Operating System Assignment 3作业讲解

用词约定

关键词 = 搜索词 = search pattern

各类Class说明：内容枯燥但重要

Node 类是一个代表share list链表节点的基本结构。每个节点在这里不仅仅是用于基本的链表结构，还增加了一些属性来满足特定的需求，例如与书籍数据和搜索模式有关的需求。以下是关于这个类的详细解释：

init(self, data, book_title=None):

- 初始化方法用于设置节点的基本属性。
- data 参数表示节点的内容，是一段文本数据。
- book_title 参数是一个可选参数，表示与当前节点相关联的书籍标题。

属性attributes:

- data: 存储在节点中的实际数据。
- next: 通常用于指向链表中的下一个节点。
- book_next: 在同一本书的内容中指向下一个节点。这可能是为了将多个节点（可能是多个数据块）从同一本书组合在一起。
- next_frequent_search: 指向最近的下一个包含关键词的节点。book_title: 存储与此节点相关联的书籍标题。
- pattern_count: 用于统计在节点的数据中特定模式出现的次数。
- index: 为每个节点分配一个唯一的索引值。这有助于跟踪和引用特定的节点。

contains_pattern(self, pattern):

- 这是一个实例方法，用于检查当前节点的数据中是否包含给定的模式。
- 它接受一个pattern参数，并在节点的数据中搜索这个模式。
- 如果模式存在，则返回True，否则返回False。

总体逻辑：
Node类表示一个链表节点，并增加了与书籍和搜索模式有关的属性和方法。
除了基本的链表功能外，这个结构还为特定的应用场景（如频繁搜索或与书籍数据相关的操作）提供了支持。

SharedList是一个线程安全的链表结构。这个结构设计用于存储多个节点，并且在多线程环境中能够安全地操作这些节点。下面详细解释这个类的每一部分：

init():

- 初始化函数为链表设置了一些基本属性，如头节点、尾节点等。
- 为链表添加了一个锁，这样在多线程操作时能够确保数据的完整性和一致性。
- 初始化了一些指针和索引，这些将用于后续的操作。

append(node):

- 向链表追加新节点。
- 使用lock确保这个操作是线程安全的。
- 每一个新节点都会被分配一个唯一的索引。
- 新节点会被追加到链表的尾部，或者如果链表是空的，那么它会成为头节点。

get_next_unprocessed_node():
- 返回链表中的下一个未处理的节点。
- 使用lock确保这个操作是线程安全的。
- 遍历链表查找一个未被处理的节点。
- 标记找到的节点为已处理，并更新最后处理的节点的指针。
- 如果所有节点都已经被处理，那么返回None。

update_next_frequent_search(current_node_with_pattern):
- 这个函数是用于更新包含特定模式的节点之间的连接。
- 使用lock确保这个操作是线程安全的。
- 如果链表之前已经存储了一个包含特定模式的节点，那么这个函数会决定是否应该更新这个节点，使其指向当前节点作为下一个频繁搜索的节点。
- 最后，更新最后一个包含特定模式的节点的指针。

总体逻辑：
- SharedList是一个链表结构，设计用于在多线程环境中存储和操作数据。
- 使用内置的锁可以确保每个操作都是线程安全的，这意味着即使多个线程同时访问和修改链表，数据也不会出现不一致的情况。
- 这个结构提供了方法来追加新节点、查找未处理的节点以及更新包含特定模式的节点的连接关系。

ClientHandler类主要是为了处理从客户端接收的数据。其设计用于非阻塞方式读取来自客户端的数据，并在数据接收完毕后保存书籍。接下来是对每一部分的详细解释：

init():
- 初始化函数设置了多个属性，例如与客户端的连接对象，共享数据列表，书籍内容的起始和末尾节点等。
- 使用selectors模块进行I/O多路复用，这样可以在单线程中处理多个I/O事件。
- 将连接设置为非阻塞，并为读事件注册回调函数。

read():
- 试图从客户端接收数据。
- 如果数据存在，则更新最后接收数据的时间，解码数据，并创建一个新节点来存储该数据。
- 根据接收到的数据更新书的内容的头和尾节点。
- 将新创建的节点添加到共享列表。
- 如果没有数据或发生错误，则注销选择器，关闭连接，并保存书籍。

save_book():
- 如果保存书籍的文件夹不存在，则创建它。
- 定义要保存的文件名并打开文件进行写入。
- 从书籍的头节点开始，运用book_next指针，遍历每个节点并将其数据写入文件。

run():
- 持续地使用选择器等待事件，超时时间设为1秒。
- 如果发生事件，则执行相应的回调函数。
- 如果超过5秒没有接收到任何数据，则输出信息，注销选择器，关闭连接，并保存书籍。

总体逻辑:

- ClientHandler对象在其生命周期中与单个客户端关联。
- 它使用selectors来异步地从该客户端读取数据，并在每次接收数据后更新内部的书籍内容。
- 当5秒内没有接收到新数据或客户端关闭连接时，ClientHandler将保存其接收到的书籍内容并结束其运行。
- 这种设计允许服务器以非阻塞方式处理多个客户端，每个客户端都在其自己的ClientHandler实例中被处理。

AnalysisThread 类是一个用于在独立线程中分析数据的工具。它继承自 threading.Thread 类，这意味着它可以作为一个独立的线程运行。该类的目标是并行地在链表中搜索给定的模式，并统计该模式出现的次数。以下是该类的详细解释:

init(self, shared_list, pattern, thread_id):

- shared_list: 共享的链表对象，多个线程会同时对其进行操作。
- pattern: 要在数据中搜索的模式或关键字。
- thread_id: 此线程的唯一标识符。

run(self):

- 重写了threading.Thread类的run方法，确定线程的主要行为。
- 在无限循环中，线程尝试获取链表中的下一个未处理的节点。
- 如果没有未处理的节点，线程会休眠2秒后再次尝试。
- 如果找到一个节点，线程会检查该节点是否包含所需的模式。如果是，线程会更新相关的统计数据，并尝试输出结果。

attempt_to_output_results(self):

- 此方法尝试输出分析结果，但只有当上次输出时间超过5秒时才实际执行输出。

output_results(self):

- 输出模式匹配计数的结果。
- 它首先获取所有书籍标题及其对应的模式匹配计数，并按匹配计数进行排序。
- 接着，它将结果打印到控制台。

get_sorted_books_by_pattern_count(self):

- 从共享链表中收集每本书的模式匹配计数。
- 使用一个字典来存储每本书的标题及其对应的匹配计数。
- 最后，它返回按匹配计数降序排序的书籍。

注意事项:
在多线程环境中，为确保数据的线程安全，代码在进行关键操作时多次使用了锁。这是为了防止多个线程同时修改共享资源，导致数据不一致或其他未定义的行为。

总结:
AnalysisThread 类的主要目的是在独立线程中搜索和分析数据，查找特定的模式，并统计该模式的出现次数。此外，该线程还会定期输出其分析结果。

AnalysisThreadHandler 类是一个用于管理和控制数据分析线程的工具。当有一个共享的链表和一个要搜索的模式时，这个类可以根据给定的线程数量创建多个线程来并行地分析数据。以下是关于这个类的详细解释:

类级别变量:

- last_output_time: 用于记录上次输出时间的静态或类级别变量。这可以用于确定何时执行某些任务或输出。
- output_lock: 是一个线程锁，确保在多线程环境中对输出进行线程安全的操作。

init(self, shared_list, pattern, thread_count):
- 初始化方法用于设置类的属性。
- shared_list 参数是一个共享的链表对象，所有线程都会对其进行操作。
- pattern 参数是要在数据中搜索的模式或关键字。
- thread_count 参数指定要创建的线程的数量。

属性:
- shared_list: 指向传入的共享链表对象的引用。
- pattern: 要搜索的模式或关键字。
- thread_count: 要创建的线程数量。
- threads: 一个空列表，用于存储所有创建的线程对象。

start_threads(self):
- 此方法负责创建和启动指定数量的分析线程。
- 使用一个循环，对于每一个要创建的线程，它都会创建一个AnalysisThread对象（在此解释中未给出这个类的详细内容）。
- 线程被启动后，它会添加到threads列表中。

join_threads(self):
- 此方法负责等待所有已启动的线程完成。
- 使用一个循环，对于threads列表中的每一个线程，它都会调用线程的join方法。这确保主线程（或调用此方法的线程）将等待每一个分析线程完成其任务。

总体逻辑：
AnalysisThreadHandler 类提供了一个方便的方式来创建、启动和管理一组用于数据分析的线程。
它的主要目的是并行地搜索共享链表中的某个模式，并确保在多线程环境中的操作是线程安全的。

EchoServer 类定义:
初始化函数 (__init__):
输入参数：
- host: 服务器要绑定的IP地址。
- port: 服务器要绑定的端口号。
- shared_list: 一个共享列表对象，可能是一种线程安全的数据结构，用于在多个线程之间共享数据。
- 定义一个新的TCP套接字。
- 将套接字绑定到给定的主机和端口。
- 开始监听连接，队列大小为5（这意味着一次可以有5个等待的连接）。
- 将shared_list参数保存为类的一个属性。
- 定义一个book_counter属性，并初始化为1。这可能是为每个客户端连接提供唯一标识的方式。

accept_client():
- 该函数没有参数。
- 使用accept方法接受来自客户端的连接。
- 打印出连接的客户端地址。
- 创建一个新的ClientHandler对象，该对象接受客户端连接、共享列表和当前的书计数器作为参数。
- 启动一个新的线程来处理客户端的请求。

- 递增书计数器。

run():
- 该函数没有参数。
- 打印出一个消息，表示服务器正在监听。
- 在无限循环中，持续接受客户端的连接。

总结：
这是一个简单的服务器类，其核心目的是接受来自客户端的连接并为每个连接启动一个新的处理线程。从代码中可以看出，它可能与某种"书"相关的应用程序有关（由book_counter推断）。每次接受一个新的客户端连接时，它会递增书计数器。

Part1

代码中定义了三个核心类：Node、SharedList和ClientHandler。下面是这些类及其之间关系的概述：

Node 类：
此类定义了一个数据节点，它具有与数据、其他节点和搜索模式相关的属性。
contains_pattern 方法用于检查该节点的数据是否包含搜索词。

SharedList 类：
这是一个共享列表，设计为线程安全的，因此多个线程可以同时访问和修改它。
共享列表中有一些方法，如 append（用于添加新节点），get_next_unprocessed_node（用于获取下一个未处理的节点），和 update_next_frequent_search（用于更新下一个包含搜索模式的节点的引用）。

ClientHandler 类：
此类负责处理从客户端接收的数据。它使用selectors模块实现非阻塞I/O。
当收到新数据时，数据被解码并包装在一个Node实例中，并添加到SharedList实例。
客户端发送的数据分块表示一本书的一部分，每次接收到数据块时，都会将其添加到共享列表并更新书籍的头和尾。
如果客户端连接超时或断开，该类将保存接收的书籍数据。

EchoServer 类**:**
用于监听和接受客户端的连接请求。
对于每个接受的连接，它会创建一个新的ClientHandler实例，并在一个新的线程中启动该处理程序。

整体流程：
流程概述：
1. 服务器启动并开始监听，等待客户端的连接。
2. 当客户端成功建立连接，服务器为该连接启动一个新的线程进行处理。
3. 在这个新线程中，客户端处理程序（ClientHandler类）从客户端接收数据块。
4. 每接收一个数据块，ClientHandler就将其包装成一个节点，并逐渐构建一个表示书籍的链表。

5. 同时，这些新创建的节点也被并发地添加到SharedList中。由于可能存在多个线程同时操作此列表，因此它被设计为线程安全。
6. 数据块继续从客户端接收，每个数据块被视为书籍的一个部分。
7. 当客户端处理程序检测到来自客户端的连接断开或在指定时间内未接收到新的数据时，它会认为整本书已完整地被接收。
8. 最后，这整本书的内容被保存到磁盘上。

应用场景：从代码来看，创建了一个多线程服务器，该服务器从多个客户端并行接收书籍数据。这些数据首先被存储在链表中，然后保存到磁盘上。

特点：

- 使用selectors模块进行非阻塞I/O，允许服务器在等待数据时继续执行其他任务。
- 使用线程锁确保多线程访问的线程安全。

Part 2:

在part 1的基础上，新增2个class。

AnalysisThread Class:

- 目的: 在多线程环境中，为数据节点（例如书籍）执行搜索和分析。
- init(): 初始化函数设定共享链表、搜索模式和线程ID。
- run(): 当线程开始时，此方法会持续地搜索共享链表中未处理的节点。对于找到的每个节点，该方法会检查节点（例如，书的部分）是否包含指定的模式或关键词，并相应地更新计数。
- attempt_to_output_results(): 尝试输出结果，但在多线程环境中确保每5秒只有一个线程可以输出。
- output_results(): 输出当前线程处理的模式的出现次数最多的书籍。
- get_sorted_books_by_pattern_count(): 搜索整个共享链表，汇总每本书中模式的出现次数，并按次数进行排序。

AnalysisThreadHandler Class:

- 目的: 管理分析线程，例如创建、启动和联接线程。
- 类级变量: last_output_time用于跟踪上一次的输出时间，而output_lock用于确保线程安全的输出。
- init(): 初始化函数设定共享链表、搜索模式和线程数量。
- start_threads(): 创建指定数量的线程并启动它们。
- join_threads(): 等待所有的分析线程完成。

总体逻辑:

- EchoServer接受来自客户端的连接并创建新的客户端处理线程，这些线程将数据收集到共享链表中。
- AnalysisThreadHandler管理一组AnalysisThread，它们不断地从共享链表中查找数据，并分析这些数据是否包含给定的模式或关键字。当找到匹配的模式时，它们会更新节点的模式计数。这些线程还有责任周期性地输出其找到的匹配模式的数量。

- 在这种设计中，共享链share list表扮演了生产者（客户端处理线程）和消费者（分析线程）之间的中介角色，生产者添加新的数据，而消费者对这些数据进行处理和分析。

补充问题：

问题：在consumer同步的情况下，如何保证next_frequent_search正确指向最近的下一个包含搜索词的节点？即：演示update_next_frequent_search()这个函数是如何工作的。

假设我们有以下节点链表：

node1 -> node2 -> node3 -> node4 -> node5

其中，每个节点都有一个index属性，从1开始递增：

node1(index: 1)
node2(index: 2)
node3(index: 3)
node4(index: 4)
node5(index: 5)

现在，假设这些节点中只有node2, node4, 和 node5 包含搜索词。

假设 node4 先被一个分析线程检测：
last_node_with_pattern 尚未定义。
设置 last_node_with_pattern = node4

假设紧接着 node5 被另一个分析线程检测：
last_node_with_pattern = node4
node4.next_frequent_search 尚未定义，因此它会被设置为 node5。
更新 last_node_with_pattern = node5

最后，node2 被另一个线程检测：
last_node_with_pattern = node5
node5.next_frequent_search 尚未定义。但是我们需要判断当前节点（node2）与 last_node_with_pattern（node5）之间的索引关系。由于 node2 的索引小于 node5，我们不会更新node5.next_frequent_search。
更新 last_node_with_pattern = node2。但是我们知道node2后面的node4和node5都包含搜索词，并且node4比node5更接近node2。因此，我们将 node2.next_frequent_search 设置为 node4。

最后的结果是：
node2.next_frequent_search -> node4
node4.next_frequent_search -> node5
这样，每个包含搜索词的节点的next_frequent_search都指向了最近的下一个包含搜索词的节点。符合我们的要求。

怎样运行程序？

例：server端：Python part1_and_2.py -l 12345 -p "the"
client端：nc localhost 12345 -i 1 <big_text_file.txt