**电子科技大学**

课程设计（大型作业）任务书

（2023/2024学年第二学期）

**课程名称**  嵌入式系统设计技术

**院（系）**  电子科技大学（深圳）高等研究院

**设计名称**

**组 长**

**小组成员**

**时 间**

一、设计目的

本课程设计的主要目的是让学生深入理解和掌握嵌入式系统设计中多核处理器优化、操作系统虚拟化和容错恢复机制。通过本课程设计，：

* 掌握多核处理器嵌入式系统内核设计的关键概念，包括并行处理、任务分配和核间通信。
* 理解并实现嵌入式操作系统的虚拟化技术，能够在单一硬件平台上高效管理和调度多个操作系统实例。
* 学习并应用容错和恢复机制，确保嵌入式系统在遇到不可预测的硬件或软件失败时能够稳定运行或快速恢复。
* 通过实际项目设计和实现，提升分析、评估和优化嵌入式系统设计的能力。
* 提升团队合作、问题解决和创新能力。

二、设计选题

本课程共提供四个候选题目，每组需要完成两道题，其一可在a)、b)、c)中任选1个，其二d)为必做，具体选题内容如下：

1. **面向多核处理器的嵌入式操作系统内核设计**：设计并实现一个嵌入式操作系统内核，该内核应支持多核并行处理、任务分配、同步以及高效的核间通信。设计目标是模拟一个嵌入式多核环境，其中内核能够有效地管理和调度多个核上的任务，实现资源共享和任务同步，同时保证系统的高性能和响应性。
2. **嵌入式操作系统的虚拟化机制设计**：设计并实现一个嵌入式操作系统的虚拟化机制，该机制应支持在单个硬件平台上同时运行多个隔离的操作系统实例。设计目标是创建一个轻量级的虚拟化层（Hypervisor），能够有效地管理和调度多个虚拟机实例，实现资源的隔离和共享，同时保证每个虚拟机实例的高性能和响应性。
3. **嵌入式操作系统的容错和恢复机制设计**：设计并实现一个嵌入式操作系统的容错和恢复机制，该机制应支持系统在遇到硬件故障或软件异常时继续保持稳定运行或快速恢复。设计目标是构建一套机制，能够实时监测系统状态，识别并隔离故障，重新分配任务，并恢复系统至正常运行状态，确保系统的实时性和可靠性不受故障影响。
4. **嵌入式操作系统的系统调用编译设计**：设计并实现一个嵌入式操作系统的系统调用，并将其编译进Linux内核中。系统调用（System Calls）是应用程序与操作系统之间进行通信的重要方式。它们提供了一种应用程序能够请求操作系统执行特定任务的机制。设计目标是遍历Linux进程，实现以树形结构显示进程之间关系的系统调用，包括每个进程以及其父进程、子进程，了解各个进程之间的关系。

三、设计内容、要求及组织形式

课程设计要求学生从本课程提供候选题目中选择主题完成其功能模块。学生需要掌握与嵌入式系统设计相关的基本理论知识，包括但不限于多核处理技术、操作系统虚拟化、容错机制、并发控制等。

学生将应用所学课程知识，设计并实现针对嵌入式操作系统的项目，这可以是优化现有系统的性能、实现新的虚拟化机制或开发先进的容错策略。此外，学生需要制作课程设计报告，详细记录项目的设计理念、实现过程、测试结果和分析，并在课程结束时进行汇报。

课程设计鼓励团队合作，学生可以根据实际情况分组，每组建议4人。团队成员需积极参与讨论和共同实施项目，每位成员都应承担相应的责任。所有学生都应保持团队合作精神和诚信，严禁抄袭或剽窃。

四、考核形式及成绩评定办法

本课程设计最终每个组的总评成绩分两个部分组成：

1.项目展示与答辩：对项目设计与实现包括项目背景、设计思路、实验结果、分析与讨论等进行展示和答辩，占总成绩的70%；

2.项目实现：实现的功能、技术难度、创新性等，占总成绩的30%；

五、面向多核处理器的嵌入式操作系统内核设计选题

1. 功能实现具体要求

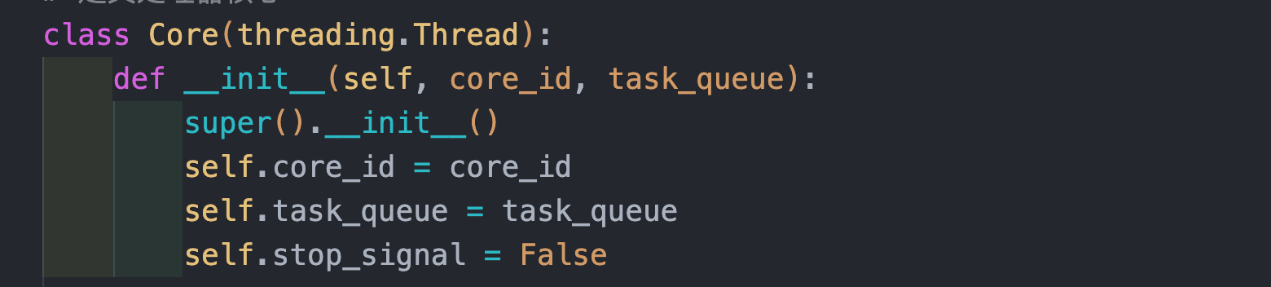
1. 多核处理器仿真：利用Python的多线程或多进程库模拟多核处理器环境，每个核作为一个独立的线程或进程。
2. 任务调度器仿真：设计一个任务调度器，根据定义的调度策略分配任务到不同的核上，并管理任务的生命周期。
3. 通信机制实现：使用Python的队列、管道、共享内存等机制实现核间的通信和数据共享。
4. 同步机制实现：利用Python的线程或进程同步机制（如锁、事件、条件变量）实现核间的同步。

2. 实现指南

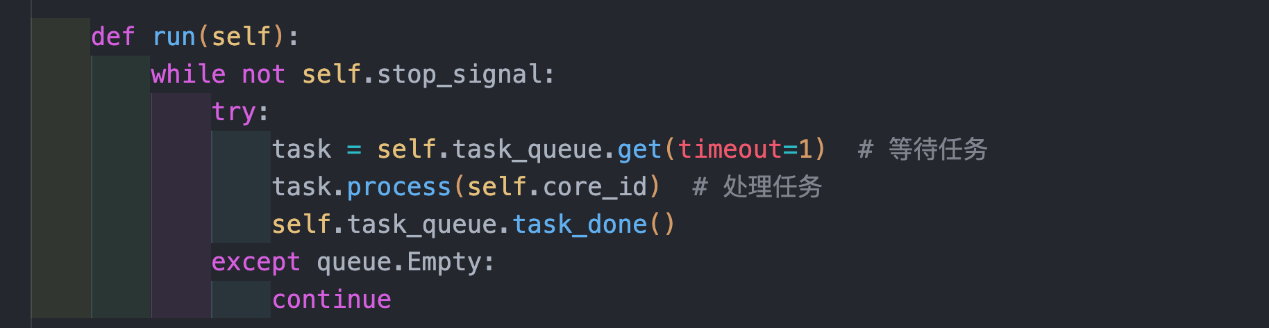
步骤 1: 设计仿真环境的基础结构

a. 定义处理器核心（Core 类）

* 初始化 (\_\_init\_\_ 方法): 初始化每个核心实例，为其分配一个唯一的ID（core\_id）和一个任务队列（task\_queue），用于从中获取任务。还需要一个信号（stop\_signal），用于通知核心何时停止执行。



* 运行 (run 方法): 在一个循环中，每个核心尝试从任务队列中获取任务。如果获取到任务，则调用该任务的处理方法（process）。如果队列为空，则捕获queue.Empty异常并继续等待。

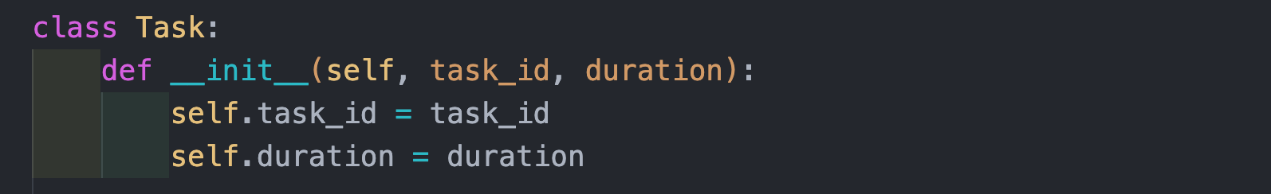


* 停止 (stop 方法): 设置stop\_signal为True，使得核心可以优雅地停止执行。

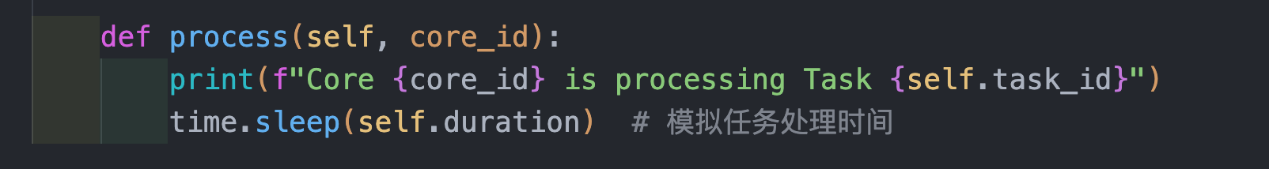


b. 定义任务（Task 类）

* 初始化 (\_\_init\_\_ 方法): 为每个任务实例分配一个唯一的ID（task\_id）和一个执行时间（duration），模拟不同任务的处理需求。

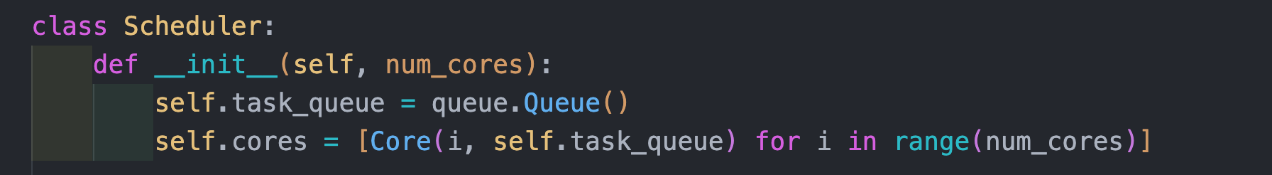


* 处理 (process 方法): 模拟任务处理过程，可以简单地使用time.sleep(duration)来模拟。

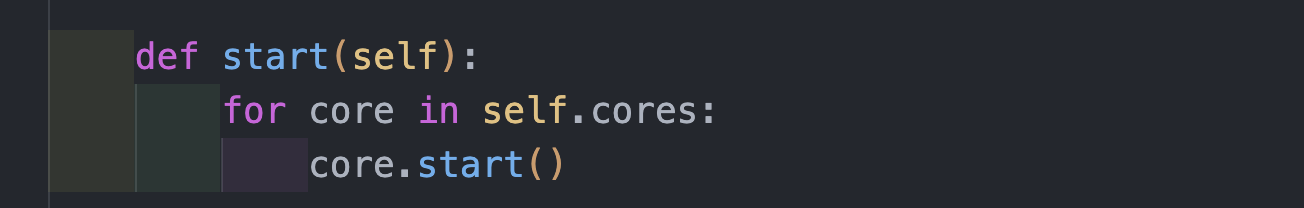


c. 定义调度器（Scheduler 类）

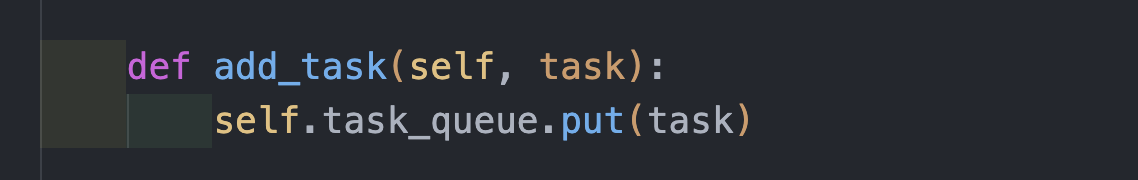
* 初始化 (\_\_init\_\_ 方法): 初始化调度器实例，创建一个任务队列和指定数量的核心实例。



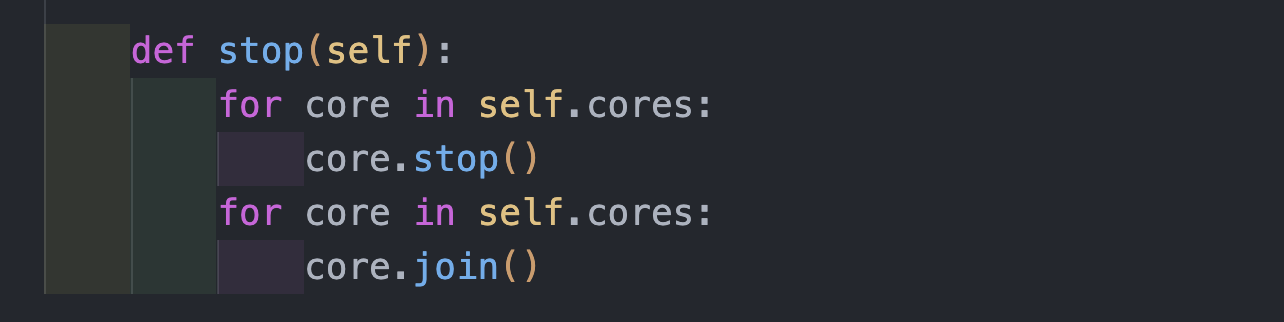
* 启动 (start 方法): 启动每个核心的线程，使它们开始等待并处理任务。



* 添加任务 (add\_task 方法): 允许向任务队列中添加新任务，这些任务将被核心线程处理。

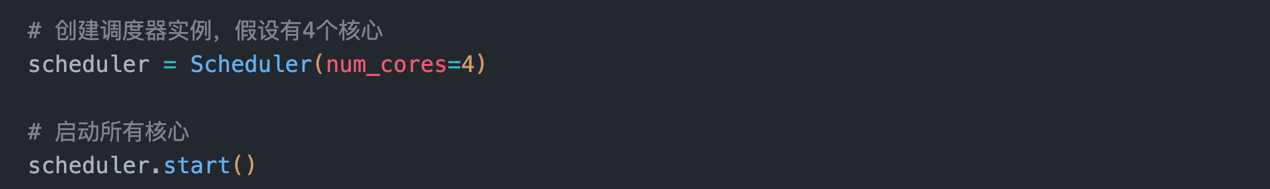


* 停止 (stop 方法): 通知每个核心停止运行，并等待它们全部停止。

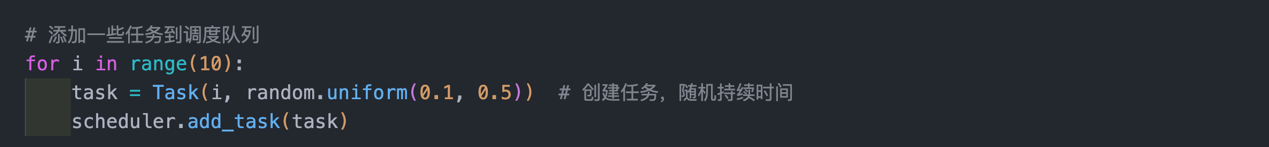


步骤 2: 实现简单的任务调度

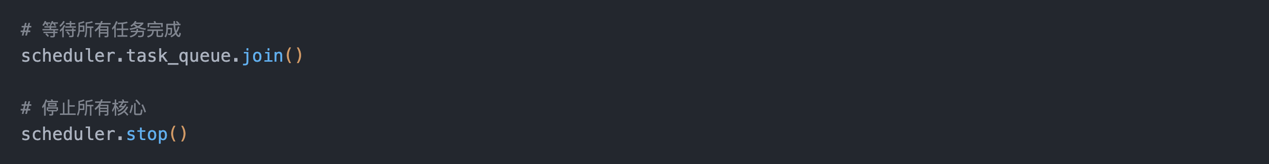
a. 创建调度器并启动核心: 实例化调度器，指定核心数量。调用调度器的start方法，启动所有核心。



b. 添加任务到调度器: 循环创建多个Task实例，每个实例有唯一的task\_id和随机的duration。使用调度器的add\_task方法将每个任务添加到任务队列中。



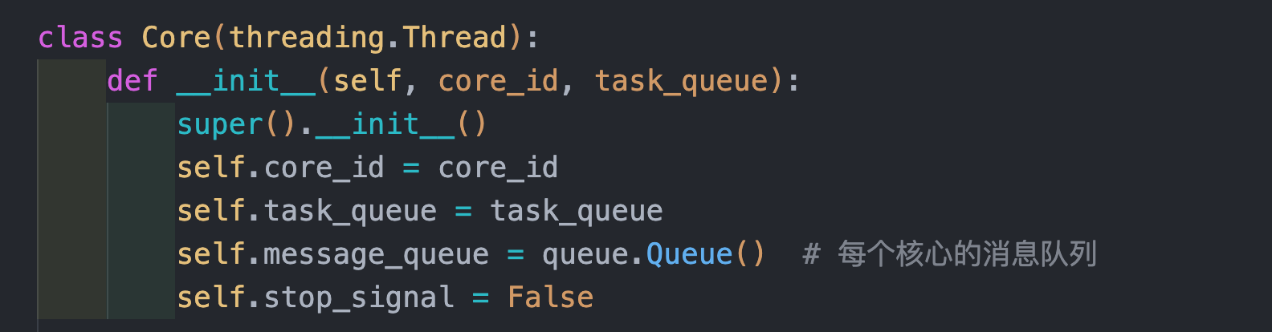
c. 等待任务完成并停止核心: 使用task\_queue.join()等待所有任务完成。调用调度器的stop方法，停止所有核心。



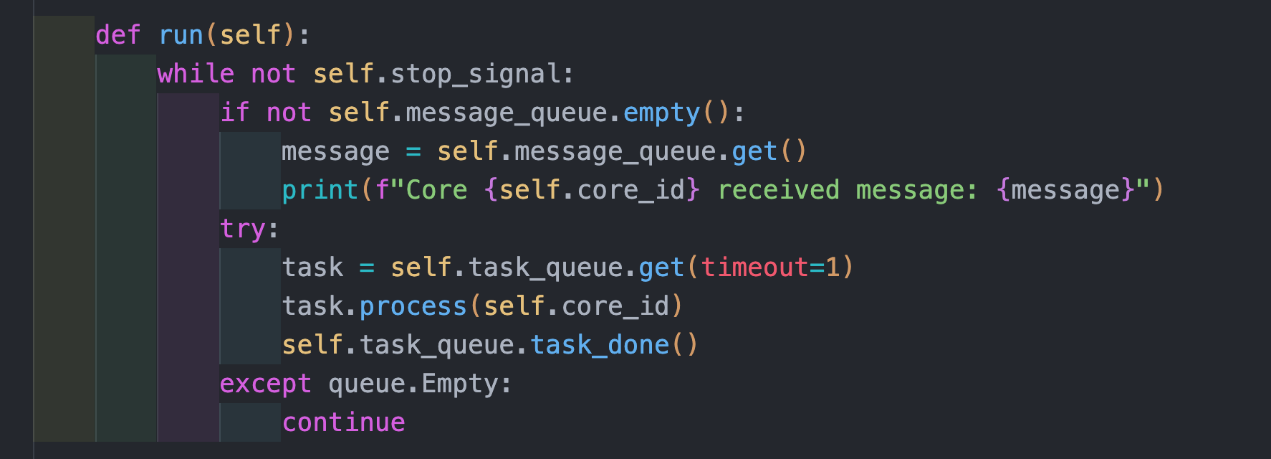
步骤 3: 添加核间通信机制

a. 在核心类中添加消息队列

* 修改核心类 (Core): 在每个Core实例中添加一个message\_queue，用于存储接收到的消息。



* 接收消息: 在Core的run方法中，检查message\_queue是否有消息，如果有，则打印或处理该消息。

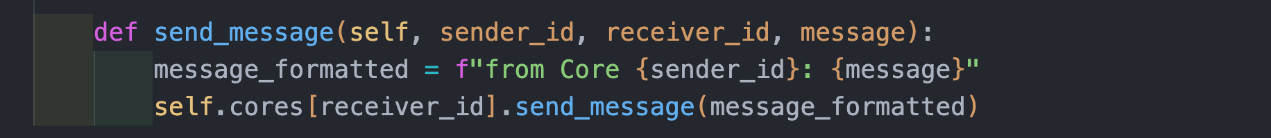


* 发送消息: 添加一个send\_message方法，允许核心将消息放入另一个核心的message\_queue中。



b. 调度器中的消息传递方法

* 添加发送消息方法 (send\_message): 在Scheduler类中添加一个方法，允许调度器协调核心之间的消息发送。



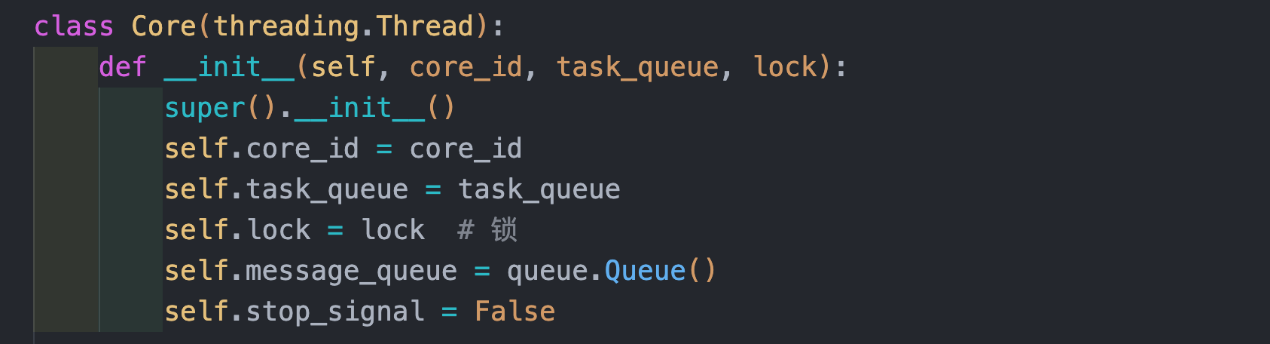
步骤 4: 实现同步机制

a. 在调度器中定义锁: 在Scheduler类中定义一个锁（self.lock），这个锁将被所有核心共享。



b. 在核心操作中使用锁

* 修改核心类: 更新Core类，使其在执行关键操作时（如处理任务或读取消息）尝试获取锁。



* 处理任务和消息时获取锁: 在run方法中，使用with self.lock语句块来确保在核心访问共享资源时获得锁。



六、嵌入式操作系统虚拟化机制设计选题

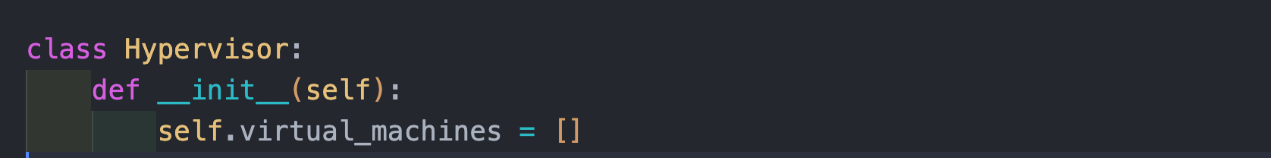
1. 功能实现具体要求

1. 虚拟机监控器（Hypervisor）实现: 设计一个轻量级的Hypervisor框架，支持加载和运行虚拟机实例。实现硬件资源的抽象化和虚拟化，包括CPU、内存和I/O设备。
2. 虚拟CPU管理: 开发虚拟CPU调度器，模拟CPU时间片分配给不同的虚拟机。实现虚拟CPU状态管理，包括创建、运行、挂起和恢复等状态。
3. 虚拟内存管理:设计并实现虚拟内存管理器，为每个虚拟机提供独立的地址空间。实现地址转换机制，将虚拟地址映射到物理地址。
4. 虚拟I/O设备实现:实现虚拟网络接口和虚拟存储设备，确保虚拟机的I/O操作隔离。开发虚拟设备驱动，支持虚拟机与虚拟设备的交互。
5. 虚拟机间通信与协作:设计并实现虚拟机间的通信机制，支持数据交换和协作。实现虚拟网络或共享内存机制，以支持虚拟机间的高效通信。

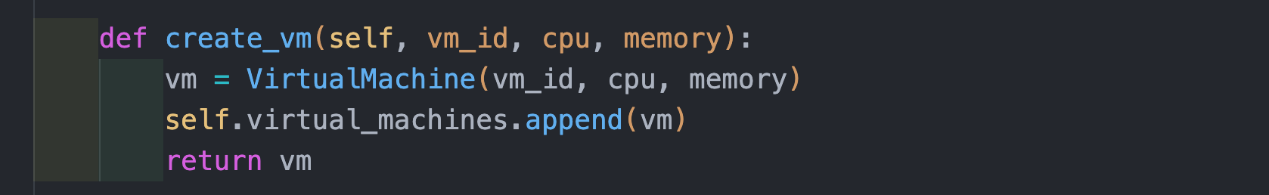
2. 实现指南

步骤 1: 实现虚拟机监控器（Hypervisor）

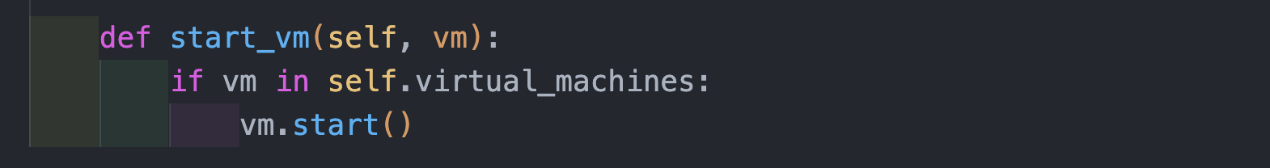
1. 初始化Hypervisor： 初始化虚拟机列表。



1. 创建虚拟机： 创建并初始化虚拟机实例。

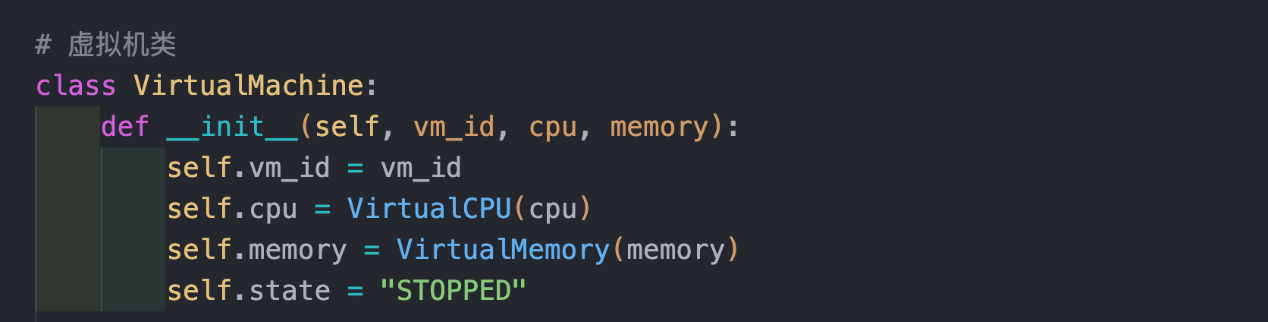


1. 启动虚拟机： 更改虚拟机状态并模拟启动过程。



步骤 2：实现虚拟机类

a. 初始化 VirtualMachine：定义 VirtualMachine 类，初始化其 ID，CPU，内存和状态。

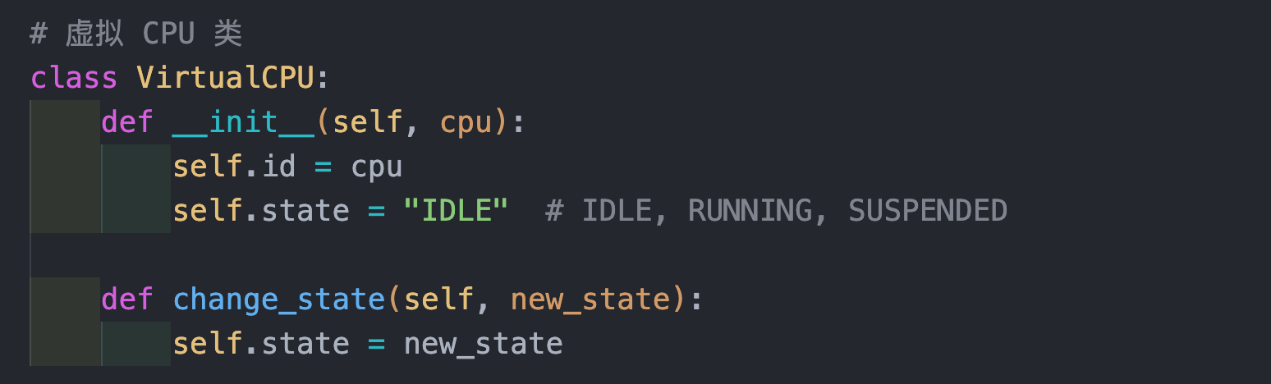


b. 启动虚拟机：实现启动方法，改变虚拟机状态为“RUNNING”。

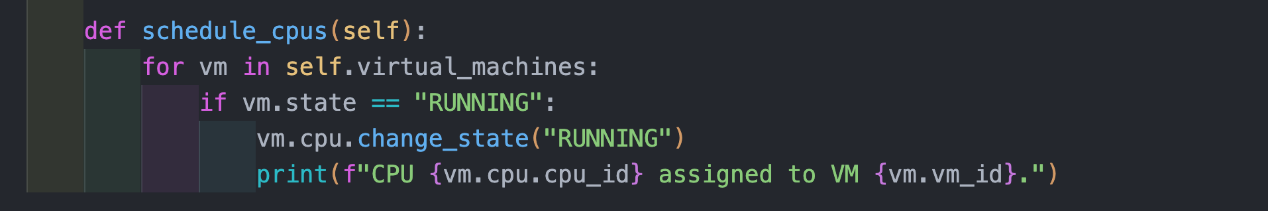


步骤 2：实现虚拟CPU管理

1. 定义VirtualCPU类： 包括状态和状态更改方法。



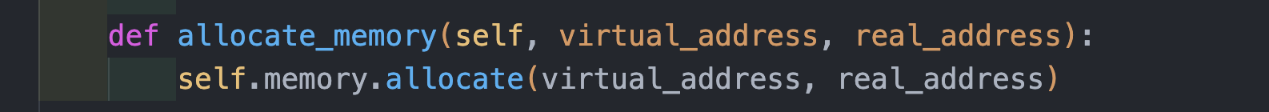
1. CPU调度（示例化为轮询调度）： 在Hypervisor中模拟CPU调度。



步骤 3：实现虚拟内存管理

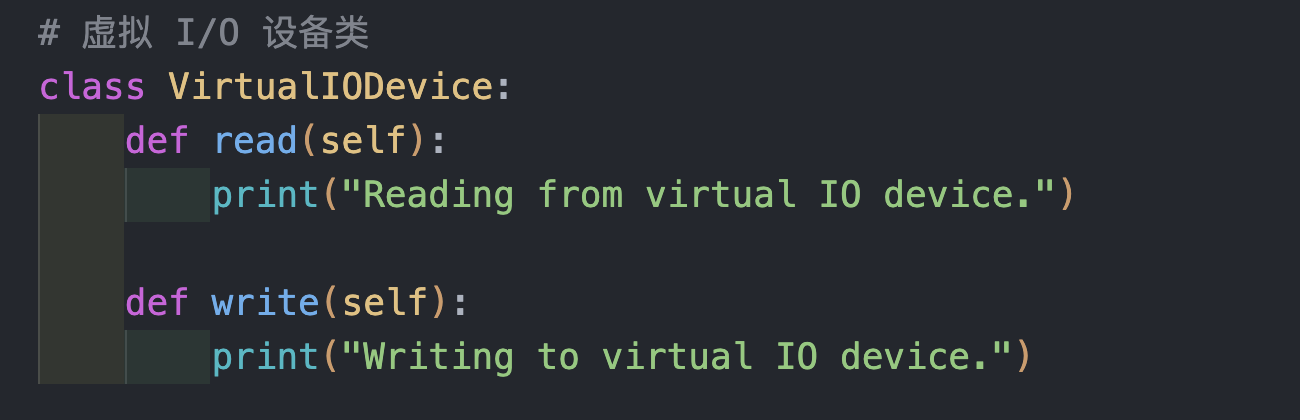
1. 定义VirtualMemory类： 包含内存大小和分配方法。

b. 在虚拟机中使用虚拟内存：在VirtualMachine中添加allocate\_memory方法

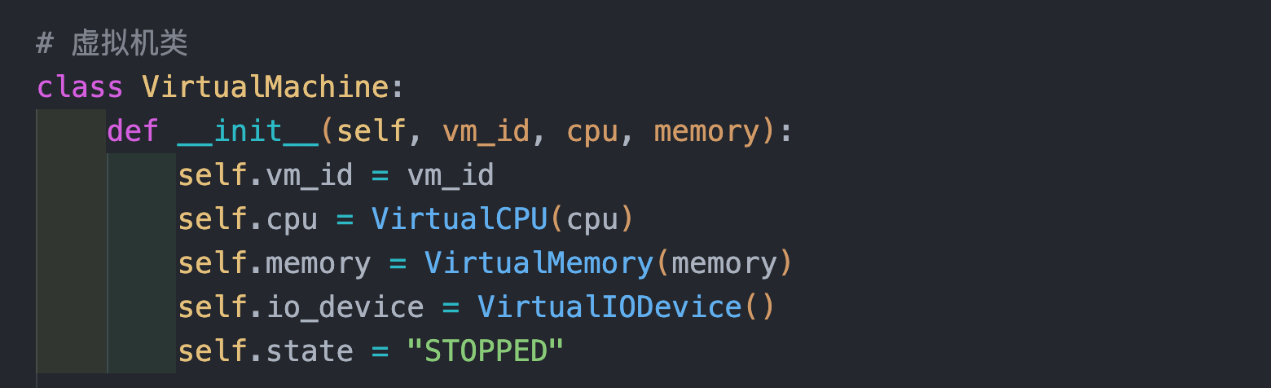


步骤 4：实现虚拟I/O设备

1. 定义VirtualIODevice类： 模拟I/O读写操作。

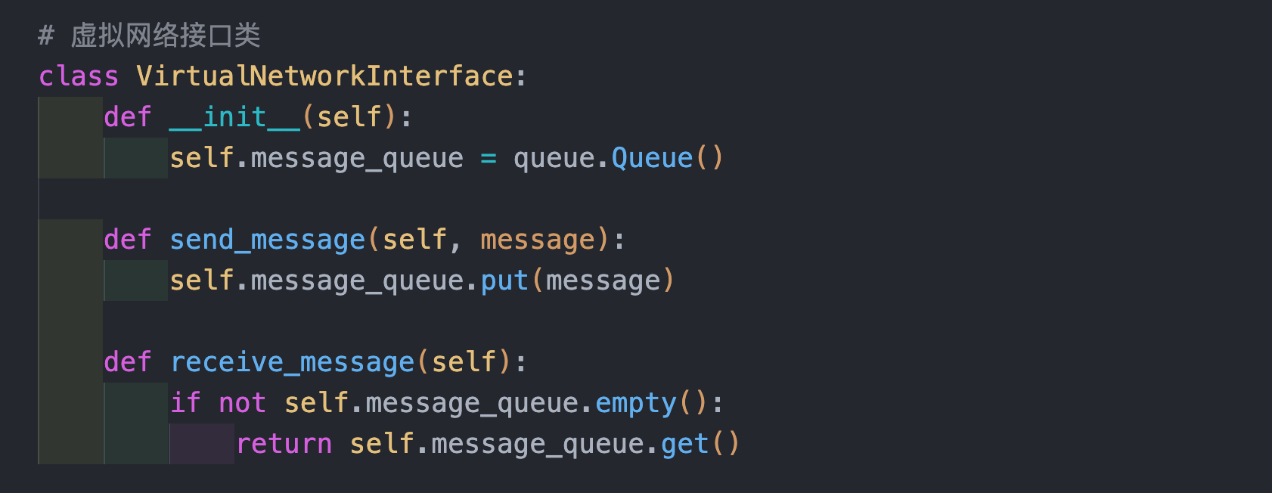


1. 在虚拟机中集成虚拟I/O设备：

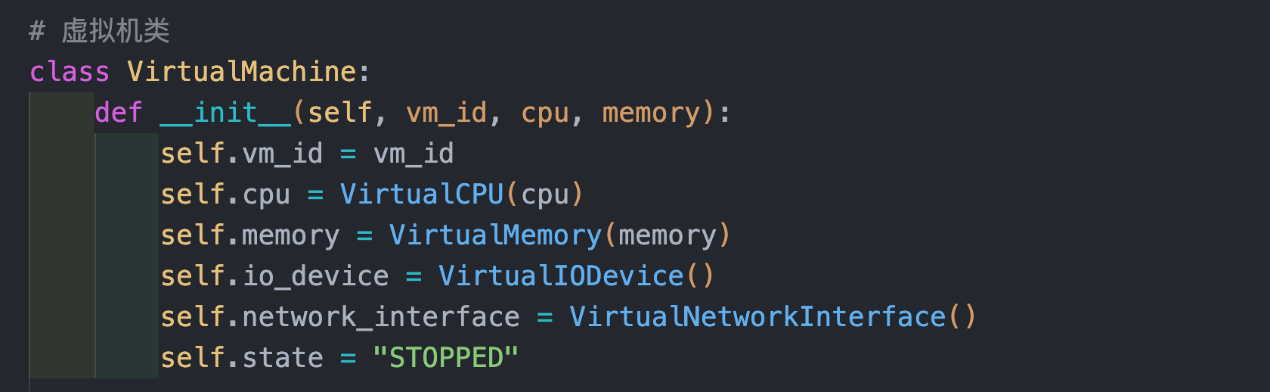


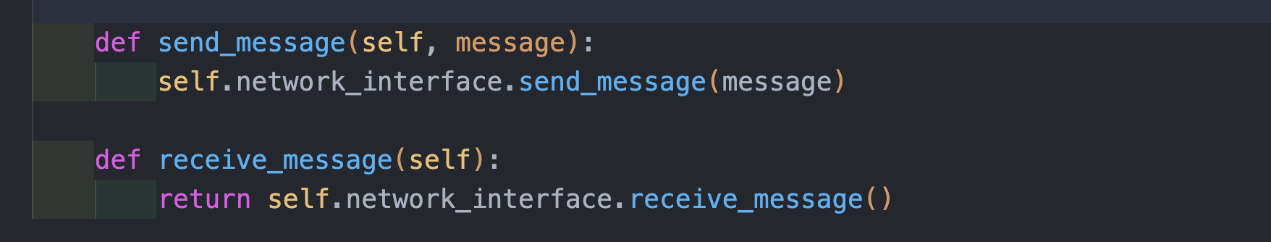
步骤5: 实现虚拟机间通信与协作

1. 定义VirtualNetworkInterface类： 实现消息队列来模拟网络接口。



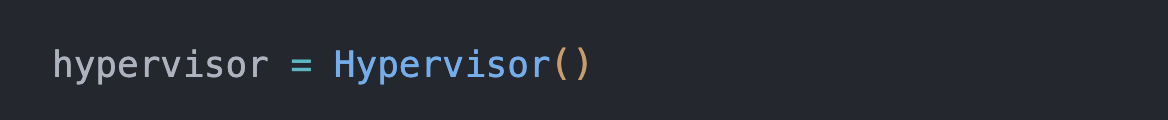
1. 在虚拟机中使用网络接口来发送和接收消息：



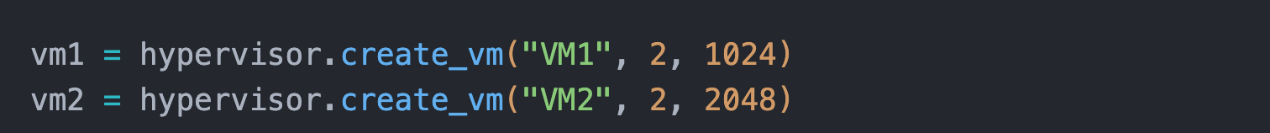


步骤 6：实现简单的虚拟机管理

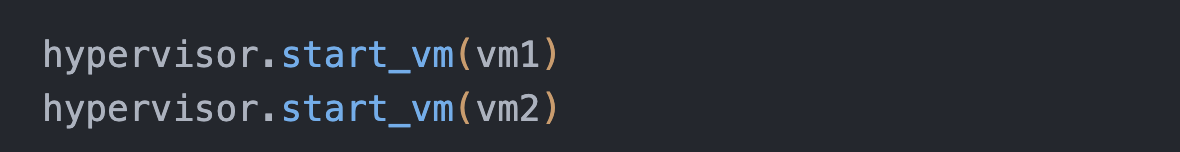
1. 初始化 Hypervisor：创建一个 Hypervisor 实例。



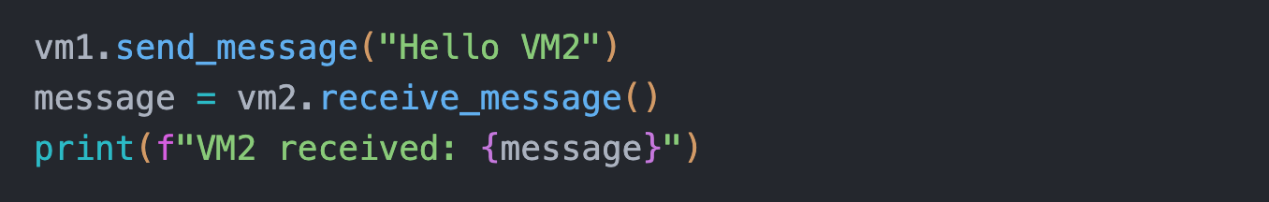
1. 创建虚拟机实例：使用 Hypervisor 创建两个虚拟机实例，分别为 vm1 和 vm2。vm1 和 vm2 被分配了不同的资源配置。



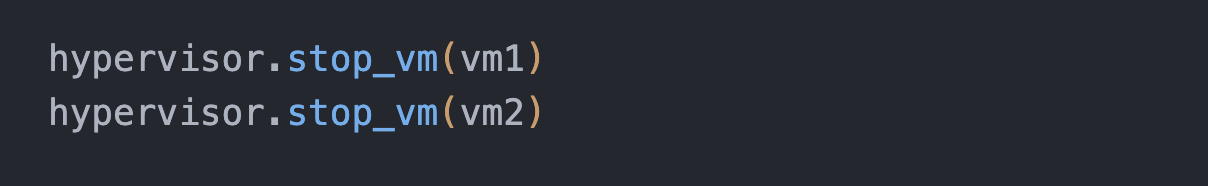
1. 启动虚拟机：启动两个虚拟机实例。



1. 模拟虚拟机间通信：vm1 发送消息给 vm2。vm2 接收来自 vm1 的消息。vm2 应该接收到来自 vm1 的 "Hello VM2" 消息，并打印出来。



1. 停止虚拟机：停止两个虚拟机实例。



七、嵌入式操作系统容错和恢复机制设计选题

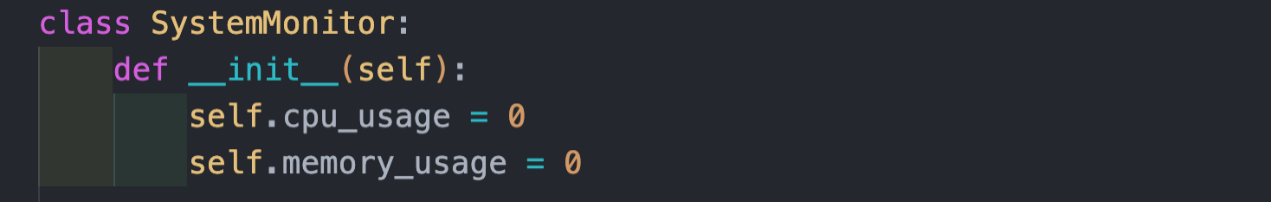
1. 功能实现具体要求

1. 故障检测机制：开发一套机制能够实时监测系统状态，如CPU使用率、内存使用情况、I/O状态等，以便于及时检测潜在的硬件故障和软件异常。并且能够基于收集的系统状态信息，应用一定的逻辑或算法来识别是否存在硬件故障或软件异常。
2. 自动故障隔离：当系统检测到故障时，自动执行一系列操作来隔离故障，防止故障影响到系统的其他部分。这可能包括关闭故障组件、重启服务、切换备用系统等措施。
3. 任务重分配策略：在检测到某任务或处理核心发生故障时，设计一套策略自动将受影响的任务迁移到其他正常的核心或处理器上继续执行。确保任务迁移后，相关的资源能够被正确分配和初始化，以便任务可以在新的环境中继续运行。
4. 状态备份与恢复：实现一个定期备份机制，定期备份任务的执行状态和关键数据，以便在发生故障时能够快速恢复。当任务因故障迁移到新的核心或处理器上时，从最近的备份点恢复任务的状态和数据，尽量减少故障对任务执行的影响。
5. 系统恢复策略：设计一个系统级的恢复策略，以确保在关键组件发生故障后，整个系统能够迅速恢复到稳定状态，恢复正常的操作和服务。

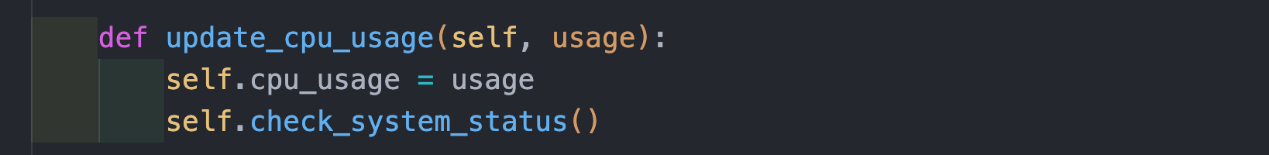
2. 实现指南

步骤1: 实现实时系统状态监测

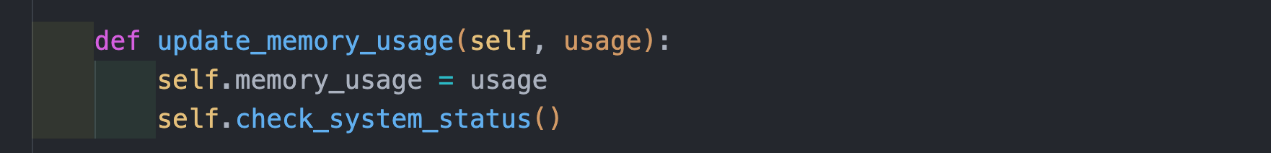
1. SystemMonitor 类\_\_init\_\_ 方法初始化系统资源使用率：在类的构造函数中初始化 CPU 和内存使用率为 0。



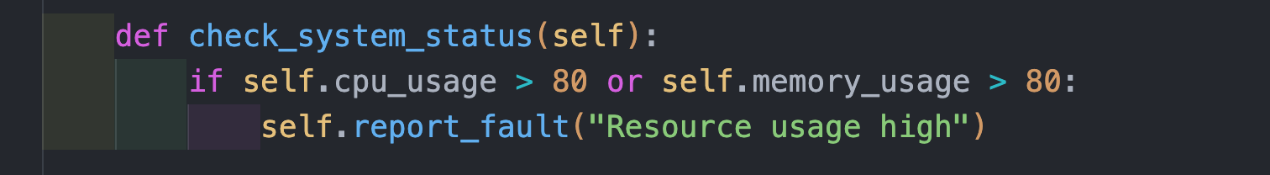
1. update\_cpu\_usage 方法更新 CPU 使用率：接收当前 CPU 使用率作为参数，并更新内部状态。更新 CPU 使用率后，调用 check\_system\_status 方法来评估系统整体健康状况。



1. update\_memory\_usage 方法更新内存使用率：接收当前内存使用率作为参数，并更新内部状态。更新内存使用率后，调用 check\_system\_status 方法来评估系统整体健康状况。



1. check\_system\_status 方法评估系统状态：判断当前的 CPU 和内存使用率是否超出正常阈值（例如超过 80%）。如果任一资源使用率过高，调用 report\_fault 方法来报告潜在的系统故障。

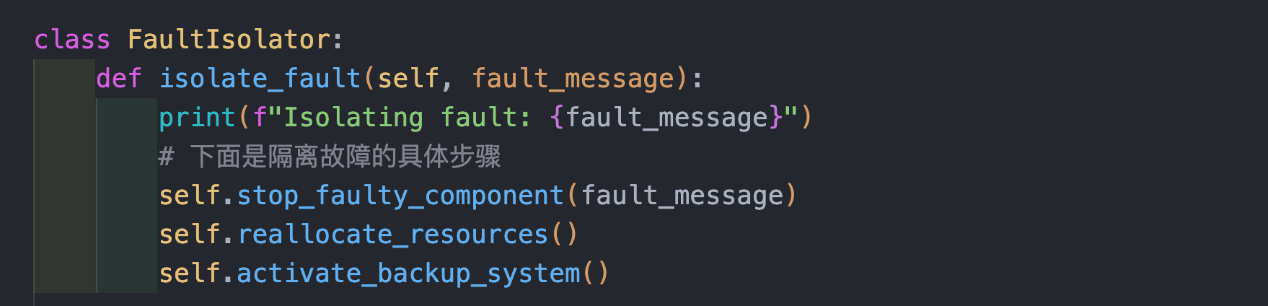


1. report\_fault 方法输出故障信息：打印出故障信息，实际系统中这里可能包括将故障信息记录到日志系统，或触发故障响应流程。



步骤2: 实现自动故障隔离机制

1. 实现FaultIsolator类中isolate\_fault 方法启动故障隔离流程。调用内部方法来停止故障组件，重新分配资源，和激活备用系统。



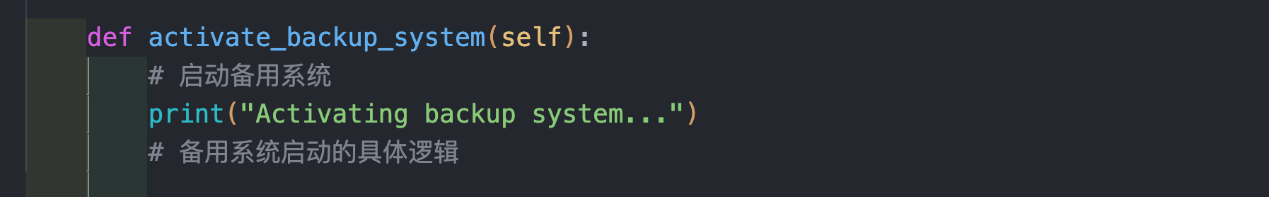
1. stop\_faulty\_component 方法停止故障组件以避免进一步的影响。模拟停止组件的操作，实际应用中可能需要根据fault\_message来确定停止哪些组件。



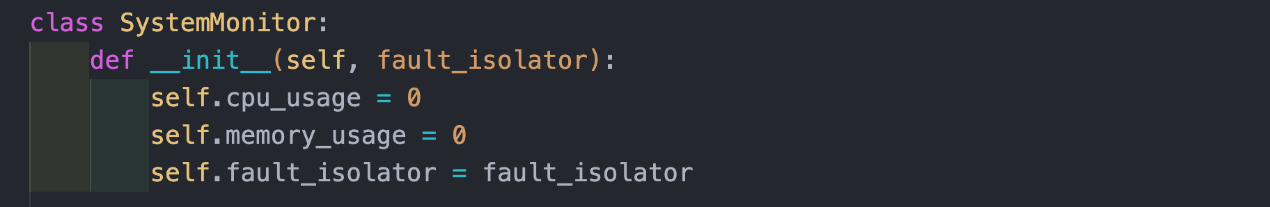
1. reallocate\_resources 方法由于故障组件已停止，需要重新分配其资源。模拟资源重新分配的操作，实际应用中需要具体策略来决定资源的重新分配。



1. activate\_backup\_system 方法如果主系统无法恢复，启动备用系统以继续提供服务。模拟激活备用系统的操作，实际应用中可能涉及复杂的切换逻辑。



1. 在SystemMonitor类中，添加对FaultIsolator的引用，并在检测到故障时调用其isolate\_fault方法。



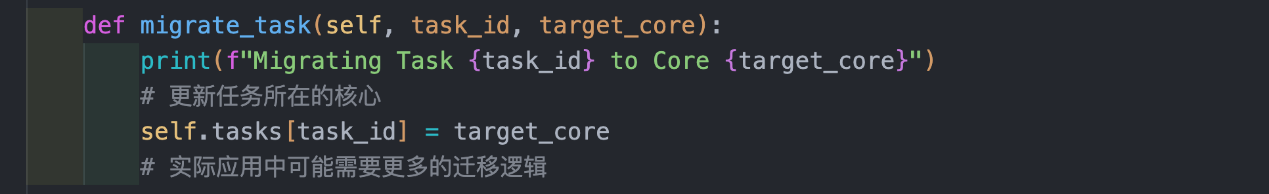


步骤3: 实现任务重分配机制

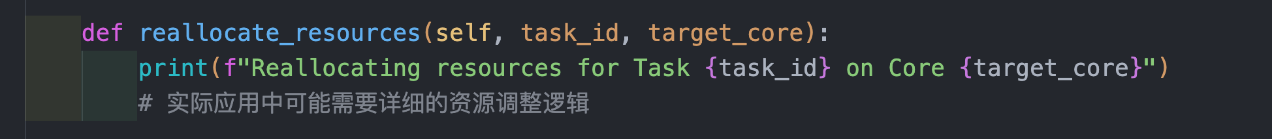
1. TaskManager类负责管理任务的分配、迁移和资源分配。其中\_\_init\_\_ 方法初始化TaskManager实例，设置一个字典来存储任务信息，其中键为任务ID，值为任务所在的核心。



1. migrate\_task 方法将特定任务迁移到指定的核心。打印迁移信息，表明哪个任务将迁移到哪个核心。更新tasks字典中的信息，将任务的核心更改为目标核心。实际应用中，此步骤可能涉及更新任务的运行环境，保存和迁移任务的状态等。



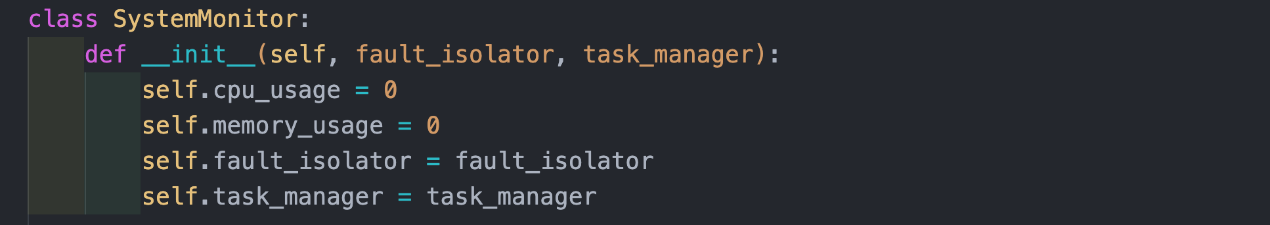
1. reallocate\_resources 方法为迁移的任务在新核心上重新分配资源。打印资源重新分配的信息。实际应用中，可能需要调整目标核心的资源分配，如分配CPU时间片、内存空间等，以适应迁移过来的任务。



1. update\_task\_status 方法更新系统中任务的状态。打印状态更新信息。实际应用中，可能需要在任务管理系统中更新任务的状态，以便其他系统组件可以获取最新状态信息。



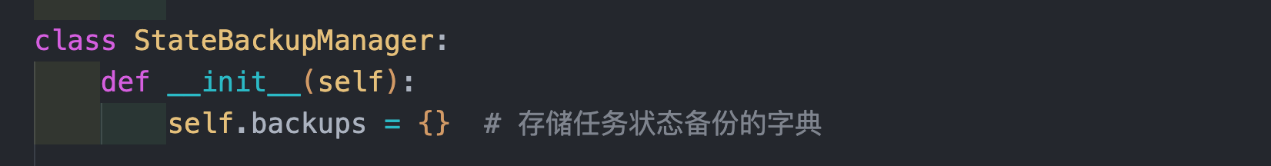
1. 在SystemMonitor类中引入TaskManager实例，以便在检测到故障时调用任务迁移和资源重新分配逻辑。增加了task\_manager作为参数，并在发现故障时调用task\_manager的migrate\_task和reallocate\_resources方法，模拟对任务的迁移和资源的重新分配。





步骤4：实现系统状态备份与恢复策略

1. StateBackupManager 类\_\_init\_\_ 方法: 初始化StateBackupManager实例，创建一个字典backups用于存储任务的状态备份。



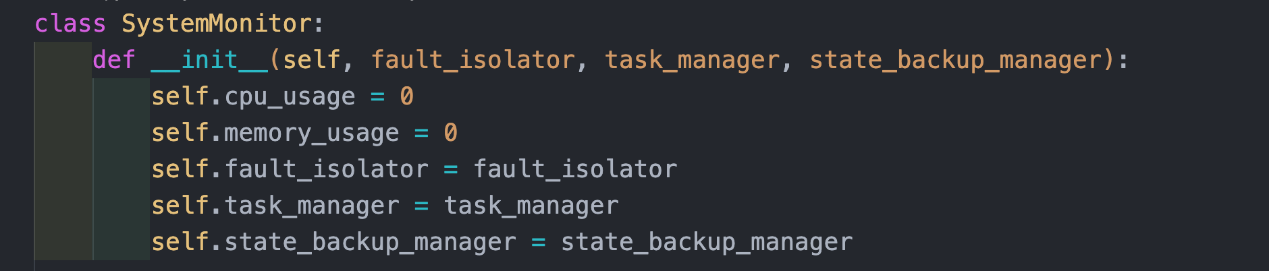
1. backup\_state 方法备份指定任务的状态。打印备份状态的信息。将任务状态存储在backups字典中，键为任务ID，值为任务状态。实际应用中，可能需要将状态序列化并存储到文件系统或数据库中以实现持久化。

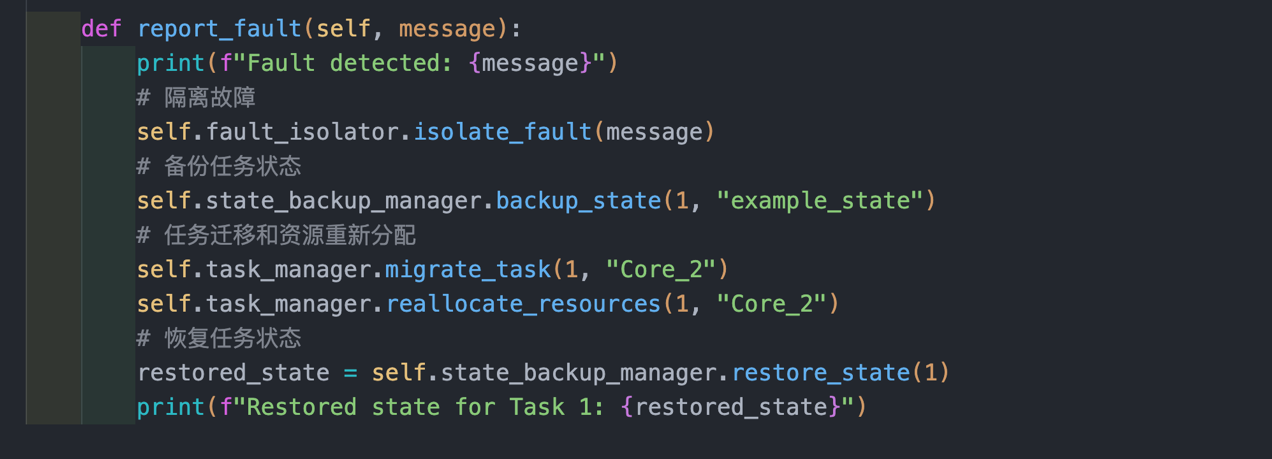


1. restore\_state 方法恢复指定任务的状态。打印恢复状态的信息。从backups字典中获取任务状态。实际应用中，如果状态存储在持久化存储中，可能需要先反序列化后再返回。



1. 在SystemMonitor中引入StateBackupManager的实例，并在适当的时刻调用其备份和恢复方法。



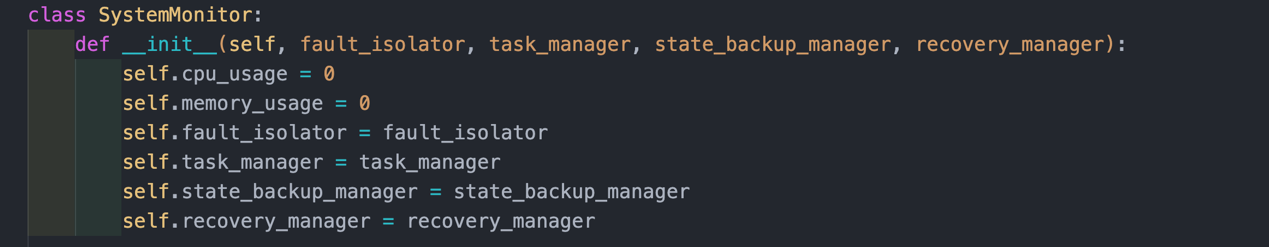


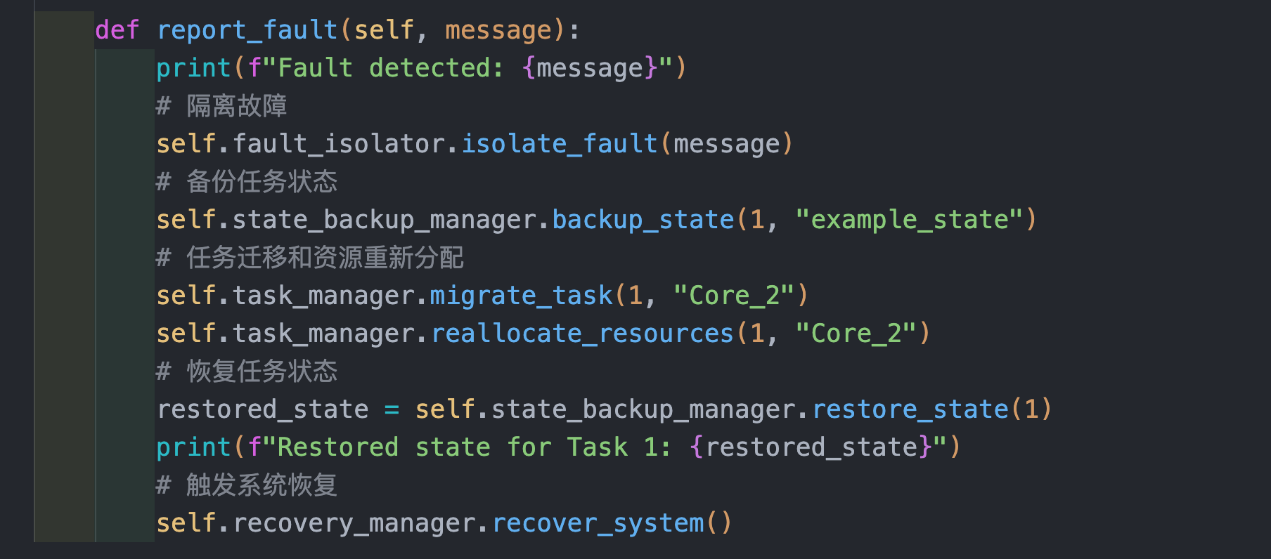
步骤5: 实现系统恢复策略

1. 定义SystemRecoveryManager类，负责系统恢复相关的操作。recover\_system方法模拟了系统恢复的过程。



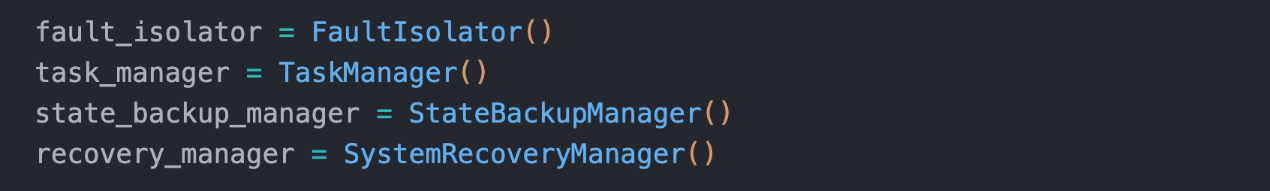
1. 将SystemRecoveryManager集成到SystemMonitor类中，以便在检测到系统故障时自动触发系统恢复流程。





步骤6: 实现简单的系统容错与恢复机制

1. 初始化，FaultIsolator负责隔离系统中检测到的故障。当系统监控发现问题时，FaultIsolator将尝试隔离故障，防止问题扩散。TaskManager管理系统中的任务，特别是在故障发生时，它负责将任务从出现故障的核心迁移到健康的核心，并重新分配必要的资源。StateBackupManager负责备份和恢复任务的状态。在故障发生前，它将任务的状态进行备份，故障恢复后，可以将这些状态恢复，以确保任务可以从故障前的状态继续执行。SystemRecoveryManager当系统发生故障需要恢复时，SystemRecoveryManager负责整个系统恢复的流程，确保系统能够尽快恢复到正常工作状态。



1. 系统监控: SystemMonitor通过update\_cpu\_usage方法模拟了CPU使用率的上升，这是一个监控系统资源使用情况的例子。



1. 故障检测: 当CPU使用率超过预定阈值（这里是80%）时，SystemMonitor的check\_system\_status方法将检测到这一情况，并判断为系统存在故障。



八、嵌入式操作系统的系统调用编译设计选题

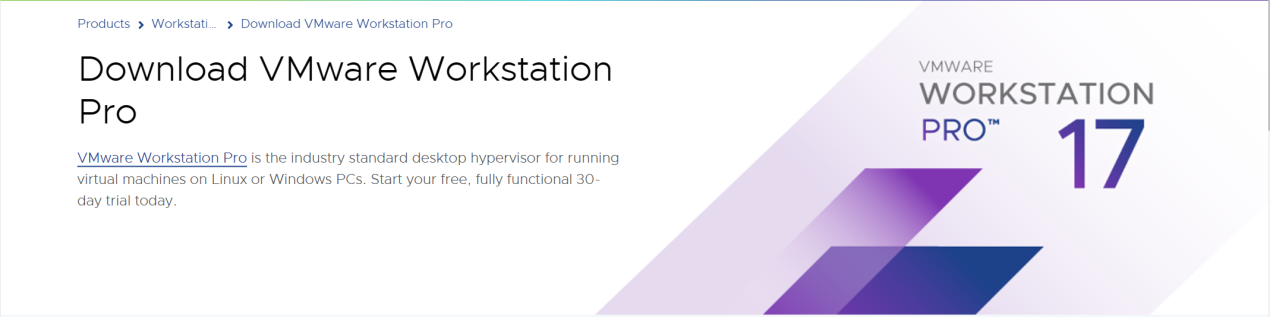
1. 功能实现具体要求

1. 安装Linux系统。
2. 构建Linux进程树：Linux进程之间的关系有父子关系和兄弟关系，显示系统中所有进程的层级关系，以树形结构展示每个进程及其父进程、子进程。
3. 编写内核模块：利用内核模块的高权限（运行在内核态），“篡改” 系统调用表，直接将系统调用函数 “注入” 到内核中，实现系统调用的添加，由于该方法不用编译内核，适合用于前期的系统调用debug阶段，快速进行系统调用程序的调试。
4. 系统调用方法的程序测试：首先，设计一个测试进程生成程序并进行编译，产生一个可以正确显示的进程树。其次，设计一个树状图生成程序并进行编译，构建一个树状图。最后，编译内核模块，可以正确显示进程树，通过树形结构清晰展示进程之间的层级关系。
5. 系统调用编译进内核：将系统调用方法编译进内核后，需要重新将新的内核映像加载到系统中。要注意，将系统调用编译进 Linux 内核是一项高级任务，修改内核源代码可能会导致一些不稳定性或者兼容性问题，因此需要谨慎对待。

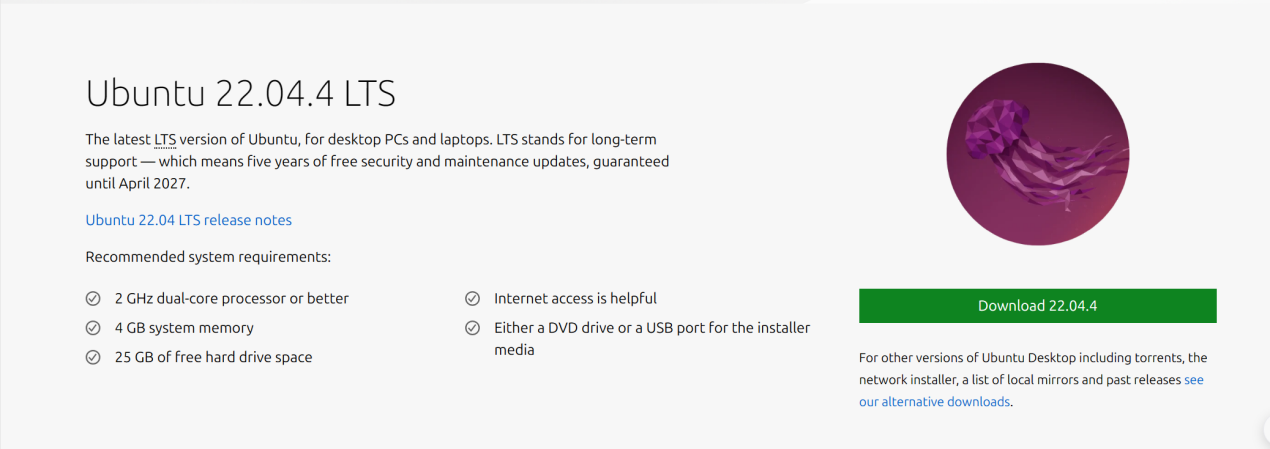
2. 实现指南

步骤1 使用VMware安装Linux系统

1. 安装VMware，官网可以提供30天的试用或者自行寻找安装包。

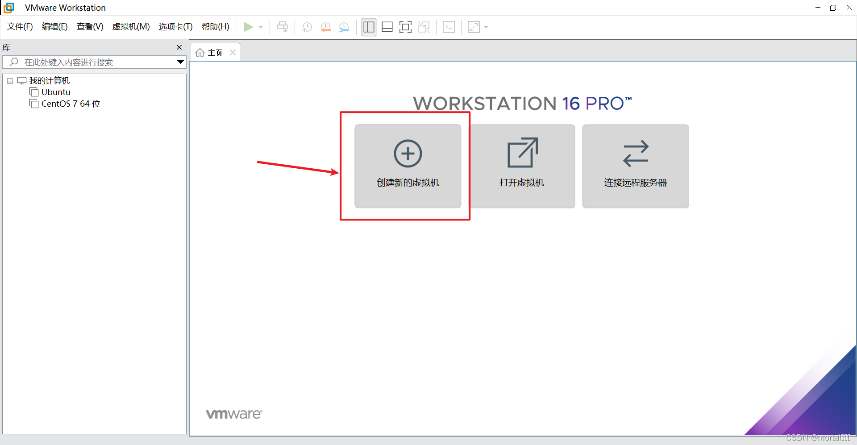


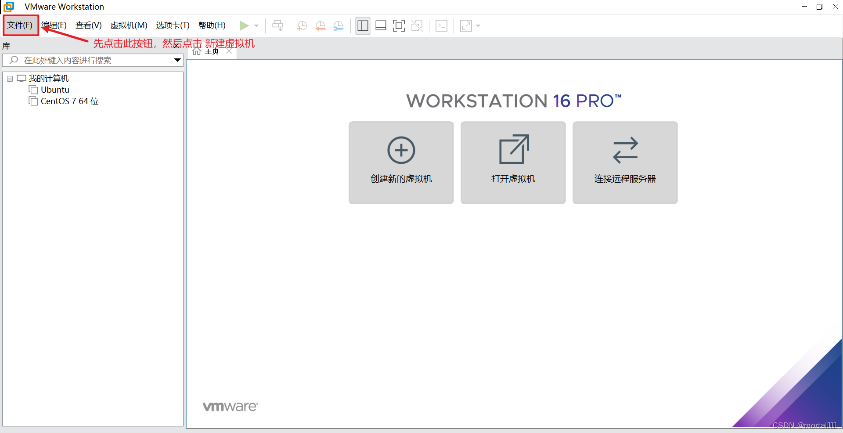
1. 下载Ubuntu镜像。



1. 创建虚拟机，安装Ubuntu。

* 打开 VMware，点击【创建新的虚拟机】。





* 选择 “自定义(高级)” , 然后点击【下一步】。



* 选择【Workstation 15.x】，然后点击【下一步】。



* 在安装过程中可以设定虚拟机名称和存放位置，然后点击【下一步】。



* 在安装过程中可以根据需求自行设定处理器配置、内存配置以及磁盘大小。
* 完成上述步骤后，虚拟机列表中会出现刚刚新建的虚拟机：

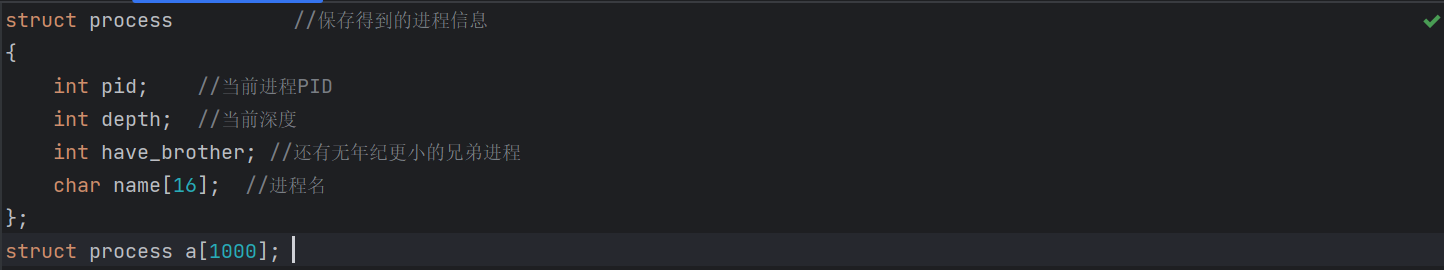


* 点击【开启此虚拟机】：



步骤2 构建Linux进程树

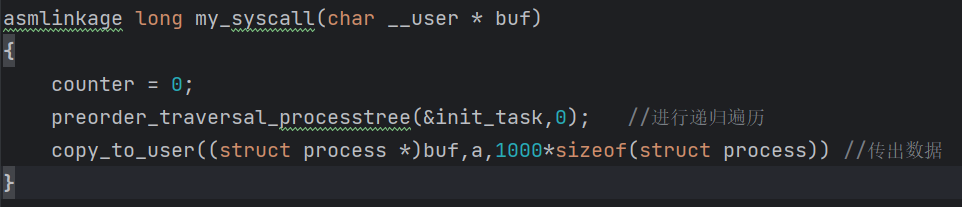
1. 进程结构定义，利用进程结构体来实现进程的各种信息记录，例如进程号、进程名、进程的兄弟关系等。



1. 进程多叉树遍历，由于系统调用的基本功能为遍历Linux进程树。因此需要采用一种合适的遍历方法，考虑到Linux进程树为一颗多叉树，所以用深度搜索中的前向遍历比较合适。preorder\_traversal\_processtree()函数采用前向遍历方法，list\_for\_each为内核封装的链表遍历函数。

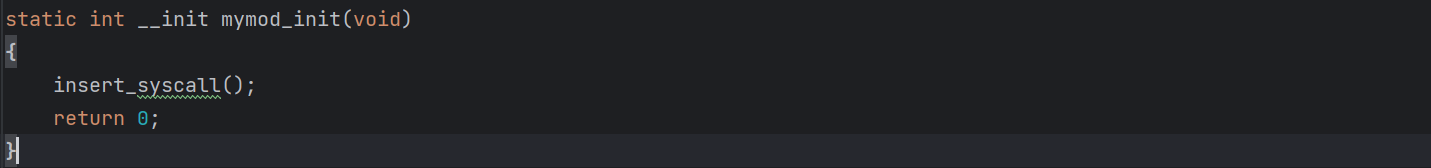


1. 系统调用主函数，该函数为系统调用的入口，即进行系统调用时，陷入内核后，根据系统调用号会跳转到这个函数执行。preorder\_traversal\_processtree()函数为进程多叉树遍历函数，其中&init\_task为根进程。​copy\_to\_user()可以理解为内核与用户之间的memcpy()，由于内核与用户之间不共享内存，所以运行于内核态的系统调用无法将获取到的进程数据直接传送给用户，只能通过copy\_to\_user()来实现。

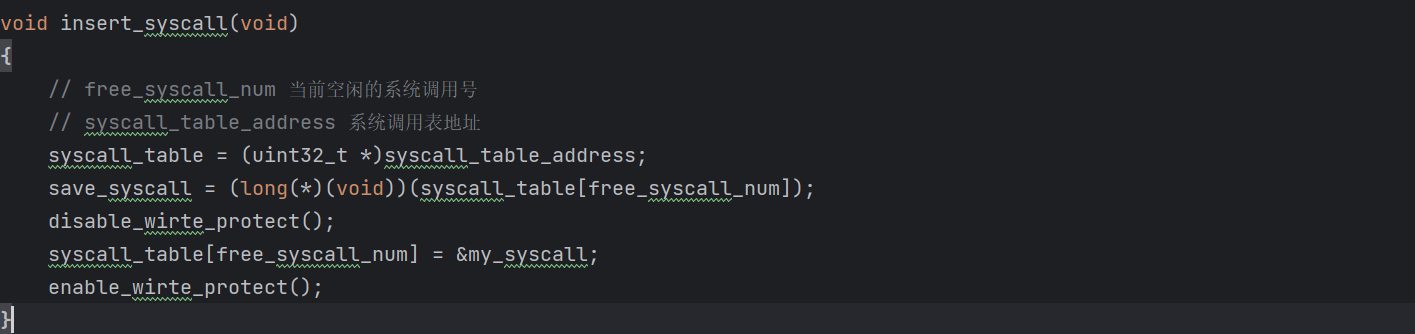


步骤3 编写内核模块

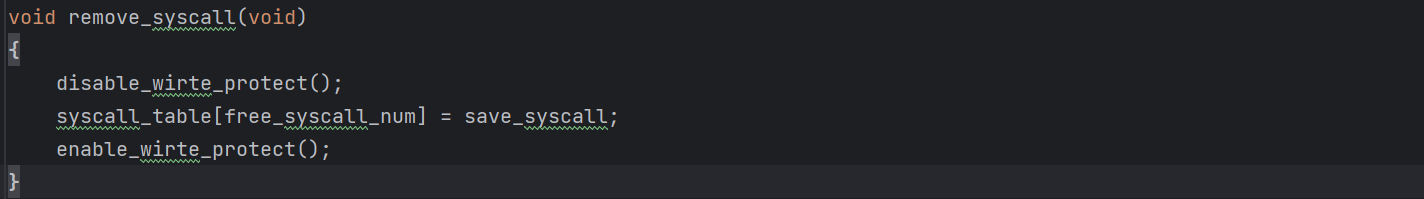
1. 内核模块初始化函数mymod\_init()，由于内核模块运行在内核态，所以其拥有极高的权限，可以利用该权限，将系统调用临时替换到系统调用表中的空位置，达到快速调试代码，不用重新编译内核的目的。



1. insert\_syscall()函数用于修改系统调用表，将系统调用表的指定位置指向自己写的系统调用函数。其中，disable\_wirte\_protect()和enable\_wirte\_protect()函数分别为关闭写保护（允许改写系统调用表）和开启写保护。



1. remove\_syscall()函数用于移除系统调用的过程。



步骤4 系统调用方法的程序测试

1. 树状图构建程序编写，构建出一个树状图。利用系统调用传回的数组，构建出Linux进程树图，表现出进程之间的父子兄弟关系。
2. 测试进程构建程序编写，运行该程序可以产生一个设计好的进程树。
3. 内核模块构建程序编写，内核模块的构建程序形式较为固定。



1. 程序编译并运行。

编译测试进程生成程序(setprocess.c)



编译树状图生成程序(mymod\_test.c)



编译内核模块(系统调用)程序(mymod.c)，虽然没有显式地定义并生成mymod.o，但是make有强大的自动推导功能，它可以自动推导出所需依赖



步骤5 系统调用编译进内核

1. 下载对应的Linux内核并解压。
2. 添加系统调用号。



1. 添加系统调用函数声明(不能放在宏定义里)。



1. 编译内核并安装。

