实验四：内存页面淘汰算法（命中率）实验

课程：操作系统

姓名：赖景康
学号：2023214564
指导教师：赵玉琦
提交日期：2025.12.03

1 实验目的

- 理解并实现三种主流页面置换算法：FIFO、LRU 与 OPT（最佳算法）并比较其缺页性能；
- 探索页框数量变化对缺页率的影响（尤其观察 Belady 现象）；
- 能用程序模拟不同算法并统计缺页次数，为理解页面置换策略提供直观实验数据。

2 问题描述与实验要求

本实验针对同一批参考串，比较 FIFO、LRU、OPT 三种页面置换算法在不同页框数下的缺页次数；并对比 FIFO/LRU 在 k=2,3,4 下的缺页情况；以及构造 Belady 现象的示例序列，验证 FIFO 在增加页框时可能出现缺页次数反而增加的异常情况。

3 实验环境

- 开发语言：Rust
- 开发工具：VS Code
- 编译/运行：Cargo（注：在当前编辑环境中无法直接运行 Cargo，我在报告中给出解析结果；建议在本地运行 ‘cargo run‘ 来获取控制台输出并替换本文中的数值以作核验。）

4 实现思路与主要函数

实现采用单线程事件驱动式模拟：给定页面访问参考串与页框数，分别实现三种算法的模拟函数，返回缺页次数。主要函数签名如下（节选自 ‘task4/src/main.rs‘）：

```
fn simulate_fifo(reference: &[i32], frames: usize) -> usize
fn simulate_lru(reference: &[i32], frames: usize) -> usize
fn simulate_opt(reference: &[i32], frames: usize) -> usize
```

实现要点：

- FIFO 使用一个队列（VecDeque）维护页面入框顺序，缺页时弹出队头并加入新页；
- LRU 通过维护每个页的最后使用时间（索引）来选取最近最久未使用页进行替换；

- OPT 在发生缺页且页框已满时，对在当前点之后每个在内页的下一次出现位置做查找，选择下一次出现最远的页进行替换；若某页不再出现则优先替换它（实现为寻找 position 返回 None 的帧）。

5 代码展示（节选）

下面给出程序中用于模拟三种页面置换算法的关键实现节选(完整代码见 ‘task4/src/main.rs‘)。

```rust
use std::collections::{HashMap, HashSet, VecDeque};

fn simulate_fifo(reference: &[i32], frames: usize) -> usize {
  assert!(frames > 0, "页框数必须大于0");
  let mut faults = 0;
  let mut frame_queue: VecDeque<i32> = VecDeque::new();
  let mut in_memory: HashSet<i32> = HashSet::new();

  for &page in reference {
    if in_memory.contains(&page) { continue; }
    faults += 1;
    if frame_queue.len() == frames {
      if let Some(evicted) = frame_queue.pop_front() {
        in_memory.remove(&evicted);
      }
    }
    frame_queue.push_back(page);
    in_memory.insert(page);
  }
  faults
}
```

```rust
fn simulate_lru(reference: &[i32], frames: usize) -> usize {
  assert!(frames > 0, "页框数必须大于0");
  let mut faults = 0;
  let mut frames_vec: Vec<i32> = Vec::new();
  let mut last_used: HashMap<i32, usize> = HashMap::new();

  for (idx, &page) in reference.iter().enumerate() {
    if frames_vec.contains(&page) {
      last_used.insert(page, idx);
      continue;
```

```
    }
    faults += 1;
    if frames_vec.len() < frames {
      frames_vec.push(page);
    } else {
      let (replace_idx, _) = frames_vec
        .iter()
        .enumerate()
        .min_by_key(|(_, &p)| last_used.get(&p).copied().unwrap_or(0))
        .expect("至少有一个页框");
      frames_vec[replace_idx] = page;
    }
    last_used.insert(page, idx);
  }
  faults
}
```

```
fn simulate_opt(reference: &[i32], frames: usize) -> usize {
  assert!(frames > 0, "页框数必须大于 0");
  let mut faults = 0;
  let mut frames_vec: Vec<i32> = Vec::new();

  for (idx, &page) in reference.iter().enumerate() {
    if frames_vec.contains(&page) { continue; }
    faults += 1;
    if frames_vec.len() < frames {
      frames_vec.push(page);
      continue;
    }
    let mut target_idx = 0;
    let mut farthest_distance: Option<usize> = None;
    for (frame_idx, &frame_page) in frames_vec.iter().enumerate() {
      match reference[idx + 1..].iter().position(|&p| p == frame_page) {
        None => { target_idx = frame_idx; break; }
        Some(dist) => {
          let dist = dist + 1;
          if farthest_distance.map_or(true, |current| dist > current) {
```

```
                farthest_distance = Some(dist);
                target_idx = frame_idx;
            }
        }
    }
    frames_vec[target_idx] = page;
  }
  faults
}
```

6 实验数据与参考串

本次实验使用程序中给定的参考串（见 ‘task4/src/main.rs‘）：

```
reference = [1,2,3,2,4,1,5,2,1,2,3,4,5,2,1]
```

以及用于演示 Belady 现象的构造序列：

```
belady_sequence = [1,2,3,4,1,2,5,1,2,3,4,5]
```

7 实验结果（解析计算）

注：由于当前环境无法直接执行 ‘cargo run‘，下面的缺页次数为对 ‘task4/src/main.rs‘ 中算法逻辑的逐步模拟/分析计算得到的结果；你可以在本地运行程序以验证并把控制台输出粘贴回本报告。

7.1 同一批数据下（页框 = 3）

- FIFO 缺页次数：12
- LRU 缺页次数：12
- OPT 缺页次数（参考）：8

7.2 FIFO 在不同页框数下（k=2,3,4）

7.3 LRU 在不同页框数下（k=2,3,4）

7.4 Belady 现象示例（FIFO）

对构造序列 ‘belady$_s$$equence$‘ $FIFO$

页框数	2	3	4
缺页次数	13	12	11

表 1: FIFO 在不同页框数下的缺页次数

页框数	2	3	4
缺页次数	13	12	9

表 2: LRU 在不同页框数下的缺页次数

页框 =3：缺页次数 = 9

页框 =4：缺页次数 = 10

这说明在该构造序列下，FIFO 随着页框数增加反而导致缺页次数上升，即出现 Belady 异常。

8 图表与表格展示

为了避免对额外 LaTeX 包的依赖（如 ‘pgfplots‘），下面以表格形式展示统计数据，便于在任何标准 TeX 安装下编译。

算法	缺页次数 (页框 =3)
FIFO	12
LRU	12
OPT	8

表 3: 同一批数据下 (页框 =3) 三种算法缺页次数比较

9 分析与讨论

- OPT（最佳算法）由于能预知未来，缺页次数最低；在本例中，OPT(3) = 8，显著优于 FIFO/LRU(3)=12。
- 对于该参考串，FIFO 与 LRU 在 k=3 时得到相同的缺页次数，但在 k=4 时 LRU 明显优于 FIFO（LRU(4)=9，FIFO(4)=11），体现出 LRU 更善于利用局部性。
- Belady 现象：FIFO 在构造序列下出现缺页次数随页框增加反而增加的情况 (3 -> 4：9 -> 10)，说明 FIFO 不是单调最优的；而 LRU/OPT 不会出现此类异常（LRU 通常避免 Belady)。

页框数	FIFO	LRU
2	13	13
3	12	12
4	11	9

表 4: FIFO 与 LRU 在不同页框数下的缺页次数比较

页框数	FIFO (Belady 序列)
3	9
4	10

表 5: Belady 构造序列下 FIFO 缺页次数（页框 3 vs 4）

- 实验提示：在实际系统中应优先使用带有最近使用信息的替换策略（如 LRU 或其近似算法）以获得更稳定的命中率；简单的 FIFO 尽管实现简单，但可能在某些工作集上表现极差。

10 实验结论

本次实验通过对比 FIFO、LRU、OPT 三种页面置换算法，验证了：

- OPT 为理论最优；
- LRU 在多数具有局部性的参考串上优于 FIFO；
- FIFO 可能会发生 Belady 现象，即增加页框反而增加缺页次数；因此实际系统中应谨慎采用单纯的 FIFO 替换策略。

11 附录：关键源代码（节选）

以下为 ‘task4/src/main.rs‘ 的主要实现节选：

```
use std::collections::{HashMap, HashSet, VecDeque};

fn simulate_fifo(reference: &[i32], frames: usize) -> usize { ... }

fn simulate_lru(reference: &[i32], frames: usize) -> usize { ... }

fn simulate_opt(reference: &[i32], frames: usize) -> usize { ... }
```

```
fn main() {
  let reference = vec![1,2,3,2,4,1,5,2,1,2,3,4,5,2,1];
  // ...
}
```

如需完整代码，请参见仓库路径：‘task4/src/main.rs‘（已附在项目中）。

12 参考文献

- Belady, L. A. (1966). A study of replacement algorithms for a virtual-storage computer. (经典页面置换问题论文)
- 教材与讲义：操作系统课程相关讲义（参见课程资料）
- Github 仓库：https://github.com/shufufufu/OS-TASKS