ĐẠI HỌC BÁCH KHOA HÀ NỘI

TRƯỜNG CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

——————— * ———————

BÀI TẬP LỚN

HỌC PHẦN: NGUYÊN LÝ HỆ ĐIỀU HÀNH

(Mã học phần: IT3070)

Đề tài:

Minh họa các giải thuật thay thế trang trong hệ điều hành

Giảng viên hướng dẫn: **TS. Đỗ Quốc Huy**

Sinh viên thực hiện : **Nguyễn Việt Anh 20215307**

Nguyễn Lê Đức Anh 20215305

Nguyễn Việt An 20225244

Đoàn Ngọc Toàn 20225100

Lớp: 157502

Hà Nội, tháng 6 năm 2025

Mục lục

LỜI CẢM ƠN

Lời đầu tiên, nhóm sinh viên xin trân trọng cảm ơn và bày tỏ lòng biết ơn sâu sắc nhất tới thầy Đỗ Quốc Huy – Giảng viên Trường Công nghệ Thông tin & Truyền thông, Đại học Bách Khoa Hà Nội – người đã tận tình giảng dạy, hướng dẫn và chỉ bảo nhóm trong suốt quá trình thực hiện bài tập lớn môn Hệ điều hành.

Nhóm cũng xin gửi lời cảm ơn chân thành tới các bạn sinh viên trong lớp và bạn bè đã luôn động viên, chia sẻ và hỗ trợ để nhóm có thể hoàn thành tốt đề tài này.

Mặc dù cả nhóm đã nỗ lực hết mình để hoàn thiện bài làm, tuy nhiên chắc chắn không thể tránh khỏi những thiếu sót nhất định. Nhóm mong muốn nhận được những góp ý quý báu và nhận xét thẳng thắn từ Thầy để có thể rút kinh nghiệm và hoàn thiện hơn trong các dự án học thuật tiếp theo.

Một lần nữa, xin chân thành cảm ơn Thầy Đỗ Quốc Huy vì sự đồng hành và tận tâm trong suốt quá trình nhóm thực hiện bài tập lớn.

Xin trân trọng cảm ơn!

Chương 1: Giới thiệu đề tài

1.1. Lý do chọn đề tài

Trong hệ điều hành hiện đại, quản lý bộ nhớ đóng vai trò then chốt trong việc đảm bảo hiệu năng, tính ổn định và khả năng phản hồi của hệ thống. Khi các chương trình yêu cầu sử dụng nhiều bộ nhớ hơn khả năng cung cấp của bộ nhớ vật lý (RAM), hệ điều hành sẽ kích hoạt cơ chế bộ nhớ ảo (virtual memory) để đảm bảo chương trình vẫn có thể thực thi bình thường mà không bị gián đoạn.

Một trong những thành phần cốt lõi của bộ nhớ ảo là cơ chế thay thế trang (page replacement) – kỹ thuật xác định trang nào nên bị loại khỏi bộ nhớ để nhường chỗ cho trang mới khi không gian bộ nhớ đã đầy. Việc lựa chọn giải thuật thay thế trang phù hợp có ảnh hưởng trực tiếp đến số lần xảy ra lỗi trang (page faults), từ đó tác động đáng kể đến tốc độ xử lý và hiệu quả vận hành của toàn bộ hệ thống.

Trong thực tế, có nhiều giải thuật thay thế trang được thiết kế với những tiêu chí và chiến lược khác nhau như: FIFO (First-In, First-Out), LRU (Least Recently Used), OPT (Optimal), LFU (Least Frequently Used), MFU (Most Frequently Used),… Mỗi thuật toán đều có ưu – nhược điểm riêng, và hiệu quả của chúng phụ thuộc vào đặc điểm truy cập bộ nhớ của chương trình cụ thể.

Vì lý do đó, nhóm chúng tôi lựa chọn thực hiện đề tài "Mô phỏng các giải thuật thay thế trang trong hệ điều hành" nhằm trực quan hóa cách hoạt động của từng thuật toán, từ đó so sánh hiệu quả của chúng dựa trên số lỗi trang sinh ra với cùng một chuỗi truy cập bộ nhớ. Đề tài không chỉ giúp nhóm hiểu sâu hơn về cơ chế vận hành của bộ nhớ ảo mà còn rèn luyện kỹ năng thiết kế thuật toán, lập trình mô phỏng và tư duy phân tích hiệu năng – những kỹ năng quan trọng cho một kỹ sư phần mềm tương lai.

1.2. Mục tiêu của đề tài

Mục tiêu của đề tài là xây dựng một chương trình mô phỏng hoạt động của các giải thuật thay thế trang phổ biến trong hệ điều hành. Chương trình cho phép người dùng nhập vào chuỗi truy cập trang và số lượng khung trang (frame) tương ứng, sau đó thực hiện mô phỏng từng giải thuật để tính toán và hiển thị số lỗi trang mà mỗi giải thuật gặp phải. Thông qua kết quả thu được so sánh và phân tích hiệu quả của các thuật toán nhằm đánh giá sự phù hợp của từng giải thuật trong các tình huống truy cập bộ nhớ khác nhau.

1.3. Phạm vi và đối tượng nghiên cứu

Phạm vi nghiên cứu của đề tài tập trung vào việc mô phỏng một số giải thuật thay thế trang trong hệ điều hành, bao gồm các thuật toán kinh điển như FIFO, LRU, OPT, LFU và MFU. Đề tài chỉ xét trong ngữ cảnh quản lý bộ nhớ ảo với cơ chế phân trang, không đi sâu vào các cơ chế quản lý bộ nhớ khác như phân đoạn hay phân chia động.

Đối tượng nghiên cứu là cách thức hoạt động của từng thuật toán thay thế trang, cách chúng xử lý chuỗi truy cập bộ nhớ đầu vào và ảnh hưởng của chúng đến số lượng lỗi trang xảy ra trong quá trình thực thi chương trình. Việc mô phỏng được thực hiện thông qua một ứng dụng đơn giản cho phép người dùng nhập vào chuỗi trang và số lượng khung trang để đánh giá và so sánh hiệu quả của từng thuật toán trong các kịch bản cụ thể.

1.4. Phương pháp thực hiện

Nghiên cứu lý thuyết: Tìm hiểu về khái niệm bộ nhớ ảo, cơ chế phân trang và các giải thuật thay thế trang như FIFO, LRU, OPT, LFU và MFU để hiểu rõ nguyên lý hoạt động, ưu nhược điểm của từng thuật toán.

Cài đặt chương trình và thử nghiệm mô phỏng: Sử dụng ngôn ngữ lập trình Python để xây dựng chương trình mô phỏng hoạt động của từng giải thuật thay thế trang.Tiến hành chạy chương trình với hai chuỗi truy cập trang khác nhau và một số lượng khung trang cố định để quan sát số lỗi trang của mỗi thuật toán.

So sánh và phân tích kết quả: Dựa trên kết quả thử nghiệm, so sánh hiệu quả hoạt động giữa các giải thuật

1.5. Cấu trúc báo cáo

Chương 1: Giới thiệu đề tài

Chương 2: Cơ sở lý thuyết

Chương 3: Thiết kế và cài đặt chương trình

Chương 4: Thực nghiệm và đánh giá

Chương 5: Kết luận và hướng phát triển

Chương 2: Cơ sở lý thuyết

2.1. Khái niệm bộ nhớ ảo

Bộ nhớ ảo là một kỹ thuật quản lý bộ nhớ được hệ điều hành sử dụng nhằm tạo ra ảo giác cho các ứng dụng rằng chúng đang có một khối bộ nhớ lớn và liên tục, ngay cả khi bộ nhớ vật lý (RAM) bị hạn chế. Kỹ thuật này cho phép các ứng dụng có kích thước lớn hơn có thể chạy trên các hệ thống có dung lượng RAM nhỏ hơn.

Mục tiêu chính của bộ nhớ ảo là hỗ trợ đa chương trình (multiprogramming). Một ưu điểm quan trọng mà bộ nhớ ảo mang lại là tiến trình đang chạy không cần phải được nạp toàn bộ vào bộ nhớ. Các chương trình có thể lớn hơn bộ nhớ vật lý hiện có. Bộ nhớ ảo cung cấp một lớp trừu tượng cho bộ nhớ chính, giúp loại bỏ các lo ngại về giới hạn lưu trữ.

Một hệ thống phân cấp bộ nhớ, bao gồm bộ nhớ trong hệ thống máy tính và bộ nhớ đĩa, cho phép một tiến trình chỉ cần một phần không gian địa chỉ của nó nằm trong RAM, từ đó cho phép nhiều tiến trình cùng tồn tại trong bộ nhớ.

Bộ nhớ ảo được triển khai kết hợp cả phần cứng và phần mềm. Nó ánh xạ các địa chỉ bộ nhớ được chương trình sử dụng, gọi là địa chỉ ảo, sang các địa chỉ vật lý trong bộ nhớ máy tính.

Tất cả các tham chiếu bộ nhớ trong một tiến trình đều là địa chỉ logic và được dịch động sang địa chỉ vật lý khi chương trình chạy. Điều này có nghĩa là tiến trình có thể được hoán đổi vào hoặc ra khỏi bộ nhớ chính và chiếm các vị trí khác nhau trong bộ nhớ chính tại các thời điểm khác nhau trong quá trình thực thi.

Một tiến trình có thể được chia thành nhiều phần, và các phần này không nhất thiết phải nằm liền kề nhau trong bộ nhớ chính khi đang chạy. Sự kết hợp giữa dịch địa chỉ động tại thời gian chạy và việc sử dụng bảng trang hoặc bảng đoạn cho phép điều này.

Có hai kỹ thuật chính để quản lý và tổ chức không gian địa chỉ ảo, trong khuôn khổ đề tài chỉ xét bộ nhớ ảo phân trang.

2.2. Quá trình phân trang và thay thế trang

Phân trang là một kỹ thuật quản lý bộ nhớ cho phép bộ nhớ ảo được chia nhỏ thành các khối cố định gọi là trang (pages), đồng thời bộ nhớ vật lý cũng được chia thành các khung trang (frames) có kích thước tương đương. Khi một tiến trình thực thi, các trang của nó sẽ được nạp vào các khung trang trong bộ nhớ chính (RAM) theo nhu cầu.

Quá trình phân trang bắt đầu khi tiến trình gửi yêu cầu truy cập đến một địa chỉ ảo. Bộ nhớ ảo sẽ chuyển địa chỉ ảo này thành địa chỉ vật lý tương ứng thông qua bảng trang (page table), một cấu trúc dữ liệu dùng để lưu ánh xạ giữa trang ảo và khung trang vật lý.

Khi một trang cần truy cập không nằm trong RAM, hệ điều hành sẽ tạo ra một sự kiện gọi là lỗi trang (page fault). Lỗi trang báo hiệu cho hệ điều hành biết rằng trang này chưa được nạp vào bộ nhớ chính và cần phải được tải vào từ bộ nhớ phụ (thường là ổ cứng).

Quá trình thay thế trang xảy ra khi bộ nhớ chính đã đầy khung trang, và hệ điều hành cần giải phóng không gian để nạp trang mới. Hệ điều hành sẽ sử dụng các giải thuật thay thế trang như FIFO, LRU, Optimal, LFU, MFU,... để quyết định trang nào sẽ bị loại bỏ khỏi RAM nhằm nhường chỗ cho trang mới.

Các bước chính trong quá trình phân trang và thay thế trang bao gồm:

Phát hiện lỗi trang khi tiến trình truy cập trang chưa có trong RAM.

Tạm dừng tiến trình và chuyển sang trạng thái chờ để hệ điều hành xử lý lỗi trang.

Lựa chọn khung trang trống hoặc chọn trang cần thay thế dựa trên giải thuật thay thế trang.

Tải trang cần thiết từ bộ nhớ phụ vào khung trang vật lý.

Cập nhật bảng trang để phản ánh trạng thái mới.

Đưa tiến trình trở lại trạng thái sẵn sàng và tiếp tục thực thi.

Thông qua quá trình phân trang và thay thế trang, hệ điều hành đảm bảo rằng các tiến trình có thể sử dụng bộ nhớ một cách linh hoạt và hiệu quả, dù kích thước bộ nhớ vật lý bị giới hạn.

2.3. Khái niệm lỗi trang

Lỗi trang (Page Fault) là một sự kiện xảy ra khi một tiến trình cố gắng truy cập vào một trang bộ nhớ ảo mà trang đó chưa được nạp vào bộ nhớ vật lý (RAM). Khi lỗi trang xảy ra, hệ điều hành phải can thiệp để xử lý bằng cách tải trang cần thiết từ bộ nhớ phụ (thường là ổ cứng) vào bộ nhớ chính.

Lỗi trang là một phần quan trọng trong cơ chế quản lý bộ nhớ ảo, giúp hệ điều hành có thể chạy các chương trình lớn hơn bộ nhớ vật lý hiện có bằng cách hoán đổi các trang dữ liệu ra vào bộ nhớ một cách linh hoạt. Mỗi khi xảy ra lỗi trang, tiến trình tạm thời bị dừng lại để hệ điều hành xử lý việc nạp trang, sau đó tiến trình được tiếp tục thực thi.

Tuy nhiên, việc xử lý lỗi trang gây ra một chi phí về hiệu năng do thời gian truy xuất bộ nhớ phụ (ổ cứng) lâu hơn nhiều so với bộ nhớ RAM. Vì vậy, một mục tiêu quan trọng trong thiết kế hệ điều hành là giảm thiểu số lần lỗi trang xảy ra thông qua các giải thuật thay thế trang hiệu quả.

2.4. Các giải thuật thay thế trang

Hiện nay, có nhiều giải thuật thay thế trang được nghiên cứu và áp dụng, mỗi giải thuật có ưu nhược điểm riêng tùy thuộc vào đặc điểm truy cập bộ nhớ của chương trình và yêu cầu của hệ thống. Mục tiêu của các thuật toán là để giảm số lượng lỗi trang. Trong phần này, chúng ta sẽ tìm hiểu và mô phỏng các giải thuật thay thế trang phổ biến như FIFO (First-In-First-Out), LRU (Least Recently Used), Optimal (tối ưu), LFU (Least Frequently Used) và MFU (Most Frequently Used).

2.4.1. FIFO

Là một trong những giải thuật thay thế trang đơn giản và phổ biến nhất trong quản lý bộ nhớ ảo. Nguyên tắc hoạt động của FIFO rất trực quan: trang nào được nạp vào bộ nhớ trước sẽ là trang bị loại bỏ đầu tiên khi cần giải phóng khung trang cho trang mới.

Cách thức hoạt động cụ thể của FIFO như sau: khi bộ nhớ đã đầy và một trang mới cần được tải vào, hệ điều hành sẽ loại bỏ trang nằm ở vị trí lâu nhất trong bộ nhớ (trang "vào trước nhất") để nhường chỗ. Quá trình này giống như một hàng đợi, nơi trang được thêm vào ở cuối và loại bỏ từ đầu.

Ưu điểm:

- Đơn giản và dễ triển khai vì chỉ cần quản lý thứ tự các trang được nạp vào bộ nhớ.

- Thích hợp cho các hệ thống nhỏ hoặc làm tiêu chuẩn so sánh với các giải thuật khác.

Nhược điểm:

- Có thể gây ra lỗi trang không cần thiết khi trang lâu nhất vẫn đang được sử dụng (hiện tượng Belady).
- Hiệu suất thường thấp hơn các giải thuật như LRU hoặc Optimal trong nhiều trường hợp thực tế.

2.4.2. LRU

Giải thuật LRU (Ít được sử dụng gần đây nhất) là một trong những thuật toán thay thế trang phổ biến và hiệu quả trong nhiều trường hợp thực tế. Nguyên lý hoạt động của LRU là: khi cần thay thế một trang trong bộ nhớ, hệ điều hành sẽ chọn trang ít được truy cập nhất trong thời gian gần đây để loại bỏ.

Giả định được đặt ra trong giải thuật này là các trang được sử dụng gần đây có nhiều khả năng sẽ tiếp tục được sử dụng trong tương lai gần, trong khi các trang không được dùng trong một thời gian dài có khả năng ít được dùng hơn.

Ưu điểm:

- Hiệu suất tốt nhờ khả năng dự đoán tương đối chính xác trang nào sẽ không được sử dụng trong tương lai gần.
- Không gặp hiện tượng Belady.

Nhược điểm:

- Phức tạp hơn khi triển khai vì cần sử dụng thêm cấu trúc dữ liệu để theo dõi thứ tự sử dụng của các trang.
- Phụ thuộc nhiều vào cách mà các trang được truy cập trong quá trình thực thi chương trình. (tức là thuật toán có thể hoạt động tốt hoặc xấu tùy vào cách chương trình truy cập các trang trong bộ nhớ**)**

2.4.3. OPT

Giải thuật thay thế trang Optimal (OPT) là một thuật toán lý tưởng trong quản lý bộ nhớ ảo, được thiết kế nhằm đạt hiệu suất cao nhất có thể về mặt lý thuyết. Thuật toán này hoạt động dựa trên nguyên tắc: khi xảy ra lỗi trang, sẽ thay thế trang mà trong tương lai sẽ

không được sử dụng trong khoảng thời gian lâu nhất. Điều này giúp giảm thiểu tối đa số lỗi trang có thể xảy ra đối với một chuỗi truy cập trang đã biết trước.

Ưu điểm:

- Cho số lỗi trang ít nhất có thể với một chuỗi truy cập cụ thể.
- Không bị hiện tượng Belady's Anomaly như FIFO.
- Là mốc chuẩn (benchmark) để đánh giá các giải thuật khác như FIFO, LRU, v.v.

Nhược điểm: Không thể triển khai trong thực tế, vì hệ điều hành không thể biết trước tương lai, không biết chương trình sẽ truy cập trang nào tiếp

2.4.4. LFU

Mục tiêu chính của thuật toán này là thay thế trang có tần suất truy cập thấp nhất, với giả định rằng trang ít được sử dụng trong quá khứ có khả năng ít được sử dụng trong tương lai.

Nguyên lý hoạt động: Mỗi trang trong bộ nhớ sẽ đi kèm với một bộ đếm tần suất (frequency counter), theo dõi số lần trang đó được truy cập. Khi cần thay thế một trang (do bộ nhớ đầy và xảy ra lỗi trang), LFU sẽ chọn trang có bộ đếm nhỏ nhất – tức là trang ít được truy cập nhất kể từ khi nó được nạp vào bộ nhớ. Nếu có nhiều trang có cùng tần suất thấp nhất, có thể chọn một trang theo thứ tự đến trước (giống FIFO) hoặc áp dụng tiêu chí bổ sung khác (như tuổi trang, thời điểm truy cập gần nhất,...).

Ưu điểm:

- Hiệu quả hơn FIFO: do xét đến tần suất truy cập thay vì thời gian nạp vào bộ nhớ.
- Phù hợp với các ứng dụng có tính truy cập lặp lại (locality): các trang thường xuyên được sử dụng sẽ được giữ lại lâu hơn.
- Hạn chế lỗi trang trong các chương trình có mô hình truy cập ổn định.

Nhược điểm:

- Cần thêm cấu trúc dữ liệu để theo dõi số lần truy cập của từng trang → tăng độ phức tạp và tốn tài nguyên.
- Trang mới có thể bị loại bỏ sớm nếu chưa có cơ hội tăng tần suất truy cập (gọi là "cold start problem").
- Không phản ánh tốt tính thời gian gần đây (recency): một trang từng được dùng nhiều nhưng sau đó không còn dùng vẫn có thể được giữ lại rất lâu.

2.4.5. MFU

Giải thuật MFU (Most Frequently Used) là một thuật toán thay thế trang trái ngược với LFU. Thay vì chọn trang ít được sử dụng nhất để thay thế, MFU lại chọn thay thế trang được sử dụng nhiều nhất.

Nguyên lý hoạt động: MFU dựa trên giả thuyết rằng các trang có tần suất truy cập cao trong quá khứ có thể đã hoàn thành "nhiệm vụ" của chúng và ít có khả năng được truy cập trong tương lai gần. Khi cần thay thế trang, MFU chọn trang có bộ đếm tần suất cao nhất để loại bỏ. Mục tiêu là ưu tiên giữ lại những trang mới hoặc ít sử dụng hơn, với giả định chúng có khả năng cần thiết hơn.

Ưu điểm:

- Có thể hữu ích trong một số trường hợp đặc biệt khi các trang "hot" thực ra không cần giữ lâu.
- Đơn giản về ý tưởng, cũng như dễ kết hợp với các phương pháp khác.

Nhược điểm:

- Phần lớn thời gian, MFU không phù hợp với các mô hình truy cập thực tế vì nó thường loại bỏ những trang đang được dùng nhiều, gây ra nhiều lỗi trang không cần thiết.
- Khó dự đoán hiệu quả, thường cho kết quả kém hơn LFU hay LRU trong thực tế.
- Cũng cần bộ đếm tần suất như LFU, làm tăng chi phí quản lý bộ nhớ.

Chương 3: Thiết kế và cài đặt chương trình

3.1. Mục tiêu

Chương này tập trung vào việc xây dựng chương trình mô phỏng hoạt động của các giải thuật thay thế trang trong hệ điều hành. Mục tiêu chính của chương là:

- Phát triển các mô-đun chương trình bằng ngôn ngữ Python cho từng giải thuật thay thế trang được nghiên cứu.
- Chương trình sẽ nhận đầu vào gồm chuỗi truy cập trang và số lượng khung trang (frame) có sẵn trong bộ nhớ. Kết quả đầu ra là số lượng lỗi trang của từng thuật toán.
- Chương trình tạo hình ảnh minh họa trực quan cho quá trình thay thế trang ở từng bước, giúp dễ dàng hình dung cách các thuật toán quản lý bộ nhớ hoạt động trong thực tế.

Việc triển khai chương trình theo từng bước cụ thể nhằm đảm bảo tính chính xác trong quá trình mô phỏng, đồng thời giúp dễ dàng kiểm tra, mở rộng và tối ưu về sau.

3.2. Cài đặt chương trình

Các thành phần của chương trình được trình bày và phân tích theo trình tự dưới đây.

3.2.1. Xử lý hình ảnh và hiển thị

Phần này chịu trách nhiệm vẽ trực quan trạng thái bộ nhớ sau mỗi bước truy cập trang, lưu hình ảnh từng bước, rồi gom các hình ảnh thành một lưới để dễ quan sát kết quả.

Các thành phần chính:

- Tạo thư mục lưu ảnh
  Hàm clear_old_images(img_dir) kiểm tra nếu thư mục ảnh chưa tồn tại thì tạo mới, nếu tồn tại thì xóa các ảnh cũ .png để tránh trộn lẫn kết quả.
- Vẽ trạng thái bộ nhớ tại mỗi bước truy cập
  Hàm draw_it(...) dùng thư viện matplotlib để:

    Vẽ các ô hình chữ nhật biểu diễn các khung trang hiện tại trong bộ nhớ.

    Đánh dấu trang mới được thêm bằng màu cam, các trang khác màu xanh da trời.

Hiển thị văn bản chú giải phía trên để người đọc biết bước này đang xảy ra gì (ví dụ: trang mới vào, trang bị thay thế, số lỗi trang đến hiện tại...).

Lưu hình ảnh vào file nếu save_path được cung cấp.

- Ghép các hình ảnh nhỏ thành một hình ảnh lớn
  Hàm combine_images_to_grid(image_paths, output_path) gom các ảnh từng bước thành một lưới (grid) để tổng hợp kết quả trực quan dễ nhìn.
- Điều phối việc vẽ và ghép ảnh
  Hàm visualize_and_combine(...) gọi hàm vẽ cho từng bước mô phỏng, lưu từng ảnh, sau đó gọi hàm ghép ảnh thành lưới và xóa các ảnh nhỏ sau khi đã ghép xong.

Code:

```
import matplotlib.pyplot as plt
import textwrap
import math
import os
from PIL import Image

def clear_old_images(img_dir):
    if not os.path.exists(img_dir):
        os.makedirs(img_dir)
    elif not os.path.isdir(img_dir):
        raise NotADirectoryError(f"{img_dir} không phải là
thư mục!")
    else:
        for file in os.listdir(img_dir):
            if file.endswith('.png'):
                os.remove(os.path.join(img_dir, file))

def draw_it(lst, s, algo, pages, step, save_path=None):
    plt.figure(figsize=(max(6, len(lst) * 0.8), 4))
    ax = plt.gca()
    ax.set_aspect('equal')

    w, h = 0.8, 0.8
    spacing = 0.1

    wrapped_text = "\n".join(textwrap.wrap(f'Step {step}: '
+ s, width=80))
    ax.text(len(lst) * w / 2, 1.5, wrapped_text,
fontsize=11, ha='center', va='center', color='navy')
```

```
    for i, item in enumerate(lst):
        color = 'orange' if i == len(lst) - 1 else
'skyblue'
        rect = plt.Rectangle((i * (w + spacing), 0), w, h,
facecolor=color, edgecolor='black', linewidth=2)
        ax.add_patch(rect)
        ax.text(i * (w + spacing) + w / 2, h / 2,
str(item), ha='center', va='center', fontsize=12,
weight='bold')

    ax.set_xlim(-0.5, len(lst) * (w + spacing))
    ax.set_ylim(-2, 2)
    ax.axis('off')

    plt.title(f'Algorithm: {algo.upper()} | Page Sequence:
{pages}', fontsize=13, color='darkred', pad=20)

    if save_path:
        plt.savefig(save_path, bbox_inches='tight')
    plt.close()

def combine_images_to_grid(image_paths, output_path,
padding=10, bg_color=(255, 255, 255)):
    if not image_paths:
        return

    images = [Image.open(p) for p in image_paths]
    widths, heights = zip(*(img.size for img in images))
    max_width, max_height = max(widths), max(heights)

    n = math.ceil(math.sqrt(len(images)))

    grid_width = n * max_width + (n + 1) * padding
    grid_height = n * max_height + (n + 1) * padding
    combined = Image.new('RGB', (grid_width, grid_height),
bg_color)

    for index, img in enumerate(images):
        row, col = divmod(index, n)
        x = padding + col * (max_width + padding)
        y = padding + row * (max_height + padding)
        combined.paste(img, (x, y))
```

```
    combined.save(output_path)

def visualize_and_combine(algo, steps, pages, img_dir,
visualize=True):
    if not visualize:
        return
    image_files = []
    for i, (state, s) in enumerate(steps):
        img_file = os.path.join(img_dir,
f'{algo}_step_{i}.png')
        draw_it(lst=state, s=s, algo=algo.upper(),
pages=pages, step=i, save_path=img_file)
        image_files.append(img_file)

    output_grid = os.path.join(img_dir, f'{algo}_grid.png')
    combine_images_to_grid(image_files, output_grid)

    for img_path in image_files:
        if os.path.exists(img_path):
            os.remove(img_path)
```

3.2.2. Các giải thuật

Trước khi đi vào chi tiết từng giải thuật, cần lưu ý rằng trong mô phỏng này, mỗi truy cập trang được thực hiện tại một thời điểm riêng biệt. Tức là, chương trình xử lý tuần tự từng trang trong chuỗi truy cập, và tại mỗi bước, bộ nhớ được kiểm tra và cập nhật theo quy tắc của giải thuật thay thế trang đang được áp dụng.

- FIFO

Trang được đưa vào bộ nhớ đầu tiên sẽ là trang bị thay thế đầu tiên khi bộ nhớ đầy.

```
def fifo_victim(q, **_):
    return q[0]
```

- LRU

Loại bỏ trang lâu nhất không được truy cập gần đây nhất.

```
def lfu_victim(q, freq, last_used, **_):
    min_freq = min(freq[p] for p in q)
    candidates = [p for p in q if freq[p] == min_freq]
    return max(candidates, key=lambda p: last_used.get(p,
-1))
```

- OPT

Loại bỏ trang sẽ không được truy cập trong tương lai gần nhất, tức là trang mà thời điểm truy cập tiếp theo xa nhất.

```
def opt_victim(q, pages, i, **_):
    future = {p: next((j for j in range(i + 1,
len(pages)) if pages[j] == p), float('inf')) for p in q}
    return max(q, key=lambda p: future[p])
```

- LFU

Loại bỏ trang được truy cập ít nhất (tần suất thấp nhất).

```
def lfu_victim(q, freq, last_used, **_):
    min_freq = min(freq[p] for p in q)
    candidates = [p for p in q if freq[p] == min_freq]
    return max(candidates, key=lambda p: last_used.get(p,
-1))
```

- MFU

Loại bỏ trang được truy cập nhiều nhất, giả định rằng trang có tần suất truy cập cao có thể sắp ít được truy cập tiếp.

```
def mfu_victim(q, freq, last_used, **_):
    max_freq = max(freq[p] for p in q)
    candidates = [p for p in q if freq[p] == max_freq]
    return max(candidates, key=lambda p: last_used.get(p,
-1))
```

3.2.3. Mô phỏng (Simulate)

Phần này tạo một hàm mô phỏng việc truy cập từng trang theo chuỗi truy cập, đồng thời vận hành các giải thuật thay thế trang đã chọn.

Ở mỗi bước truy cập:

- Kiểm tra trang có trong bộ nhớ hay không
- Nếu không có và bộ nhớ đầy, dùng giải thuật tương ứng để chọn trang cần thay thế
- Thêm trang mới vào bộ nhớ
- Cập nhật tần suất và thời điểm truy cập cuối cùng của từng trang
- Ghi lại trạng thái bộ nhớ và mô tả diễn biến ở bước đó để dùng cho việc hiển thị trực quan

Code:

```
def simulate(pages, frame_size, algo):
    q, steps, faults = [], [], 0
    freq, last_used = {}, {}
    victim_funcs = {
        'fifo': fifo_victim,
        'opt': opt_victim,
        'lru': lru_victim,
        'lfu': lfu_victim,
        'mfu': mfu_victim
    }
    victim_fn = victim_funcs[algo]

    for i, page in enumerate(pages):
        s = f'Next page: {page}\n'
        if page not in q:
            s += f'{page} not in memory.\n'
            if len(q) == frame_size:
                faults += 1
                victim = victim_fn(q, pages=pages, i=i,
freq=freq, last_used=last_used)
                q.remove(victim)
                s += f'Memory full. Removed {victim}.\n'
            q.append(page)
            s += f'Added {page}.\n'
```

```
        else:
            s += f'{page} already in memory.\n'
        freq[page] = freq.get(page, 0) + 1
        last_used[page] = i
        s += f'Page faults so far: {faults}\n'
        steps.append((list(q), s))
    return faults, steps
```

3.2.4. Chương trình chính

Là điểm khởi đầu, điều phối toàn bộ luồng chạy của chương trình: từ đọc dữ liệu, thực thi mô phỏng các giải thuật, hiển thị kết quả đến ghi file đầu ra.

Nhiệm vụ:

- Đọc dữ liệu đầu vào từ file JSON (chuỗi truy cập trang, kích thước khung, danh sách giải thuật, cờ hiển thị hình ảnh)
- Tạo thư mục lưu trữ hình ảnh (nếu chưa có)
- Xóa các ảnh cũ trong thư mục để tránh lẫn lộn
- Gọi hàm mô phỏng cho từng giải thuật, thu thập kết quả
- Gọi hàm hiển thị và kết hợp hình ảnh minh họa từng bước của các giải thuật
- Ghi kết quả số lỗi trang của từng giải thuật ra file JSON đầu ra

Code:

```
def run_simulation(input_file):
    BASE_DIR = os.path.dirname(os.path.abspath(__file__))
    IMG_DIR = os.path.join(BASE_DIR, 'img')
    OUTPUT_DIR = os.path.join(BASE_DIR, 'output.json')

    if not os.path.exists(IMG_DIR):
        os.makedirs(IMG_DIR)

    with open(input_file, 'r') as f:
        data = json.load(f)
    pages = data.get('pages', [])
    frame_size = data.get('frame_size', 3)
    algorithms = data.get('algorithms', ['fifo', 'opt',
'lru', 'lfu', 'mfu'])
```

```
    visualize = data.get('visualize', True)

    if not pages or not all(isinstance(p, int) for p in
pages):
        return
    if frame_size <= 0 or not isinstance(frame_size, int):
        return

    results = {}
    clear_old_images(IMG_DIR)

    for algo in algorithms:
        faults, steps = simulate(pages, frame_size, algo)
        results[algo] = {'faults': faults}
        visualize_and_combine(algo, steps, pages, IMG_DIR,
visualize=visualize)

    with open(OUTPUT_DIR, 'w') as f:
        json.dump(results, f)

# Chạy mô phỏng chính
INPUT_DIR =
os.path.join(os.path.dirname(os.path.abspath(__file__)),
'input.json')
print(f'Input_dir = {INPUT_DIR}')
run_simulation(INPUT_DIR)
```

Chương 4: Thực nghiệm và đánh giá

4.1. Mục tiêu

Chương này nhằm đánh giá hiệu quả của các giải thuật thay thế trang bộ nhớ đã được triển khai ở chương trước, bao gồm: FIFO, LRU, OPT, LFU và MFU. Thông qua việc thực hiện chương trình với các tổ hợp đầu vào khác nhau, chương trình mô phỏng sẽ cung cấp số lỗi trang tương ứng của từng giải thuật.

Mục tiêu chính gồm:

- Thực thi chương trình với các chuỗi truy cập trang và số lượng khung trang khác nhau, đại diện cho các tình huống truy cập bộ nhớ đa dạng.
- Thu thập kết quả từ quá trình mô phỏng, bao gồm số lỗi trang ứng với từng thuật toán trong mỗi tổ hợp đầu vào.
- Trực quan hóa dữ liệu thông qua bảng số liệu và biểu đồ cột để so sánh hiệu suất các thuật toán.
- Đánh giá định tính và định lượng từng thuật toán dựa trên kết quả thu được, từ đó rút ra các nhận định về tính hiệu quả, điểm mạnh và hạn chế của từng thuật toán.

Qua đó, chương này đóng vai trò quan trọng trong việc liên kết giữa lý thuyết và thực tiễn, giúp người đọc hiểu sâu hơn về hành vi của từng thuật toán trong các điều kiện khác nhau.

4.2. Mô phỏng

4.2.1. Các tổ hợp đầu vào

Chúng tôi lựa chọn 6 tổ hợp đầu vào gồm các chuỗi truy cập trang khác nhau kết hợp với số khung trang tương ứng để đánh giá hiệu quả các giải thuật thay thế trang. Mỗi tổ hợp đầu vào được mô tả như sau:

Tên tổ hợp	Chuỗi truy cập trang	Số khung trang (Frames)
Tổ hợp 1	1,2,3,4,1,2,5,1,2,3	3
Tổ hợp 2	7,0,1,2,0,3,0,4,2,3	4
Tổ hợp 3	1,2,3,4,5,6,7,8,9,10	5
Tổ hợp 4	2,3,2,1,5,2,4,5,3,2	3
Tổ hợp 5	3,2,1,0,3,2,4,3,2,1	4
Tổ hợp 6	0,4,1,4,2,4,3,4,2,4	3

Chú ý: Chuỗi truy cập trang được lựa chọn sao cho bao quát được các kiểu truy cập phổ biến như truy cập tuần tự, truy cập lặp lại, truy cập ngẫu nhiên,... nhằm đánh giá toàn diện hiệu năng của các giải thuật thay thế trang.

4.2.2. Thực hiện

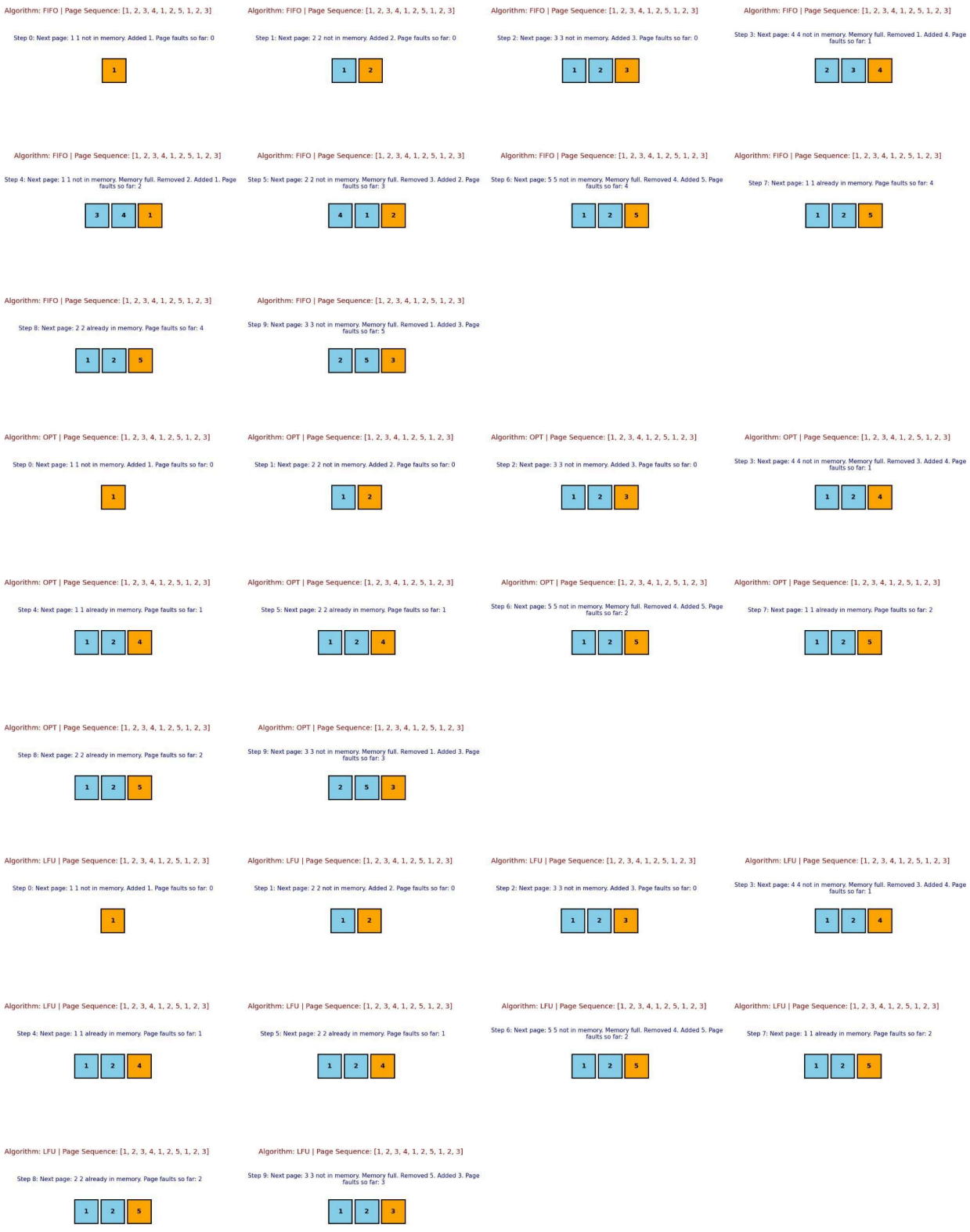
Để đảm bảo tính khách quan và nhất quán trong quá trình đánh giá, chúng tôi đã thực hiện chạy mô phỏng từng giải thuật thay thế trang (FIFO, LRU, OPT, LFU, MFU) với tất cả 6 tổ hợp đầu vào nêu trên.

Quá trình thực hiện bao gồm các bước sau:

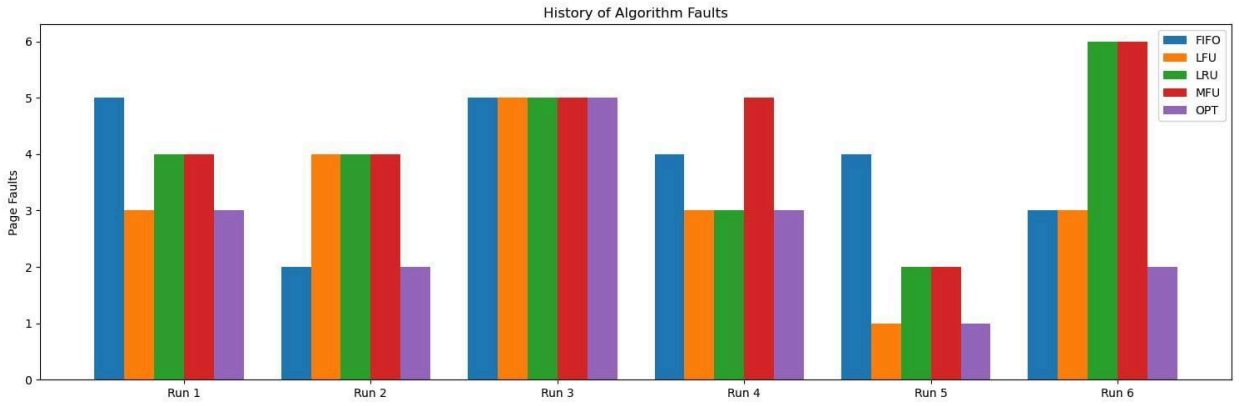
- Nhập từng chuỗi truy cập trang và số khung trang cho từng tổ hợp vào file input.jsol
- Chạy chương trình (bằng VS Code,…)
- Ghi nhận và tổng hợp kết quả lỗi trang cho từng tổ hợp và giải thuật.

Xét ví dụ đầu tiên, ta có đầu ra là file output.jsol và các hình ảnh minh họa trong file img:

```
Log Output:

Algorithm: FIFO
Page faults: 5

Algorithm: LFU
Page faults: 3

Algorithm: LRU
Page faults: 4

Algorithm: MFU
Page faults: 4

Algorithm: OPT
Page faults: 3
```


Algorithm: FIFO | Page Sequence: [1, 2, 3, 4, 1, 2, 5, 1, 2, 3]
Step 0: Next page: 1 1 not in memory. Added 1. Page faults so far: 0
1
Algorithm: FIFO | Page Sequence: [1, 2, 3, 4, 1, 2, 5, 1, 2, 3]
Step 1: Next page: 2 2 not in memory. Added 2. Page faults so far: 0
1 2
Algorithm: FIFO | Page Sequence: [1, 2, 3, 4, 1, 2, 5, 1, 2, 3]
Step 2: Next page: 3 3 not in memory. Added 3. Page faults so far: 0
1 2 3
Algorithm: FIFO | Page Sequence: [1, 2, 3, 4, 1, 2, 5, 1, 2, 3]
Step 3: Next page: 4 4 not in memory. Memory full. Removed 1. Added 4. Page faults so far: 1
2 3 4
Algorithm: FIFO | Page Sequence: [1, 2, 3, 4, 1, 2, 5, 1, 2, 3]
Step 4: Next page: 1 1 not in memory. Memory full. Removed 2. Added 1. Page faults so far: 2
3 4 1
Algorithm: FIFO | Page Sequence: [1, 2, 3, 4, 1, 2, 5, 1, 2, 3]
Step 5: Next page: 2 2 not in memory. Memory full. Removed 3. Added 2. Page faults so far: 3
4 1 2
Algorithm: FIFO | Page Sequence: [1, 2, 3, 4, 1, 2, 5, 1, 2, 3]
Step 6: Next page: 5 5 not in memory. Memory full. Removed 4. Added 5. Page faults so far: 4
1 2 5
Algorithm: FIFO | Page Sequence: [1, 2, 3, 4, 1, 2, 5, 1, 2, 3]
Step 7: Next page: 1 1 already in memory. Page faults so far: 4
1 2 5
Algorithm: FIFO | Page Sequence: [1, 2, 3, 4, 1, 2, 5, 1, 2, 3]
Step 8: Next page: 2 2 already in memory. Page faults so far: 4
1 2 5
Algorithm: FIFO | Page Sequence: [1, 2, 3, 4, 1, 2, 5, 1, 2, 3]
Step 9: Next page: 3 3 not in memory. Memory full. Removed 1. Added 3. Page faults so far: 5
2 5 3
Algorithm: OPT | Page Sequence: [1, 2, 3, 4, 1, 2, 5, 1, 2, 3]
Step 0: Next page: 1 1 not in memory. Added 1. Page faults so far: 0
1
Algorithm: OPT | Page Sequence: [1, 2, 3, 4, 1, 2, 5, 1, 2, 3]
Step 1: Next page: 2 2 not in memory. Added 2. Page faults so far: 0
1 2
Algorithm: OPT | Page Sequence: [1, 2, 3, 4, 1, 2, 5, 1, 2, 3]
Step 2: Next page: 3 3 not in memory. Added 3. Page faults so far: 0
1 2 3
Algorithm: OPT | Page Sequence: [1, 2, 3, 4, 1, 2, 5, 1, 2, 3]
Step 3: Next page: 4 4 not in memory. Memory full. Removed 3. Added 4. Page faults so far: 1
1 2 4
Algorithm: OPT | Page Sequence: [1, 2, 3, 4, 1, 2, 5, 1, 2, 3]
Step 4: Next page: 1 1 already in memory. Page faults so far: 1
1 2 4
Algorithm: OPT | Page Sequence: [1, 2, 3, 4, 1, 2, 5, 1, 2, 3]
Step 5: Next page: 2 2 already in memory. Page faults so far: 1
1 2 4
Algorithm: OPT | Page Sequence: [1, 2, 3, 4, 1, 2, 5, 1, 2, 3]
Step 6: Next page: 5 5 not in memory. Memory full. Removed 4. Added 5. Page faults so far: 2
1 2 5
Algorithm: OPT | Page Sequence: [1, 2, 3, 4, 1, 2, 5, 1, 2, 3]
Step 7: Next page: 1 1 already in memory. Page faults so far: 2
1 2 5
Algorithm: OPT | Page Sequence: [1, 2, 3, 4, 1, 2, 5, 1, 2, 3]
Step 8: Next page: 2 2 already in memory. Page faults so far: 2
1 2 5
Algorithm: OPT | Page Sequence: [1, 2, 3, 4, 1, 2, 5, 1, 2, 3]
Step 9: Next page: 3 3 not in memory. Memory full. Removed 1. Added 3. Page faults so far: 3
2 5 3
Algorithm: LFU | Page Sequence: [1, 2, 3, 4, 1, 2, 5, 1, 2, 3]
Step 0: Next page: 1 1 not in memory. Added 1. Page faults so far: 0
1
Algorithm: LFU | Page Sequence: [1, 2, 3, 4, 1, 2, 5, 1, 2, 3]
Step 1: Next page: 2 2 not in memory. Added 2. Page faults so far: 0
1 2
Algorithm: LFU | Page Sequence: [1, 2, 3, 4, 1, 2, 5, 1, 2, 3]
Step 2: Next page: 3 3 not in memory. Added 3. Page faults so far: 0
1 2 3
Algorithm: LFU | Page Sequence: [1, 2, 3, 4, 1, 2, 5, 1, 2, 3]
Step 3: Next page: 4 4 not in memory. Memory full. Removed 3. Added 4. Page faults so far: 1
1 2 4
Algorithm: LFU | Page Sequence: [1, 2, 3, 4, 1, 2, 5, 1, 2, 3]
Step 4: Next page: 1 1 already in memory. Page faults so far: 1
1 2 4
Algorithm: LFU | Page Sequence: [1, 2, 3, 4, 1, 2, 5, 1, 2, 3]
Step 5: Next page: 2 2 already in memory. Page faults so far: 1
1 2 4
Algorithm: LFU | Page Sequence: [1, 2, 3, 4, 1, 2, 5, 1, 2, 3]
Step 6: Next page: 5 5 not in memory. Memory full. Removed 4. Added 5. Page faults so far: 2
1 2 5
Algorithm: LFU | Page Sequence: [1, 2, 3, 4, 1, 2, 5, 1, 2, 3]
Step 7: Next page: 1 1 already in memory. Page faults so far: 2
1 2 5
Algorithm: LFU | Page Sequence: [1, 2, 3, 4, 1, 2, 5, 1, 2, 3]
Step 8: Next page: 2 2 already in memory. Page faults so far: 2
1 2 5
Algorithm: LFU | Page Sequence: [1, 2, 3, 4, 1, 2, 5, 1, 2, 3]
Step 9: Next page: 3 3 not in memory. Memory full. Removed 5. Added 3. Page faults so far: 3
1 2 3

Biểu đồ cột sau đây so sánh số lỗi trang của các giải thuật đối với 6 tổ hợp đầu vào:



4.3. Đánh giá

Tổng thể, ta nhận thấy sự đánh đổi giữa độ phức tạp cài đặt và hiệu quả thuật toán. OPT lý tưởng nhưng không thực tế; LRU hiệu quả và khả thi trong triển khai; FIFO đơn giản nhưng kém hiệu quả; LFU và MFU có thể phù hợp trong những trường hợp đặc biệt nhưng khó tối ưu tổng quát.

FIFO:

- FIFO thường có số lỗi trang cao hơn so với các giải thuật thông minh hơn như LRU, OPT, LFU trong tất cả các tổ hợp đầu vào. Điều này chứng tỏ FIFO chưa tối ưu trong việc dự đoán và giữ lại các trang cần thiết, dẫn đến hiệu quả bộ nhớ thấp hơn.

- Hiện tượng Belady (nghịch lý tăng số lỗi trang khi tăng số khung trang) có thể xảy ra với FIFO, bởi đặc tính đơn giản và không xét đến tần suất hay thời gian truy cập các trang. Điều này làm giảm độ tin cậy và hiệu quả của FIFO trong các hệ thống thực tế khi số khung trang thay đổi.
- Mặc dù vậy, ưu điểm của FIFO là dễ hiểu, dễ cài đặt và tốn ít tài nguyên xử lý, phù hợp với các hệ thống hạn chế về phần cứng hoặc yêu cầu đơn giản.

Tóm lại, FIFO không phải là lựa chọn tối ưu cho các hệ thống cần hiệu suất cao và ổn định, nhưng nó vẫn đóng vai trò quan trọng trong việc hiểu cơ chế thay thế trang và làm chuẩn so sánh với các giải thuật khác.

LRU:

- LRU có **số lỗi trang thấp hơn rõ rệt so với FIFO** trong tất cả các tổ hợp đầu vào, cho thấy hiệu quả vượt trội trong việc duy trì các trang cần thiết và giảm thiểu số lỗi trang.
- So với giải thuật tối ưu OPT, LRU vẫn có số lỗi trang cao hơn một chút, nhưng sự chênh lệch không lớn, khẳng định LRU là giải thuật thay thế trang gần tối ưu nhất trong thực tế, vì OPT đòi hỏi biết trước tương lai truy cập trang.
- LRU tránh được hiện tượng Belady, nhờ khả năng thích ứng với các mẫu truy cập trang dựa trên lịch sử sử dụng, giúp duy trì tính ổn định khi số khung trang thay đổi.
- Nhược điểm của LRU là chi phí tính toán và quản lý cao hơn FIFO do cần theo dõi và cập nhật thứ tự truy cập trang. Điều này có thể đòi hỏi tài nguyên phần cứng hoặc phần mềm bổ sung.

Ta thấy LRU cân bằng tốt giữa hiệu quả giảm lỗi trang và độ phức tạp thực thi, là lựa chọn phù hợp cho nhiều hệ điều hành và ứng dụng thực tế.

OPT:

- OPT đạt **số lỗi trang thấp nhất trong tất cả các tổ hợp đầu vào**, minh chứng cho tính tối ưu tuyệt đối của nó trong lý thuyết.Do yêu cầu biết trước toàn bộ chuỗi truy cập trang tương lai, OPT không khả thi để triển khai trực tiếp trong hệ thống thực tế, nhưng vẫn được dùng làm chuẩn so sánh cho các giải thuật khác.
- So với LRU, OPT ít lỗi trang hơn, nhưng sự khác biệt không quá lớn, cho thấy LRU là một giải thuật rất gần với tối ưu.

- OPT không gặp hiện tượng Belady bởi vì nó có tầm nhìn “tương lai” trong việc chọn trang để thay thế, giúp tránh thay thế nhầm trang có thể sắp được sử dụng.

Như vậy OPT là giải thuật mẫu mực về hiệu quả, thường dùng để đánh giá các thuật toán thay thế trang khác, nhưng vì không thực tế trong môi trường vận hành thông thường, nó chủ yếu là thước đo lý thuyết cho hiệu suất tối ưu nhất.

LFU:

- Số lỗi trang của LFU nhìn chung cao hơn so với OPT và LRU, nhưng thấp hơn FIFO và MFU trong phần lớn các tổ hợp đầu vào.
- LFU hoạt động tốt trong những trường hợp chuỗi truy cập có sự lặp lại ổn định của các trang được dùng thường xuyên, vì nó ưu tiên giữ lại những trang được truy cập nhiều.
- Tuy nhiên, LFU có thể bị "kẹt" với các trang cũ đã từng được dùng nhiều nhưng hiện tại không còn cần thiết nữa (hiện tượng giữ lại trang "lỗi thời"), làm giảm hiệu quả trong một số trường hợp.
- So với OPT và LRU, LFU chưa thực sự tối ưu trong việc giảm lỗi trang vì nó chỉ dựa vào tần suất, không xét đến thời gian gần đây nhất của các lần truy cập.
- Không gặp hiện tượng Belady, nhưng do cách đánh giá tần suất lâu dài, LFU có thể chậm thích nghi với các thay đổi đột ngột trong mô hình truy cập trang.

LFU là giải thuật có hiệu quả trung bình khá, phù hợp với môi trường truy cập ổn định, tuy nhiên vẫn còn hạn chế về khả năng thích ứng nhanh với các truy cập thay đổi.

MFU:

- MFU có số lỗi trang cao nhất trong tất cả các thuật toán thử nghiệm, chứng tỏ hiệu quả thấp hơn rõ rệt.
- Lý do là MFU thường ưu tiên loại bỏ những trang đang được dùng nhiều, điều này đi ngược lại với nguyên tắc giữ lại các trang quan trọng.

Kết quả cho thấy MFU không phù hợp với mô hình truy cập trang phổ biến hiện nay, khi mà các trang được truy cập nhiều thường vẫn còn cần thiết trong thời gian tới. MFU là thuật toán có hiệu quả kém nhất trong bộ thử nghiệm, không được khuyến khích sử dụng trong môi trường quản lý bộ nhớ ảo thực tế.

Chương 5: Kết luận và hướng phát triển

5.1. Kết luận

Qua việc thiết kế, cài đặt và mô phỏng trên các tổ hợp đầu vào cụ thể, ta đã đánh giá được ưu nhược điểm, hiệu suất của từng thuật toán thông qua số lỗi trang.

Kết quả thực nghiệm cho thấy:

- Giải thuật OPT đạt hiệu quả tối ưu nhất về số lỗi trang nhưng khó áp dụng thực tế do yêu cầu biết trước các lần truy cập tương lai.
- LRU là giải thuật có hiệu suất tốt, phù hợp với nhiều trường hợp thực tế, nhờ khả năng lựa chọn trang thay thế dựa trên lịch sử truy cập gần nhất.
- FIFO đơn giản nhưng dễ gặp hiện tượng Belady làm giảm hiệu quả.
- LFU và MFU ít được ứng dụng vì không thích nghi tốt với mô hình truy cập trang phổ biến, đặc biệt MFU cho kết quả kém nhất.

Việc mô phỏng và so sánh giúp hiểu sâu hơn về nguyên lý hoạt động và tác động của từng giải thuật đối với quản lý bộ nhớ trong hệ điều hành.

5.2. Hướng phát triển

Tiếp tục nghiên cứu và thử nghiệm các thuật toán thay thế trang nâng cao, như thuật toán lai hoặc dựa trên trí tuệ nhân tạo, để cải thiện hiệu quả xử lý lỗi trang trong các môi trường phức tạp hơn.

Phát triển giao diện người dùng trực quan, thân thiện cho chương trình mô phỏng, giúp người dùng dễ dàng nhập dữ liệu, quan sát kết quả và so sánh hiệu năng của các thuật toán.

Mở rộng chương trình để hỗ trợ đa dạng dạng đầu vào thực tế, đồng thời tích hợp các công cụ phân tích, biểu đồ trực quan giúp đánh giá thuật toán nhanh chóng và hiệu quả.

DANH MỤC TÀI LIỆU THAM KHẢO

1. Bài giảng học phần Nguyên lý Hệ điều hành - thầy Đỗ Quốc Huy.
2. Nguyên lý Hệ điều hành (NXB Giáo Dục Việt Nam) - Hồ Đắc Phương.ss
3. Wikipedia.com
4. geeksforgeeks.com
5. *Operating System Concepts* - Abraham Silberschatz, Peter Baer Galvin và Greg Gagne.