实验报告

基本信息

- **班级**: 计科 23 级 1 班
- **学号**: 202305133315
- **姓名**: 周潍可
- **实验名称**: 页面置换算法模拟程序设计

1. 实验目的

> 本次实验旨在理解和掌握操作系统中虚拟内存管理的核心机制——页面置换算法.通过编程实现三种常见页面置换算法(OPT、FIFO、LRU)以及 LFU 算法,模拟页面访问过程,统计缺页中断次数和页面置换次数,从而对比不同算法的性能表现.
>
> 实验要求:
>
> - 随机生成长度为 30 的页面访问序列;
> - 内存物理块数固定为 3;
> - 模拟四种页面置换算法并输出结果;
> - 对比分析各算法在相同测试用例下的缺页率和置换效率.

2. 数据结构说明

描述你在程序中使用的数据结构及其作用.

> 本实验中使用了以下数据结构:

> - `std::vector<int>` :表示页面访问序列,包含 30 个整数,取值范围为 0~9.
> - `std::set<int>` 或 `std::unordered_set<int>` :用于快速判断当前页面是否已在内存中.
> - `std::queue<int>` :在 FIFO 算法中维护页面进入内存的顺序.
> - `std::map<int, int>` :在 LRU 算法中记录每个页面最近一次被访问的时间戳.

> - `std::list<int>` 或 `std::deque<int>` :在 LRU 算法中维护访问顺序,便于查找最久未使用的页面.
> - `std::map<int, int>` :在 LFU 算法中记录每个页面的访问频率.

这些数据结构分别服务于不同算法的需求,共同实现了页面置换过程的模拟.

3. 处理过程(流程图或盒图)


```
A[开始] --> B[遍历页面访问序列]
B --> C{当前页面是否已存在于内存中?}
C -->|是| D[更新相关信息]
C -->|否| E[发生缺页中断]
E --> F{内存是否已满?}
F -->|否| G[将页面加入内存]
F -->|是| H[选择页面进行替换]
H --> I[根据算法选择页面]
I -->|OPT| J[选择最远被访问的页面]
I -->|FIFO| K[替换最早进入内存的页面]
I -->|LRU| L[替换最久未访问的页面]
I -->|LFU| M[替换访问次数最少的页面]
D --> N[统计缺页中断次数和页面置换次数]
G --> N
J --> N
K --> N
L --> N
M --> N
N --> O[结束]
```

4. 源代码

```
#ifndef PAGE_REPLACER_H
#define PAGE_REPLACER_H

#pragma once
#include <algorithm>
#include <deque>
#include <limits>
#include <list>
#include <map>
#include <optional>
#include <vector>
enum class PageReplaceAlgo {
  OPT,
  FIFO,
  LRU_K,  // LUR_K
  LFU
};

class PageReplacer {
 public:
  PageReplacer(
      PageReplaceAlgo algorithm, size_t frame_capacity,
      size_t k = 2,  // default K=2
      const std::optional<std::vector<size_t>> &ref_seq = std::nullopt);

  void access_page(size_t page_num);  // 访问 某个页面

  std::vector<size_t> &get_frames();  // 当前的 frame 有哪些页

  size_t get_page_fault();     // 当前的 缺页数目
  double page_fault_ratio();   // 计算缺页率
  size_t get_replace_count();  // 当前的 置换次数

 private:
  PageReplaceAlgo algorithm_;
  size_t frame_capacity_;
  std::vector<size_t> frames_;
  size_t k_value_;  // LRU/LFU -K

  std::optional<std::vector<size_t>> ref_seq_;
```

```
  size_t access_index_ = 0;   // 当前访问位置
  size_t page_faults_ = 0;    // 缺页数目
  size_t replace_count_ = 0;  // 置换次数

  std::list<size_t> fifo_queue_;
  std::map<size_t, std::deque<size_t>>
      access_history_;  // 每个元素的访问历史 5:{2,4,5} 第2 4 5 次访问了
                        // 5这个元素

  size_t find_opt();
  size_t find_fifo();
  size_t find_lru_k();
  size_t find_lfu();
};

#endif  // PAGE_REPLACER_H
```

```cpp
#include "lab4/PageReplacer.h"

#include <algorithm>
#include <deque>
#include <limits>
#include <list>
#include <map>
#include <optional>
#include <stdexcept>
#include <vector>
PageReplacer::PageReplacer(PageReplaceAlgo algo, size_t capacity, size_t k,
                           const std::optional<std::vector<size_t>> &ref_seq)
    : algorithm_(algo),
      frame_capacity_(capacity),
      k_value_(k),
      ref_seq_(ref_seq) {
  //
  if (frame_capacity_ == 0) {
    throw(std::invalid_argument("Check Capacity!!"));
  }
  if (!ref_seq.has_value()) {
    throw(std::invalid_argument("Check RefSeq!!"));
  }
}

void PageReplacer::access_page(size_t pg) {
  //
  auto it = std::find(frames_.begin(), frames_.end(), pg);
  access_index_++;

  // Hit 命中
  if (it != frames_.end()) {
    if (algorithm_ == PageReplaceAlgo::LFU ||
        algorithm_ == PageReplaceAlgo::LRU_K) {
      //
      access_history_[pg].push_back(access_index_);

      if (access_history_[pg].size() > k_value_) {
        access_history_[pg].pop_front();
      }
    }
  }
```

```
else {
  // 未命中
  page_faults_++;
  // 未满
  if (frames_.size() < frame_capacity_) {
    frames_.push_back(pg);
    if (algorithm_ == PageReplaceAlgo::FIFO) {
      fifo_queue_.push_back(pg);
    } else if (algorithm_ == PageReplaceAlgo::LFU ||
               algorithm_ == PageReplaceAlgo::LRU_K) {
      //
      access_history_[pg].push_back(access_index_);
    }
  } else {
    // 已经满了 需要置换
    size_t replace_idx;
    size_t pg_to_replace;
    switch (algorithm_) {
      case PageReplaceAlgo::OPT:
        replace_idx = find_opt();
        break;
      case PageReplaceAlgo::FIFO:
        replace_idx = find_fifo();
        break;
      case PageReplaceAlgo::LRU_K:
        replace_idx = find_lru_k();
        break;
      case PageReplaceAlgo::LFU:
        replace_idx = find_lfu();
        break;
      default:
        throw std::runtime_error("Algorithm Error!");
    }
    // 把要置换的页面 pg_to_replace换成 pg
    pg_to_replace = frames_[replace_idx];
    frames_[replace_idx] = pg;
    replace_count_++;

    switch (algorithm_) {
      case PageReplaceAlgo::FIFO: {
        // FIFO
        auto fifo_it =
            std::find(fifo_queue_.begin(), fifo_queue_.end(), pg_to_replace);
```

```
          if (fifo_it != fifo_queue_.end()) {
            fifo_queue_.erase(fifo_it);
          }
          fifo_queue_.push_back(pg);
          break;
        }

        case PageReplaceAlgo::LRU_K:
        case PageReplaceAlgo::LFU: {
          // LRU-K 和 LFU
          access_history_.erase(pg_to_replace);
          access_history_[pg].push_back(access_index_);
          break;
        }

        default:
          break;
      }
    }
  }
}

std::vector<size_t> &PageReplacer::get_frames() {
  //
  return frames_;
}

size_t PageReplacer::get_page_fault() { return page_faults_; }

size_t PageReplacer::get_replace_count() { return replace_count_; }

double PageReplacer::page_fault_ratio() {
  if (access_index_ == 0) {
    return 0.0;
  }
  return static_cast<double>(get_page_fault()) / access_index_;
}

size_t PageReplacer::find_opt() {
  if (!ref_seq_.has_value() || ref_seq_->empty()) {
    throw std::runtime_error("Reference sequence is invalid or empty");
  }
  std::map<size_t, size_t> next_use;
```

```
  for (size_t page : frames_) {
    next_use[page] = std::numeric_limits<size_t>::max();
    for (size_t i = access_index_; i < ref_seq_->size(); ++i) {
      if ((*ref_seq_)[i] == page) {
        next_use[page] = i;
        break;
      }
    }
  }
  size_t max_distance = 0;
  size_t replace_idx = 0;
  for (size_t i = 0; i < frames_.size(); ++i) {
    size_t page = frames_[i];
    if (next_use[page] > max_distance) {
      max_distance = next_use[page];
      replace_idx = i;
    }
  }
  return replace_idx;
}

size_t PageReplacer::find_fifo() {
  //
  if (fifo_queue_.empty()) {
    throw std::runtime_error("FIFO Queue Is Empty!!");
  }

  size_t pg_to_find = fifo_queue_.front();

  auto it = std::find(frames_.begin(), frames_.end(), pg_to_find);

  if (it == frames_.end()) {
    throw std::runtime_error("FIFO Page not Found in Frames");
  }
  // 返回该页面在 frames_ 中的idx
  return std::distance(frames_.begin(), it);
}

size_t PageReplacer::find_lru_k() {
  size_t best_idx = 0;  // 最优页面的最后一次访问时间
  size_t min_k_access_time =
      std::numeric_limits<size_t>::max();  // 当前最优页面的第 K 次访问时间
  size_t min_cnt =
```

```
    std::numeric_limits<size_t>::max();  // 当前最优页面的历史访问次数
size_t last_access_tb =
    std::numeric_limits<size_t>::max();  // 最优页面的最后一次访问时间

for (size_t i = 0; i < frames_.size(); ++i) {
  size_t current_pg = frames_[i];
  const auto &history = access_history_[current_pg];

  size_t cnt = history.size();
  size_t k_access_time;  // 当前页面的 __倒数__ 第 K 次访问时间

  size_t last_access_time =
      history.empty() ? 0 : history.back();  // 当前页面的第 K 次访问时间
  // 2:history:{4,5,7,10} k=2:
  // 倒数第 k=2次就是 his[4-2] = his[2]=7

  // 如果访问次数没有 k 次 设置为0
  /*
  access_history_ = {
    1: [10, 20, 30],
    2: [15, 25],
    3: [5]
  }
  */
  if (cnt < k_value_) {
    k_access_time = 0;
  } else {
    k_access_time = history[cnt - k_value_];
  }
  /*
  优先考虑历史访问次数小于 K 的页面 (说明使用频率低)
  若多个页面都满足此条件,选访问次数最少的
  如果访问次数相同,则选最后一次访问时间最早的
  否则,比较第 K 次访问时间
  越早越好(表示很久没用了)
  如果相同,则看最后一次访问时间(越早越好)
  */
  bool is_better = false;

  // 优先考虑访问次数小于 K 的页面
  if (k_access_time == 0) {
    if (min_k_access_time == 0) {
      if (cnt < min_cnt) {
```

```
          is_better = true;
        } else if (cnt == min_cnt) {
          if (last_access_time < last_access_tb) {
            is_better = true;
          }
        }
      } else {
        is_better = true;
      }
    } else {
      if (min_k_access_time != 0) {
        if (k_access_time < min_k_access_time) {
          is_better = true;
        } else if (k_access_time == min_k_access_time) {
          if (last_access_time < last_access_tb) {
            is_better = true;
          }
        }
      }
    }
    if (is_better) {
      min_k_access_time = k_access_time;
      min_cnt = cnt;
      last_access_tb = last_access_time;
      best_idx = i;
    }
  }
  return best_idx;
}
size_t PageReplacer::find_lfu() {
  size_t best_idx = 0;
  size_t min_freq = std::numeric_limits<size_t>::max();  // 最小访问频率
  size_t last_access_tb =
      std::numeric_limits<size_t>::max();  // 对应页面的最后访问时间

  for (size_t i = 0; i < frames_.size(); ++i) {
    size_t current_pg = frames_[i];
    const auto &history = access_history_[current_pg];

    size_t cnt = history.size();
    size_t last_access_time = history.empty() ? 0 : history.back();

    // 如果当前页面访问频率更低,则更新最优页面
```

```
    if (cnt < min_freq) {
      min_freq = cnt;
      last_access_tb = last_access_time;
      best_idx = i;
    }
    // 如果访问频率相等,比较最后一次访问时间
    else if (cnt == min_freq) {
      if (last_access_time < last_access_tb) {
        last_access_tb = last_access_time;
        best_idx = i;
      }
    }
  }

  return best_idx;
}
```

5.实验总结

> 通过本次实验,我深入理解了操作系统中虚拟内存管理的关键技术之一——页面置换算法.在实际编码过程中,我掌握了如何模拟内存访问行为,并对不同算法的实现机制进行了比较分析.

> 在实现过程中,我遇到了一些挑战:

> - 如何高效地查找下一个不再使用的页面(OPT);
> - 如何维护访问时间戳以支持 LRU 的判断;
> - 如何避免 FIFO 中可能出现的“Belady 异常”;
> - 如何准确记录页面访问频率并及时更新 LFU 表.
> - 通过合理选择合适的数据结构(如 set、map、deque),这些问题都得到了有效解决.实验结果显示,在大多数情况下,OPT 性能最优,而 LRU 和 LFU 表现接近,FIFO 相对较差但实现简单.

此次实验不仅加深了我对操作系统的理解,也提升了我在实际开发中运用算法解决问题的能力.未来可以尝试引入更复杂的页面访问模型,如基于局部性原理的工作集模型,进一步优化模拟效果.