TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
ĐẠI HỌC QUỐC GIA TP.HCM

KHOA CÔNG NGHỆ THÔNG TIN

ASSIGNMENT 02.01

HỆ ĐIỀU HÀNH

Lớp: 21CLC03

Sinh viên: Trần Nguyên Huân – 21127050

Nguyễn Hoàng Phúc – 21127671

Giảng viên hướng dẫn: Thái Hùng Văn, Đặng Trần Minh Hậu

THÀNH PHỐ HỒ CHÍ MINH - 2023

Contents

I. Bảng phân công công việc, mức độ hoàn thành và các tiêu chí làm được

I.1. Bảng phân công công việc và mức độ hoàn thành

Họ và tên	MSSV	Công việc thực hiện	Mức độ hoàn thành
Trần Nguyên Huân	21127050	- Viết báo cáo	100%
Nguyễn Hoàng Phúc	21127671	- Code chương trình demo	100%

I.2. Các tiêu chí làm được

	Tiêu chí	Tỉ lệ hoàn thành
1	Trình bày tổng quan về deadlock và critical section	100%
2	So sánh / đánh giá các giải pháp xử lí deadlock	100%
3	Nêu các trạng thái cần tránh trong deadlock	100%
4	Trình bày một số thuật toán trong Deadlock Prevention và Deadlock Avoidance	100%
5	So sánh / đánh giá các giải pháp xử lí Critical Section	100%
6	Chương trình mô phỏng thuật toán Banker (giải quyết Deadlock)	100%
7	Tổng kết chương trình và đánh giá điểm mạnh, điểm yếu của deadlock và hướng khắc phục của thuật toán	100%

❖ Chú thích:

- Trong báo cáo này nhóm em làm phần chương trình demo về xử lí Deadlock nên phần lí thuyết nhóm em trình bày phần Deadlock có phần chi tiết hơn Critical Section.

- Chương trình có thể chạy trên Microsoft Visual Studio (2022), các IDE còn lại thì có thể sẽ không chạy được.

II. Tổng quan về Deadlock và Critical Section

II.1. Deadlock

- Deadlock là một trạng thái của hệ thống trong đó các tiến trình hoặc luồng đang chờ đợi tài nguyên từ các tiến trình khác mà không ai sẵn sàng nhượng bộ tài nguyên của mình. Deadlock là một vấn đề phức tạp trong hệ thống máy tính và có thể xảy ra khi các tiến trình hoặc luồng cố gắng truy cập cùng một tài nguyên hoặc tài nguyên khác nhau mà không thể hoàn tất nhiệm vụ của chúng.

- Các tài nguyên bao gồm CPU, bộ nhớ, đĩa cứng, bàn phím, màn hình và các thiết bị ngoại vi khác. Khi các tiến trình cố gắng truy cập vào các tài nguyên này, có thể xảy ra tình trạng chặn đầu cuối (deadlock) khi một tiến trình yêu cầu tài nguyên của tiến trình khác mà không nhường tài nguyên của mình. Khi điều này xảy ra, các tiến trình sẽ chờ đợi tài nguyên của nhau và không thể tiếp tục thực thi. Nếu deadlock xảy ra, hệ thống sẽ dừng lại và không thể hoàn thành nhiệm vụ được yêu cầu.

- Để xử lý deadlock, các hệ thống máy tính cung cấp các giải pháp như giải pháp bắt buộc tài nguyên (resource allocation), giải pháp thu hồi tài nguyên (resource deallocation) và giải pháp giải quyết theo thời gian (timeouts). Việc xử lý deadlock là một phần quan trọng của quản lý tài nguyên trong hệ thống máy tính, giúp đảm bảo rằng các tiến trình có thể truy cập tài nguyên một cách an toàn và hiệu quả.

II.2. Critical Section

- Trong hệ điều hành, critical section là một khối lệnh trong một tiến trình hoặc luồng mà các tiến trình hoặc luồng khác không được phép truy cập vào cùng một lúc. Critical section được sử dụng để đảm bảo tính nhất quán và đúng đắn của các tài nguyên chia sẻ giữa các tiến trình hoặc luồng khác nhau.

- Để đảm bảo tính đúng đắn của critical section, hệ điều hành cung cấp các cơ chế đồng bộ hóa như mutex, semaphore, monitor, hoặc spinlock. Các cơ chế này đảm bảo rằng chỉ một tiến trình hoặc luồng được phép truy cập vào critical section tại một thời điểm. Nếu một tiến trình hoặc luồng đang thực thi trong critical section, các tiến trình hoặc luồng khác phải đợi cho đến khi tiến trình hoặc luồng đang thực thi kết thúc.

- Các cơ chế đồng bộ hóa này được sử dụng trong các ứng dụng như đồ họa, đa phương tiện, hệ thống phân tán và các ứng dụng mạng. Việc sử dụng các cơ chế đồng bộ

hóa giúp đảm bảo rằng các tiến trình hoặc luồng sẽ không xảy ra xung đột dữ liệu (data race) hoặc tình trạng bất đồng bộ (inconsistency) trong việc truy cập các tài nguyên chia sẻ.

- Tuy nhiên, việc sử dụng các cơ chế đồng bộ hóa có thể gây ra tình trạng chậm trễ trong việc thực thi chương trình, vì các tiến trình hoặc luồng phải đợi cho đến khi tiến trình hoặc luồng khác kết thúc trong critical section. Do đó, việc quản lý và sử dụng các cơ chế đồng bộ hóa là một phần quan trọng trong việc phát triển phần mềm đa tiến trình hoặc đa luồng trên các hệ điều hành.

III. So sánh và đánh giá các giải pháp xử lí Deadlock và Critical Section

III.1. Các giải pháp xử lí Deadlock

Giải pháp	**Mô tả**	**Ưu điểm**	**Khuyết điểm**
Chặn và phát hiện (Prevention and Detection)	Hệ điều hành sẽ theo dõi tài nguyên và các tiến trình để phát hiện deadlock	Dễ triển khai và hiệu quả trong nhiều trường hợp	Chi phí và tài nguyên phải được sử dụng để theo dõi tài nguyên và tiến trình
Phòng tránh deadlock (Avoidance)	Hệ điều hành sử dụng các thuật toán để tránh tình trạng deadlock	Hiệu quả và đáng tin cậy trong nhiều trường hợp	Có thể dẫn đến sự trì hoãn và ảnh hưởng đến hiệu suất hệ thống
Giải quyết deadlock (Resolution)	Hệ điều hành giải quyết tình trạng deadlock khi nó xảy ra	Hiệu quả trong việc giải quyết vấn đề.	Chi phí cao và ảnh hưởng đến hiệu suất hệ thống
Thông báo và khôi phục (Notification and Recovery)	Hệ điều hành tự động khởi động lại các tiến trình bị treo để khắc phục tình trạng deadlock	Đảm bảo hệ thống luôn hoạt động và không bị gián đoạn	Ảnh hưởng đến hiệu suất hệ thống và dữ liệu đang được xử lý có thể bị mất
Đảo ngược tài nguyên (Resource preemption)	Hệ điều hành có thể giải phóng tài nguyên của một tiến trình bị treo để cấp cho tiến trình khác.	Không ảnh hưởng đến hiệu suất hệ thống và không làm mất dữ liệu.	Có thể gây ra sự bất công cho tiến trình bị giải phóng tài nguyên. Nếu tài nguyên được giải phóng quá nhiều lần, có thể dẫn đến

			sự trì hoãn dài hơn và làm giảm hiệu suất hệ thống.

- Mỗi giải pháp xử lý deadlock trong hệ điều hành đều có ưu điểm và nhược điểm riêng, và tùy thuộc vào tình huống cụ thể, các giải pháp này có thể được kết hợp để đạt được hiệu quả cao nhất. Tuy nhiên, có một vài đánh giá tổng quan về các giải pháp này như sau:
    - Giải pháp chặn và phát hiện (Prevention and Detection): Giải pháp này đảm bảo rằng deadlock không xảy ra, nhưng nó có thể tốn nhiều tài nguyên và có thể làm giảm hiệu suất hệ thống. Tuy nhiên, nếu đảm bảo rằng các tiến trình không thể bị treo do deadlock, hệ thống sẽ hoạt động ổn định và hiệu quả.
    - Giải pháp thông báo và khôi phục (Notification and Recovery): Giải pháp này tập trung vào việc phát hiện và khôi phục deadlock. Nó có thể được sử dụng để giải quyết deadlock một cách nhanh chóng và hiệu quả, nhưng cần phải cẩn thận để đảm bảo rằng không có dữ liệu nào bị mất trong quá trình khôi phục.
    - Giải pháp phân giải mâu thuẫn (Avoidance): Giải pháp này được thiết kế để tránh xảy ra deadlock bằng cách giữ cho các tiến trình đang chạy trong trạng thái an toàn. Tuy nhiên, giải pháp này có thể dẫn đến trì hoãn và làm giảm hiệu suất hệ thống.
    - Giải pháp đảo ngược tài nguyên (Resource preemption): Giải pháp này có thể giải quyết deadlock bằng cách giải phóng tài nguyên của tiến trình treo và cấp phát cho các tiến trình khác. Tuy nhiên, giải pháp này có thể gây ra sự bất công và làm giảm hiệu suất hệ thống nếu tài nguyên được giải phóng quá nhiều lần.

III.2. Deadlock Prevention

III.2.1. Trạng thái mutual exclusion

Mutual exclusion là một trạng thái trong hệ thống khi hai hoặc nhiều tiến trình đồng thời yêu cầu truy cập vào cùng một tài nguyên và không thể thực hiện được vì tài nguyên đó chỉ có thể được sử dụng bởi một tiến trình tại một thời điểm. Để tránh tình trạng mutual exclusion, cần thỏa mãn các điều kiện sau đây:

- Tài nguyên phải được cấp cho một tiến trình duy nhất vào một thời điểm: Điều này đảm bảo rằng chỉ có một tiến trình được truy cập vào tài nguyên đó tại một thời điểm, do đó tránh được trường hợp hai tiến trình yêu cầu truy cập vào tài nguyên cùng một lúc.

- Tiến trình không được giữ tài nguyên mà không sử dụng: Điều này đảm bảo rằng tài nguyên sẽ được sử dụng bởi các tiến trình khác khi nó không được sử dụng bởi tiến trình hiện tại. Nếu tiến trình giữ tài nguyên mà không sử dụng, nó sẽ gây lãng phí tài nguyên và có thể làm giảm hiệu suất của hệ thống.
- Tiến trình không được giữ tài nguyên trong khi đợi tài nguyên khác: Điều này đảm bảo rằng các tiến trình không sẽ giữ tài nguyên mà không sử dụng, trong khi đang chờ đợi tài nguyên khác. Nếu các tiến trình giữ các tài nguyên mà không sử dụng trong khi đang chờ đợi các tài nguyên khác, nó có thể gây deadlock.

- Các kĩ thuật để tránh mutual exclusion xảy ra trong hệ thống

- Sử dụng khóa (lock): Phương pháp này sử dụng một biến khóa (lock) để đồng bộ hóa truy cập vào tài nguyên chung. Khi một tiến trình muốn sử dụng tài nguyên, nó phải yêu cầu khóa trước khi truy cập. Nếu khóa đang được sử dụng bởi một tiến trình khác, tiến trình đang yêu cầu phải đợi cho đến khi khóa được giải phóng. Khi một tiến trình đã sử dụng xong tài nguyên, nó sẽ giải phóng khóa để cho tiến trình khác sử dụng.
- Sử dụng biến cờ (flag): Phương pháp này sử dụng một biến cờ (flag) để đồng bộ hóa truy cập vào tài nguyên chung. Khi một tiến trình muốn sử dụng tài nguyên, nó sẽ yêu cầu sử dụng biến cờ và đặt giá trị của biến này thành true để báo hiệu rằng tài nguyên đang được sử dụng bởi tiến trình này. Khi tiến trình đã sử dụng xong tài nguyên, nó sẽ đặt giá trị của biến cờ thành false để cho phép các tiến trình khác sử dụng.
- Sử dụng semaphores
  - Semaphore là một biến đồng bộ hóa được sử dụng để đồng bộ hóa truy cập vào tài nguyên. Semaphore bao gồm một giá trị nguyên và hai hoạt động đồng bộ hóa là P (proberen) và V (verhogen). Khi một tiến trình muốn truy cập tài nguyên, nó sẽ gọi hoạt động P để giảm giá trị của semaphore. Nếu giá trị của semaphore là 0, tiến trình sẽ bị khóa cho đến khi semaphore trở thành khả dụng.
  - Khi tiến trình sử dụng xong tài nguyên, nó sẽ gọi hoạt động V để tăng giá trị của semaphore lên một đơn vị, cho phép các tiến trình khác sử dụng tài nguyên. Semaphore còn được sử dụng để đồng

bộ hóa truy cập vào các khu vực nhớ, thực hiện đồng bộ hóa giữa các tiến trình, xử lý báo hiệu và nhiều hơn nữa.

- Có hai loại semaphore chính là binary semaphore và counting semaphore. Binary semaphore có giá trị chỉ bằng 0 hoặc 1 và được sử dụng để đồng bộ hóa truy cập vào tài nguyên đơn lẻ. Counting semaphore có giá trị bằng số nguyên không âm và được sử dụng để đồng bộ hóa truy cập vào nhiều tài nguyên.
- Tuy nhiên, sử dụng semaphores cần phải chú ý để tránh các vấn đề như deadlock hoặc livelock. Chẳng hạn, nếu một tiến trình yêu cầu semaphore và bị khóa, nhưng không giải phóng semaphore, các tiến trình khác sẽ bị chặn và hệ thống sẽ bị deadlock. Hoặc nếu một tiến trình giải phóng semaphore quá nhanh, semaphore sẽ trở nên không đồng bộ và gây ra livelock. Do đó, việc sử dụng semaphores cần được thực hiện cẩn thận và có kế hoạch đầy đủ để tránh các vấn đề trên.

III.2.2. Trạng thái “Hold and Wait”

- Để đảm bảo rằng điều kiện "hold-and-wait" không bao giờ xảy ra trong hệ thống, chúng ta phải đảm bảo rằng khi một tiến trình yêu cầu một tài nguyên, nó không giữ bất kỳ tài nguyên nào khác. Một giao thức mà chúng ta có thể sử dụng là yêu cầu mỗi tiến trình yêu cầu và được cấp phát tất cả các tài nguyên của nó trước khi nó bắt đầu thực thi. Chúng ta có thể thực hiện điều này bằng cách yêu cầu các lệnh hệ thống yêu cầu tài nguyên cho một tiến trình đứng trước tất cả các lệnh hệ thống khác.

- Một giao thức thay thế cho phép một tiến trình chỉ yêu cầu các tài nguyên khi nó không có bất kỳ tài nguyên nào. Một tiến trình có thể yêu cầu một số tài nguyên và sử dụng chúng. Trước khi nó có thể yêu cầu bất kỳ tài nguyên bổ sung nào, nó phải giải phóng tất cả các tài nguyên mà nó đang được cấp phát.

- Để minh họa sự khác biệt giữa hai giao thức này, chúng ta xem xét một tiến trình sao chép dữ liệu từ ổ đĩa DVD đến một tập tin trên đĩa cứng, sắp xếp tập tin đó và sau đó in kết quả ra máy in. Nếu tất cả các tài nguyên phải được yêu cầu ở đầu tiên của quá trình, thì tiến trình phải yêu cầu ban đầu ổ đĩa DVD, tập tin đĩa cứng và máy in. Nó sẽ giữ máy in trong suốt quá trình thực thi, mặc dù nó chỉ cần máy in ở cuối.

- Phương pháp thứ hai cho phép tiến trình chỉ yêu cầu ban đầu ổ đĩa DVD và tập tin đĩa cứng. Nó sao chép từ ổ đĩa DVD vào đĩa cứng và sau đó giải phóng cả ổ đĩa DVD và tập tin đĩa cứng. Sau đó, tiến trình phải yêu cầu tập tin đĩa cứng và máy in. Sau khi sao chép tập tin đĩa cứng vào máy in, nó giải phóng hai tài nguyên này và kết thúc.

- Có hai nhược điểm chính của cả hai giao thức này. Thứ nhất, sử dụng tài nguyên có thể thấp, vì các tài nguyên có thể được cấp phát nhưng không sử dụng trong một khoảng thời gian dài. Trong ví dụ được đưa ra, chúng ta có thể giải phóng ổ đĩa DVD và tệp đĩa, và sau đó yêu cầu tệp đĩa và máy in, chỉ nếu chúng ta có thể đảm bảo dữ liệu của chúng tôi sẽ vẫn nằm trên tệp đĩa. Nếu không, chúng ta phải yêu cầu tất cả các tài nguyên ở đầu tiên cho cả hai giao thức.

- Thứ hai, đó là đói đến tài nguyên. Một quá trình cần nhiều tài nguyên phổ biến có thể phải đợi vô thời hạn, vì ít nhất một trong số các tài nguyên mà nó cần luôn được phân bổ cho một quá trình khác.

III.2.3. Trạng thái không có quyền ưu tiên (No Preemption)

- Để tránh No Preemption, ta cần giải pháp để có thể thu hồi các tài nguyên đang được sử dụng bởi một tiến trình để có thể cấp cho một tiến trình khác, và giải pháp này được gọi là preemption.

- Có nhiều cách để thực hiện preemption, một số phương pháp được sử dụng phổ biến như:

1. Priority Inversion Protocol: Trong giao thức này, tiến trình có ưu tiên cao hơn sẽ được cấp tài nguyên trước, nhưng nếu một tiến trình có ưu tiên thấp đang sử dụng tài nguyên mà tiến trình ưu tiên cao cần sử dụng, tiến trình ưu tiên thấp sẽ giữ tài nguyên đó và tiến trình ưu tiên cao sẽ bị chặn (blocked) đến khi tiến trình ưu tiên thấp hoàn thành việc sử dụng tài nguyên.
2. Aging: Đây là một kỹ thuật giúp tránh tình trạng tiến trình bị đóng băng (starvation) bằng cách tăng độ ưu tiên của các tiến trình đã chờ đợi một thời gian dài.

3. Deadline-Monitoring Protocol: Trong giao thức này, hệ thống sẽ kiểm tra các tiến trình đang chờ tài nguyên xem chúng có thể hoàn thành trước một deadline nhất định hay không. Nếu không thể hoàn thành trong thời gian đó, tiến trình sẽ bị gián đoạn (interrupt) để tài nguyên của nó được thu hồi và cấp cho các tiến trình khác.

- Tuy nhiên, mỗi phương pháp đều có nhược điểm riêng. Ví dụ, trong Priority Inversion Protocol, tiến trình ưu tiên cao có thể bị chặn bởi một tiến trình ưu tiên thấp, làm giảm hiệu quả của hệ thống. Trong khi đó, trong Aging, độ ưu tiên của một tiến trình có thể quá cao sau một thời gian dài chờ đợi, dẫn đến ưu tiên của tiến trình này vượt

qua các tiến trình khác. Trong Deadline-Monitoring Protocol, nếu deadline được đặt quá ngắn, nhiều tiến trình sẽ bị gián đoạn, gây giảm hiệu suất của hệ thống. Ngược lại, nếu deadline được đặt quá dài, các tiến trình khác sẽ phải chờ đợi quá lâu để sử dụng tài nguyên.

III.2.4. Trạng thái Circular Wait

- Có nhiều cách để tránh trạng thái circular wait:

1. Ordering resources: Xếp hàng đợi các tài nguyên theo một thứ tự cố định và yêu cầu các tiến trình tuân theo thứ tự này khi yêu cầu tài nguyên. Ví dụ, nếu tài nguyên A phải được sử dụng trước tài nguyên B, thì bất kỳ tiến trình nào yêu cầu tài nguyên A cũng phải yêu cầu tài nguyên B trước khi được cấp phát tài nguyên A. Phương pháp này có thể đảm bảo không xảy ra trạng thái circular wait nhưng cần thiết phải biết trước thứ tự ưu tiên của các tài nguyên.
   ➔ Phương pháp này đảm bảo không xảy ra trạng thái circular wait nhưng yêu cầu biết trước thứ tự ưu tiên của các tài nguyên và có thể gây ra hiệu suất kém nếu thứ tự này không được thiết lập đúng cách.

2. Resource allocation denial: Từ chối cấp phát tài nguyên nếu việc này sẽ dẫn đến trạng thái circular wait. Tuy nhiên, phương pháp này có thể gây ra tình trạng hạn chế sử dụng tài nguyên và có thể dẫn đến hiệu suất kém nếu các tiến trình không thể tiếp tục hoạt động vì không thể cấp phát tài nguyên.
   ➔ Phương pháp này đảm bảo không xảy ra trạng thái circular wait nhưng có thể dẫn đến hạn chế sử dụng tài nguyên và gây ra hiệu suất kém nếu các tiến trình không thể tiếp tục hoạt động vì không thể cấp phát tài nguyên.

3. Resource preemption: Giải phóng các tài nguyên của một tiến trình và cấp phát lại cho các tiến trình khác nếu cần thiết để tránh trạng thái circular wait. Tuy nhiên, phương pháp này có thể gây ra tình trạng độc quyền tài nguyên, khi một tiến trình có thể giữ các tài nguyên lâu hơn so với các tiến trình khác.
   ➔ Phương pháp này giải quyết trạng thái circular wait bằng cách giải phóng các tài nguyên của một tiến trình và cấp phát lại cho các tiến trình khác. Tuy nhiên, phương pháp này có thể gây ra tình trạng

độc quyền tài nguyên và làm giảm hiệu suất nếu các tiến trình phải chờ đợi quá lâu để có thể sử dụng tài nguyên.

4. Wait-for graph and detection: Sử dụng đồ thị chờ đợi để phát hiện trạng thái circular wait và giải quyết vấn đề bằng cách tìm kiếm chu trình trong đồ thị và giải phóng tài nguyên. Phương pháp này hiệu quả trong việc giải quyết trạng thái circular wait nhưng tốn nhiều thời gian và tài nguyên tính toán.
   ➔ Phương pháp này hiệu quả trong việc giải quyết trạng thái circular wait nhưng tốn nhiều thời gian và tài nguyên tính toán để phát hiện và giải quyết vấn đề. Bên cạnh đó, phương pháp này có thể dẫn đến tình trạng chậm hơn nếu cần phải tạo và xử lý các đồ thị chờ đợi lớn.

III.3. Một số thuật toán Deadlock Avoidance

III.3.1. Thuật toán Resource-Allocation-Graph (RAG)

- Thuật toán Resource-Allocation-Graph là một trong những phương pháp được sử dụng trong việc phát hiện và tránh deadlock trong hệ thống máy tính. Thuật toán này sử dụng đồ thị tài nguyên (Resource Allocation Graph) để mô hình hóa việc sử dụng tài nguyên và quan hệ giữa các tiến trình.

- Các bước của thuật toán Resource-Allocation-Graph như sau:

1) Tạo đồ thị chờ đợi tài nguyên (Resource Allocation Graph) bằng cách tạo một đỉnh (vertex) cho mỗi tiến trình (process) và một đỉnh cho mỗi tài nguyên (resource). Tạo một cạnh (edge) từ mỗi tiến trình đến tài nguyên mà nó đang yêu cầu, và tạo một cạnh từ mỗi tài nguyên đến tiến trình nào đang giữ tài nguyên đó.
2) Kiểm tra xem đồ thị có chứa chu trình không. Nếu có chu trình, điều này cho thấy rằng hệ thống đang ở trạng thái deadlock.
3) Tìm các tiến trình có thể được giải phóng tài nguyên mà chúng đang giữ. Nếu có ít nhất một tiến trình có thể được giải phóng tài nguyên, hệ thống không đang ở trạng thái deadlock.
4) Nếu không có tiến trình nào có thể được giải phóng tài nguyên, hệ thống đang ở trạng thái deadlock và không thể tiếp tục hoạt động.
5) Nếu hệ thống đang ở trạng thái deadlock, giải quyết vấn đề bằng cách giải phóng một số tài nguyên. Có thể giải phóng tài nguyên

bằng cách tạm dừng một số tiến trình hoặc giải phóng các tài nguyên được sử dụng không hiệu quả.

6) Sau khi đã giải phóng tài nguyên, kiểm tra lại đồ thị chờ đợi tài nguyên để đảm bảo rằng không còn chu trình và hệ thống có thể tiếp tục hoạt động.

➔ Thuật toán Resource-Allocation-Graph giúp cho việc phát hiện trạng thái deadlock trở nên dễ dàng hơn, tuy nhiên, nó không phải là phương pháp hoàn hảo và có thể bị sai sót trong một số trường hợp đặc biệt. Ngoài ra, nó cũng không giải quyết được các trường hợp xảy ra deadlock mà không có chu trình trong đồ thị tài nguyên.

III.3.2. Thuật toán Banker (Banker's Algorithm)

- Thuật toán đồ thị cấp phát tài nguyên (resource-allocation-graph algorithm) không áp dụng được cho hệ thống cấp phát tài nguyên có nhiều phiên bản của từng loại tài nguyên. Thuật toán tránh bế tắc mà chúng tôi mô tả tiếp theo có thể áp dụng cho một hệ thống như vậy nhưng kém hiệu quả hơn sơ đồ đồ thị phân bổ tài nguyên. Thuật toán này được biết đến là thuật toán banker. Tên được đặt như vậy vì thuật toán có thể được sử dụng trong một hệ thống ngân hàng để đảm bảo rằng ngân hàng không bao giờ phân phát tiền sẵn có của nó theo cách mà nó không còn có thể đáp ứng nhu cầu của tất cả khách hàng của mình.

- Khi một tiến trình mới vào hệ thống, nó phải khai báo giá trị tối đa số lượng phiên bản của từng loại tài nguyên mà nó có thể cần. Con số này có thể không vượt quá tổng số tài nguyên trong hệ thống. Khi người dùng yêu cầu một tập hợp các tài nguyên trong hệ thống, hệ thống phải xác định xem liệu việc phân bổ các tài nguyên này tài nguyên sẽ rời khỏi hệ thống trong trạng thái an toàn hay không. Nếu an toàn, các tài nguyên sẽ được phân bổ; nếu không, tiến trình phải chờ cho đến khi một số quy trình khác phát hành đủ tài nguyên.

- Một số cấu trúc dữ liệu phải được duy trì đẻ cài đặt thuật toán banker. Các cấu trúc dữ liệu này mã hóa trạng thái cấp phát tài nguyên hệ thống. Chúng ta cần các cấu trúc dữ liệu sau, trong đó n là số tiến trình trong hệ thống và m là số loại tài nguyên:

- **Available:** Một vector kích thước m để chỉ số tài nguyên có sẵn của mỗi loại. Nếu Available[j] bằng k, thì k tài nguyên của loại Rj có sẵn.
- **Max:** Một ma trận n x m định nghĩa nhu cầu tối đa có thể đạt được của từng tiến trình. Nếu Max[i][j] bằng k, thì tiến trình Pi có thể yêu cầu tối đa k tài nguyên của loại Rj.

- **Allocation:** Một ma trận n x m định nghĩa số tài nguyên của từng loại hiện tại phân bổ cho từng tiến trình. Nếu Allocation[i][k] bằng k, thì tiến trình Pi được phân bổ k tài nguyên loại Rj.
- **Need:** Một ma trận n x m chỉ ra nhu cầu tài nguyên còn lại của mỗi quá trình. Nếu Need[i][j] bằng k, thì tiến trình Pi có thể cần hơn k tài nguyên của loại Rj để hoàn thành nhiệm vụ của nó. Chú ý rằng Need[i][j] bằng [i][j] – Allocation[i][j].

- Để đơn giản hóa việc trình bày thuật toán banker, giả sử X và Y là các vector kích thước n. Chúng ta nói rằng $X \leq Y$ nếu và chỉ nếu $X[i] \leq Y[i]$ với mọi i = 1, 2, …, n. Ví dụ nếu X = (1,7,3,2) và Y = (0,3,2,1) thì $Y \leq X$. Ngoài ra, $Y < X$ nếu $Y \leq X$ và $Y \neq X$.

- Chúng ta có thể coi mỗi hàng trong ma trận Allocation và Need là các vector và gọi chúng là $Allocation_i$ và $Need_i$. Vector $Allocation_i$ chỉ định các tài nguyên hiện được phân bổ cho tiến trình P_i; vector $Need_i$ chỉ định tài nguyên bổ sung rằng P_i vẫn có thể yêu cầu để hoàn thành nhiệm vụ của mình.

III.4. Các giải pháp xử lí critical section

- Critical section là một phần của chương trình mà nhiều luồng (thread) có thể truy cập đồng thời và thay đổi trạng thái của nó, có thể dẫn đến tình trạng không đồng bộ hoặc lỗi dữ liệu. Để giải quyết vấn đề này, hệ điều hành (OS) cung cấp nhiều giải pháp khác nhau để xử lý Critical section. Dưới đây là một bảng so sánh/đánh giá các giải pháp này:

Giải pháp	Mô tả	Ưu điểm	Khuyết điểm
Peterson's Solution	Giải pháp dựa trên việc sử dụng biến flag và turn để đảm bảo đồng bộ giữa 2 luồng thực thi.	- Đơn giản, dễ triển khai. - Thích hợp cho các ứng dụng đơn giản có ít luồng thực thi.	- Không thể mở rộng cho nhiều hơn 2 luồng thực thi. - Không đảm bảo tính đồng bộ và an toàn dữ liệu trên nhiều luồng.
Synchronization Hardware	Giải pháp dựa trên việc sử dụng phần cứng hỗ trợ đồng bộ, chẳng hạn như Interlocked Operations trên CPU.	- Hiệu suất cao. - Không tốn nhiều tài nguyên hệ thống.	- Khó triển khai và hỗ trợ, do yêu cầu phải có phần cứng hỗ trợ đồng bộ. - Không thể sử dụng trong các ứng dụng đa nền tảng. - Không đảm bảo tính đồng bộ và an

			toàn dữ liệu trên nhiều luồng và tiến trình.
Mutex Locks	Giải pháp dựa trên việc sử dụng Mutex để đảm bảo đồng bộ và an toàn dữ liệu giữa các luồng thực thi.	- Đảm bảo tính đồng bộ và an toàn dữ liệu giữa các luồng thực thi. - Có thể sử dụng cho nhiều luồng thực thi.	- Tốn nhiều tài nguyên hệ thống. - Có thể gây tình trạng đợi (deadlock) nếu không được sử dụng đúng cách.
Semaphores	Giải pháp dựa trên việc sử dụng Semaphore để đảm bảo đồng bộ và an toàn dữ liệu giữa các luồng thực thi.	- Đảm bảo tính đồng bộ và an toàn dữ liệu giữa các luồng thực thi. - Có thể sử dụng cho nhiều luồng thực thi. - Giải quyết được vấn đề bất đồng bộ (race condition) trong các ứng dụng đa luồng.	- Có thể dễ dàng gây hiện tượng deadlock (khi các luồng đang chờ nhau và không tiến hành thực thi) - Tốn tài nguyên hệ thống hơn so với Mutex Locks.

- Việc lựa chọn giải pháp xử lý Critical section phụ thuộc vào tính chất và yêu cầu của ứng dụng cụ thể. Dưới đây là một số trường hợp cụ thể và giải pháp tương ứng:

- Nếu ứng dụng chỉ có hai luồng thực thi và yêu cầu đơn giản, Peterson's Solution có thể là sự lựa chọn tốt nhất.
- Nếu ứng dụng yêu cầu tính đồng bộ và an toàn dữ liệu tốt nhất, Mutex Locks hoặc Semaphores có thể là giải pháp tốt nhất. Semaphore có thể được sử dụng để đồng bộ hóa giữa nhiều luồng và giải quyết vấn đề bất đồng bộ (race condition), trong khi Mutex Locks thường được sử dụng để đảm bảo đồng bộ và an toàn dữ liệu giữa các luồng.
- Nếu ứng dụng đòi hỏi tính hiệu suất cao và không muốn tốn nhiều tài nguyên hệ thống, Synchronization Hardware có thể là giải pháp tốt nhất. Tuy nhiên, việc triển khai và hỗ trợ giải pháp này có thể gặp nhiều khó khăn, do đó chỉ được sử dụng trong các trường hợp đặc biệt.
- Ngoài ra, nếu ứng dụng yêu cầu sự đồng bộ và an toàn dữ liệu giữa các tiến trình trên nhiều máy tính khác nhau, phải sử dụng các giải pháp phức tạp như

Distributed Lock Manager (DLM) hoặc các thuật toán đồng bộ hóa khác trong hệ thống phân tán.

IV. Chương trình mô phỏng sử dụng thuật toán Banker (xử lí Deadlock)

IV.1. Mô tả chương trình

- Đây là chương trình đa luồng cài đặt dựa trên thuật toán Banker đã được trình bày trong phần III.3.2 để quản lí việc cấp phát tài nguyên cho các tiến trình. Chương trình giả lập một ngân hàng với hai loại tài nguyên: tiền mặt và giấy tờ. Mỗi khách hàng yêu cầu rút tiền với một số lượng tiền mặt và giấy tờ cụ thể. Nếu số lượng tài nguyên yêu cầu có sẵn trong ngân hàng, yêu cầu sẽ được chấp nhận và khách hàng sẽ có thể rút tiền. Ngược lại, yêu cầu sẽ không được chấp nhận và khách hàng phải chờ đợi đến khi có đủ tài nguyên.

- Cụ thể, chương trình định nghĩa một lớp Bank để lưu trữ thông tin về tài nguyên hiện có, tài nguyên tối đa mà mỗi tiến trình có thể yêu cầu và số tài nguyên đang được cấp phát cho mỗi tiến trình. Nó cũng cung cấp các phương thức để yêu cầu tài nguyên từ ngân hàng và giải phóng tài nguyên khi các tiến trình hoàn thành.

- Chương trình sử dụng các thread để tạo ra nhiều tiến trình đồng thời và gọi phương thức yêu cầu tài nguyên từ lớp Bank. Nếu yêu cầu tài nguyên được chấp nhận, tiến trình đang chạy sẽ giữ tài nguyên trong một khoảng thời gian được giả lập bằng cách gọi hàm sleep_for(), sau đó trả lại tài nguyên cho ngân hàng. Nếu yêu cầu tài nguyên không được chấp nhận, tiến trình đó sẽ thông báo cho người dùng.

- Chương trình sử dụng mutex để đồng bộ hóa các thread khi truy cập vào tài nguyên chia sẻ (như các biến trong lớp Bank và màn hình để hiển thị thông báo).

IV.2. Các phương thức(hàm) trong chương trình

- Các phương thức công khai của lớp **Bank** bao gồm:

- **Bank::Bank(const std::vector<int>& resources, const std::vector<std::vector<int>>& max_resources)**: là hàm khởi tạo lớp **Bank**, chấp nhận hai đối số là **resources** và **max_resources**. Hàm tạo vector **allocation** bằng số lượng process và số lượng tài nguyên. Nó cũng

tạo ra vector **need** bằng **max_resources.**

```
Bank(const std::vector<int>& resources, const std::vector<std::vector<int>>& max_resources)
    : available(resources), max(max_resources) {
    int process_count = max_resources.size();
    allocation = std::vector<std::vector<int>>(process_count, std::vector<int>(resources.size(), 0));
    need = max_resources;
}
```

- **Bank::is_safe(int process, const std::vector<int>& request)**: kiểm tra xem nếu khách hàng yêu cầu một số lượng tài nguyên cụ thể thì có đủ tài nguyên để đáp ứng yêu cầu không. Phương thức này sử dụng giải thuật an toàn để đảm bảo rằng không có khách hàng nào bị bế tắc vì không đủ tài nguyên để hoàn thành yêu cầu của mình.

```
bool is_safe(int process, const std::vector<int>& request) {
    std::vector<int> temp_available = available;
    std::vector<std::vector<int>> temp_allocation = allocation;
    std::vector<std::vector<int>> temp_need = need;

    for (size_t i = 0; i < request.size(); ++i) {
        temp_available[i] -= request[i];
        temp_allocation[process][i] += request[i];
        temp_need[process][i] -= request[i];
    }

    std::vector<bool> finish(max.size(), false);
    bool found;

    do {
        found = false;
        for (size_t i = 0; i < max.size(); ++i) {
            if (!finish[i] && std::equal(temp_need[i].begin(), temp_need[i].end(), temp_available.begin(), temp_available.end(), [](int a, int b) { return a <= b; })) {
                for (size_t j = 0; j < temp_available.size(); ++j) {
                    temp_available[j] += temp_allocation[i][j];
                }
                finish[i] = true;
                found = true;
            }
        }
    } while (found);

    return std::all_of(finish.begin(), finish.end(), [](bool a) { return a; });
}
```

- **Bank::request_resources(int process, const std::vector<int>& request)**: kiểm tra xem có thể cấp phát tài nguyên được yêu cầu cho một process và nếu có, thực hiện cấp phát và trả về **true**, ngược lại trả về **false**.

```
bool request_resources(int process, const std::vector<int>& request) {
    std::unique_lock<std::mutex> lock(mtx);
    if (!is_safe(process, request)) {
        lock.unlock();
        return false;
    }

    for (size_t i = 0; i < request.size(); ++i) {
        available[i] -= request[i];
        allocation[process][i] += request[i];
        need[process][i] -= request[i];
    }
    lock.unlock();
    return true;
}
```

- **Bank::release_resources(int process)**: giải phóng các tài nguyên của một process, thực hiện trả lại các tài nguyên cấp phát và khởi tạo lại các giá trị của tài nguyên cần thiết và tài nguyên đã cấp phát.

```
void release_resources(int process) {
    std::unique_lock<std::mutex> lock(mtx);
    for (size_t i = 0; i < available.size(); ++i) {
        available[i] += allocation[process][i];
        allocation[process][i] = 0;
        need[process][i] = max[process][i];
    }
    lock.unlock();
}
```

- Các phương thức khác:

- **atm_withdrawal(Bank& bank, int process, const std::vector<int>& request):** Đầu tiên, hàm kiểm tra xem yêu cầu rút tiền có thể được thực hiện hay không bằng cách gọi hàm **request_resources** của đối tượng **Bank**. Nếu yêu cầu được chấp nhận, thì hàm thông báo cho người dùng biết yêu cầu được chấp nhận và tiến hành giả lập việc xử lý yêu cầu trong 1 giây bằng cách sử dụng hàm **std::this_thread::sleep_for**. Sau khi hoàn thành xử lý yêu cầu, hàm thông báo cho người dùng biết yêu cầu đã được hoàn thành và giải phóng tài nguyên đã sử dụng bằng cách gọi hàm **release_resources** của đối tượng **Bank**. Nếu yêu cầu không được chấp nhận do thiếu tài nguyên, thì hàm thông báo cho người dùng biết yêu cầu không được chấp nhận.

```
void atm_withdrawal(Bank& bank, int process, const std::vector<int>& request) {
    if (bank.request_resources(process, request)) {
        {
            std::unique_lock<std::mutex> lock(print_mtx);
            std::cout << "Yeu cau rut tien " << process + 1 << " duoc chap nhan.\n";
            lock.unlock();
        }
        // Simulate processing time
        std::this_thread::sleep_for(std::chrono::seconds(1));
        {
            std::unique_lock<std::mutex> lock(print_mtx);
            std::cout << "Yeu cau rut tien " << process + 1 << " hoan thanh va tra lai tai nguyen.\n";
            lock.unlock();
        }
        bank.release_resources(process);
    }
    else {
        std::unique_lock<std::mutex> lock(print_mtx);
        std::cout << "Yeu cau rut tien " << process + 1 << " khong duoc chap nhan do thieu tai nguyen\n";
        lock.unlock();
    }
}
```

IV.3. Mô tả hàm main và chạy chương trình demo

- Hàm main của chương trình bao gồm các bước sau:

1. Nhập số lượng loại tài nguyên và số lượng tiến trình từ người dùng.
2. Nhập số lượng tài nguyên có sẵn cho mỗi loại tài nguyên từ người dùng.
3. Nhập số lượng tài nguyên tối đa cho mỗi tiến trình từ người dùng.
4. Khởi tạo một đối tượng Bank với tài nguyên và tài nguyên tối đa đã nhập.
5. Nhập các yêu cầu từng tiến trình cho các loại tài nguyên.
6. Sử dụng thread để xử lý yêu cầu của mỗi tiến trình.
7. Đợi tất cả các thread kết thúc.
8. Kết thúc chương trình.

```
int main() {
    // Input resources and max resources
    std::vector<int> resources;
    std::vector<std::vector<int>> max_resources;
    int num_processes;

    std::cout << "Enter the number of resource types for the bank: ";
    int num_resources;
    std::cin >> num_resources;
    std::cout << "\n";
    std::cout << "Enter the available resources for each type\n";
    for (int i = 0; i < num_resources; ++i) {
        int resource;
        std::cout << "Enter the available resources for resource's type " << i + 1 << ": ";
        std::cin >> resource;
        resources.push_back(resource);
    }
    std::cout << "\n";
    std::cout << "Enter the number of processes: ";
    std::cin >> num_processes;
    std::cout << "\n";
    std::cout << "Enter the maximum resources for each process:\n";
    for (int i = 0; i < num_processes; ++i) {
        std::vector<int> process_max;
        std::cout << "Process " << i + 1 << ": \n";
        for (int j = 0; j < num_resources; ++j) {
            int max_resource;
            std::cout << "Enter the maximum resources for resource's type " << j + 1 << ": ";
            std::cin >> max_resource;
            process_max.push_back(max_resource);
        }
        max_resources.push_back(process_max);
    }
```

Hình ảnh bước nhập trong hàm main

```
    Bank bank(resources, max_resources);

    // Input requests
    std::vector<std::vector<int>> requests;
    std::cout << "Enter the requests for each process:\n";
    for (int i = 0; i < num_processes; ++i) {
        std::vector<int> process_request;
        std::cout << "Process " << i + 1 << ": \n";
        for (int j = 0; j < num_resources; ++j) {
            int request;
            std::cout << "Enter the request resource for resource's type " << j + 1 << ": ";
            std::cin >> request;
            process_request.push_back(request);
        }
        requests.push_back(process_request);
    }

    // Process requests
    std::vector<std::thread> threads;
    for (int i = 0; i < num_processes; ++i) {
        threads.emplace_back(atm_withdrawal, std::ref(bank), i, requests[i]);
    }

    for (auto& t : threads) {
        t.join();
    }
```

Hình ảnh bước xử lí các tiến trình và xuất ra trong hàm main

- Ví dụ về input và output của chương trình khi chạy

```
Enter the number of resource types for the bank: 2

Enter the available resources for each type
Enter the available resources for resource's type 1: 2000
Enter the available resources for resource's type 2: 1000

Enter the number of processes: 2

Enter the maximum resources for each process:
Process 1:
Enter the maximum resources for resource's type 1: 1000
Enter the maximum resources for resource's type 2: 500
Process 2:
Enter the maximum resources for resource's type 1: 1200
Enter the maximum resources for resource's type 2: 800

Enter the requests for each process:
Process 1:
Enter the request resource for resource's type 1: 800
Enter the request resource for resource's type 2: 400
Process 2:
Enter the request resource for resource's type 1: 1300
Enter the request resource for resource's type 2: 900
The withdrawal request 1 has been accepted.
The withdrawal request 2 is not accepted due to lack of resources
The withdrawal request 1 has been completed and resource returned.
```

IV.4. Tổng kết về chương trình và thuật toán Banker

- Chương trình chúng em đã có xử lí các chủ đề: “multithreading”, “preventing race conditions”, và “deadlock avoidance”

- Về điểm mạnh và điểm yếu của thuật toán Banker:

✓ Điểm mạnh:

- Phòng ngừa tình trạng deadlock: thuật toán Banker có khả năng phân bổ tài nguyên sao cho không gây ra deadlock trong hệ thống.
- Hiệu quả cao: Thuật toán Banker có độ phức tạp thời gian O(n^2), tương đối nhẹ so với các thuật toán phân bổ tài nguyên khác.
- Đảm bảo tối ưu hóa tài nguyên: Thuật toán Banker cho phép tối ưu hóa sử dụng tài nguyên của hệ thống. Nó sẽ phân phối tài nguyên theo cách tối ưu để đảm bảo rằng các tiến trình có thể hoàn thành công việc của mình mà không cần phải chờ đợi quá lâu.

➢ Điểm yếu

- Khó xử lý với các yêu cầu tài nguyên động: Thuật toán Banker yêu cầu phải biết trước các yêu cầu tài nguyên của các tiến trình, điều này là không khả thi đối với các hệ thống yêu cầu tài nguyên động.
- Tiềm ẩn rủi ro bảo mật: Thuật toán Banker có thể bị tấn công và lộ thông tin về tài nguyên trong hệ thống.

Cách khắc phục:

❖ Khắc phục vấn đề yêu cầu tài nguyên ban đầu phải biết trước: Có thể sử dụng các thuật toán phát hiện yêu cầu tài nguyên động để giải quyết vấn đề này. Các thuật toán này cho phép hệ thống xác định số lượng tài nguyên mà một tiến trình cần để hoàn thành công việc của nó trong quá trình thực thi.

❖ Để xử lý các yêu cầu tài nguyên động, có thể sử dụng các thuật toán phân bổ tài nguyên động khác như thuật toán Wound Wait hay thuật toán Wait Die.

❖ Để giải quyết vấn đề bảo mật, cần áp dụng các biện pháp bảo mật phù hợp như mã hóa dữ liệu, kiểm tra xác thực, giám sát và bảo vệ hệ thống khỏi các cuộc tấn công.

V. Nguồn tham khảo

(1): Sách: Operating-System-Concepts.9th.Edi.Abraham-Silberschatz.2012 (https://drive.google.com/drive/folders/1-bz_MZPNp8DHevgNVAYQs6Kj-65qmU4R)

(2): Thư viện **mutex**: https://cplusplus.com/reference/mutex/mutex/

(3): Thư viện **thread**: https://cplusplus.com/reference/thread/thread/

(4): Thuật toán **Banker**: https://en.wikipedia.org/wiki/Banker%27s_algorithm#:~:text=Banker's%20algorithm%20is%20a%20resource,conditions%20for%20all%20other%20pending