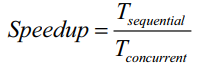
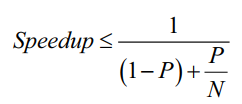
**MASTERING CONCURRENTY PROGRAMMING WITH JAVA 8**

1. **Concurrency Design Principle**
   1. Khái niệm cơ bản

* **Chạy đồng thời vs chạy song song (Concurrency vs parallelism):** các định nghĩa giữa 2 khái nhiệm này khá giống nhau ví dụ:
  + Conccurrency để nói về trường hợp thực thi nhiều tác vụ trên HDH đơn luồng, HDH sẽ lập lịch để chuyển giữa các tác vụ với nhau (với thời gian chuyển đổi rất nhỏ) để trông như các tác vụ chạy đồng thời
* **Đồng bộ (synchornization):** bao gồm 2 loại
  + Đồng bộ điều khiển (control synchornization): một task sẽ phụ thuộc vào sự kết thúc hoặc kết của 1 task khác, task thứ 2 chỉ được bắt đầu khi task thứ nhất kết thúc.
  + Đồng bộ truy cập dữ liệu (Data access synchonization): khi 2 hoặc nhiều tác vụ muốn truy cập đến 1 biến chia sẻ chung nhưng tại 1 thời điểm chỉ có 1 tác vụ được phép truy cập
* Một số cơ chế đồng bộ phổ biến trong các hệ thống chạy đồng thời:
  + **Semaphore:** là cơ chế kiểm soát việc truy cập đến một hay nhiều khối tài nguyên.
    - Nó sử dụng 1 biến để lưu trữ số lượng tài nguyên được sử dụng và 2 toán từ nguyên tử (atomic operation) để kiểm soát giá trị của biến
    - **Mutex** (viết tắt của *mutal exclusion*) là 1 kiểu đặc biệt của semaphore chỉ chứa 2 giá trị là “tài nguyên bận” hoặc “tài nguyên không bận” và chỉ có tiến trình thiết lập mutex busy mới có thể giải phóng nó.
  + **Monitor:** là cơ chế loại trừ lần trên các tài nguyên được chi sẻ
* **Thread safe:** Một method hay một đối tượng được coi là luồng an toàn (thread safe) nếu tất cả việc sử dụng dữ liệu được chia sẻ được bảo vệ bởi cơ chế đồng bộ, so sánh và trao đổi không chặn( nonblocking compare-and swap) nguyên thủy hoặc dữ liệu không thể bị thay đổi (immutable).
* **Immuatable Objec (các đối tượng không thể thay đổi được):**  là các đối tượng có cơ chế đặc biệt, ta không thể thay đổi trạng thái hiển thị được của nó (bao gồm giá trị của các thuộc tính),
  + Nếu muốn thay đổi thì phải tạo một đối tượng mới.
  + Các đối tượng này là thread-safe
  + Các đối tượng thuộc kiểu String là một ví dụ
* **Atomic operation and variable (biến và thao tác nguyên tử):** 
  + Thao tác nguyên tử là kiểu thao tác thực hiện gần như tức thời với phần tác vụ còn lại của chương trình
  + Biến nguyên tử: là kiểu biến cho phép thao tác nguyên tử với việc lấy giá trị và gán giá trị cho biến đó. Có thể triển khai bằng cơ chế đồng bộ hoặc phương pháp lock-free sử dụng CAS( cơ chế này không đòi hỏi bất cứ sự đồng bộ nào)
* **Shared memory vs message passing (bộ nhớ dùng chung vs truyền tin nhắn ):** các tác vụ có thể sử dụng 2 method khác nhau để giao tiếp với nhau qua 2 kiểu bộ nhớ dùng chung và truyền tin nhắn
  + **Shared Memory:** sử dụng trong trường hợp tác vụ chạy trên cùng hệ điều hành, các tác vụ sẽ dùng chung vùng nhớ để đọc và ghi, để tránh các lỗi việc truy cập vào vùng nhớ dùng chung cần phải được bảo vệ bởi cơ chế đồng bộ
  + **Message passing:** Sử dụng trong trường hợp các tác vụ chạy trên khác HDH. Khi các tác vụ giao tiếp với nhau chúng sẽ gửi các thông điệp tới nhau theo một giao thức nhất định. Việc giao tiếp cần được đồng bộ nếu người gửi được khóa để đợi phản hổi hoặc không cần đồng bộ nếu người gửi tiếp tục thực thi các nhiệm vụ sau khi gửi tin nhắn.
  1. **Các vấn đề có thể trong các ứng dụng chạy đồng thời**
* **Race condition (Điều kiện thực hiện):** xảy ra khi có 2 hoặc nhiều tác vụ ghi đồng thời vào biến dùng chung (shared variable) mà không có bất cứ cơ chế đồng bộ nào.
* **Deadlock (khóa chết):** xảy ra khi hai hoặc nhiều luồng đợi để sử dụng tài nguyên đang được khóa bởi một luồng khác, khi đó các luồng sẽ đợi mãi cho đến khi tài nguyên được giải phóng. Khóa chết sẽ xảy ra khi có đồng thời 4 điều kiện cùng xảy ra trên cùng 1 hệ thống (được gọ là Coffman’s Condition) bao gồm:
  + **Mutual exclusion (loại trừ lẫn nhau):** tài nguyên liên quan đến deadlock phải là tài nguyên *không được chia sẻ.* Chỉ có 1 task có thể sử dụng tài nguyên ở 1 thời điểm
  + **Hold and wait condition (giữ và đợi điều kiện):** task đang đợi tài nguyên từ 1 task khác không thực hiện giải phóng bất cứ tài nguyên nào.
  + **No pre-emption ( không có quyền ưu tiên):** Tài nguyên đang bị task nắm giữ không thể được giải phóng bởi task khác.
  + **Circural wait (đợi vòng trong):** task 1 đợi tài nguyên từ task 2, task 2 đợi tài nguyên từ task 3… task n-1 đợi tài nguyên từ task n và task n lại đợi tài nguyên từ task 1.
* **Một số cơ chế giúp tránh deadlock:**
  + **Bỏ qua:** Nếu xảy ra thì hậu quả có thể là sẽ phải dừng ứng dụng để xử lý.
  + **Phát hiện:** Hệ thống có những task đặc biệt có thể giúp chúng ta phát hiện ra deadlock, và khi phát hiện hệ thống có thể có những cơ chế để khắc phục.
  + **Ngăn chặn:** Có thể ngăn chặn deadlock bằng cách loại bỏ 1 hoặc nhiều điều kiện của Coffman
  + **Tránh né:** Có thể tránh khóa chết bằng cách phân tích tài nguyên mà một luông có thể sử dụng trước khi luồng đó bắt đầu thực thi.
* **Livelock (khóa sống):** xảy ra khi 2 hoặc nhiều task luôn luôn thay đổi trạng thái của nó dẫn đến task khác cũng thay đổi
* **Resource starvation (đói tài nguyên):** xảy ra khi một hoặc nhiều luồng cùng đợi tài nguyên để thực thi.
  + Khi tài nguyên được giải phóng, hệ thống sẽ phải chọn lựa luồng nào được phép sử dụng tiếp theo. Nếu thuật toán không đủ tốt, một số luồng có thể phải đợi trong thời gian rất dài.
  + **Fairness (sự công bằng)** là giải pháp cho vấn đề này, tất cả các luồng sẽ phải có được tài nguyên sau một chu kỳ nào đó, tuy nhiên.
* **Priority inversion (đảo ngược độ ưu tiên):** xảy ra khi các task có độ ưu tiên thấp nắm giữ tài nguyên mà các task có độ ưu tiên cao hơn cần. Bởi vậy task có độ ưu tiên thấp sẽ thực thi trước task có độ ưu tiên cao.
  1. **Phương pháp để thiết kế thuật toán đồng thời**
* **Bước 1: Phân tích**
  + Tìm các phần code có thể thực thi song song với nhau, chúng ta nên tập trung vào các phần chiếm nhiều thời gian thực thi.
  + Đối tượng lý tưởng là các tiến trình được lặp đi lặp lại mà một bước không phụ thuộc vào các bước khác hoặc các phần code không phụ thuộc vào phần khác.
* **Bước 2: Thiết kế:**
  + Sự thay đổi thiết kế code có thể ảnh hưởng đến 2 phần: *cấu trúc code và tổ chức của cấu trúc dữ liệu.*
  + Có 2 cách tiếp cận cho vấn đề thiết kế ứng dụng chạy đồng thời:
    - **Phân tách tác vụ:**  Thực hiện khi ta muốn phân chia code thành 2 hoặc nhiều các tác vụ mà có thể thực thi trong 1 lần. Có thể một vài tác vụ sẽ được thực thi theo một thứ tự nhất định hoặc phải đợi cùng một thời điểm, ta cần chú ý cơ chế đồng bộ với các trường hợp này
    - **Phân tách dữ liệu:** Thực hiện khi ta có nhiều thể hiện của cùng 1 tác vụ nhưng làm việc với tập con của một tập dữ liệu. Tập dữ liệu này là dữ liệu dùng chung (*share resource*), vì thế tập dữ liệu này phải là thread safe
  + Mục đích của việc thiết kế các ứng dụng đồng thời là để tăng hiệu năng cho hệ thống vì thế ta cần phải chú ý đến số nhân và số luồng của CPU.
  + Nếu Thiết kế quá nhiều luông có thể làm giảm hiệu năng của hệ thống, nhưng nếu thiết kế quá ít luồng sẽ không tận dụng được tài nguyên.
* **Bước 3: Thực hiện**
  + Sử dụng ngôn ngữ hoặc các thư viện hỗ trợ để phát triển các ứng dụng
* **Bước 4: Kiểm thử**
* **Bước 5: Điều chỉnh (tuning):**
  + Mục đích chính là so sánh và điều chỉnh hiệu năng giữa khi ta thực hiện tác vụ đồng thời với tuần tự. Trong trường hợp hiệu năng không như mong muốn, ta cần phải tìm hiều nguyên nhân.
  + Có nhiều chỉ số để xem xét khả năng cải thiện hiệu năng:
    - *Speedup : Chỉ số đánh giá độ cải thiện hiệu năng giữa chạy tuần tự và chạy song song*



* + - *Amdahl's law: Tính toán khả năng cải thiện hiệu năng tối đa có thể đạt được: (P: % phần code có thể thực hiện chạy song song, N: số nhân của máy tính )*



* + - *Gustafson-Barsis' law:đề cập đến việc khi máy tính có càng nhiều nhân, càng nhiều tác vụ có thể xử lý đồng thời:*



* **Kết Luận:**
  + Không phải tất cả các thuật toán đều thực hiện song song , nếu muốn triển khai ứng dụng chạy đồng thời ta cần phải tuân theo các điểm sau:
    - **Efficiency(Tính hiệu quả):**  Các thuật toán để chạy song song cần phải có thời gian thực thi nhỏ hơn chạy tuần tự.
    - **Simplicity(Tính đơn giản):** Cần đảm bảo thuật toán được triển khai đơn giản nhất có thể.
    - **Portability(Tính linh động):** Các thuật toán cần triển khai cần phải chạy được trên nhiều nền tảng khác nhau.
    - **Scalability(Tính mở rộng):** trường hợp tài nguyên hệ thống đước mở rộng (số nhân, số luồng CPU), thuật toán cần tận dụng được tối đa sự mở rộng này.
  1. **API hỗ trợ đa luồng trong JAVA**
* **Các class đa luồng cơ bản**:
  + *Thread:* Đại điện cho tất cả các luồng được thực thi trong JAVA
  + *Runable:* Cách khác để tạo luồng trong JAVA
  + *ThreadLocal:* Class chứa các biến cục bộ cho 1 luồng
  + *ThreadFactory:* là một dạng của mẫu thiết kế FACTORY, dùng để tạo các luồng có thể chỉnh sửa được
* **Cơ chế đồng bộ:**
  + Java concurency API bao gồm các cơ chế đồng bộ khác nhau cho phép:
    - Định nghĩa phần quan trọng (critical section) cho việc truy cập đến tài nguyên dùng chung
    - Đồng bộ các tác vụ khac nhau tại một điểm chung
  + Các cơ chế đồng bộ quan trọng nhất bao gồm:
    - *Synchronized* keyword: cho phép định nghĩa các phần quan trọng trong một khối code hoặc toàn bộ method
    - *Lock interface:* cung cấp sự đồng bộ mềm dẻo hơn so với *synchronized* keyword:
      * *ReentrantLock*: impl của Lock có thể kết hợp thêm với các điều kiện
      * *ReentrantReadWriteLock:* tách biệt hành động đọc và ghi
      * *StampedLock:* (được bổ sung trong java8) bao gồm 3 chế độ để kiểm soát hoạt động đọc và ghi
      * *Semaphore*
      * *CountDownlatch*
      * *CyclicBarrier*
      * *Phaser*
* **Executor:** Là cơ chế đặc biệt cho phép tách biệt việc tạo và quản lý các luồng. Chúng ta không cần quan tâm đến việc tạo và quản lý các luồng, chỉ cần tạo và gửi chúng đến executor. Một số lớp chính liên quan đến Excutor bao gồm:
  + *Executor and ExecutorService interface:* Bao gồm các hàm chung cho tất cả excutor
  + *ThreadPoolExcutor:* Cho phép lấy các *excutor* từ ThreadPool và cấu hình số lượng tối đa luồng chạy song song
  + *ScheduledThreadPoolExcutor:* là một kiểu đặc biệt của excutor cho phép thực thi task sau một khoảng thời gian hoặc theo một chu ky nào đó.
  + *Callable:*  là một inteface thay thế cho Runable, cho phép các luông riêng biệt có thể trả về giá trị.
  + *Future:* là một interface bao gồm các method cho phép lấy được các giá trị trả về từ *Callable* và kiểm soát trạng thái của nó
* **Fork/Join framework:** định nghĩa ra cá excutor đặc biệt cho giải pháp về vấn đề về kĩ thuật chia và trị (*divide and conquer technique*).
* **Parallel Streams (Luồng song song):**Java8 đã cung cấp thêm các tính năng mới vô cùng quan trọng là *Stream và Lambda Expression.*  Một số các yếu tố quan trọng trong luông song song bao gồm:
  + *Stream interface*
  + *Optional:*  là một đối tượng chứa đựng (container object) định nghĩa để chứa hoặc không chứa 1 giá trị non-null
  + *Collector*
  + *Lambda Expression.*
* **Concurrency data Structures:**
  + Hầu hết các cấu trúc dữ liệu thông thường (ArrayList, HastTable) không sử dụng trong các hệ thống đa luồng.
    - Nếu muốn sử dụng chúng ta cần thêm cơ chế đồng bộ cho chúng, nhưng sẽ làm giảm hiệu năng của hệ thống.
    - Nếu sử dụng có thể gây ra *race condition,* hoặc bị ném ra các ngoại lệ.
  + Java Concurrency API cung cấp nhiều cấu trúc dữ liệu sử dụng cho các ứng dụng đa luồng, có thể phân chúng làm 2 loại:
    - Blocking Data Structure: bao gồm các method sẽ thực hiện chặn các luồng khác khi mà, ví dụ như bị rỗng hoặc muốn lấy dữ liệu.
    - Non-blocking Data Structure: Nếu hành động được thực thi nay lập tức, nó sẽ không chặn luồng đang thực thi, các luồng khác gọi đến sẽ được trả về giá trị null hoặc ném ra một ngoại lệ
  + Một vài cấu trúc dữ liệu:
    - *ConcurrentLinkedDeque: This is a non-blocking list*
    - *ConcurrentLinkedQueue: This is a non-blocking queue*
    - *LinkedBlockingDeque: This is a blocking list*
    - *LinkedBlockingQueue: This is a blocking queue*
    - *PriorityBlockingQueue: This is a blocking queue that orders its elements based on its priority*
    - *ConcurrentSkipListMap: This is a non-blocking navigable map*
    - *ConcurrentHashMap: This is a non-blocking hash map*
    - *AtomicBoolean, AtomicInteger, AtomicLong, and AtomicReference: These are atomic implementations of the basic Java data types*
* **Concurrency Design Pattern:**
  + **Signaling:**  Sử dụng trong trường hợp 1 luồng cần thông báo sự kiện cho luông khác, cách dễ dàng nhất để triển khai là sử dụng method *wait()* hoặc *notity()* thông qua việc sử dụng các class *ReentrantLock hoặc Semaphore.*
  + **Rendezvous:** mục đích giống với Signaling, tuy nhiên dùng trong trường hợp luồng thứ nhất đợi event từ luồng thứ 2 và luồng thứ 2 đợi event từ luồng thứ nhất
  + **Mutex (mutal exclusion):** đảm bảo tài nguyên dùng chung được đồng bộ hóa trong việc truy cập, cách đơn giản nhất là dùng *synchronize keyword* hoặc sử dụng *ReentrantLock , Semaphore*
  + **Multiplex:** cũng giống như mutex tuy nhiên ta có thể quyết định được số luông có thể truy cập tại 1 thời điểm, cách dễ nhất để triển khai là sử dụng *Semaphore* để khởi tại số luông có thể thực thi vào vùng quan trọng tại cùng 1 thời điểm.
  + **Barrier:** Sử dụng trong trường hợp đồng bộ tất cả các luồng đến 1 thời điểm, không một luông nào có thể tiếp tục thực hiện cho đến khi tất cả các luồng đến một điểm đồng bộ. Có thể sử dụng Cyclicbarrier để triển khai.
  + **Double-checked locking**: sử dụng trong trường hợp ta yêu cầu khóa và check lại điều kiện, nếu điều kiện ko thỏa mãn thì phải có giải pháp để yêu cầu khóa đối tượng
  + **Read**-**write lock:** Sử dụng trong trường hợp bảo vệ truy cập tới các biến dùng chung tuy nhiên tách biệt giữa việc đọc và ghi để tăng hiệu năng,
    - Kiểu thiết kế này định nghĩa 2 loại *khóa trong*(internal lock): lock cho hoạt động đọc và lock cho hoạt động ghi.
    - Nếu 1 luông đang thực hiện đọc và luồng khác cũng muốn đọc thì nó sẽ không bị chặn.
    - Nếu 1 luồng đang đọc, 1 luồng khác muốn ghi thì nó phải đợi đến khi luông kia đọc xong.
    - Nếu 1 luồng đang ghi, 1 luồng khác muốn đọc hoặc ghi thì cần phải đợi cho đến khi luồng kia ghi xong.
  + **Thread Pool:** Sử dụng với mục đích tránh việc tạo quá nhiều thread dẫn đến giảm hiệu năng hệ thống.
    - Cách thiết kế này sẽ có 1 tập các luông và 1 queue chứa các tác vụ cần xử lý.
    - Sau khi xử lý xong task, thread sẽ không bị kill đi mà tiếp tục lấy các task trong queue ra xử lý hoặc đợi task được đẩy vào queue.
    - Java concurrency API cung cấp ExecutorService để triển khai thiết kế này.
  + **Thread local storage:** sử dụng trong trường hợp sử dụng biến toàn cục hoặc các biến static
* **Mô hình bộ nhớ trong java:** Mô hình bộ nhớ JAVA mô tả cách mà các luồng tương tác nhớ nhau qua bộ nhớ và khi có một thay đổi do 1 luồng gây ra thì các luồng khác cũng nhận được sự thay đổi đó
  + Trong trường hợp các ứng dụng đa luồng chạy trên CPU nhiều nhân sẽ có 1 số vấn đề với bộ nhớ đệm, tuy chúng làm tăng hiệu năng của hệ thông tuy nhiên lại gây ra mất đồng bộ dữ liệu: Khi 1 luồng chỉnh sửa thông tin của 1 biến, nó sửa đổi trên bộ nhớ đệm chứ không sửa đổi trên bộ nhớ chính ngay lập tức, Nếu 1 luồng khác đọc giá trị của biến này trước khi nó được cập nhật vào bố nhớ chính thì nó chỉ đọc được dữ liệu cũ.
  + Có nhiều mô hình bộ nhớ khác nhau: một số rất chặt chẽ (tất cả các luông luôn luôn được truy cập vào biến), một số khác thì ít chặt chẽ hơn (chỉ 1 số trường hợp cập nhật và bộ nhớ chính)
  + Mô hình bộ nhớ JAVA được định nghĩa theo các điểm dưới đây:
    - Định nghĩa các hành vi của các keyword: violite, synchronized, final.
    - Đảm bảo các ứng dụng chạy đa luông đã được đồng bộ chạy đúng trên tất cả các kiến trúc
    - Đảm bảo thứ tự thực hiện trước và sau trong các trường hợp *volatile read, volatile write, lock, and unlock*
    - Khi 1 luồng yêu cầu 1 sự giám sát (monitor), bộ nhớ đệm sẽ bị vô hiệu hóa
    - Khi luông giải phóng sự giám sát, dữ liệu đệm sẽ được thực hiện vào bộ nhớ chính.
  + Mục đích chính của mô hình bộ nhớ JAVA là giúp cho ứng dụng chạy đa luồng sẽ thực hiện đúng trên tất cả các máy ảo JAVA (JVM) cũng như trên mọi hệ điều hành.
* **Một số thủ thuật trong thiết kế thuật toán đa luồng:**
  + **Xác định đúng các tác vụ chạy độc lập**:
    - Các task chỉ có thể thực hiện đồng thời nếu như chúng độc lập với nhau.
    - Một trường hợp khác là một tác vụ có nhiều điều kiện tiên quyết và các điều kiện này không phụ thuộc vào nhau thì ta có thể thực hiện các điều kiện này đồng thời sau đó đồng bộ thứ tự thực hiện các bước tiếp theo sau khi hoàn thành các điều kiện này.
  + **Triển khai đa luồng ở mức cao nhất có thể :** Việc này đem lại các lợi ích
    - Không cần lo lắng về việc tạo và kiểm soát các thread
    - Tối ưu hiệu năng so với việc sử dụng thread trực tiếp
    - Bao gồm các đặc tính tiên tiến giúp cho API mạnh mẽ hơn
    - Dẽ dàng để chuyển các ứng dụng từ OS này sang OS khác
    - Các ứng dụng có thể nhanh hơn trong các phiên bản java tiếp theo.
  + **Tính đến khả năng mở rộng:**
    - Mục đích chính của các thuật toán đa luồng là sử dụng tối đa tài nguyên của hệ thống nên cần tính đến trường hợp tài nguyên của hệ thống được nâng cấp (số nhân của CPU).
    - Trường hợp thiết kế thuật toán đa luồng sử dụng phân tách dữ liệu (data decomposite) thì nên lấy được tài nguyên của hệ thống ví dụ như số nhân, bộ xử lý để có thiết kế phù hợp.
    - Trường hợp thiết kế thuật toán đa luồng sử dụng phân tách tác vụ (task decomposite) cần tính toán số lượng luồng đủ lớn để tận dùng tài nguyên hệ thống cũng như tránh quá tải gây là giảm hiệu năng.
  + **Sử dụng thread-safe API:** Khi sử dụng thư viện JAVA trong các ứng dụng đa luồng cần tìm hiểu xem có phải là thread-safe hay không, nếu không cần làm theo các bước sau:
    - Sử dụng một thread-safe khác thay thế
    - Nếu không có thread-safe thay thế nào, cần sử dụng cơ chế đồng bộ để tránh các lỗi liên quan đến đa luồng đặc biệt là race condition
  + **Đừng bao giờ giả sử các luồng sẽ được thực hiện theo 1 thứ tự:** Thứ tự thực hiện các luồng trong 1 ứng dụng chạy đa luồng trong trường hợp ko sử dụng sự đồng bộ là 1 cơ chế ko thể tự quyết định được.
    - Thứ tự thực hiên hoặc việc lập lịch cho các luồng do hệ điều hành quyết định nó, thứ tự này có thể thay đổi liên lục
    - Hậu quả của việc giả thiết các luồng chạy theo thứ tự là gây ra race condition
  + **Sử dụng các biến thread local thay vì biến static hoặc biến dùng chung:**
    - Các biến cục bộ của luông (thread local variable) là một kiểu biến đặc biệt: Mỗi luông sẽ có 1 giá trị riêng cho biến này bởi vậy ta ko cần cơ chế đồng bộ cho việc truy cập đến biến này.
  + **Tìm kiếm các phiên bản đa luồng dễ dàng cho việc triển khai;**
  + **Sử dụng đối tượng không thể thay đổi (immutable object) khi có thể:** đối tượng ko thể thay đổi là các đối tượng mà ta không thể sửa đổi các thuộc tính của nó sau khi khởi tạo. Nếu muốn thay đổi các thuộc tính của đối tượng này ta phải tạo đối tượng mới
    - Sử dụng immutable object có thể tránh được lỗi race condition
    - Ưu điểm:
      * Không cần các cơ chế đồng bộ cho các phương thức của class
      * Không xảy ra trường hợp dữ liệu không nhất quán (data inconsistency )
    - Nhược điểm: Khi tạo quá nhiều đối tượn sẽ ảnh hướng đến hiệu năng và bộ nhớ của hệ thống
  + **Tránh khóa chết bằng cách khóa theo thứ tự:** Cách thự hiện đơn giản nhất là gán số thứ tự cho các tài nguyên . khi 1 luồng muốn sử dụng nhiều tài nguyên chúng cần phải yêu cầu theo thứ tự **.**
  + **Sử dụng biến nguyên tử (atomic variable) thay vì cơ chế đồng bộ :**
    - Atomic variable được bổ sung trong API ở phiên bản JAVA 5 để hỗ hợ các hoạt đọng nguyên tử trên 1 biến
    - Đôi khi ta có thể sử dụng từ khóa volatile thay vì cơ chế đồng bộ trong trường hợp chỉ có 1 luồng ghi và các luồng khác chỉ đọc
  + **Giữ khóa đối tượng trong thời gian ngắn nhất:**để tránh làm ảnh hưởng đến hiệu năng của hệ thống
  + **Có các biện pháp phòng ngừa khi sử dụng khởi tạo muộn (lazy initialization)**
    - Ưu điểm của *lazy initialization*  là tiết kiệm bộ nhớ vì chỉ khi cần sử dụng đói tượng ta mới khởi tạo
    - Tuy nhiên có một vấn đề là khi 2 luông cùng yêu cầu khởi tạo sẽ tạo ra 2 object khác nhau ảnh hưởng đến singleton class
  + **Tránh sử dụng hành động chặn bên trong phần được bảo vệ:**
    - Hành động chặn được hiểu là chặn các task gọi đến chúng cho đến khi một sự kiện nào đó xảy ra ví dụ như việc đọc/ghi từ file ra console các task muốn gọi đến file này cần phải đợi cho đến khi hành động đọc/ghi được thực hiện xong

1. **QUẢN LÝ ĐA LUỒNG**
   1. **Giới thiệu về executors**

* Có 2 cách để sử dụng đa luông trong JAVA
  + Class implement Runable inteface:
  + Sử dụng 1 instance của lớp Thread
* Với 2 cách này rất khó để tạo và quản lý thread hoặc quản lý cơ chế đồng bộ giữa 2 thread với nhau nếu ta tạo quá nhiều thread có thể làm ảnh hưởng đến hiệu năng của cả hệ thống

1. **Đặc trưng cơ bản của executors**

* Không cần phải tạo các đối tượng Thread, nếu muốn thực thi task đồng thời, ta chỉ cần tạo instance của task và gửi chúng vào executors, nó sẽ tự quản lý luồng thực thi task này
* Executor sẽ giảm thiểu được việc quá tải bằng cách tái sử dụng thread thay vì khởi tạo chúng, nó quản lý một thread pool gọi là worker-thread
* Dễ dàng quản lý tài nguyên bằng executors bằng cách giới hạn số thread tối đa trong pool nếu gửi nhiều task hơn so với số thread nó sẽ được gửi vào queue và khi 1 thread thực hiện xong task nó sẽ lấy task mới từ queue ra để thực thi
* Bạn phải kết thúc sự thực thi của 1 executor một cách rõ ràng, cần chỉ rõ rằng khi nào thì executor sẽ kết thúc quá trình thực thi và kill tất cả các thread nếu không executor sẽ không kết thúc và ứng dụng của bạn cũng sẽ không kết thúc

1. **Cấu tạo cơ bản của executors framework**

* **Executor inteface:** định nghĩa các method cơ bản cho phép gửi Runnable object vào executor
* **ExecutorService inteface:** bổ sung thêm các hàm chức năng
* **ThreadPoolExecutor class:**  bao gồm các method cho phép lấy được trạng thái của executor, các method thiết lập các tham số cho executor và các method cho phép mở rộng và đáp ứng các chức năng
* **Executors class:** cung cấp các hàm tiện ích cho việc tạo đối tượng *Executor*  và các class liên quan

1. **GETTING MAXIMUM FROM EXECUTORS**
   1. **Đặc tính nâng cao của executors**
2. *Hủy tác vụ*

* Ta có thể hủy thực thi một task sau khi đã gửi nó vào executor bằng cách sử dụng hàm *cancel()*
* Có một số trường hợp không thể hủy task:
  + Task đã bị hủy
  + Task đã được thưc thi xong
  + Truyền tham số *false* vào hàm *cancel()*
  + Một số lý do khác

1. *Lập lịch thực thi cho task*

* Java cung cấp đặc tính cho phép lập lịch cho việc thực thi các task: *ScheduledThreadPoolExecutor.* Trong đó, ta có thể:
  + Thực thi task sau 1 khoảng trễ
  + Thực thi task theo chu kỳ: fixed rate hoặc fixed delay

1. *Ghi* *đè phương thưc của executor*

* beforeExecute(): được triệu hồi trước khi thực thi 1 task trong executos
* afterExecutor(): được triệu hồi ngày sau khi thực thi1 task trong executor
* newTaskFor(): Tạo 1 task mới sẽ thực thi Runable Object gửi vào executor bằng hàm submit

1. *Thay đổi một số tham số khởi tạo*

* BlockingQueue<Runable>: là queue mặc định chứa các task đang đợi thực thi
* ThreadFactory: được dùng để tạo các thread cho executor
* RejectExecutionHandler: sau khi gọi hàm shutdown() hoặc shutdownNow() thì tất cả các task được gửi vào executor sẽ bị từ chối

1. **Lấy dữ liệu từ các tasks - Callable và Future Interface**
   1. ***Giới thiệu về Callable và Future Interface***

* Với executor framework cho phép ta làm việc với các task mà không cần khởi tạo và quản lý các thread. Ta chỉ cần tạo các task và gửi chúng vào executor. Executor có thể xử lý 2 loại task:
  + *Task dựa trên Runable interface* : task triển khai hàm *run()* và không trả về kết quả
  + *Task dựa trên Callable interfacte*: task triển khai hàm *call()* và trả về kết quả là một Object.Kiểu trả về sẽ được xác định dựa trên kiểu tham số generic của Callable interface.

1. *The callable interface*

* Các đặc điểm chính của interface:
  + Là một generic interface. Có một tham số đơn là kiểu dữ liệu tương ứng với kiểu trả về của hàm *call().*
  + Cần khai báo hàm *call():* hàm này sẽ đươc thực thi bởi executor khi nó thực thi task và cần trả về một đối tượng của kiểu dữ liệu đã khai báo.
  + Hàm call () có thể ném ra một checked exception. Ta có thể triển khai ngoại lệ này trong executor bằng các ghi đè lại phương thức *afterExecute()*

1. *The future interface*

* Khi ta gửi 1 Callable task vào executor thì nó sẽ trả về 1 triển khai của Future Interface cho ta kiểm soát qua trình thực thi và trạng thái của task để lấy ra kết quả.
* Các đặc điểm chính của future interface:
  + Có thể hủy task bằng cách gọi hàm *cancel()*
  + Có thể kiểm tra task được hủy hoặc đã hoàn thành hay chưa.
  + Có thể lấy giá trị trả về bằng hàm *get() .* Có 2 phương thức riêng biệt:
    - Hàm get() không có tham số đầu vào: trả về giá trị nếu task đã thực thi xong
    - Hàm get() chứa 2 thám số: chu kỳ và TimeUnit của chu kỳ, nếu sau khoản thời gian này mà task chưa thực hiện xong thì sẽ ném ra một ngoại lệ *TimeoutException*

1. **Thực thi các task phân chia theo giai đoạn- Phase class trong JAVA**
   1. ***Giới thiệu về phaser Class***

* Phaser Class là một cơ chế bất đồng bộ được thiết kế để thực hiện các thuật toán bằng cách chia thanh nhiều giai đoạn trong cách thực thi đa luồng.
* Được sử dụng trong trường hợp nếu một tiến trình có thể được chia thành nhiều giai đoạn mà cần phải thực hiện phaser 1 trước khi chuyển sang giai đoạn 2
* Các đặc điểm chính của class Phaser:
  + Phaser cần phải biết được số lượng task nó kiểm soát. Java mô tả điểu này bằng cách ghi danh các người tham gia (participant). 1 người tham gia có thể đăng ký tại bất kỳ thời điểm nào
  + Task phải thông báo cho phaser khi nó hoàn thành 1 phase. Phaser sẽ thực hiện làm cho các task này sleep cho đến khi
  + Phaser lưu một số tự nhiên để đếm số phase nó đã thay đổi
  + Người tham gia có thể rời khỏi sự kiểm soat của phaser tại bất kỳ thời điểm nào. Java tham chiếu hành động đó như một sự hủy đăng ký của một người tham gia
  + Ta có thể thực thi custome code khi phaser thực hiện sự thay đổi giữa các pharse
  + Ta có thể kiểm soát sự kết thúc của phaser. Khi phaser kết thúc sẽ không có người tham gia mới nào được phép và không có sự đồng bộ nào giữa các task được tạo ra.
  + Có thể sử dụng một vài hàm để biết được trạng thái và số người tham gia trong phaser.

1. *Đăng ky và hủy đăng kí của người tham gia*

* Người tham gia được đăng kí tại thơi điểm ban đầu của việc thực thi, nhưng một người tham gia có thể đăng ký tại bất kỳ thời điểm nào .
* Việc đăng ký có thể sử dụng các phương thức nhau:
  + Constructor: Phaser() hoặc Phaser(int parties): tạo ra 1 phaser với số người tham gia xác định
  + bulkRegister(int parties): đăng ký với số người tham gia xác định tại 1 thời điểm
  + register(): đăng ký cho người tham gia mới
* Khi 1 task hoàn thành việc thực thi nó cần phải hủy đăng ký khỏi phaser hiện tại, nếu không phaser sẽ đợi kết thúc cho đến khi phaser tiếp theo thay đổi.
* Để hủy đăng ký , có thể gọi hàm arriveAndDerigister() để chỉ thị rằng task đã hoàn thành và sẽ không tham gia trong phase tiếp theo

1. *Đồng bộ sự thay đổi các phaser*

* Class Phaser cung cấp 3 method để ra tín hiệu task đã hoàn thành phase, nếu 1 task không được gọi một trong các method này các task tham gia còn lại sẽ bị chặn bởi phaser một mãi mãi
* Để tham gia phase tiếp theo, có thể sử dụng một trong các method sau:
  + arriveAndAwaitAdvance(): task đã hoàn thành phase hiện tại và đợi sang phase tiếp theo
  + awaitAdvance(int phase): hàm chỉ thị rằng phaser cần đợi hoàn nếu thành số task được cấu hình từ tham số đầu vào bằng với số task thực tế,nếu không bằng thì hàm sẽ return ngay lập tực

1. *Các hàm khác*

* Khi tất cả các task đã hoàn thành 1 phase và trước khi thực hiện phase tiếp theo nó sẽ thực thi hàm onAdvance(). Hàm này nhận 2 tham số đầu vào:
  + Phase: số lượng phase đã hoàn thành, phase đầu tiền sẽ là 0.
  + registeredParties: số lượng task tham gia
* Nếu muốn thực thi một đoạn code giữa 2 phase thì cần triển khai 1 phase (extend class Phaser) và ghi đè method.
* Một phase sẽ có 2 trạng thái:
  + ***Active :*** Phaser ở trạng thái này khi nó được tạo ra và task tham dự mới được đăng kí và tiếp tục thực hiện cho đến khi kết thúc. Ở trạng thái này phaser sẽ chấp nhận task tham gia mới.
  + ***Terminal:*** Phaser ở trạng thái này nếu hàm *onAdvance()* trả về *true .* Mặc định hàm này trả về true nếu như tất cả các task tham gia được hủy ghi danh.
* Ngoài ra Phaser còn có một số hàm khác như: getRegisterParties(), getPhaser

1. **Giải pháp cho việc tối ưu chia và trị- Fork/Join framework**
2. **Processing Massive Datasets with Parallel Streams – TheMap and Reduce Model**
   1. ***Giới thiệu về stream***

* Stream là một dữ liệu tuần tự cho phép thực hiện các chuỗi các hành động theo cách tuần tự hoặc song song.
* Stream được sử dụng nhiều trong các ngôn ngữ lập trình hướng chức năng như Scala

1. Đặc tính cơ bản

* Stream không chứa các thành phần, nó chỉ lấy các thành phần từ source rồi gửi vào các phép toán thông qua mô hình gọi là đường ống (pipline)
* Có thể làm việc với stream mà không cần phải làm thêm bất cứ việc gì. Hàm *stream()*  để tạo một stream tuần tự hoặc *parallelStream()* đê tạo một stream thực hiện song song. Có thể chuyển đôi stream tuần tự và stream song song bất cứ lúc nào.
* Không thể tác động vào stream để chúng thực hiện các hành động là tuần tự hay song song, Stream song song sử dụng Fork/Join framwork để thực thi các phép toán song song.
* Stream được sử dụng một tốt trong lập trình hàm, có thể sử dụng lambda expression như một các đẻ thực thi các thuật toán trong tác phép toán trên stream.
* Stream không thể tái sử dụng. Nếu bạn muốn thực thi một phép toán khác trên cùng dữ liệu, bạn cần phải tạo ra một stream mới.
* Stream thực hiện lazy processing với data, có nghĩa stream chỉ lấy data khi cần thực hiện.
* Không thể truy cập các phần tử của stream. Stream thường xử lý các phần tử một cách đồng nhất. Bạn không thể biết vị trí của phần tử cũng như các thành phần bên cạnh nó. Trong trường hợp stream là tuần tự các thành phần có thể thực thi theo một các được sắp xếp bất kỳ.
* Stream không cho phép sửa dữ liệu nguồn. Khi stream sử dụng 1 dữ liệu nguồn nếu muốn lưu trữ kết quả nó sẽ phải tạo ra một list khác, điều này cho phép ta có thể trả về kết quả là một list dữ liệu khác mà không ảnh hưởng đến dữ liệu nguồn.

1. Các phần của stream:

* Source of Stream: tạo ra dữ liệu sẽ được thực hiện bởi Stream Object.
  + Có thể tạo ra stream từ nhiều nguồn:
    - Collection như ArrayList, LinkedList, HashSet…
    - Array: nếu tạo một stream từ mảng int sẽ tạo ra một IntStream.
  + Có Thể tạo và sử dụng stream để xư lý nội dung của một thư mục hoặc File
* Phép toán trung gian :
  + Một phép toán trung gian sẽ return kết quả là một stream và luôn luôn tạo ra kết quả là một stream.
  + Một số phép toán quan trọng bao gồm:
    - *distinct()* ; trả về một stream với các giá trị không trùng lặp
    - *filter()*: trả về một stream với các thành phần thỏa mãn điều kiện
    - *flatMap()*: sử dụng để biến đổi một stream của các stram stream (ví dụ List stream,Set stream) thanh một stream đơn
    - *limit():* trả về một stream chứa số lượng xác định các thành phần.
    - *map():* biến đổi phần tử của một stream từ kiểu dữ liệu này sang kiểu dữ liệu khác
    - *peek():* trả về cùng một stream, nhưng nó được xử lý một vài đoạn code, thông thường được dùng để ghi log dữ liệu
    - *skip():* bỏ qua phần tử ban đầu (hoặc số lượng phần tử truyền vào trong tham số) của stream
    - *sorted():* sắp xếp các phần tử của stream.
* Phép toán kết thúc:
  + phép toán sẽ trả về kết quả là một đối tượng không trả về một stream.
  + Các phép toán kết thúc quan trọng nhất bao gồm:
    - *collect():* phép toán giảm số lượng các thành phần của dữ liệu nguồn theo một cách xác định.
    - *count(): trả về số lượng các phần tử*
    - *min(), max(): trả về giá trị lớn nhất và nhỏ nhất.*
    - *reduce(): biến đổi các thành phần của stream thành một đối tượng không trùng lặp*
    - *forEach/forEachOrder(): áp dụng một hành động trên tất cả các phần tử của một stream.*
    - *findFirst()/findAny(): trả về một phần tử hoặc một phần tử đầu tiên nếu chúng tồn tại.*
    - *anyMatch()/allMatch()/noneMatch(): nhận một thuộc tính như một tham số đầu vào và trả về một giá trị Boolean nếu có bất kỳ/tất cả/ không có phần tử nào trong stream thỏa mãn điều kiện.*
* ***MapReduce vs MapCollect:***
  + MapReduce là một mẫu lập trình để xử lý một lượng dữ liệu rất lớn trong môi trường phần tán với nhiều server hoạt động theo cluster. Kỹ thuật này bao gồm 2 bước:
    - Map: lọc và biến đổi dữ liệu.
    - Reduce: áp dụng các phép toán tổng hợp vào dữ liệu

1. **Diving into Concurent Data Structure and Synchronization Utilities**
   1. **Concurrent Data Structure**
      * Mọi phần mềm đều làm việc với dữ liệu, các chương trình sử dụng các cấu trúc dữ liệu để lưu trữ dữ liệu trên bộ nhớ
      * Trong môi trường đa luồng , cần cần trọng khi sử dụng các cấu trúc dữ liệu dùng chung giữa các thread, có thể gây ra lỗi race data conditional. Để tránh lỗi này, có thể sử dụng các phương pháp:
      * Sử dụng cấu trúc dữ liệu không đồng bộ và tự thêm cơ chế đồng bộ
      * Sử dụng cấu trúc dữ liệu cung cấp bởi các thư viện java , đã được bổ sung cơ chế đồng bộ và tối ưu hóa trong việc sử dụng trong các ứng dụng đa luồng. Cách này được khuyến khích sử dụng.
      * Cấu trúc dữ liệu blocking và non blocking:
      * Cấu trúc dữ liệu blocking: Cung cấp các hàm insert và delete, trong trường hợp các hành động này không thể thực hiện được thì thread được gọi sẽ được block cho đến khi thực hiện xong
      * Cấu trúc dữ liệu non-blocking: Cung cấp các hàm insert và delete, nếu các hành động này không được thực hiện ngay lập tức sẽ trả về kết quả đặc biệt hoặc ném ra một ngoại lệ