**操作系统课程项目文档**

**内存管理：请求调页**



实验名称 内存管理项目：请求调页

姓名 刘震

学号 2352471

学院 计算机科学与技术学院

专业 软件工程

任课教师 张惠娟

日期 2025.5.11

1. **项目概览**
   1. **项目简介**

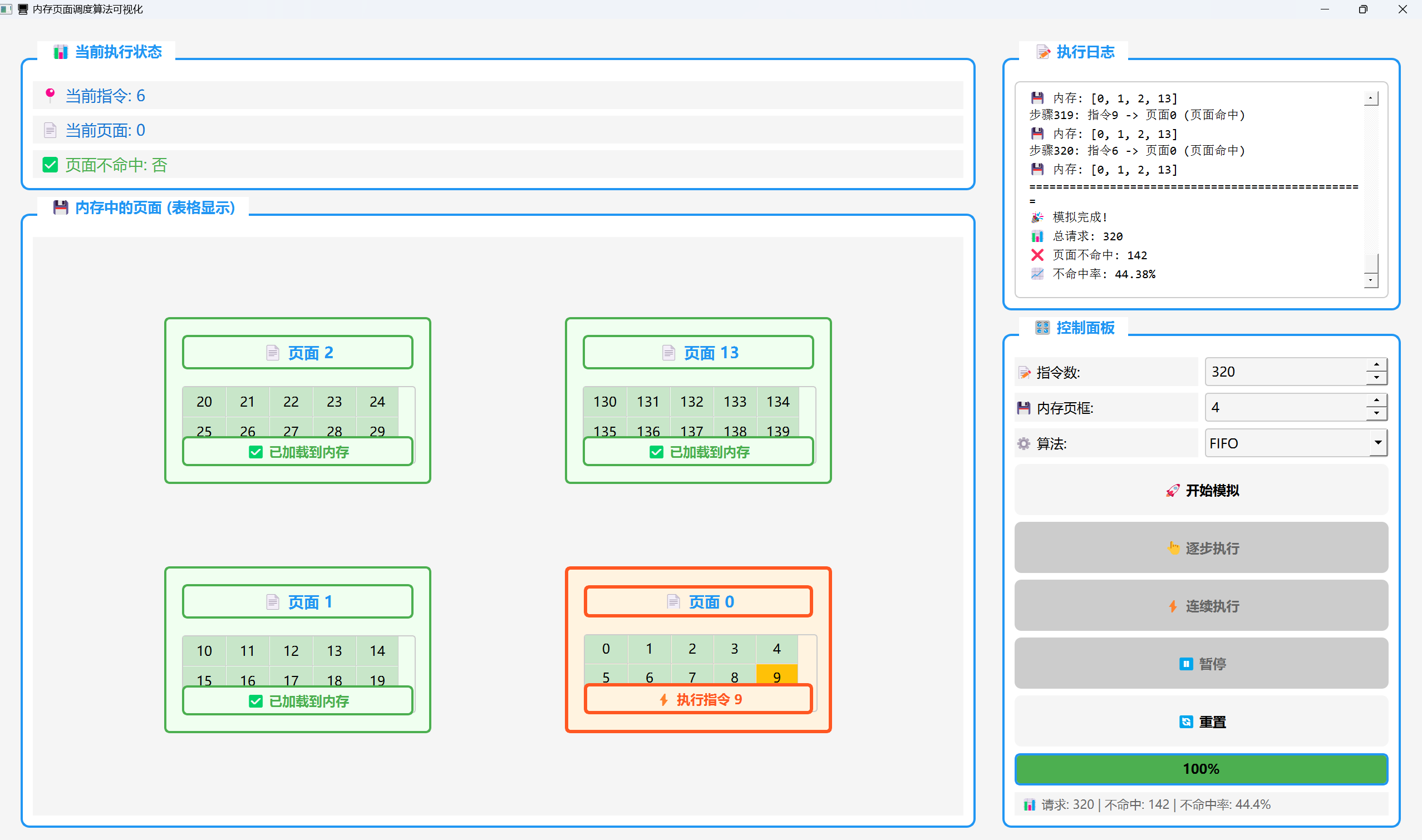
本项目是一个基于请求分页存储管理的内存调度模拟系统，目标是实现操作系统中页面切换的基本原理。

系统默认为一个作业包含320条指令，每10条为一页，共分成32页，物理内存仅有4个页框。当指令所在页未调入内存时，就会发生缺页，如果内存已满，就需要执行页面置换操作。

指令序列依据后续性和分布性生成，展现运行进程中指令的常规性。页面置换支持FIFO和LRU算法，FIFO根据先进先出原则选择替换页，LRU根据最久未使用原则进行切换。项目提供了可视化界面，用于展示当前指令、页面加载情况和内存页框内容，并支持手动或自动执行操作。执行过程中将记录指令执行情况和缺页次数，最终给出统计结果，包括不命中率（缺页率）。

项目通过效果显示和简洁的代码实现，有效地实践了分页存储管理的核心内容。

* 1. **界面概览**

****

上面的图片展示了本次内存管理项目的 UI 设计，可以通过按钮调整使用的置换算法、指令数量，同时在右上角可以清除的看见指令的执行情况，以及调页的情况；最后可以在执行日志上看见缺页率（不命中率的数值）。

1. **类设计说明**

本项目代码包含多个 Python 文件，分别定义了不同的类来实现内存分页调度算法的可视化和模拟。下面将对主要文件中的核心类进行详细描述。

* 1. **指令序列的生成：allocation.py 文件**

此文件主要负责生成指令序列以及管理整个模拟过程的调度。

* + 1. **Allocation类**

 Allocation 类用于生成一个模拟的指令执行序列，并管理当前正在执行的指令。它模拟了程序在运行过程中访问不同逻辑地址（在此简化为指令序号）的模式。这个类首先根据用户定义的总指令数和请求的指令序列长度，生成一个具有一定随机性和局部性特征的指令流。然后，它提供方法来获取当前指令、下一条指令，并向前推进指令序列。

函数设计和作用如下：

|  |  |
| --- | --- |
| **函数名** | **作用描述** |
| \_\_init\_\_(self, order\_nums, request\_order\_nums) | 初始化类实例。设置总指令数 (order\_nums) 和请求的指令序列长度 (request\_order\_nums)，调用 gen\_seq 生成指令序列，并将当前指令索引 cur\_index 初始化为0。 |
| gen\_seq(self) | 生成指令执行序列。序列的生成具有一定的模式：从一个随机的起始指令开始，然后是它的顺序后继；接着可能跳转到该起始指令之前的某个随机位置及其后继，再跳转到之前的某个随机位置之后的某个随机位置及其后继。这个过程会重复直到达到请求的指令序列长度。 |
| cur\_order(self) | 获取并返回当前指令序列中由 cur\_index 指向的指令的序号。 |
| next\_order(self) | 获取当前指令的下一条指令的序号，但不移动 cur\_index 指针。如果当前指令已经是序列的最后一条，则返回 None。 |
| go\_next(self) | 将当前指令索引 cur\_index向前移动一位，指向序列中的下一条指令。如果已到达序列末尾，则不进行任何操作。 |

**2.1.2 枚举 State**

State 枚举定义了内存调度模拟过程可能处于的几种不同状态。它用于 MemoryDispatch 类中跟踪和管理模拟的生命周期。

枚举值：

* NOT\_STRAT: 表示模拟尚未开始。
* IN\_PROGRESS: 表示模拟正在进行中。
* FINISHED: 表示模拟已经完成。



* + 1. **MemoryDispatch 类**

MemoryDispatch 类是内存调度模拟的主控制器。它整合了指令生成（通过 Allocation 类）和页面调度逻辑（通过 Dispatcher 类）。该类负责启动、执行（获取下一条指令并请求相应页面）、以及结束整个模拟过程，并维护模拟的当前状态。

函数设计和作用如下：

|  |  |
| --- | --- |
| **函数名** | **作用描述** |
| \_\_init\_\_(self) | 初始化模拟控制器。设置初始状态为 State.NOT\_STRAT，创建一个 Dispatcher 实例（默认4个页框）用于页面调度，以及一个 Allocation 实例（默认总指令320条，请求320条）用于指令序列管理。 |
| start(self) | 启动模拟过程，将当前状态设置为 State.IN\_PROGRESS。 |
| next\_order(self) | 如果模拟正在进行 (State.IN\_PROGRESS)，则获取 Allocation 实例中的当前指令。同时，它会检查是否所有指令都已执行完毕，如果是，则调用 finish 方法结束模拟。 |
| finish(self) | 结束模拟过程，将当前状态设置为 State.FINISHED。 |
| get\_page(self) | 处理当前的页面请求。它从 allocation 对象获取当前指令的序号，通过整除10计算出该指令所在的页面号，然后调用 dispatcher 对象的 accept\_request 方法来请求这个页面。 |

* 1. **指令的分页与置换：dispatch.py 文件**

此文件定义了内存页面的数据结构以及核心的页面调度算法实现。

**2.2.1 数据类 Page**

Page 是一个数据类 (dataclass)，用于表示内存中的一个页面。它封装了与页面相关的信息，这些信息对于实现不同的页面替换算法（如 FIFO 和 LRU）至关重要。

属性说明**：**

|  |  |  |
| --- | --- | --- |
| **属性名** | **类型** | **作用描述** |
| index | int | 页面的唯一标识符（页号）。 |
| order | int | 用于 FIFO (先进先出) 算法。表示页面进入内存的顺序，数值越大通常表示越早进入。 |
| priority | int | 用于 LRU (最近最少使用) 算法。表示页面的使用近况，数值越小表示最近被使用（优先级高）。 |

**2.2.2 类 Dispatcher**

Dispatcher 类是内存页面调度逻辑的核心实现。它模拟操作系统内存管理器如何处理页面访问请求，包括处理页错误（Page Fault）以及根据所选算法（FIFO 或 LRU）执行页面替换。该类维护内存中当前页面的状态、总页框数、请求次数、缺页次数等统计信息。

函数设计和作用：

|  |  |
| --- | --- |
| **函数名** | **作用描述** |
| \_\_init\_\_(self, sum\_page\_number, dispatch\_method="FIFO") | 初始化调度器。设置内存的总页框数 (sum\_page\_number) 和页面替换算法 (dispatch\_method，默认为 "FIFO")。同时初始化用于存储当前占用页面的字典 \_occupy\_page、已占用页框数 \_occupy\_page\_num、总请求次数 \_request\_times 和缺页次数 \_fault\_times。会校验参数的有效性。 |
| \_update\_occupy\_page(self, request\_index) | 内部辅助方法，用于在一次页面访问后更新内存中所有已存在页面的 order 和 priority 属性。对于 LRU，除了当前访问的页面，其他页面的 priority 会增加（表示更久未被访问）。对于 FIFO，其他页面的 order 会增加（表示在内存中停留时间更长）。 |
| accept\_request(self, request\_index) | 处理一个页面访问请求。首先增加总请求次数。如果请求的页面已在内存中（命中），则对于 LRU 算法，会更新该页面的 priority 为0（表示最新访问）。如果页面不在内存中（缺页），则增加缺页次数。若内存未满，则直接调入新页面；若内存已满，则根据选定的替换算法 (dispatch\_FIFO 或 dispatch\_LRU) 替换一个旧页面。调入新页面后，会调用 \_update\_occupy\_page。 |
| dispatch\_FIFO(self, request\_index) | 实现 FIFO 页面替换算法。它会找到当前内存中 order 值最大的页面（即最早进入内存的页面），将其移出，然后将新的 request\_index 对应的页面调入，并设置其 order 和 priority 为0。之后调用 \_update\_occupy\_page。 |
| dispatch\_LRU(self, request\_index) | 实现 LRU 页面替换算法。它会找到当前内存中 priority 值最大的页面（即最近最少使用的页面），将其移出，然后将新的 request\_index 对应的页面调入，并设置其 order 和 priority 为0。之后调用 \_update\_occupy\_page。 |

* 1. **UI设计：ui.py 文件**

此文件包含了构建图形用户界面 (GUI) 的所有类，用于可视化内存分页调度的过程。

**2.3.1 类 MemoryPageWidget (继承自 QFrame)**

MemoryPageWidget 是一个自定义的 Qt 组件，用于在 GUI 中可视化地表示内存中的一个物理页框。它可以显示该页框当前装载的逻辑页面号、该页面包含的指令范围（以表格形式），以及页面的状态（如“已加载”、“空闲”、“正在执行指令”等）。通过不同的样式和文本提示，用户可以直观地了解每个页框的情况。

函数设计和作用：

|  |  |
| --- | --- |
| **函数名** | **作用描述** |
| \_\_init\_\_(self, page\_id) | 初始化页框组件。设置其固定的尺寸和基础样式，并调用 setup\_ui 来构建内部的 UI 元素。page\_id 参数用于初始化时可能传入的页面号。 |
| setup\_ui(self) | 构建并布局组件内部的 UI 元素，包括显示页面号的标题标签 (title\_label)、一个 2x5 的 QTableWidget 用于显示10条指令的序号，以及一个状态标签 (status\_label)。 |
| set\_page\_content(self, page\_id) | 更新组件以显示指定 page\_id 的内容。它会更新标题标签的文本为新的页面号，并填充指令表格，显示从 page\_id \* 10 到 page\_id \* 10 + 9 的指令号。 |
| set\_empty(self) | 将组件设置为表示一个空闲（未使用）的页框。这会更新标题和状态标签的文本为“空页框”或“未使用”，改变边框样式为虚线，并将指令表格内容清空或显示占位符（如 "--"）。 |
| set\_loaded(self, is\_loaded=True) | 根据 is\_loaded 参数设置组件的加载状态。如果为 True，则边框变实线（通常为绿色），状态标签显示“已加载”，表格背景也可能改变颜色以示区分。如果为 False，则恢复到未加载的默认外观。 |
| highlight\_instruction(self, instruction\_index) | 高亮显示当前正在执行的指令。如果传入的 instruction\_index 属于此页框所显示的页面范围，则该页框组件的边框会特别高亮（例如变为橙色），状态标签会提示正在执行的指令号，并且指令表格中对应的指令单元格背景色会改变。 |

* + 1. **类 PagingUI (继承自 QWidget)**

PagingUI 是整个内存分页调度模拟程序的主界面窗口类。它负责创建和管理所有的GUI元素，包括参数设置区（指令数、内存页框数、替换算法选择）、控制按钮区（开始、单步、连续、暂停、重置）、内存页框显示区（由多个 MemoryPageWidget 组成）、执行日志区以及状态显示区（当前指令、当前页面、是否缺页、进度条、统计信息等）。此类还处理用户交互，驱动 MemoryDispatch 对象进行模拟，并实时更新界面以反映模拟状态。

函数设计和作用：

|  |  |
| --- | --- |
| **函数名** | **作用描述** |
| \_\_init\_\_(self) | 初始化主窗口。调用 initUI 方法来构建界面。初始化一些内部状态变量，如 memory\_dispatch (模拟控制器)、memory\_widgets (存储页框组件的字典)、页框与页面的映射关系 (frame\_to\_page, page\_to\_frame)、自动执行定时器 auto\_timer 等。 |
| initUI(self) | 设置窗口的标题、大小、位置和整体样式。调用 create\_left\_panel 和 create\_right\_panel 创建界面的左右两大主要区域，并使用 QHBoxLayout 进行布局。 |
| create\_left\_panel(self) | 创建并返回界面的左侧面板。该面板通常包含当前执行状态的显示（当前指令、当前页面、是否缺页）以及内存中页面的可视化展示区域（一个网格布局，用于放置 MemoryPageWidget 实例）。 |
| create\_right\_panel(self) | 创建并返回界面的右侧面板。该面板通常包含执行日志的文本区域 (QTextEdit) 和控制面板（包含参数设置、操作按钮、进度条和统计信息标签）。 |
| start\_simulation(self) | 根据用户在控制面板中设置的参数（指令数、内存页框数、替换算法）来初始化并开始一个新的模拟。它会创建 MemoryDispatch、Dispatcher 和 Allocation 的实例，调用 init\_memory\_frames 初始化内存页框显示，清空日志，重置进度条，并更新按钮的可用状态。 |
| init\_memory\_frames(self, frame\_count) | 根据给定的 frame\_count（页框数量），在内存显示区域创建相应数量的 MemoryPageWidget 实例，并将它们初始设置为空闲状态。同时初始化 frame\_to\_page 和 page\_to\_frame 映射。 |
| step\_execute(self) | 执行模拟的单个步骤。它会从 memory\_dispatch 获取当前指令和页面，调用 dispatcher.accept\_request 处理页面请求，记录是否发生缺页。如果发生缺页导致页面替换或加载到空闲帧，则调用 update\_frame\_mapping。然后调用 update\_display 和 update\_memory\_display 更新界面，记录日志，并使 allocation 前进到下一条指令。检查模拟是否结束。 |
| update\_frame\_mapping(self, old\_pages, new\_pages, new\_page) | 当发生页面替换或新页面载入空闲页框时，此方法负责更新 frame\_to\_page（页框到页面的映射）和 page\_to\_frame（页面到页框的映射）这两个字典。它会找出被替换出去的页面和新载入的页面，并更新它们在哪个物理页框中的记录。同时在日志中记录这一变化。 |
| auto\_execute(self) | 启动连续执行模式。它会启动一个 QTimer (auto\_timer)，该定时器会以一定的时间间隔（例如100毫秒）重复调用 step\_execute 方法。同时更新相关按钮（如“连续执行”禁用，“暂停”启用）的状态。 |
| pause\_execute(self) | 暂停连续执行模式。它会停止 auto\_timer 定时器。同时更新相关按钮（如“连续执行”启用，“暂停”禁用）的状态。 |
| finish\_simulation(self) | 当所有指令执行完毕或模拟被强制结束时调用。它会停止任何活动的定时器，将 memory\_dispatch 的状态设为完成，禁用单步和连续执行按钮，启用开始按钮，并在日志中显示最终的统计结果（如总请求数、总缺页数、缺页率）。 |
| reset\_simulation(self) | 重置整个模拟环境和用户界面到初始状态。停止定时器，清除 memory\_dispatch 实例，重置所有统计数据和日志，清空内存页框显示，并恢复按钮的初始可用状态，以便用户可以开始新的模拟。 |
| update\_display(self, current\_instruction, current\_page, page\_fault) | 更新界面上显示当前状态的标签，包括当前执行的指令号、当前访问的页面号、是否发生缺页的提示（通常用不同颜色或文本表示）。同时更新进度条的值和统计信息标签（请求数、缺页数、缺页率）。 |
| update\_memory\_display(self) | 遍历所有 MemoryPageWidget 实例（代表物理页框），根据当前的 frame\_to\_page 映射关系更新每个页框显示的内容（是空闲还是加载了哪个页面）。如果页框加载了页面，则调用其 set\_page\_content 和 set\_loaded 方法。同时，如果当前有正在执行的指令，会调用相应页框的 highlight\_instruction 方法来高亮显示。 |
| clear\_memory\_display(self) | 清除内存显示区域的所有 MemoryPageWidget 实例。在重置模拟或开始新的模拟之前调用，以确保界面干净。 |

**3．置换算法说明**

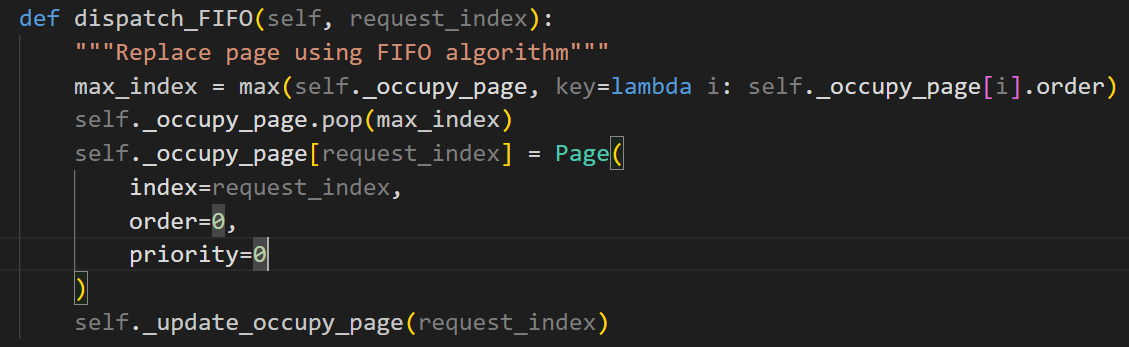
在操作系统中，当发生缺页中断且内存已满时，需要选择一个已在内存中的页面将其换出，以便为新的页面腾出空间。本项目实现了两种经典的页面替换算法：FIFO (先进先出) 和 LRU (最近最少使用)。这些算法在 Dispatcher 类中实现。

**3.1 FIFO 先进先出置换算法**

FIFO 算法的原理非常简单：选择最早进入内存的页面进行替换。这种算法实现起来容易，但性能上可能不是最优的，因为它可能会替换掉一个经常被访问的页面，仅仅因为它进入内存的时间较早。

在本项目中，Page 数据类包含一个 order 属性，用于记录页面进入内存的相对顺序。当一个新页面被调入内存时，其 order 初始化为0。而每次页面访问（无论是命中还是缺页导致新页面调入）后，\_update\_occupy\_page 方法会遍历内存中所有**其他**已存在的页面，并将它们的 order 值加1。这样，order 值最大的页面就是最早进入内存的页面。

下面是 dispatch\_FIFO 方法的实现代码：

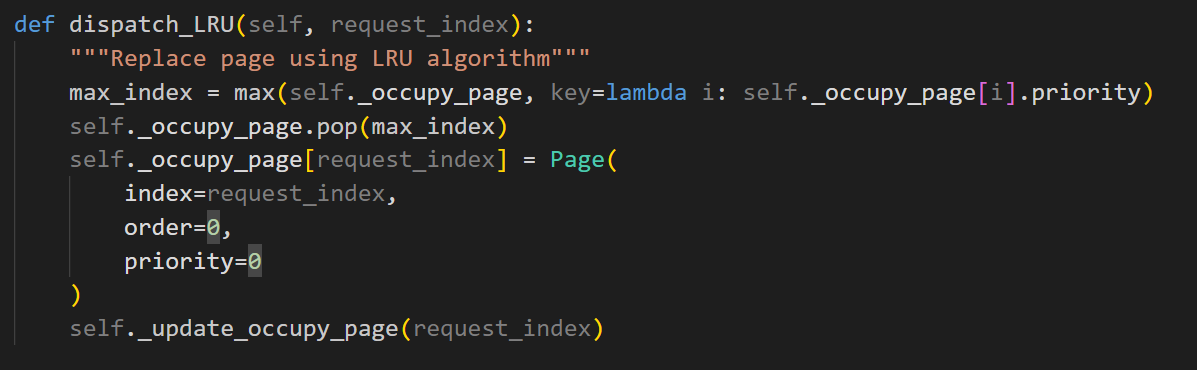


当 accept\_request 方法检测到缺页且内存已满时，如果当前选择的算法是 "FIFO"，就会调用 dispatch\_FIFO(request\_index)。该函数首先通过 max(self.\_occupy\_page, key=lambda i: self.\_occupy\_page[i].order) 找到 \_occupy\_page 字典中 order 属性值最大的那个页面的索引（即页号 max\_index）。然后，使用 self.\_occupy\_page.pop(max\_index) 将这个最老的页面从内存记录中移除。紧接着，为请求的新页面 request\_index 创建一个新的 Page 实例，其 order 初始化为0（表示它是最新进入的），并将其添加到 \_occupy\_page 字典中。最后，调用 self.\_update\_occupy\_page(request\_index)，这个方法会遍历当前内存中的所有页面，对于那些不是刚刚调入的 request\_index 的页面，将其 order 值加1，以此来维持页面进入内存的相对顺序。

**3.2 LRU (最近最少使用) 算法**

LRU 算法的原理是：当需要替换页面时，选择在最近一段时间内最久未被访问过的页面。这种算法通常能提供较好的性能，因为它倾向于保留那些近期被频繁访问的页面，符合程序的局部性原理。

为了实现 LRU，Page 数据类包含一个 priority 属性。这个属性用于追踪页面的“新鲜度”：priority 值越小，表示页面越新被访问过；priority 值越大，表示页面越久未被访问。具体实现如下：



当 accept\_request 方法检测到缺页且内存已满，并且当前选择的算法是 "LRU" 时，就会调用 dispatch\_LRU(request\_index)。该函数通过 max(self.\_occupy\_page, key=lambda i: self.\_occupy\_page[i].priority) 找到 \_occupy\_page 字典中 priority 属性值最大的页面索引 (max\_index)，这个页面就是最近最少使用的页面。然后，使用 self.\_occupy\_page.pop(max\_index) 将其移除。之后，为请求的新页面 request\_index 创建一个新的 Page 实例，其 priority 初始化为0（表示它是最新被使用的），并添加到 \_occupy\_page 中。最后，调用 self.\_update\_occupy\_page(request\_index)，此方法会增加内存中其他所有页面的 priority 值，反映出它们又经历了一个未被访问的“时间单位”。

**4.项目小结**

本项目的核心是通过一个具有可视化界面的模拟系统，实现对请求分页存储管理过程的全路跟踪。系统通过随机生成的320条指令，分成32页，每页 10 条，进入最多仅有4个页框的内存系统中进行进程执行。当指令所在页未被加载时触发缺页，系统根据指定的置换算法执行页面替换，这个过程被统计、显示并通过 UI 载图显示。

项目的重点也包括展现 FIFO 和 LRU 这两种置换算法在实际运行过程中的差异性和效果。FIFO 依据页面被调入的顺序进行替换，简单直观但在面对有序访问时易造成页面排除失败；LRU 则根据最久未被调用的页面进行替换，在大多数情况下能提供更优算的效果。