS 切换，保持协程切换轻量化特征的同时，维系了 TLS 的跨协程安全性（兼顾高性能和“透明”虚拟化）。

本实施例提供构建动态运行库的方法，通过 hook 机制构建原生运行库中第一库函数的线程库函数钩子，通过线程库函数钩子中的协程上下文替换线程原始的线程描述符，通过全局变量标记当前运行主体，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，可以使构建的协程具有与线程相同的能力，提高了协程调度线程调度协程时的可靠性。

针对传统技术中，协程库没有妥善的解决协程锁和线程锁的兼容性问题。在公开的一些可选实现方式中，上述基于应用程序的原生运行库，构建动态运行库还包括：

采用 hook 机制构建与第二库函数相关的锁库函数钩子，第二库函数为原生运行库中分别与锁、条件变量、信号量相关的库函数；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。

本可选实现方式中，基于应用程序的请求，当前运行的主体可以通过动态运行库构建的协程，也可以通过运行原生运行库构建得到线程；通过全局变量，可以识别到运行主体的类型是协程还是线程，当运行主体是线程时，线程库函数钩子进一步判断当前的运行主体是协程调度线程还是其它线程，当当前的运行主体是协程调度线程时，不采用任何一种锁或信号量机制，通过该机制，可以保证协程调度线程不会因为使用锁或信号量导致其自身挂起。

当当前的运行主体是除协程调度线程之外的线程（如主线程或原生线程），基于当前的操作系统，采用与该操作系统对应的锁机制传递唤醒信号。例如，当操作系统是 Linux 系统时，采用 futex 机制传递唤醒信号；当操作系统是非 Linux 系统时，采用管道通信与 pollwait 的方式传递唤醒信号。

Futex 机制：一种用户态和内核态混合的同步机制，是 Linux 的基础构件之一。用户态与内核态之间通过基于一段共享内存实现的原子操作进行状态同步。用户态进程通过原子操作判断 futex 变量是否存在资源竞争，如需等待则陷入系统调用将线程挂起；唤醒 futex 时，也先通过原子操作检查是否存在挂起等待的任务，需要唤醒时才陷入系统调用执行“wake up”操作。简单地说，就是通过用户态原子操作，避免在没有资源竞争的情况下陷入内核态，从而大大提升“低竞争”场景下的运行效率。

本可选实现方式中，上述当前运行主体还可以是无超时等待操作的操作对象和有超时等待操作的操作对象，上述基于当前运行主体，给出当前运行主体相应的锁子机制包括：对于无超时等待操作，直接将协程上下文结构放入锁对象或条件变量对象的等待队列，并对该协程上下文结构调用“挂起”方法（协程模式下为 yield）。

而对于有超时等待操作，则需要将协程上下文结构同时放入相应的等待队列、以及时间队列中，再对协程上下文结构调用“挂起”方法。

本实施例实现了“透明”兼容协程和线程的锁机制，在对 C 运行库中锁相关库函数的 hook 中，自动识别当前处于协程上下文还是线程上下文，从而采取不同的行为模式。

本可选实现方式提供的锁库函数钩子构建方法，对原生运行库中线程锁相关

库函数进行 hook，实现了一种同时兼容线程和协程的锁方案，保证了运行时虚拟机系统对线程锁与协程锁的兼容性。

在本公开的一些可选实现方式中，上述基于应用程序的原生运行库，构建动态运行库还包括：

采用 hook 机制构建与第三库函数相关的切换库函数钩子，第三库函数为原生运行库中能够导致线程切换或挂起操作的库函数；切换库函数钩子用于采用协程切换或挂起机制代替线程切换或挂起机制。

本可选实现方式提供的切换库函数钩子构建方法，对原生运行库中能够导致线程切换或挂起操作的函数进行 hook，可以在切换或挂起线程时采用协程的切换与挂起方式，提升了操作系统的并发性能。

进一步参考图 2，作为对上述各图所示运行时虚拟化方法的实现，本公开提供了运行时虚拟化装置的一个实施例，该装置实施例与图 2 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 2 所示，本公开的实施例提供了一种运行时虚拟化装置 200，该装置 200 包括：构建单元 201、生成单元 202、任务接收单元 203、协程调度单元 204。其中，上述构建单元 201，可以被配置成基于应用程序的原生运行库，构建动态运行库，动态运行库以协程方式模拟原生运行库中实现的线程。上述生成单元 202，可以被配置成在启动应用程序时加载动态运行库，生成运行时虚拟化环境以及运行于运行时虚拟化环境中的协程调度线程，协程调度线程用于调度至少一个协程，运行时虚拟化环境为替换原生运行库所生成的运行时环境后的环境。上述任务接收单元 203，可以被配置成在运行时虚拟化环境下，接收应用程序的任务请求。上述协程调度单元 204，可以被配置成基于任务请求和动态运行库，采用协程调度线程调度协程，以使协程完成任务请求对应的任务。

在本实施例中，运行时虚拟化装置 200 中，构建单元 201、生成单元 202、任务接收单元 203、协程调度单元 204 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 101、步骤 102、步骤 103、步骤 104。

在一些实施例中，上述调度单元 204 被配置成：基于任务请求和动态运行库，创建至少一个协程；将至少一个协程发送给协程调度线程，以采用协程调度线程

调度至少一个协程。

在一些实施例中，上述协程调度线程包括：调度器以及用于放置至少一个协程的无锁队列，调度单元 204 进一步被配置成：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。

在一些实施例中，上述装置 200 还包括：原生接收单元，被配置成接收应用程序的原生请求；线程调度单元，被配置成基于原生请求和动态运行库，创建至少一个线程，通过至少一个线程执行原生请求对应的任务。

在一些实施例中，上述构建单元 201 被配置成：采用 hook 机制构建与第一库函数相关的线程库函数钩子，第一库函数为原生运行库中创建线程、获取线程描述符的库函数；在线程库函数钩子中，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。

在一些实施例中，上述构建单元 201 被配置成：采用 hook 机制构建与第二库函数相关的锁库函数钩子，第二库函数为原生运行库中分别与锁、条件变量、信号量相关的库函数；在锁库函数钩子中，通过全局变量，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。

在一些实施例中，上述构建单元 201 被配置成：采用 hook 机制构建与第三库函数相关的切换库函数钩子，第三库函数为原生运行库中能够导致线程切换或挂起操作的库函数；切换库函数钩子用于采用协程切换或挂起机制代替线程切换或挂起机制。

本公开的实施例提供的运行时虚拟化装置，首先，构建单元 201 基于应用程序的原生运行库，构建动态运行库；其次，生成单元 202 在启动应用程序时加载动态运行库，生成运行时虚拟化环境以及运行于运行时虚拟化环境中的协程调度线程，协程调度线程用于调度至少一个协程，运行时虚拟化环境为替换原生运行库所生成的运行时环境后的环境；再次，任务接收单元 203 在运行时虚拟化环境

下，接收应用程序的任务请求；最后，协程调度单元 204 基于任务请求和动态运行库，采用协程调度线程调度协程，以使协程完成任务请求对应的任务。由此，本公开通过动态运行库生成运行时虚拟化环境，透明替换应用程序的运行时环境，在不对应用程序进行任何修改或重新编译的前提下，提升了应用程序的并发性能；通过动态运行库生成的具有线程能力的协程支持轻量化的协程调度，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。

针对上述运行时虚拟化方法，本公开提供了一种运行时虚拟机系统，该系统具体可以应用于各种电子设备中。

如图 3 所示，本公开的实施例提供了一种运行时虚拟机系统 300，该系统 300 包括：运行在运行时虚拟化环境 301 下的主线程 302、协程调度线程 303、事件驱动线程 304 以及异步输入输出和定时器线程 305，其中，运行时虚拟化环境 301 通过加载动态运行库生成，且动态运行库基于应用程序的原生运行库构建得到，动态运行库以协程方式模拟原生运行库中实现的线程；协程调度线程 303 用于调度至少一个协程，以使协程完成应用程序的任务请求；主线程 302 在应用程序启动时创建；事件驱动线程 304 用于维护文件句柄的状态，并在读或写缓存就绪、且存在阻塞在文件句柄上的读写任务时，唤醒相应的协程；异步输入输出和定时器线程 305，获取异步输入输出任务结果，唤醒处于 IO 等待的协程；在系统调用超时返回时，弹出超时耗尽的任务并唤醒；在系统调用异步 IO 任务结果时，唤醒处于 IO 等待的协程。

主线程 302，应用启动时就会创建，可以看作是一个特殊的系统原生线程。

协程调度线程 303，协程调度线程负责周而复始的从无锁队列中取出就绪态协程任务，使用 resume（包括刚刚启动的协程）方式将协程上下文换入，并继续执行相应的协程。协程发生让出（yield，包括协程结束）时，如果属于非等待态协程（即，只是暂时让出 CPU），就需要把对应的协程放回唤醒队列，重新接受调度。本公开的运行时虚拟机系统在实践中采用了 boost 的 context 库，用以实现协程栈的切换，以及协程函数的换入换出。

异步输入输出和定时器线程 305，异步 IO 和定时器的后台部分，可以在同一个后台线程中实现。这是因为，等待异步 IO 事件的系统调用接口可以指定超时

时间，并且这类超时的时间精度也足够高，适合 timer 的实现。作为负载较低的后台线程，将前述二者功能合并，有助于进一步降低后台线程个数。异步输入输出和定时器线程 305 通过两种数据结构与系统中其他组件进行交互：

AIO task queue: 用以传递异步 IO 任务的 `io_event`，后台线程调用 `io_getevents` 返回结果后，会搜索此队列，传递 IO 操作结果，并唤醒处于 IO 等待的协程。

Timed queue: 管理超时等待状态的协程上下文。这个队列存储的是一种 key-value 结构，以超时时间为 key，COCTX 为 value，并以 key 值排序（key 值较小——即超时时间离当前时间间隔最短——的任务排在前面）。

该后台线程周而复始的调用 `io_getevents` 系统调用，当此调用超时返回时，处理 timed queue，将超时耗尽的任务（既可能是协程，也可能是原生线程）弹出并唤醒；当系统调用返回异步 IO 任务结果时，则传递 `io_event` 结构的内容，并唤醒对应协程。

事件驱动线程 304，又叫 Epoll 线程，`epoll` 线程负责系统中已经打开的所有支持 poll 操作的文件句柄的 POLLIN & POLLOUT 状态维护。因为动态运行库已经 hook 了包括 `socket`、`open`、`close` 等库函数，因而，当文件句柄（主要是 `socket`）刚刚被创建时，就被放入 `sockfd map` 中进行维护。该线程通过调用 `epoll_wait`，周而复始的更新 `sockfd map` 中所有文件句柄的读/写缓存状态，当遇到读/写缓存就绪、且当前存在阻塞在此文件句柄上的读写任务或者 poll wait 任务（如，在 `poll`、`select`、`epoll_wait` 等库函数钩子中陷入等待的协程）时，即唤醒对应的协程。

针对上述运行时虚拟机系统，最小化的启动一个挂载运行时虚拟机的应用程序，仅需创建 4 个线程，即：1 个主线程、1 个协程调度线程、1 个事件驱动线程、以及 1 个异步输入输出和定时器线程。即便以上述最小化资源配置启动的进程（也就是仅需 4 个线程），仍然能够轻松驾驭数以百计的小型并发任务，而不出现卡顿。而面对资源占用率比较高的大型应用程序（如数据库）时，可以考虑使用与主机 CPU 核数相同个数的协程调度线程，其他后台线程按比例缩减的方式配置。也就是说，无论是小任务多进程，还是大任务多线程的应用场景，本公开的运行时虚拟机都可以很好的给予支持。具体举例如下：

第一种情况，所有进程都以最小化配置启动（4 线程），每个进程的负载平均为 100 个小任务。当启动 1000 个进程时，系统整体负载达到 10 万并发，但只需创建 4000 个线程。

第二种情况，一个 64 核系统中，只启动一个大型应用（如数据库），其中：主线程 1 个，协程调度线程 64 个，epoll 线程 8 个，AIO/timer 线程 8 个，数据库的后台线程 10 个（如：发布订阅、备份恢复、监控、日志清理、表空间回收整理等，均使用系统原生线程），总共 91 个线程。这样的配置，足以应对数以 10 万计的数据库并发访问，而总的线程个数还不到 100 个。

本公开的实施例提供的运行时虚拟机系统，通过动态库注入技术，提供一套完整的协程化的动态运行库，所有和操作系统交互的途径已经都被本公开中动态运行的库函数钩子接管了（也就是实现了一整套协程版本的库函数），应用程序虚拟运行在另一个与原生系统完全兼容的虚拟系统上，即协程虚拟化环境，宿主系统中的任意应用都可以不做任何修改，直接运行于协程化虚拟环境，透明化的实现了协程替代线程的方案。

在本公开的另一个实施例中，又提供了一种运行时虚拟机系统 400，该系统 400 包括：运行在运行时虚拟化环境 401 下的主线程 402、协程调度线程 403、事件驱动线程 404、异步输入输出和定时器线程 305 以及系统原生线程 306。

相对于图 3 所示实施例，图 4 所示实施例中具有系统原生线程 306，图 4 所示实施例相对于图 3 所示实施例，生成的动态运行库保留了原生运行库的特征，通过原生运行库可以在应用程序有需求时调取原生运行库，生成系统原生线程 306。

由于线程管理相关的库函数都已经被 hook，默认情况下加载动态运行库后创建出的都是协程，在应用程序有生成系统原生线程的需求时，使用者可以通过指定线程函数的函数名选择将对应线程以原生线程方式启动；也可以通过实现创建线程的库函数对应的回调接口，动态指定某个新创建的线程是以协程方式启动还是以原生线程方式启动。

可选地，本实施例提供的运行时虚拟机系统，除参数配置接口、日志接口等常见接口外；还可以实现了一套回调（callback）接口，动态运行库中大部分库函数钩子都可以配备对应的回调函数。开发者可以选择实现某些或全部回调，也可以不使用任何回调。通过这些回调接口，开发者可以部分改变库函数行为，甚至完全接管某些特定的库函数，从而实现诸如：

将某些原本需要修改系统底层机制才能实现的优化，直接注入到库函数中；

而这种注入只针对当前应用，不会影响其他正在运行的程序；

基于本公开的运行时虚拟机系统进行二次开发，实现另一套运行时虚拟机系统。（比如，接管文件 IO 相关的所有库函数，把本地文件读写改为云盘读写，即可实现一个基于云盘的运行时虚拟机）

以一个 recv（接收网络包）接口的 callback 为例，说明回调函数在本公开的运行时虚拟机系统中的运行逻辑。当开发者指定了回调函数时，本公开的运行时虚拟机系统的 recv 函数会把从 socket 中接收到的网络包先传递给回调函数，并根据回调函数的返回值，确定 recv 函数最终的返回方式。回调函数可以选择不做任何介入，直接返回这个包；也可以选择丢弃这个包；甚至进一步对包的内容进行修改，向应用返回修改后的包。

本实施例提供的运行时虚拟机系统提供回调接口，开发者可以通过实现回调函数，向协程库的 hook 中注入新的逻辑或完全改变 hook 函数的逻辑，从而实现对原始应用程序的扩展、增强，或者借助此运行时虚拟机框架实现一套全新的运行时虚拟化环境。

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 RAM 503 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：基于应用程序的原生运行库，构建动态运行库；在启动应用程序时加载动态运行库，生成协程调度线程，协程调度线程用于调度至少一个协程；接收应用程序的任务请求；基于任务请求和动态运行库，采用协程调度线程调度协程，以使协程完成任务请求对应的任务。

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）。

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。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，包括构建单元、生成单元、任务接收单元、协程调度单元。其中，这些单元的名称在某种情况下并不构成对该单元本身的限定，例如，构建单元还可以被描述为“被配置成基于应用程序的原生运行库，构建动态运行库”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。

权利要求书

- 1、一种运行时虚拟化方法，所述方法包括：
- 基于应用程序的原生运行库，构建动态运行库，所述动态运行库以协程方式
- 5 模拟所述原生运行库中实现的线程；
- 在启动所述应用程序时加载所述动态运行库，生成运行时虚拟化环境以及运行于所述运行时虚拟化环境中的协程调度线程，所述协程调度线程用于调度至少一个协程，所述运行时虚拟化环境为替换所述原生运行库所生成的运行时环境后的环境；
- 10 在所述运行时虚拟化环境下，接收所述应用程序的任务请求；
- 基于所述任务请求和所述动态运行库，采用所述协程调度线程调度所述协程，以使所述协程完成所述任务请求对应的任务。
- 2、根据权利要求 1 所述的方法，其中，所述基于所述任务请求和所述动态运行库，采用所述协程调度线程调度协程，以使所述协程完成所述任务请求对应的任务包括：
- 15 基于所述任务请求和所述动态运行库，创建至少一个协程；
- 将所述至少一个协程发送给所述协程调度线程，以采用所述协程调度线程调度所述至少一个协程。
- 20 3、根据权利要求 2 所述的方法，其中，所述协程调度线程包括：调度器以及用于放置所述至少一个协程的无锁队列，所述采用所述协程调度线程调度所述至少一个协程包括：
- 所述调度器从所述无锁队列中获取所述至少一个协程中处于就绪状态的协
- 25 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，所述调度器从所述无锁队列中获取下一个处于就绪状态的协程，并同步所述下一个协程的协程上下文与所述线程控制块中的线程上下文；
- 30 在所述下一个协程主动让出之后，所述调度器继续从所述无锁队列中获取协

程并同步所述线程控制块，直至所述无锁队列中无任何协程为止。

4、根据权利要求 1 所述的方法，其中，所述方法还包括：

接收所述应用程序的原生请求；

5 基于所述原生请求和所述动态运行库中的原生运行库，创建至少一个线程，通过所述至少一个线程执行所述原生请求对应的任务。

5、根据权利要求 1-4 之一所述的方法，其中，所述基于应用程序的原生运行库，构建动态运行库包括：

10 采用 hook 机制构建与第一库函数相关的线程库函数钩子，所述第一库函数为所述原生运行库中创建线程、获取线程描述符的库函数；

在所述线程库函数钩子中，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所述协程上下文结构中的线程本地存储信息，所述同步管理模块用于在所述协程调度线程调度协程时，同步该协程的线程本地存储。

15

6、根据权利要求 5 所述的方法，其中，所述基于应用程序的原生运行库，构建动态运行库还包括：

采用 hook 机制构建与第二库函数相关的锁库函数钩子，所述第二库函数为

20 所述原生运行库中分别与锁、条件变量、信号量相关的库函数；

在所述锁库函数钩子中，通过所述全局变量，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。

7、根据权利要求 5 所述的方法，其中，所述基于应用程序的原生运行库，构建动态运行库还包括：

25

采用 hook 机制构建与第三库函数相关的切换库函数钩子，所述第三库函数为所述原生运行库中能够导致线程切换或挂起操作的库函数；

所述切换库函数钩子用于采用协程切换或挂起机制代替线程切换或挂起机制。

30

8、一种运行时虚拟化装置，所述装置包括：

构建单元，被配置成基于应用程序的原生运行库，构建动态运行库，所述动态运行库以协程方式模拟所述原生运行库中实现的线程；

生成单元，被配置成在启动所述应用程序时加载所述动态运行库，生成运行时虚拟化环境以及运行于所述运行时虚拟化环境中的协程调度线程，所述协程调度线程用于调度至少一个协程，所述运行时虚拟化环境为替换所述原生运行库所生成的运行时环境后的环境；

任务接收单元，被配置成在所述运行时虚拟化环境下，接收所述应用程序的任务请求；

10 协程调度单元，被配置成基于所述任务请求和所述动态运行库，采用所述协程调度线程调度协程，以使所述协程完成所述任务请求对应的任务。

9、根据权利要求 8 所述的装置，其中，所述调度单元被配置成：基于任务请求和动态运行库，创建至少一个协程；将至少一个协程发送给协程调度线程，以采用协程调度线程调度至少一个协程。

10、根据权利要求 8 所述的装置，其中，所述协程调度线程包括：调度器以及用于放置至少一个协程的无锁队列，调度单元进一步被配置成：

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；

20 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。

25

11、根据权利要求 8 所述的装置，其中，所述装置还包括：原生接收单元，被配置成接收应用程序的原生请求；

线程调度单元，被配置成基于原生请求和动态运行库，创建至少一个线程，通过至少一个线程执行原生请求对应的任务。

30

12、根据权利要求 8-11 之一所述的装置，其中，所述构建单元被配置成：采用 hook 机制构建与第一库函数相关的线程库函数钩子，第一库函数为原生运行库中创建线程、获取线程描述符的库函数；

5 在所述线程库函数钩子中，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。

13、根据权利要求 12 所述的装置，其中，所述构建单元被配置成：采用 hook 10 机制构建与第二库函数相关的锁库函数钩子，第二库函数为原生运行库中与锁相关的库函数；

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。

15 14、根据权利要求 12 所述的装置，其中，所述构建单元被配置成：

采用 hook 机制构建与第三库函数相关的切换库函数钩子，第三库函数为原生运行库中能够导致线程切换或挂起操作的库函数；

切换库函数钩子用于采用协程切换或挂起机制代替线程切换或挂起机制。

20 15、一种运行时虚拟机系统，所述系统包括：运行在运行时虚拟化环境下的协程调度线程、主线程、事件驱动（EPOLL）线程以及异步输入输出（AIO）和定时器线程，其中，所述运行时虚拟化环境通过加载动态运行库生成，且所述动态运行库基于应用程序的原生运行库构建得到，所述动态运行库以协程方式模拟所述原生运行库中实现的线程；

25 所述协程调度线程用于调度至少一个协程，以使所述协程完成所述应用程序的任务请求；

所述主线程在所述应用程序启动时创建；

所述事件驱动线程用于维护文件句柄的状态，并在读或写缓存就绪、且存在阻塞在文件句柄上的读写任务时，唤醒相应的协程；

30 所述异步输入输出和定时器线程，获取异步输入输出任务结果，唤醒处于 IO

等待的协程；在系统调用超时返回时，弹出超时耗尽的任务并唤醒；在系统调用异步 IO 任务结果时，唤醒处于 IO 等待的协程。

16、一种电子设备，包括：

5 一个或多个处理器；

存储装置，其上存储有一个或多个计算机程序；

当所述一个或多个计算机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 1-7 中任一项所述的方法。

10 17、一种非瞬时性计算机可读存储介质，其上存储有计算机程序，其中，该计算机程序被处理器执行时实现如权利要求 1-7 中任一项所述的方法。