C,C++ Concurrency

Linh La

Table of Contents

[1. Giới thiệu 1](#_Toc481449173)

[2. Quản lý threads 10](#_Toc481449174)

[3. Chia sẻ dữ liệu 18](#_Toc481449175)

[4. Đồng bộ các tiến trình 41](#_Toc481449176)

### Giới thiệu

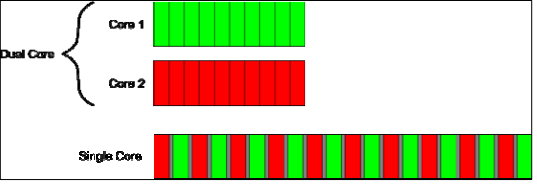
##### Thế nào là xử lí tranh chấp

Trên các hệ thống đơn lẻ, chúng xử lí các tác vụ một cách độc lập và đồng thời, giống như kiểu tuần tự hơn. Đây không phải hiện tượng mới: hệ điều hành đa nhiệm cho phép máy tính đươn lẻ chạy nhiều ứng dụng một lúc qua việc chuyển tác vụ (task switching), một thao tác phổ biến.

Trên các máy nhiều vi xử lí mới thực sự gọi là xử lí tranh chấp.

Trong lịch sử, hầu hết mấy tính chỉ có 1 vi xử lí, chỉ thực thi một tác vụ một thời điểm, nhưng chúng có thể task switching nhiều lần một giây, tức làm cái này một chút, rồi cái kia một chút, và diễn ra rất nhanh làm chúng ta có cảm giác rằng chúng đang xử lí đồng thời.

Các máy tình hiện đại được xây dựng trên hệ thống một vi xử lý nhưng nhiều nhân, và dần trở nên phổ biến hơn. Nghĩa là chúng có thể thực thi nhiều tác vụ đồng thời, chứ không phải tuần tự. Chúng ta gọi nó là sự xử lí tranh chấp về phần cứng.

Hình dưới nêu ý tưởng về cách hoạt động của máy đơn nhân và hai nhân. Trên máy hai nhân, mỗi tác vụ thực thi trên mỗi trên của nó. Máy đơn nhân phải làm task switching.

Điểm khác biệt là ở máy đơn nhân có một khoảng lặng giữa các tác vụ (thanh màu xám), khoảng lặng này là thời gian hệ thống làm thao tác chuẩn bị chuyển ngữ cảnh mỗi lần task-switching. Tức hệ điều hành phải lưu trạng thái CPU và con trỏ chỉ thị tác vụ đang chạy, sau đó bắt đầu chuyển qua thực thi tác vụ khác. Khi trở về tác vụ cũ, nó nạp lại trạng thái CPU, tìm con trỏ chỉ thị đang làm dở và thực thi tiếp.

CPU có khả năng cấp vùng nhớ cho các chỉ thị xử lí và dữ liệu cho tác vụ mới vào cache, giúp CPU có thể thực thi bất kì chỉ thị nào mà không xảy ra độ trễ.

Do đó, khả năng thực thi đồng thời phần cứng là hiển nhiên với đa vi xử lí hay đa nhân, một số vi xử lí có thể thực thi nhiều thread trên một core. Nhân tố quan trọng để xem xét là số threads cứng: thước đo xem bao nhiêu tác vụ độc lập mà phần cứng có thể chạy đồng thời. Thậm chí với hệ thống có phần cứng xử lí đồng thời, sẽ rất dễ dàng làm nhiều tác vụ song song hơn, task-switching vẫn được dùng.

Ví dụ, máy để bàn thông thường có hàng trăm tác vụ, vẫn chạy ngầm, dù máy tính đang trạng thái chờ. Task switching cho phép tác vụ chạy nền, cho phép bạn chạy đủ thứ: xử lí văn bản, biên dịch, biên tập và duyệt web một lúc.



Hình trên minh họa task switching gồm 4 tác vụ độc lập trên máy 2 nhân, các tác vụ được chia đều cho 2 nhân. Trên thực tế, có nhiều phát sinh khiên cho việc phân chia tác vụ không đồng đều. Một số sẽ được đề cập trong chương 8, khi xem xét các nhân tố ảnh hưởng đến hiệu suất của mã lệnh thực thi đồng thời.

Tất cả kỹ thuật, hàm và lớp trong tài liệu này có thể dùng cho hệ máy một nhân hay đa nhân, không bị ảnh hưởng bởi yếu tố đồng thời có đạt được hay không qua task switching hay bởi phần cứng hỗ trợ xử lí đồng thời hay không.

Tuy nhiên, bạn cũng phải hình dung làm thế nào sử dụng được khả năng xử lí đồng thời trên ứng dụng của bạn khi phải phụ thuộc vào khả năng phần cứng có cho phép hay không. Điều này được đề cập ở chương 8, khi thiết kế câu lệnh bằng C++.

##### Phương pháp xử lí tranh chấp

1. QUẢN LÝ TỔ CHỨC

Hình dung hai lập trình viên làm việc trên cùng một project. Trường hợp mỗi người ở một văn phòng khác nhau, họ có thể tiến hành công việc độc lập, không làm phiền nhau, và có tài liệu tham khảo riêng. Tuy nhiên, vấn đề giao tiếp thì không thông suốt: giống như là nói chuyện qua qua, hời hợt, họ dùng phone và email hay thức dậy, đi dạo. Và bạn cũng quá tải khi phải kiểm soát hai văn phòng, hàng tá tài liệu tham khảo khác nhau phải mua.

Bây giờ tưởng tượng rằng bạn chuyển các lập trình viên vào chung một phòng. Họ có thể nói chuyện với nhau thoải mái cũng như thảo luận thiết kế chương trình, và có thể dễ dàng vẽ biểu đồ trên giấy hay bảng trắng để giúp minh họa ý tưởng hay khi giải thích với đồng nghiệp. Giờ bạn chỉ phải quản lý một văn phòng, một bộ tài liệu.

1. VẤN ĐỀ PHÁT SINH

Khá tiện nhưng có một rắc rối, nhân viên sẽ khó tập trung làm việc, phát sinh trong việc cùng chia sẻ tài liệu làm việc (thỉnh thoảng có ai đó lại hỏi “tài liệu làm việc đi đâu goài?” blab la bla).

Có hai giải pháp tổ chức văn phòng cho bạn, cũng như hai phương pháp cơ bản để thực thi đồng thời trên máy tính. Mỗi lập trình viên đại diện cho một threads, văn phòng là bộ vi xử lí.

* Cách đầu tiên là tổ chức thành nhiều bộ xử lí đơn thread, giống như mỗi lập trình viên ngồi trong phòng chính anh ta.
* Cách thứ hai là tổ chức thành một bộ xử lí đa thread, giống như hai lập trình viên ngồi một phòng.

Bạn có thể kết hợp tùy ý các kiểu và có nhiều bộ xử lí, một số sẽ là đa luồng, và một số sẽ là đơn luồng, nhưng cơ chế thì giống nhau. Hãy cùng quan sát hai phương pháp xử lí đồng thời trên ứng dụng

1. Xử lí đồng thời với nhiều bộ xử lí

Trong phương pháp đầu tiên, chúng ta sử dụng tính xử lí tranh chấp trong ứng dụng bằng cách chia ứng dụng thành nhiều tiến trình đơn thread riêng rẽ, cùng chạy, giống như bạn đang duyệt nhiều trang web cùng lúc vậy. Những tiến trình riêng rẽ có thể truyền tin nhắn cho nhau qua kênh giao tiếp nội bộ (IPC: signals, socket, files, pipes,…), minh họa trong hình dưới

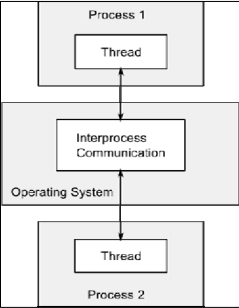


Figure 1.3 Communication between a pair of processes running concurrently

Khuyết điểm là cách giao tiếp giữa các process thường phức tạp để thiết lập lên, xuống hay cả hai, vì hệ điều hành thường cung cấp nhiều sự bảo vệ giữa các process để tránh một process vô tình làm thay đổi dữ liệu thuộc sở hữu process khác.

Mặt khuyết thứ hai là sự tiêu tốn hiệu suất khi chạy nhiều bộ xử lí: tốn thời gian để khởi động process, hệ điều hành phải để dành tài nguyên cho việc quản lý process và vân vân..

 Dĩ nhiên, cũng có mặt tốt: hệ điều hành có cơ chế bảo vệ giữa các process và cơ chế giao tiếp cao cấp hơn, tức dễ dàng hơn để viết các mã lệnh xử lí đồng thời an toàn với nhiều process hơn là thread. Môi trường tương tự như vậy được sử dụng bởi ngôn ngữ lập trình Erlang dùng các process như là đơn vị cơ sở để xử lí đồng thời, và rất hiệu quả.

Ngoài ra bạn có thể chạy các process riêng rẽ trên hệ máy riêng biệt kết nối với nhau qua mạng. Nhưng tăng chi phí giao tiếp, khó thiết kế hệ thống để hoạt động hiệu quả trên nền tảng song song và cải thiện hiệu suất.

1. Xử lí đồng thời với đa luồng

Phương pháp thứ hai triển khai trên một bộ xử lí đơn lẻ nhưng hỗ trợ đa luồng. Thread khá giống với process con – mỗi thread chạy độc lập với nhau, và có thể chạy các chỉ thị khác nhau. Tuy nhiên, tất cả threads trong một process cùng chia sẻ vùng địa chỉ, và dữ liệu có thể truy xuất trực tiếp từ tất cả các threads – biến toàn cục vẫn ở phạm vi toàn cục, và con trỏ hay tham chiếu tới đôi tượng/dữ liệu có thể truyền qua lại giữa các thread.

Dù vậy, thỉnh thoảng có trường hợp phải chia sẻ vùng nhớ giữa các process, ngữ cảnh này khá phức tạp để thiết lập, và khó quản lý, như vùng địa chỉ dữ liệu không cần thiết phải giống nhau/chia sẻ giữa các process.

Hình bên dưới minh họa hai thread trong một process giao tiếp với nhau thông qua vùng nhớ chia sẻ

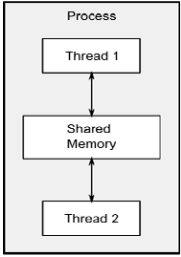


Figure 1.4 Communication between a pair of threads running concurrently in a single process

Vùng địa chỉ được chia sẻ và sự thiếu tính bảo vệ dữ liệu giữa thread nên ít tốn hiệu suất. Tính linh hoạt về dùng chung vùng nhớ có cái giá của nó: nếu dữ liệu truy xuất bởi nhiều thread, lập trình viên phải đảm bảo sự nhất quán các thread. Vấn đề phát sinh nằm ở dữ liệu dùng chung và công cụ , chỉ dẫn giúp tránh được vấn đề đã đề cập ở phần 3, 4, 5 và 8.

Vấn đề không thể khắc phục được, phải cẩn thận khi coding, nhưng chúng rất đáng để chúng ta tìm ra giải pháp giao tiếp giữa các thread.

Chi phí thấp thể hiện ở việc khởi động và giao tiếp giữa các thread trong một process, hơn là giữa các vi xử lí đơn thread. Điều mà trở thành ưu tiên lớn cho việc tìm giải pháp triển khai trên các ngôn ngữ lập trình chính, bao gồm C++, nhằm vào vấn đề chia sẻ vùng nhớ. Thêm nữa, C++ chuẩn không cung cấp bất kì hỗ trợ nào về nội giao tiếp giữa các process, nên ứng dụng dùng nhiều process sẽ phải dựa trên API của platform.

Rất tiếc là tài liệu này chỉ đề cập về xử lí đa luồng.

##### Tại sao phải xử lí đồng thời?

Có hai lý do chính: chia nhỏ vấn đề ra và bài toán hiệu suất.

Thực tế, có nhiều lý do để dùng, tùy theo ý muốn của bạn thôi.

1. Chia nhỏ vấn đề

Đây luôn là ý tưởng tốt khi viết phần mềm: nhóm các đoạn code liên quan lại với nhau, bằng cách giữ các đoạn code không liên quan tách biệt, chúng ta có thể dễ dàng test chương trình, debug…

Chúng ta có thể xử lí đồng thời cho một khu vực code riêng biệt, vì chúng cần phải chạy cùng lúc để hỗ trợ lẫn nhau.

Nếu không có xử lí đồng thời, chúng ta phải viết framework task-switching, xử lí tuần tự, hay phải gọi thực thi vùng code không liên quan trong quá trình chạy.

Xem xét một ứng dụng có mức độ xử lí cao với giao diện người dùng, như DVD player cho máy bàn. Những ứng dụng như vậy thường có hai nhiệm vụ chính

* Đọc dữ liệu từ đĩa, giải mã hình ảnh và âm thanh
* Gửi dữ liệu giải mã tới phần cứng đồ họa và âm thanh theo trình tự thời gian để không bị giựt hình/tiếng.

Nhưng nó vẫn phải xử lí tương tác với người dùng, như khi bấm “pause”, “return to menu” hay “quit”. Nếu là thread đơn lẻ, ứng dụng phải kiểm tra thao tác người dùng định kì trong suốt thời gian chơi đĩa, tức là đoạn code chơi DVD và xử lí giao diện người dùng xen kẽ vào nhau.

Khi dùng xử lí đa luồng, chúng ta tách code xử lí tương tác giao diện người dùng riêng biệt với code xử lí DVD. Đương nhiên là phải có sự tương tác giữa hai thread, khi bấm “pause”, sẽ tác động tới thread xử lí DVD.

Ví dụ trên minh họa về sự phân chia trách nhiệm, công việc, như thread xử lí tương tác giao diện sẽ phản hồi ngay lập tức yêu cầu, thậm chí là hiển thị con trỏ đang bận hay tin nhắn “please wait” trong khi yêu cầu phải chờ cho thread làm việc.

Tương tự, các thread riêng rẽ thường dùng để chạy tác vụ ngầm, tức cần phải chạy liên tục, như giám sát hệ thống file khi có bất kì thay đổi nào trên ứng dụng tìm kiếm desktop. Dùng thread theo cách này nói chung sẽ đơn giản hóa công việc nhờ phân chia logic các thread, sự tương tác giữa chúng được qui định rạch ròi

Trong trường hợp này, số lượng threads độc lập với số lượng nhân CPU, vì chia thread sẽ dựa trên sự thiết kế phần mềm hơn nỗ lực làm tăng đầu ra.

1. Bài toán hiệu suất

Hệ thống nhiều bộ vi xử lí tồn tại qua nhiều thập nhiên, nhưng cho tới gần đây, chúng ta mới đặt ra nền tảng về siêu máy tính, mainframes và hệ thống máy chủ khổng lồ. Tuy nhiên, nhà sản xuất chip phải gia tăng thiết kế đa nhân lên 2, 4 , 16 hay thêm nhiều bộ xử lí trên một chip đơn để đạt hiệu suất tốt hơn.

Do đó, máy tính đa nhân trở nên phổ biến, mức tiêu thụ điện nằng cũng tăng lên nhưng máy tính chạy tác vụ đơn nhanh hơn, nhưng do chạy đa nhiệm song song.

Trong quá khứ, lập trình viên có thể ngồi và xem chương trình chạy nhanh hơn mỗi khi hệ chip xử lí mới ra đời, mà không có nỗ lực nào để cải tiến cả, như Herb Sutter nói :”Bữa trưa miễn phí đã hết”. Nếu phần mềm được hưởng lợi từ sự gia tăng tiêu thụ điện năng, thì nó phải được thiết kế để chạy đa nhiệm song song. Lập trình viên phải đầu tư, và những ai đang bỏ qua điều này sẽ phải xem xét lại.

Có hai cách để gia tăng hiệu suất với xử lí đồng thời.

* Đầu tiên, hiển nhiên là chia tác vụ thành các phần nhỏ, chạy song song, giảm thời gian đợi. Đây gọi là xử lí tác vụ song song.  
  Nghe khá đơn giản, nó có thể là process phức tạp, vì có nhiều sự lệ thuộc giữa các phần nhỏ này. Sự phân chia này kiểu như xử lí – một thread thực thi một phần của giải thuật, trong khi thread khác làm phần khác.
* Hay theo kiểu dữ liệu – mỗi thread thực thi cùng một hàm trên các phần khác nhau của dữ liệu, đây gọi là xử lí dữ liệu song song

Giải thuật là thứ dễ ảnh hưởng tới xử lí song song thường được gọi là Embarrassingly Parallel. Vì sự triển khai

Algorithms which are readily susceptible to such parallelism are frequently called  
*Embarrassingly Parallel*. Despite the implications that you might be embarrassed to have  
code so easy to parallelize, this is a good thing: other terms I've encountered for such  
algorithms are *naturally parallel* and *conveniently concurrent.* Embarrassingly parallel  
algorithms have very good scalability properties — as the number of available hardware  
threads goes up, the parallelism in the algorithm can be increased to match.

Những giải thuật như vậy là ý tưởng hoàn hảo cho việc “Chung tay làm công việc nhẹ hơn”. Với những phần mà giải thuật không phải dạng embarrasingly parallel, bạn có thể chia giải thuật thành các tác vụ song song cố định(do đó không có tính co giãn). Kỹ thuật phân chia tác vụ giữa các thread sẽ đề cập ở chương 8.

Cách tăng hiệu suất thứ hai để xử lí song song giải quyết các vấn đề lớn – thay vì xử lí từng file một, chúng ta xử lí hai hay mười hay hai mươi. Đây thực sự chỉ là ứng dụng về xử lí dữ liệu song song, bằng cách thực thi cùng các tiến trình trên các bộ dữ liệu đồng thời..

Nó vẫn tốn cùng một lượng thời gian để xử lí một dãy dữ liệu, nhưng dữ liệu được xử lí nhiều hơn. Dĩ nhiên, có giới hạn là không hiệu quả cho mọi trường hợp, nhưng nó làm tăng hiệu quả xử lí như videos, các khu vực của tấm hình có thể xử lí đồng thời

1. Tại sao không dùng xử lí tranh chấp

Theo lý thuyết, chỉ có lý do duy nhất không sử dụng là khi lợi ích mạng lại ít hơn chi phí bỏ ra. Các đoạn code xử lí như vậy rất khó hiểu trong nhiều trường hợp, nên tốn thời gian, suy nghĩ để viết và bảo trì, đồng thời vì sự phức tạp nên cũng dẫn tới nhiều lỗi hơn.

Nếu hiệu suất đem lại không đủ lớn hay các vấn đề không đủ tính minh bạch để tách riêng rẽ và tốn thời gian lập trình, chi phí bỏ ra không tương xứng thì bạn dừng dùng xử lí tranh chấp.

Ngoài ra, hệ thống phải tốn chi phí khi khởi động một thread, hệ điều hành phải cấp phát tài nguyên kernel tương ứng và vùng nhớ stack, và sau đó đưa thread này lên bộ lập biểu, tất cả điều đó tốn khá nhiều thời gian. Nếu tác vụ chạy trên thread kết thúc nhanh chóng, thì thời gian chuẩn bị cho thread trở nên tốn công vô ích.

Thêm nữa, thread là tài nguyên có giới hạn. Nếu bạn có quá nhiều thread chạy cùng lúc, nó tiêu tốn tài nguyên OS, và làm hệ thống tổng thể chạy chậm đi. Không những thế, dùng nhiều thread có thể làm cạn kiệt vùng nhớ của process, vì mỗi thread cần một vùng nhớ stack riêng. Đây là vấn đề của bộ xử lí 32-bits vốn chỉ nhận được 4GB Ram: nếu mỗi thread chiếm 1MB stack (kích cỡ thông thường trên nhiều hệ thống), hệ thống chỉ có tối đa 4096 threads, và chưa kể đến vùng nhớ dành riêng cho static data và heap data.

Hệ thống 64-bits không giới hạn vùng nhớ, nhưng vẫn có giới hạn tài nguyên: để giới hạn threads, chúng ta phải dùng thread pools (chương 9), một viên đạn bạc.

Nếu phía server dùng mỗi threads cho một kết nối, lượng kết nối ít sẽ không có vấn đề gì cho tới khi có quá nhiều, tài nguyên hệ thống cạn kiệt nhanh chóng vì khởi động nhiều thread cho cùng một kỹ thuật quản lý phía server. Trong bối cảnh đó, dùng thread pool cẩn trọng có thể tối ưu hóa hiệu suất (chương 9).

Cuối cùng, càng nhiều thread, thì xuất hiện càng nhiều thao tác chuyển mà OS phải làm. Mỗi thao tác chuyển cảnh cần thời gian, số lượng thread tăng đều đến vào một thời điểm nào đó làm hiệu suất tổng thể ứng dụng chậm đi. Vì lý do này, nếu bạn muốn đạt hiệu suất tốt, nên điều chỉnh số lượng thread chạy phù hợp với phần cứng.

Tối ưu hiệu suất dùng phương pháp xử lí tranh chấp giống như một chiến lược – có khả năng cải thiện hiệu suất, nhưng cũng phức tạp hóa code, khó hiểu, và nhiều bug. Do đó nó chỉ tốt khi dùng cho trường hợp khủng hoảng về hiệu suất mà chúng ta lường trước được. Dĩ nhiên, hiệu suất luôn hay sự chi nhỏ giải quyết các vấn đề luôn là đối tượng xem xét cho thiết kế ứng dụng đa luồng.

##### Xử lí tranh chấp và đa luồng trên C++

Tiêu chuẩn hóa xử lí tranh chấp và đa luồng mới được đưa vào C++. Với C++0x, bạn có thể viết mã lệnh đa luồng mà không cần phải viện đến phần mở rộng xác định platform. Để hiểu nguồn gốc đằng sau các quyết định trong thư viện chuẩn mới C++, chúng ta phải dạo qua về lịch sử một chút.

1. Lịch sử đa luồng trên C++

Vào năm 1998, C++ chuẩn chưa hỗ trợ gì về thread, và tác động từ nhiều yếu tố ngôn ngữ được viết ra theo dạng cỗ máy trừu tượng tuần tự. Không chỉ vậy, hình mẫu vùng nhớ không được định nghĩa chính thức, nên bạn không thể viết ứng dụng đa luồng mà không có phần mở rộng cho trình biên dịch cho C++ chuẩn 1998.

Dĩ nhiên, trình biên dịch thương mại là miễn phí để thêm các phần mở rộng vào ngôn ngữ lập trình, và sự phổ biến của C APIs về đa luồng- như trong POSIX C chuẩn và Microsoft Window API – đã khiến nhiều nhà xuất bản trình biên dịch C++ hỗ trợ đa luồng với phần mở rộng hỗ trợ cho nhiều platform. Trình biên dịch hỗ trợ có giới hạn, cho phép sử dụng C API tương ứng cho mỗi platform, và đảm bảo thư viện runtime C++ (ví dụ như code cửa cơ chế kiểm soát lỗi) làm việc với các thread.

Do đó, vài nhà phát hành đã cung cấp hình mẫu vùng nhớ nhận diện được đa luồng chính thức, các vận hành của trình biên dịch và bộ vi xử lí trở nên tốt hơn, một lượng lớn chương trình đa luồng trên C++ được viết ra.

Chưa hài long về cách dùng C API trên mỗi platform riêng biệt để xử lí đa luồng, lập trình viên C++ phải tìm kiếm thư viện cho riêng họ nhằm cung cấp nền tảng hướng đối tượng xử lí đa luồng. Framworks như MFC và thư viện general-purpose C++ như Boost và ACE đã tích hợp một bộ thư viện C++ gồm các platform APIs và nền tảng cao hơn để xử lí đa luồng, đơn giản hóa thao tác trong lập trình.

Mặc dù thông tin chi tiết của thư viện lớp xem ra khá đa dạng, nhưng xét cụ thể về mặt khởi tạo thread, nhìn tổng quan thì khá giống nhau.

Một thiết kế quan trọng đặc thù, phổ biến trong nhiều thư viện C++, cung cấp những lợi ích cho lập trình viên, là sử dụng RAII (Resource Acquisition Is Initialization). Ý tưởng chính đằng sau RAII là một nguồn tài nguyên thu được vào thời gian khởi tạo hay không, thuộc sở hữu của một đối tượng, và đối tượng hủy sẽ tự động giải phóng tài nguyên ở một thời điểm thích hợp. Điều này cho phép C++ thông qua các RAII để hỗ trợ việc dọn dẹp các tài nguyên, từ một phương pháp giải phóng bộ nhớ nó cũng có thể được dùng để giải phóng các tài nguyên khác (file handles, mutexes, database connections, transactions, …).

**RAII** với **lock** để đảm bảo **mutexes** được mở khóa khi đã thoát ra khỏi phạm vi liên quan.

Cho nhiều trường hợp, đa luồng hỗ trợ trên trình biên dịch C++, kết hợp với các APIs platform và thư viện độc lập với platform như Boost, ACE cung cấp nền tảng tốt để viết code đa luồng với C++, và kết quả là có hàng triệu dòng lệnh C++ viết ra như một phần của ứng dụng đa luồng

Tuy nhiên, thiếu sự hỗ trợ của một mô hình quản lý vùng nhớ theo tiêu chuẩn có thể gây ra lỗi, đặc biệt đối với ai đang cố gắng giành hiệu suất cao bằng việc lạm dụng kiến thức về vi xử lí phần cứng, hay những ai viết những mã lệnh hỗ trợ nhiều platform mà đáng lẽ là để cho trình biên dịch làm việc này.

1. Xử lí tương tranh hỗ trợ trên C++ chuẩn

Không chỉ thiết kế trình quản lý vùng nhớ có khả năng nhận biết threads, mà thư viện C++ chuẩn còn được mở rộng thêm:

* lớp quản lý thread (chương 2)
* bảo vệ dữ liệu dùng chung (chương 3)
* đồng bộ hóa tiến trình giữu các threads (chương 4)
* tiến trình atomic cấp thấp (chương 5)

Thư viện về thread mới của C++ chú trọng về kinh nghiệm tích lũy từ trước qua cách dùng các thư viện lớp C++ đề cập ở trên. Cụ thể, thư viện threads của Boost được dùng làm bộ khung chính cho thư viện mới C++, các lớp, cấu trúc đều giống so với Boost.

Theo sự phát triển của Chuẩn mới, có hai trường phái, thư viện Boost thread tự thay đổi để phù hợp với thư viện C++ chuẩn, cho những ai dùng Boost cũng sẽ cảm thấy quen khi dùng C++.

Hỗ trợ xử lí tương tranh chỉ là một trong nhiều thay đổi của C++ chuẩn, có rất nhiều sự cải tiến của ngôn ngữ giúp người dùng thuận tiện hơn. Phạm vi cuốn sách sẽ không đề cập tới, bạn có thể tham khảo phụ lục A giới thiệu ngắn về tính năng mới C++.

Sự hỗ trợ tiến trình atomic trực tiếp trogn C++ cho phép lập trình viên tối ưu hóa code bằng cách định nghĩa semantics trên bất kỳ platfrom nào. Điều này thật tuyệt vời, chúng ta có thể dùng portable code: không chỉ trình biên dịch sẽ lo xử lí platform, mà sự tối ưu lập trình ở đây còn tính tới semantics (ngữ nghĩa) của tiến trình, do đó cho phép tối ưu chương trình tốt hơn.

1. Hiệu suất thư viện C++ thread

Một vấn đề lập trình viên quan tâm trong tốc độ xử lí tính toán thường phàn nàn về C++ nói chung, và thư viện C++ cung cấp nền tảng phát triển quá thấp, như trong thư viện thread chuẩn mới.

Nếu bạn quan tâm đến hiệu suất, thì phải hiểu rằng chi phí triển khai tương ứng khi dùng các nền tảng cấp cao, so sánh với cấp thấp hơn là khập khiễng.

Ủy ban C++ chuẩn đã nhận ra điều này khi thiết kế thư viện chuẩn, cũng như thư viện thread chuẩn – khi đó mục tiêu thiết kế là sẽ không tính nhiều về hiệu suất khi dùng trực tiếp API cấp thấp. Do đó thư viện C++ hiện nay được thiết kế cho phép triển khai hiệu quả trên hầu hết platforms.

Một mục tiêu khác của ủy ban là đảm bảo C++ cung cấp cơ sở nền tảng cấp thấp đủ khả năng cho những ai có nhu cầu về hiệu suất lập trình cao.

Rút cục, sự quản lý vùng nhớ mới và tất cả thư viện hỗ trợ tiến trình atomic cho phép điều khiển trực tiếp các bits và bytes riêng lẻ, và đồng bộ nội thread , và các khả năng thay đổi khác. Những kiểu dữ liệu atomic, và các tiến trình tương ứng có thể dùng trong nhiều trường hợp mà trước đây lập trình viên phải dựa vào ngôn ngữ hỗ trợ trên từng platform.

Code dùng kiểu dữ liệu chuẩn mới và các tiến trình trở nên bao quát và dễ bảo trì hơn.

Thư viện chuẩn C++ bây giờ cũng đã cung cấp nền tảng và lớp trừu tượng cấp cao hơn, nên viết thread cũng dễ dàng và ít lỗi. Thỉnh thoảng việc dùng các phương tiện này cũng tốn chi phí hiệu suất vì các mã lệnh bên ngoài thêm vào để thực thi. Tuy nhiên, chi phí này là cần thiết khi phát triển các nền tảng cao cấp hơn và không cao hơn bao nhiêu nếu bạn tự làm thủ công. Trình biên dịch cũng ngoại tuyến hàm đủ các code bên ngoài.

Trong vài trường hợp, các phương tiện này được chỉ định dùng cho mục đích cụ thể. Cũng hiếm khi nào có một hàm không dùng ảnh hưởng tới hiệu suất mã lệnh khác.

Nếu bạn đang ngắm tới hiệu suất, và chi phí bỏ ra quá lớn, thì bạn nên làm thủ công, thiết kế các hàm từ cấp thấp. Trong phần lớn trường hợp, sự phức tạp và nguy cơ lỗi mới là vấn đề hơn là chút hiệu suất đem lại.

Thậm chí có trường hợp nghẽn cổ chai khi thiết kế với thư viện chuẩn C++, đó là do sự thiết kế ứng dụng quá kém chứ không phải do thư viện tồi. Ví dụ, nếu có quá nhiều threads giành nhau một “mutex”, nó sẽ gây áp lực lên hiệu suất đáng kể. Thay vì cố càng nhỏ thời gian hoạt động mutex, thì nên tái cấu trúc ứng dụng, giảm bớt sự giằng co trên mutex. Đây là một trong vấn đề phát sinh đề cập ở chương 8.

Một vài trường hợp hiếm khi thư viện C++ chuẩn không cung cấp hiệu suất và vận hành tốt, có lẽ bạn nên dùng ngôn ngữ hỗ trợ platform đó.

1. Các phương tiện lập trình platform

Trong khi C++ cung cấp các phương tiện lập trình cho đa luồng và tương tranh, trên bất kì platform, thì các API platform vẫn đem lại nhiều lợi ích khác.

Để truy xuất dễ dàng những phương tiện này mà không phải từ bỏ lợi ích của thư viện chuẩn C++, dữ liệu trong thư viện threads C++ cung cấp hàm thành viên **native\_handle()** cho các hàm triển khai bên dưới xử lí trực tiếp dùng API của platform.

Như đặc tính tự nhiên, các tiến trình diễn ra dùng **native\_handle()** hoàn toàn phụ thuộc vào platform, và vấn đề này nằm ngoài lề cuốn sách (kể cả thư viện C++ chuẩn)

Dĩ nhiên, trước khi cân nhắc dùng API platform riêng, bạn phải nắm chắc các phương tiện trên thư viện chuẩn.

Bắt đầu với phần khởi động nào

##### Khởi động

Chương trình đa luồng cũng viết như bình thường (hàm, lớp, cấu trúc.. ). Điểm khác biệt là hàm sẽ chạy đồng thời với nhau, nên phải đảm bảo dữ liệu dùng chung được an toàn.

Để chạy đồng thời, hàm và đối tượng phải quản lý được các thread khác nhau

1. Tạo thread đầu tiên

Để khởi tạo thread bạn theo cú pháp hàm dựng thread:

**Std::thread <thread\_name>(<tên\_hàm>)**

Bắt đầu với chương trình in “Hello World”

#include <iostream>

int main(){

std::cout<<"Hello World\n";

}

Bây giờ so sánh với chương trình sau, in ra “Hello Concurrent World”, với một thread tách biệt để hiển thị.

#include <iostream>

#include <thread> #1

void hello() #2

{

std::cout<<"Hello Concurrent World\n";

}

int main(){

std::thread t(hello); #3

t.join(); #4

}

**Giải thích**

Đầu tiên bạn phải khai báo thư viện **#include <thread>** (#1) : định nghĩa kiểu dữ liệu thread và các hàm, lớp liên quan để quản lý thread. Các phần bảo vệ dữ liệu được định nghĩa trong thư viện khác.

Thứ hai là đoạn code chạy thread phải nằm trong hàm riêng biệt (#2).

Chương trình đã khởi động một thread cho hello() ở #3, số thread bây giờ là 2:

* thread bắt đầu tại main()
* thread mới tại hello().

Thread main() vẫn tiếp tục cho tới khi kết thúc chương trình, nhưng nó phải đợi cho thread hello() kết thúc để thực thi tiếp.

Tại #4, phương thứ .join() sẽ khiến thread main() phải đợi cho thread hello().

1. Tổng kết

Phần này chúng ta đã hiểu ý nghĩa của xử lí tương tranh và đa luồng, tại sao nên hay không nên dùng.

Đồng thời qua lịch sử phát triển C++0x, đã hỗ trợ rất nhiều cho lập trình viên tận dụng triệt để phần cứng để tối ưu chương trình.

Ví dụ “hello word” cho thấy viết thread khá đơn giản, khó khăn là nằm trong việc thiết kế chương trình.

### Quản lý threads

Hãy nghĩ tới các vấn đề khi bạn bắt đầu dùng thread:

* Làm sao để khởi động thread
* Làm sao kiểm tra thread đã chạy xong
* Làm sao quản lý nhiều thread

Thư viện chuẩn C++ sẽ làm tất cả việc trên dễ dàng. Đối với các tác vụ khó hiểu, thư viện cung cấp tính linh hoạt nhất để làm điều bạn muốn.

Chương này đề cập tới các vấn đề:

* Khởi động thread
* Đợi thread kết thúc hay chạy ngầm
* Truyền tham số phụ vào hàm chạy thread khi nó khởi động
* Chuyển quyền sở hữu tài nguyên từ thread này sang thread khác.
* Cuối cùng là chọn số thread để dùng, và định danh một thread.

1. Cơ bản

Mọi chương trình C++ có it nhất 1 thread khi bắt đầu C++ runtime: là thread khởi đầu chạy **main()**. Chương trình của bạn có thể khởi thêm thread cho một hàm.

Thư viện yêu cầu: <thread>

Cách một đối tượng std::thread thực thi là nó cần cung cấp một callback, để nó bắt đầu thread mới cho callback này. Callbacks có thể là:

1.) Con trỏ hàm

2.) Đối tượng hàm

3.) Hàm Lambda

Cú pháp: hàm dựng nhận tham số là con trỏ hàm/ đối tượng hàm

std::thread object\_thread(<callback>);

Bất kì thread nào đều có thể đợi thread khác thực thi xong rồi mới thoát bằng cách gọi join() cho thread đang thực thi

1. 3 cách khởi tạo thread

Tham số là con trỏ hàm:

#include <iostream>

#include <thread>

void thread\_function()

{

    for(int i = 0; i < 10000; i++);

        std::cout<<"thread function Executing"<<std::endl;

}

int main()

{

    std::thread threadObj(thread\_function);

    for(int i = 0; i < 10000; i++);

        std::cout<<"Display From MainThread"<<std::endl;

    threadObj.join();

    std::cout<<"Exit of Main function"<<std::endl;

    return 0;

}

Tham số là đối tượng hàm:

#include <iostream>

#include <thread>

class DisplayThread

{

public:

    void operator()()

    {

        for(int i = 0; i < 10000; i++)

            std::cout<<"Display Thread Executing"<<std::endl;

    }

};

int main()

{

    std::thread threadObj( (DisplayThread()) );

    for(int i = 0; i < 10000; i++)

        std::cout<<"Display From Main Thread "<<std::endl;

    std::cout<<"Waiting For Thread to complete"<<std::endl;

    threadObj.join();

    std::cout<<"Exiting from Main Thread"<<std::endl;

    return 0;

}

Nếu đối tượng hàm chứa bất kì con trỏ hay tham chiếu nào, phải đảm bảo chúng vẫn có giá trị để có thể truy xuất từ thread, nếu không sẽ gặp lỗi.

Tham số là hàm Lambda

#include <iostream>

#include <thread>

int main()

{

    int x = 9;

    std::thread threadObj([]{

            for(int i = 0; i < 10000; i++)

                std::cout<<"Display Thread Executing"<<std::endl;

            });

    for(int i = 0; i < 10000; i++)

        std::cout<<"Display From Main Thread"<<std::endl;

    threadObj.join();

    std::cout<<"Exiting from Main Thread"<<std::endl;

    return 0;

}

1. Kết thread join()

Tình huống: tạo thread trong một hàm có sử dụng biến cục bộ trong hàm.

struct func{

int& i;

//hàm dựng tham số

func(int& i\_):i(i\_){}

void operator()(){

for(unsigned j=0;j<1000;++j){

//do\_something(i); #1

}

}

};

void oops(){

int some\_local\_state = 0;

std::thread my\_thread(func(some\_local\_state));

} // #2

**Giải thích:**

Nếu hàm oops() kết thúc tại #2 mà đối tượng my\_thread chưa chạy xong: tham chiếu tới biến “i” không còn, chương trình gặp lỗi tại #1.

Chúng ta phải dùng **std::thread**. **join()** để đảm bảo rằng thread chạy oops() phải đợi **my\_thread** chạy xong trước khi kết thúc.

Ví dụ về 10 Thread làm việc, hàm main sẽ đợi cho tất cả kết thúc rồi mới thoát.

**Kết Thread dùng join()**

#include <iostream>

#include <thread>

#include <algorithm>

class WorkerThread

{

public:

    void operator()()

    {

        std::cout<<"Worker Thread "<<std::this\_thread::get\_id()<<" is Executing"<<std::endl;

    }

};

int main()

{

    std::vector<std::thread> threadList;

    for(int i = 0; i < 10; i++)

    {

        threadList.push\_back( std::thread( WorkerThread() ) );

    }

    // gọi join() cho từng đối tượng std::thread object dùng for\_each và std::mem\_fn

    std::cout<<"wait for all the worker thread to finish"<<std::endl;

    std::for\_each(threadList.begin(),threadList.end(), std::mem\_fn(&std::thread::join));

    std::cout<<"Exiting from Main Thread"<<std::endl;

    return 0;

}

**Giải thích:**

Join() đảm bảo thread được kết dù có xảy ra exception hay không, bằng cách block thread **main** đang chạy hiện tại lại cho tới lúc thread chạy **WorkerThread**() return.

1. Dùng RAII kết Thread

Thay vì phải khai báo hủy với join(), bạn có thể bao đóng thread thành đối tượng **thread\_guard**

Trong hàm hủy, trước khi join() thread cần kiểm tra xem đối tượng còn active hay không, dùng **thread.joinable()**

Cú pháp:

bool joinable() const; (since C++11)

Một thread dù đã hoàn thành code thực thi, nhưng chưa join(), vẫn xem là active thread, nên vẫn có thể joinable().

**DÙNG RAII ĐỂ ĐỢI THREAD HOÀN TẤT**

#include <thread>

class thread\_guard {

std::thread &t;

public:

explicit thread\_guard(std::thread& t\_) : t(t\_) {}

~thread\_guard() {

//kiểm tra thread đang active không

if (t.joinable())//#2

{

//Nếu có, join thread tới khi nó chạy xong

t.join();//#3

}

}

thread\_guard(thread\_guard const&) = delete //C++ 11: delete không support copy semantic

thread\_guard& operator=(thread\_guard const&) = delete; ;//#4 không support copy semantic

};

void func(int& a){//do something}

void f() {

int some\_local\_state;

std::thread t(func(some\_local\_state));

thread\_guard g(t);

//xử lí trường hợp ngoại lệ #5

}//hàm hủy thread\_guard được gọi

**Giải thích:**

Dù xảy ra ngoại lệ khay không, khi ra khỏi hàm, hàm hủy thread\_guard vẫn đảm bảo thread được kết.

**Lưu ý rằng thread dùng cơ chế RAII, tài nguyên được sở hữa bởi đối tượng thread thì không thể dùng phương pháp thông thường để gán/di chuyển cho đối tượng khác.**

**Nên khi triển khai lớp bạn nên cấm dùng hàm dựng sao chép và toán tử gán mặc định tại #4**

Nếu bạn kết thread cho đối tượng đã join, tức tài nguyên đã hủy, sẽ xảy ra lỗi.

1. Cơ chế thread ngầm

Hay còn gọi là thread ma, theo học thuyết UNIX, process ma chạy ngầm mà không thông qua tương tác người dùng. Những thread như vậy có đặc tính thời gian thực thi khá lâu: nó chạy trong hầu hết thời gian trình ứng dụng, cho phép các tác vụ như giám sát hệ thống file, xóa các entry không dùng của bộ lưu trữ đối tượng, hay tối ưu cấu trúc dữ liệu.

Nó dùng cho các cơ chế không cần quan tâm khi nào thread hoàn tất, như kiểu “bắn và quên”.

Cách triển khai giống như ví dụ trên, dùng đối tượng thread\_guard và join(), tốt cho các trường hợp hủy thread khi thoát ra phạm vi hay nó được cấp phát động.

1. Tách thread detach()

Threads sau khi tách được xem là thread ngầm. Để tách thread, chúng ta gọi std::detach() cho đối tượng std::thread

Đối tượng std::thread sau khi gọi detach(),không còn liên quan gì tới thread hiện tại nữa, sự thực thi giữa chúng độc lập với nhau. Thư viện runtime C++ sẽ tự động xóa tài nguyên liên quan khi thread kết thúc.

Khi đã tách thì bạn không cần **join** nữa. Cú pháp:

void detach(); (since C++11)

Ví dụ:

#include <thread>

#include <cassert>

void do\_something() {

}

void main() {

std::thread t(do\_something);

t.detach();

assert(!t.joinable());//assert:Neu bieu thuc ben trong khong thoa man (false)-> dung chuong trinh

}

Thỉnh thoảng việc gọi **detach**() tường minh giúp bảo trì code dễ hơn.

Tuy nhiên, để **detach**(), phải đảm bảo thread còn liên kết tới tài nguyên trên **std::thread**, cũng giống như điều kiện để **join**(). Bạn kiểm thread còn active (**joinable**() = true) trước khi gọi **detach**()

Các thread đã tách sẽ trở thành thread chạy ngầm, thư viện runtime C++ sẽ quản lý các thread này.

1. Tips sử dụng detach và join

Tình huống 1: Không bao giờ gọi **join**() hay **detach**() cho đối tượng std::**thread** không liên quan đến thread thực thi.

    std::thread threadObj( (WorkerThread()) );

    threadObj.join();

    threadObj.join(); // Khiến ứng dụng thoát

Khi join() được gọi trên đối tượng **threadObj**, thì sau khi join() return, đối tượng **threadObj** không còn liên hệ gì với thread đang thực thi **WorkerThread**() nữa.

Việc gọi lại join() trên **threadObj** sẽ khiến chương trình bị ngắt.

Tương tự như vậy, **detach**() khiến đối tượng **threadObj** không còn liên kết với bất kì thread thực thi nào nữa, việc gọi lại cũng khiến ứng dụng thoát ra.

    std::thread threadObj( (WorkerThread()) );

    threadObj.detach();

    threadObj.detach(); // Khiến ứng dụng thoát

Trước khi gọi **join**() hay **detach**() phải có bước kiểm tra xem đối tượng có thể **joinable**() hay không:

std::thread threadObj( (WorkerThread()) );

if(threadObj.joinable())

{

std::cout<<"Detaching Thread "<<std::endl;

threadObj.detach();

}

if(threadObj.joinable())

{

std::cout<<"Detaching Thread "<<std::endl;

threadObj.detach();

}

std::thread threadObj2( (WorkerThread()) );

if(threadObj2.joinable())

{

std::cout<<"Joining Thread "<<std::endl;

threadObj2.join();

}

if(threadObj2.joinable())

{

std::cout<<"Joining Thread "<<std::endl;

threadObj2.join();

}

Tình huống 2 : Không bao giờ được phép quên gọi join() hay detach()cho đối tượng thread đang có liên kết với thread thực thi.

Việc này cũng dẫn tới ứng dụng bị thoát. Vì bên trong hàm hủy đối tượng thread, nếu Thread vẫn có thể joinable() thì ngắt ứng dụng.

#include <iostream>

#include <thread>

#include <algorithm>

class WorkerThread

{

public:

    void operator()()

    {

        std::cout<<"Worker Thread "<<std::endl;

    }

};

int main()

{

    std::thread threadObj( (WorkerThread()) );

    // khi chúng ta quên không gọi join() hay detach() cho threadObj

    // hàm hủy của std::thread sẽ ngắt ứng dụng

    return 0;

}

Trường hợp xảy ra ngoại lệ cũng vậy, đối tượng thread cũng sẽ không được join() hay detach() đúng cách. Phương pháp tốt nhất là dùng đối tượng RAII bao đóng std::thread lại.

**Dùng RAII để detach và join thread**

#include <iostream>

#include <thread>

class thread\_guard

{

    std::thread & m\_thread;

    public:

        thread\_guard(std::thread  & threadObj) : m\_thread(threadObj){}

        ~thread\_guard()

        {

            // kiểm tra xem thread có thể join?

            if(m\_thread.joinable())

            {

                m\_thread.detach();

            }

        }

};

void thread\_function()

{

    for(int i = 0; i < 10000; i++);

        std::cout<<"thread\_function Executing"<<std::endl;

}

int main()

{

    std::thread threadObj(thread\_function);

    thread\_guard wrapperObj(threadObj); // Nếu comment line này, ứng dụng sẽ crash

    return 0;

}

1. Truyền tham số cho thread

Để truyền tham số tới đối tượng hàm hay hàm của thread chỉ cần truyền thêm các tham số này vào hàm dựng của std::thread

Cú pháp:

**Std::thread <tên thread>(callback, tham\_số\_1, tham\_số\_2,…)**

Nếu hàm có giá trị trả về thì sẽ được bỏ qua.

Mặc định tất cả tham số được sao chép vào vùng nhớ nội bộ của thread mới. Xem ví dụ sau:

#include <iostream>

#include <string>

#include <thread>

void threadCallback(int x, std::string str)

{

    std::cout<<"Passed Number = "<<x<<std::endl;

    std::cout<<"Passed String = "<<str<<std::endl;

}

int main()

{

    int x = 10;

    std::string str = "Sample String";

    std::thread threadObj(threadCallback, x, str);

    threadObj.join();

    return 0;

}

1. Tips nên tránh:

Tình huống 1: Không được truyền địa chỉ biến local stack vào hàm callback của thread. Vì có khả năng Thread 1 đã ra khỏi phạm vi nhưng Thread 2 vẫn sử dụng địa chỉ của biến để truy xuất.

Việc truy xuất địa chỉ không hợp lệ sẽ gây ra lỗi. Ví dụ:

#include <iostream>

#include <thread>

void newThreadCallback(int \* p)

{

    std::cout<<"Inside Thread :  "" : p = "<<p<<std::endl;

    std::chrono::milliseconds dura( 1000 );

    std::this\_thread::sleep\_for( dura );

    \*p = 19;

}

void startNewThread()

{

    int i = 10;

    std::cout<<"Inside Main Thread :  "" : i = "<<i<<std::endl;

    std::thread t(newThreadCallback,&i);

    t.detach();

    std::cout<<"Inside Main Thread :  "" : i = "<<i<<std::endl;

}

int main()

{

    startNewThread();

    std::chrono::milliseconds dura( 2000 );

    std::this\_thread::sleep\_for( dura );

    return 0;

}

Tình huống 2: Truyền con trỏ cấp phát động tới thread

Có khả năng một vài thread sẽ xóa vùng nhớ đó trước khi thread mới truy xuất tới. Việc truy xuất vùng nhớ không hợp lệ sẽ gây ra lỗi. Ví dụ:

#include <iostream>

#include <thread>

void newThreadCallback(int \* p)

{

    std::cout<<"Inside Thread :  "" : p = "<<p<<std::endl;

    std::chrono::milliseconds dura( 1000 );

    std::this\_thread::sleep\_for( dura );

    \*p = 19;

}

void startNewThread()

{

    int \* p = new int();

    \*p = 10;

    std::cout<<"Inside Main Thread :  "" : \*p = "<<\*p<<std::endl;

    std::thread t(newThreadCallback,p);

    t.detach();

    delete p;

    p = NULL;

}

int main()

{

    startNewThread();

    std::chrono::milliseconds dura( 2000 );

    std::this\_thread::sleep\_for( dura );

    return 0;

}

1. Truyền tham chiếu:

Lưu ý khi truyền tham chiếu vào hàm callback của thread, xét trường hợp:

#include <iostream>

#include <thread>

void threadCallback(int const & x)

{

    int & y = const\_cast<int &>(x);

    y++;

    std::cout<<"Inside Thread x = "<<x<<std::endl;

}

int main()

{

    int x = 9;

    std::cout<<"In Main Thread : Before Thread Start x = "<<x<<std::endl;

    std::thread threadObj(threadCallback, x);

    threadObj.join();

    std::cout<<"In Main Thread : After Thread Joins x = "<<x<<std::endl;

    return 0;

}

In Main Thread : Before Thread Start x = 9

Inside Thread x = 10

In Main Thread : After Thread Joins x = 9

Mặc dù threadCallback nhận tham số kiểu tham chiếu nhưng thay đổi giá trị trên “x” sẽ không ảnh hưởng giá trị bên ngoài thread.

Bởi vì biến “x” trong hàm threadCallback là tham chiếu tới biến tạm được sao chép tại stack thread mới tạo.

Vậy làm sao để fix? Dùng std::ref() :

#include <iostream>

#include <thread>

void threadCallback(int const & x)

{

    int & y = const\_cast<int &>(x);

    y++;

    std::cout<<"Inside Thread x = "<<x<<std::endl;

}

int main()

{

    int x = 9;

    std::cout<<"In Main Thread : Before Thread Start x = "<<x<<std::endl;

    std::thread threadObj(threadCallback,std::ref(x));

    threadObj.join();

    std::cout<<"In Main Thread : After Thread Joins x = "<<x<<std::endl;

    return 0;

}

In Main Thread : Before Thread Start x = 9

Inside Thread x = 10

In Main Thread : After Thread Joins x = 10

1. Truyền con trỏ hàm thành viên:

Truyền con trỏ hàm thành viên như hàm callback và con trỏ tới đối tượng ở tham số thứ hai, tiếp theo là các tham số đầu vào của hàm thành viên.

Ví dụ:

#include <iostream>

#include <thread>

class DummyClass {

public:

    DummyClass(){}

    DummyClass(const DummyClass & obj){}

    void sampleMemberFunction(int x)

    {

        std::cout<<"Inside sampleMemberFunction "<<x<<std::endl;

    }

};

int main() {

    DummyClass dummyObj;

    int x = 10;

    std::thread threadObj(&DummyClass::sampleMemberFunction,&dummyObj, x);

    threadObj.join();

    return 0;

}

1. Nhận dạng thread

Thông tin nhân dạng có kiểu cấu trúc **std::thread::id**, có 2 cách để lấy:

* Dùng **std::thread.get\_id() :** Nếu thread không còn active. Hàm trả về thông tin ám chỉ “không có thread”
* Dùng **std::this\_thread::get\_id()**: lấy thông tin id của thread hiện tại đang chạy.

Thông tin **std::thread::id** thường được dùng để kiểm tra thread còn thực thi hay không.

Ví dụ: Trước khi khởi động thread khác, bạn truyền tham số **thread.get\_id()** vào hàm check\_thread để kiểm tra

std::thread::id master\_thread;

void check\_thread(std::thread::id id){

if(id == master\_thread){

//do something...

}

//do something...

}

Sau khi lấy được **id** bạn có thể dùng nó để lưu trữ , so sánh và sắp xếp trong collection.

Ví dụ: **std::hash<std::thread::id>**

Có thể dùng output stream cho **id:**

std::cout<<std::this\_thread::get\_id();

Ngoài ra để trao đổi thông tin giữa các thread, bạn cũng phải biết **id** của chúng.

1. Chuyển quyền sở hữu tài nguyên
2. std::move()

Tương tự như con trỏ **unique\_ptr**, chúng ta dùng **std:move** để chuyển sở hữu giữa các thread.

Ví dụ: tạo 3 thread **t1**, **t2**, và **t3**.

void some\_function();

void some\_other\_function();

thread t1(some\_function); //#1 Khởi tạo thread t1

thread t2=std::move(t1); //#2 Khởi tạo thread t2 và chuyển sở hữa t1 sang t2

t1 = thread(some\_other\_function); //#3 t1 không sở hữu gì cả, nên có thể gán tài nguyên cho nó

thread t3(move(t2); //#4 tạo thread 3 và gán tài nguyên t2 -> t3 qua hàm dựng

t1=std::move(t3); //#6 chuyển tài nguyên t3->t1

**Giải thích:**

Tại **#6**, t1 lại sở hữu thread hàm some\_function, thread cũ some\_other\_function sẽ bị hủy.

1. Hàm trả về thread

Tham chiếu hàm tồn tại đến hết chương trình, nên chúng ta có thể truyền tham chiếu hàm ra ngoài phạm vi. Bạn không cần dùng **std::move** cho thread trả về.

**Hàm trả về đối tượng thread**  
std::thread f(){

void some\_function();

return std::thread(some\_function);

}

void some\_other\_function(int i){std::cout<<i;};

std::thread g(){

std::thread t(some\_other\_function,42);

return t;

}

1. Hàm nhận tham số thread

Chỉ có hai trường hợp truyền tham số:

* Truyền kiểu giá trị: không truyền đối tượng mà dùng hàm dựng thread.
* Truyền **std::move**: nhượng quyền sở hữu đối tượng truyền vào.

**Hàm nhận tham số thread**  
void f(std::thread t);

void g(){

void some\_function();

f(std::thread(some\_function)); //Truyền đối tượng tạm với hàm dựng khởi tạo

std::thread t(some\_function); //Khởi tạo thread t với some\_function

f(std::move(t)); //Chuyển sở hữu của t vào hàm f

}

1. Hàm dựng với std::move

Dùng tham chiếu thread trong class dễ bị lỗi: trường hợp thread đang chạy trong khi đối tượng thread bên ngoài tiến hành hủy hay trao quyền sở hữu

Khai báo std::move trong hàm dựng sẽ trao quyền quản lý thread cho đối tượng:

//scoped\_thread

class scoped\_thread{

private:

std::thread t;

public:

explicit scoped\_thread(std::thread t\_): t(std::move(t\_)){

if(!t.joinable()) //#5

throw std::string("No thread");

}

~scoped\_thread(){

if (t.joinable()) t.join();

}

scoped\_thread(scoped\_thread const&)=delete;

scoped\_thread& operator=(scoped\_thread const&)=delete;

};

void func(int&);

void f(){

int some\_local\_state;

scoped\_thread t(std::thread(func, some\_local\_state));//#1

//do\_something\_in\_current\_thread

}

1. Chạy nhiều thread với STL collection

Ví dụ dùng vector<std::thread> làm bộ chứa:

#include <thread>

#include <vector>

#include <algorithm>

void do\_work(unsigned id);

void f(){

std::vector<std::thread> threads;

for(unsigned i=0;i<20;++i){

threads.push\_back(std::thread(do\_work,i)); //#1

}

std::for\_each(threads.begin(),threads.end(), std::mem\_fn(&std::thread::join)); //#2

}

1. Chọn số thread khi runtime

Thư viện chuẩn C++ cung cấp phương thức **std::thread::hardware\_concurrency()** , cho biết số lượng thread có thể chạy song song trong chương trình. (số lượng nhân CPU)

Hàm trả về zero nếu không có thông tin hỗ trợ. Chúng ta dùng nó như một thông tin có ích để triển khai thread hợp lý.

Ví dụ sau triển khai threads để tính tổng giá trị các phần tử trong collection:

#include <iostream>

#include <thread>

#include <vector>

#include <algorithm>

#include <numeric>

using namespace std;

//dinh nghia functor tinh tong

template<typename Iterator,typename T>

struct accumulate\_block{

void operator()(Iterator first,Iterator last,T& result){

result=accumulate(first,last,result);

}

};

//Ham parallet trien khai thread

template<typename Iterator,typename T>

T parallel\_accumulate(Iterator first,Iterator last,T init){

unsigned long const length=distance(first,last);// Length: so phan tu trong collection

if(!length) return init; //#1 Neu range Iterator = 0, tra ve gia tri ban dau init

unsigned long const min\_per\_thread=25; //so phan tu toi thieu tinh toan per thread

unsigned long const max\_threads= (length + min\_per\_thread -1)/min\_per\_thread; //#2 dam bao so thread thap nhat la 1

unsigned long const hardware\_threads= thread::hardware\_concurrency();

unsigned long const num\_threads = min(hardware\_threads!=0?hardware\_threads:2,max\_threads);//#3

unsigned long const block\_size=length/num\_threads; //#4 so phan tu per thread

//vector results de luu ket qua tinh toan tu #[num\_threads] thread

vector<T> results(num\_threads);

//vector quan ly thread

vector<thread> threads(num\_threads-1); //#5 -1 cho thread main()

Iterator block\_start=first;

//Khoi tao thread

for(unsigned long i=0;i<(num\_threads-1);++i){

Iterator block\_end=block\_start + block\_size; //dinh vi lai iterator end

advance(block\_end,block\_size); //#6 nhay block\_end len vi tri cuoi cung cua block

threads[i]=thread( accumulate\_block<Iterator,T>(), block\_start,block\_end,ref(results[i])); //#7

block\_start=block\_end; //#8 dinh vi lai iterator begin

}

accumulate\_block()(block\_start,last,results[num\_threads-1]); //#9

for\_each(threads.begin(),threads.end(), mem\_fn(&thread::join)); //#10

return accumulate(results.begin(),results.end(),init); //#11

}

### Dữ liệu dùng chung

##### Vấn đề:

Nếu dữ liệu dùng chung chỉ đọc, thì không có chuyện gì xảy ra cả. Tuy nhiên, nếu một thread đang cố gắng xử lí dữ liệu, sẽ xảy ra vấn đề.

Chúng ta phải luôn nắm chắc được số phần tử trong danh sách dữ liệu, vốn sẽ bị thay đổi khi update, delete. Và đừng tin vào những điều bất biến.

Ví dụ DSLK kép, ví dụ khi xóa 1 node, chúng ta phải điều chỉnh con trỏ cho node kế tiếp và node phía sau, điều này bị phá vỡ nếu một trong 2 node liền kề đang bị update cùng lúc.

Hay khi 1 thread duyệt DSLK, trong khi 1 thread đang cố gắng xóa node đầu tiên, chương trình sẽ bị crash.

Nguyên nhân xung đột thì rất đa dạng. Chúng ta gọi chúng là race condition.

1. Race condition

Có tình huống thực tế như sau:

Bạn đi mua vé xem phim và chọn chỗ ngồi trong rạp. Quầy bán vé có nhiều thu ngân, cũng có nhiều người xem cùng phim với bạn.

Đặt trường hợp chỉ còn vài đúng 1 chỗ còn trống. Tình huống này đang xảy ra tranh chấp. Vậy ai đặt trước người đó sẽ có vé.

Hầu hết trường hợp tranh chấp, hệ quả đều chấp nhận được. Ví dụ khi 2 thread đang chèn phần tử vào hàng đợi, không thành vấn đề khi cái nào làm trước cũng được.

Race condition khá khó chịu, phá vỡ tính ổn định hệ thống, dẫn tới các hành vi không lường. C++ định nghĩa thuật ngữ data race như một kiểu dữ liệu của race condition

Các race condition thường xảy ra khi có tiến trình xử lí trên 2 hay nhiều dữ liệu, khó nhận biết và xác suất xảy ra là khá thấp. Dẫu vậy, hệ thống càng lớn, mức độ truy xuất dày đặc hơn thì tỉ lệ xung đột cũng tăng.

Khi bạn debug, thường sẽ không bao giờ thấy chúng xuất hiện vì thời gian mỗi bước nhảy trong debug đủ để thread thực thi xong.

1. Cơ chế bảo vệ

Chúng ta đề cập hai cách xử lí race condition sau đây:

* Wrap cấu trúc dữ liệu với cơ chế bảo vệ
* Xử lí thao tác trên dữ liệu như một giao dịch: như kiểu database, các dữ liệu cần truy xuất sẽ được chứa trong bản ghi giao dịch, được cấp phép thực thi trong từng bước một. Nếu thực thi không thành công do dữ liệu bị tranh chấp, giao dịch sẽ được thực thi lại. Cơ chế này gọi là Software Transactional Memory

Cơ chế bảo vệ cơ bản nhất của C++ chuẩn là **mutex**.

##### Bảo vệ dữ liệu dùng chung

Ý tưởng khá hay nếu đoạn code truy xuất dữ liệu đánh dấu là **mutually exclusive,** trong một thời điểm chỉ có 1 thread thao tác trên dữ liệu mà thôi.

Sự đồng bộ này gọi là mutaully exclusive. Trước khi truy xuất dữ liệu dùng chung, bạn bạn khóa mutex tương ứng lại, sau khi xong, bạn mở nó ra.

Thư viện thread đảm bảo rằng khi 1 thread khóa mutex, tất cả các thread nào đang cố khóa cùng mutex sẽ phải đợi khi mutex mở trở lại.

1. Kiểu BasicLockable

BasicLockable là học thuyết đơn giản về các kiểu của C++ chuẩn hoạt động trên khái niệm đồng bộ mutual exclusive cho thread.

Yêu cầu về BasicLockable

Kiểu L được xem là BasicLockable, nếu điều kiện sau thỏa mãn cho đối tượng m của kiểu L:

|  |  |  |
| --- | --- | --- |
| **Expression** | **Requires** | **Effects** |
| m.lock() |  | Khóa thread tới khi lấy được khóa m cho thực thi hiện tại. Nếu xảy ra ngoại lệ thì không lấy khóa. |
| m.unlock() | Thực thi hiện tại đang giữ khóa m | Giải phóng khóa khỏi thread hiện tại mà không ném ra ngoại lệ nào cả. |

1. Sử dụng mutex

Các bước

* Tạo đối tượng **std::mutex**
* **Std::mutex.lock()** để khóa mutex
* **Std::mutex.unlock()** để mở khóa

Tuy nhiên chúng ta không dùng trực tiếp phương thức mà dùng RAII triển khai thông qua lớp template **std::lock\_guard.**

Lớp này sẽ là wrapper của mutex, nó tự động khóa mutex trong hàm dựng, và mở khóa trong hàm hủy.

Cú pháp:

**Std::lock\_guard<std::mutex>** <tên\_wrapper>(tên\_mutex)

Ví dụ:

Bảo vệ list với mutex

#include <list>

#include <mutex>

#include <algorithm>

std::list<int> some\_list; //#1

std::mutex some\_mutex; //#2

void add\_to\_list(int new\_value){

std::lock\_guard<std::mutex> guard(some\_mutex); //#3

some\_list.push\_back(new\_value);

}

bool list\_contains(int value\_to\_find){

std::lock\_guard<std::mutex> guard(some\_mutex); //#4

return std::find(some\_list.begin(),some\_list.end(),value\_to\_find)!= some\_list.end();

}

Hai phương thức đều truy xuất đến collection list, được khai báo cùng một mutex.

Hiếm khi nào biến toàn cục được sử dụng như thế này. Thông thường mutex sẽ được bao đóng cùng với dữ liệu trong một class. Và các phương thức truy xuất liên quan tới dữ liệu cũng vậy, đây là quy tắc chung khi thiết kế class.

Tất cả hàm thành viên truy xuất đều sử dụng chung một mutex của lớp. Tuy nhiên điều này chưa đảm bảo an toàn cho dữ liệu.

Trường hợp hàm thành viên trả về tham chiếu tới dữ liệu, là lỗ hổng lớn. Do đó lớp phải được thiết kết thật tốt, không có cửa sau.

1. Cơ chế bảo vệ dữ liệu

Chúng ta nên tuân thủ các quy tắc sau:

* Các hàm truy xuất dữ liệu phải khai báo mutex với **std::lock\_guard**
* Cấm các phương thức trả về tham chiếu hay truyền tham chiếu, con trỏ của dữ liệu ra ngoài phạm vi lớp: vì chúng ta không lường trước được dữ liệu có thể bị can thiệp từ bên ngoài thế nào, dù đã đặt khóa bảo vệ bên trong lớp.
* Kiểm tra các hàm gọi bên trong hàm thành viên nếu có bất kì trường hợp như trên.

Ví dụ: về truyền functor vào tham số hàm thành viên

Lỗ hổng truyền tham chiếu về

class data{

public:

void do\_something();

};

class data\_wrapper{

private:

data data;

std::mutex m;

public:

template<typename Function>

void process\_data(Function func){

std::lock\_guard<std::mutex> l(m);

func(data); //#1

}

};

data\* unprotected;

void malicious\_function(data& protected\_data){

}

data\_wrapper x;

void foo(){

x.process\_data(malicious\_function); //#2

unprotected->do\_something(); //#3

}

Đoạn code trên có một lỗ hổng ở phương thức process\_data(), do không lường trước được tham số truyền vào functor là kiểu tham chiếu tới dữ liệu bảo vệ.

Trường hợp này xảy ra khá phổ biến, cho dù bảo vệ hoàn toàn bởi mutex nhưng race condition vẫn có khản năng xảy ra, chúng ta tìm hiểu trong phần tiếp theo.

1. Thread-safe cho stack
2. Race condition

Mutex hay cơ chế bảo vệ dữ liệu nào khác cũng thay bạn xử lí race condition – việc của bạn là triển khai mutex và thiết kế interface thật tốt.

Triển khai interface của std::stack

interface cho đối tượng std::stack container

template<typename T,typename Container=std::deque<T>>

class stack

{

public:

explicit stack(const Container &);

explicit stack(Container && = Container ());

template <class Alloc> explicit stack(const Alloc&);

template <class Alloc> stack(const Container &, const Alloc&);

template <class Alloc> stack(Container &&, const Alloc&);

template <class Alloc> stack(stack&&, const Alloc&);

bool empty() const;

size\_t size() const;

T& top();

T const& top() const;

void push(T const&);

void push(T&&);

void pop();

void swap(stack&&);

};

Nếu tuân theo quy tắc, phải đổi kiểu trả về của top() từ tham chiếu thành giá trị bản sao.

Xem đoạn code xử lí stack như sau:

stack<int> s;

if(!s.empty()){ //#1 kiểm tra stack rỗng?

int const value=s.top();//#2 lấy giá trị tham chiếu trả về

s.pop(); //#3 xóa top

do\_something(value);

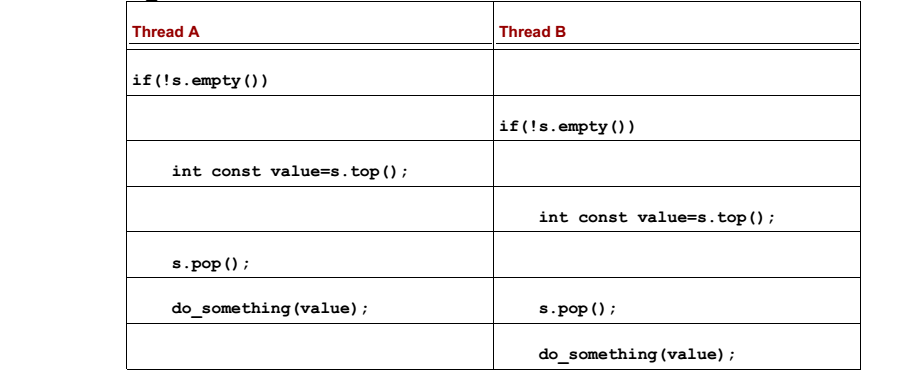
}

Đoạn code trên gặp phải 2 race condition như sau:

* Trường hợp stack còn duy nhất 1 phần tử, khi empty() vừa gọi nếu có 1 thread khác đang truy xuất stack để pop() thì hành vi tại **#3** có thể dẫn tới crash chương trình.

Nếu tung ra một ngoại lệ cho top(), chúng ta xác định được nguyên nhân, như vậy thì empty() lại tỏ ra thừa thãi, code quá rườm rà.

* Giữa lần gọi #2 và #3 ( top() và pop() ). Hình dung 2 thread đang thao tác cùng đoạn code trên, tình huống này hoàn toàn có thể xảy ra, khi bạn dùng thread để tăng hiệu suất, việc dùng lại code giữa các thread là bình thường.



Cả hai đều tham chiếu đến cùng đối tượng s.pop(). Sự kiện do\_something() rất có thể lặp lại 2 lần đưa ra kết quả không chính xác.

Điều phiền toái là tiến trình thực hiện khá suôn sẻ, không hề có lỗi nào.

Tóm lại, thiết kế đã vấp phải các vấn đề sau:

* Truyền tham chiếu ra bên ngoài
* Không có cơ chế mutex bảo vệ dữ liệu dùng chung

1. Giải pháp

Đề xuất thay đổi thiết kế interface triệt để:

* Đặt mutex cho top(), pop().

Trong lịch sử C++ STL, **std::stack** interface, Tom Cargill đã chỉ ra rằng các hàm gọi kết hợp có cùng một vấn đề, nếu hàm dựng sao chép cho các đối tượng trên stack có thể xảy ra ngoại lệ.

Vấn đề này được giải quyết hoàn toàn trên quan điểm về an toàn ngoại lệ của Herb Sutter [1999], nhưng nguy cơ từ race condition lại bùng lên.

Ví dụ: **stack**<**vector**<int>>. Hàm dựng sao chép mặc định nhận tham số vector<int>, trường hợp bộ nhớ không đủ, nó sẽ ném ra ngoại lệ **std::bad\_alloc** .

pop() vừa trả về giá trị, đồng thời xóa trong stack, nếu thao tác sao chép từ nơi gọi hàm chưa xong thì dữ liệu đã xóa. Vì Vậy top() được tách riêng khỏi pop().

Thật không may là việc tách 2 quá trình đưa tới race condition, nhưng có cách khắc phục đơn giản.

1. Truyền tham chiếu cho giá trị pop

Triển khai pop() nhận tham chiếu để gán giá trị trả về

**std**::**vector**<int> result;

some\_stack.**pop**(result);

Điểm bất lợi là bạn phải tạo bản sao cho phần tử cần pop, để truyền vào tham số. Việc tạo bản sao khá tốn chi phí. Mặt khác, nó không phải lúc nào cũng hiệu quả, hàm dựng yêu cầu phải có tham số truyền vào. Cuối cùng là không phải kiểu đối tượng người dùng nào cũng được overload toán tử gán, dù nó có move semantic đi nữa.

1. Không dùng hàm dựng sao chép hay hàm dựng move semantic no-throw

Chỉ còn duy nhất vấn đề an toàn ngoại lệ cho hàm pop() (có trả về giá trị ) nếu giá trị trả về xảy ra ngoại lệ. Có rất nhiều kiểu hàm dựng không ném exception, và kể cả hàm dựng move semantic cho kiểu tham chiếu rvalue dù hàm dựng sao chép của chúng có hỗ trợ.

Chúng ta sẽ cấm dùng các kiểu đối tượng có thể trả về giá trị mà không có xử lí ngoại lệ cho thread-safe stack.

1. Trả về con trỏ tới đối tượng pop

Điểm thuận lợi là có thể sao chép mà không cần ném ngoại lệ, tránh được vấn đề ngoại lệ Cargill.

Cho hầu hết trường hợp trả về con trỏ có vùng nhớ cấp phát, chúng ta dùng **std::shared\_ptr**, là một lựa chọn tốt, dù tốn ít chi phí.

1. Kết hợp cách 1 với 2 hoặc 3

Tùy theo bạn.

1. Triển khai thread safe cho stack

Chúng ta sẽ xử lí race conditions cho interface, áp dụng cách thứ 1 và 3: có 2 overloads cho pop(), một nhận tham chiếu cho biến nhận giá trị, và một trả về **std::shared\_ptr<>**.

Triển khai **Wrapper Interface** cho stack khá đơn giản với push() và pop():

Thread-safe stack

#include <exception>

struct empty\_stack: std::exception{

const char\* what() const throw();

};

template<typename T>

class thread\_safe\_stack{

public:

stack();

stack(const stack&);

stack& operator=(const stack&) = delete; //#1

void push(T new\_value);

std::shared\_ptr<T> pop();

void pop(T& value);

bool empty() const;

};

Để an toàn tối đa, chúng ta cấm dùng phép gán tại #1 (kể cả gán chính nó và phép gán).

class definition for a thread-safe stack

#include <exception>

#include <numeric>

using namespace std;

struct empty\_stack: exception{

const char\* what() const throw();

};

template<typename T>

class thread\_safe\_stack{

private:

stack<T> data;

mutable mutex m;

public:

stack(){}

stack(const stack& other) {//#1 không dùng danh sách khởi tạo cho constructor

lock\_guard<mutex> lock(other.m); //vì phải khóa mutex cho đối tượng truyền vào

data=other.data;

}

stack& operator=(const stack&) = delete;

void push(T new\_value) {

lock\_guard<mutex> lock(m);

data.push(new\_value);

}

shared\_ptr<T> pop(){

lock\_guard<mutex> lock(m);

if(data.empty()) throw empty\_stack(); //#2 kiểm tra empty trước khi pop

shared\_ptr<T> const res(new T(data.top()); //#3 Cấp phát memroy cho con trỏ trả về

data.pop(); //trước khi xóa khỏi stack

return res;

}

void pop(T& value) {

lock\_guard<mutex> lock(m);

if(data.empty()) throw empty\_stack();

value=data.top();

data.pop();

}

bool empty() const {

lock\_guard<mutex> lock(m);

return data.empty();

}

};

1. DeadLock
2. Vấn đề

Không nên triển khai mutex như biến toàn cục để bảo vệ phạm vi dữ liệu quá lớn, trong 1 thời điểm chỉ có 1 thread chạy , như vậy nó sẽ triệt tiêu mọi lợi ích của đa luồng,.

**Trường hợp về nhân linux (Linux kernel):**

Phiên bản đầu tiên để xử lí hệ thống đa nhân của Linux chỉ dùng một khóa toàn cục đơn lẻ. Nghĩa là hệ thống 2 nhân lại hoạt động kém hơn so với 2 hệ thống một nhân!. Thread trên nhân còn lại không thể hoạt động tốt.

Xảy ra tranh cãi về thiết kế nhân Linux, thiết kế sau đó đã có cơ chế khóa tốt hơn, hệ thống 4 nhân giờ hoạt động tốt gần bằng 4 hệ thống đơn nhân.

Tuy nhiên, thỉnh thoảng cần hơn 1 khóa mutex để bảo vệ tất cả dữ liệu trong tiến trình!

Như trường hợp mutex đang bảo vệ dữ liệu chung cho lớp. Nếu muốn có mutex ở cấp cao hơn: nghĩa là để người dùng tự lo, hoặc triển khai 1 mutex lo toàn bộ đối tượng của lớp. Cả 2 tùy chọn đều thích hợp.

Nếu bạn chọn triển khai 2 hay nhiều mutex cho tiến trình, có nguy cơ dẫn tới **deadlock**.

Trái ngược với race condition, deadlock là trường hợp các thread đợi nhau, nhưng chẳng cái nào làm gì cả.

Ví dụ: Món đồ chơi của chúng ta gồm cái trống và bộ gõ. Bạn có 2 đứa trẻ muốn chơi, nếu một đứa lấy cả 2 món, nó chơi cho tới khi thỏa thích, trong khi đứa kia ngồi buồn thiu.

Bây giờ chúng ta giấu mỗi món vào trong 2 chiếc hộp pandora, cả 2 đứa cùng nhào tới. Mỗi đứa 1 món, nhưng lại kẹt – trừ khi một đứa tốt bụng nhường đứa kia chơi trước, cả 2 đứa đều ôm đồ và đòi đứa còn lại, cả 2 đều chẳng chơi.

Liên tưởng đứa trẻ vói món đồ chơi như thread với mutex, mỗi cặp thread cần khóa cả cặp mutex để thực thi gì đó, và mỗi thread nắm 1 mutex, đợi thread kia nhả ra. Cả 2 chẳng làm gì ngoài việc ôm mutex của nó.

1. std::lock()

Phương pháp chung để chóng deadlock là chúng ta quy định khóa 2 hay nhiều mutex phải theo cùng một trật tự.

Ví dụ bạn luôn đặt mutex A rồi tới B, sau đó tới C, D …

Đó là khi các mutex dùng cho các mục đích khác nhau. Xét trường hợp các mutex đang bảo vệ các đối tượng khác nhau của lớp.

Ví dụ: thao tác so sánh 2 đối tượng cùng lớp – ta đặt mutex để dữ liệu không bị tác động khi so sánh, cả 2 mutex đều phải bị khóa.

friend bool operator<(X const& lhs, X const& rhs) {

std::lock\_guard<std::mutex> lock\_a(lhs.m,std::adopt\_lock); //#1 Mutex khóa theo trật tự std::lock\_guard<std::mutex> lock\_b(rhs.m,std::adopt\_lock); //#2 #1 và #2

return lhs.some\_detail<rhs.some\_detail;

}

bool operator<(some\_big\_object& lhs,some\_big\_object& rhs);

Do không có bước kiểm tra 2 tham chiếu, Deadlock xảy ra khi 2 tham số đều cùng trỏ tới một đối tượng, sau khi mutex #1 khóa, mutex tại #2 không thể mở được, chương trình sẽ bị treo tại đây.

Chúng ta sẽ phải thêm điều kiện so sánh 2 tham chiếu để tránh deadlock.

Ngoài ra C++ cung cấp phương thức **std::lock** để khóa nhiều mutex cùng lúc, cách dùng:

* Truyền các đối tượng mutex vào tham số của **std::lock()**
* **G**ọi **std::lock\_guard** cho từng mutex với tham số thứ hai là **std::adopt\_lock.**

**Std::adopt\_lock** sẽ báo cho **std::lock\_guard** biết mutex đã được khóa rồi, chúng sẽ sở hữu mutex mà không khóa nữa.

Chúng ta vẫn phải dùng **std::lock\_guard** để đảm bảo mutex được mở khóa đúng cách khi hàm trả về.

Note: có 1 dạng mutex cho phép cùng một thread có thể khóa nhiều lần, đó là **std::recursive\_mutex**.

**Listing 3.6 Khóa 2 mutex với std::lock() trong toán tử so sánh**

class some\_big\_object;

bool operator<(some\_big\_object& lhs,some\_big\_object& rhs);

class X{

private:

some\_big\_object some\_detail;

mutable std::mutex m;

public:

X(some\_big\_object const& sd):some\_detail(sd){}

friend bool operator<(X const& lhs, X const& rhs) {

if(&lhs==&rhs) //kiểm tra tham chiếu tới chính nó

return true;

std::lock(lhs.m,rhs.m); //#1 có thể xảy ra exception

std::lock\_guard<std::mutex> lock\_a(lhs.m,std::adopt\_lock); //#2

std::lock\_guard<std::mutex> lock\_b(rhs.m,std::adopt\_lock); //#3

return lhs.some\_detail<rhs.some\_detail;

}

};

Nếu xảy ra ngoại lệ trong hàm **std::lock** bởi 1 hay nhiều mutex, exception sẽ được truyền ra ngoài hàm, thì tất cả mutex còn lại sẽ được mở.

**std::lock** hoạt động theo cơ chế “khóa tất cả - hoặc không cái nào khóa cả”.

Đối với các trường hợp riêng lẻ, chúng ta cần phân tích và dùng kinh nghiệm để phát hiện. Giải quyết DeadLock là vấn đề khó khăn, vì xác suất xảy ra cũng thấp như race condition vậy.

1. std::unique\_lock

**std::unique\_lock** linh hoạt hơn một chút so với **std::lock\_guard**, **std::unique\_lock** lưu trữ khóa nhưng không cố định sở hữu mutex.

* Truyền **std::adopt\_lock** vào tham số thứ 2 trong hàm dựng để sở hữu và quản lý khóa mutex (trạng thái lock).
* Truyền **std::defer\_lock** để giữ mutex vẫn ở trạng thái unlock trong hàm dựng.

Mutex có thể bị khóa bằng phương thức thành viên **std::unique\_lock**::**lock()** hoặc truyền chính đối tượng **std::unique\_lock** vào tham số của **std::lock**()

Listing 3.6 sẽ được viết lại dùng **std::unique\_lock** với **std::defer\_lock** (#1) so với cách dùng **std::lock\_guard** với **std::adopt\_lock**.

Do đặc thù cơ chế phức tạp hơn nên **std::unique\_lock** có kích thước lớn hơn và chậm hơn so với **std::lock\_guard**. Đối tượng **std::unique\_lock** không sở hữu mutex nhưng cái giá là: thông tin phải được lưu trữ và phải được cập nhật.

Listing 3.9:

**Listing 3.9 Khóa 2 mutexes với lock() và std::unique\_lock trong toán tử so sánh**

class some\_big\_object;

bool operator<(some\_big\_object& lhs,some\_big\_object& rhs);

class X

{

private:

some\_big\_object some\_detail;

mutable std::mutex m;

public:

X(some\_big\_object const& sd):some\_detail(sd){}

friend bool operator<(X const& lhs, X const& rhs)

{

if(&lhs==&rhs)

return true;

std::unique\_lock<std::mutex> lock\_a(lhs.m,std::defer\_lock); //#1

std::unique\_lock<std::mutex> lock\_b(rhs.m,std::defer\_lock); //#1

std::lock(lock\_a,lock\_b); //#2

return (lhs.some\_detail < rhs.some\_detail);

}

};

**Giải thích**

Trong listing 3.9, đối tượng **std::unique\_lock** được truyền vào tham số của **std::lock()** (#2)

**std::unique\_lock** cung cấp các phương thức thành viên **lock(), try\_lock()** và **unlock().** Ngoài ra còn có một flag bên trong để nhận biết mutex có sở hữu bởi đối tượng hay không, hàm hủy **std::unique\_lock** dựa vào flag này để gọi **unlock**() cho mutex hay không.

Chúng ta dùng thành viên **owns\_lock()** để kiểm tra flag. Lý do **std::unique\_lock** chậm hơn **std::lock\_guard** vì phải cập nhật, kiểm tra flag nội bộ.

Nếu **std::lock\_guard** đáp ứng đủ nhu cầu, thì không cần phải dùng **std::unique\_lock**. Tuy nhiên, trường hợp khi cần giữ mutex nhưng không sở hữu nó, hay trường hợp quyền sở hữu khóa cần chuyển đi thì nên dùng **std::unique\_lock**

1. Giải pháp khóa phân cấp
2. Thiết kế phân tầng

Ý tưởng là thiết kế các tầng có mã số riêng cho tất cả mutex, cơ chế hoạt động tuân theo 2 luật:

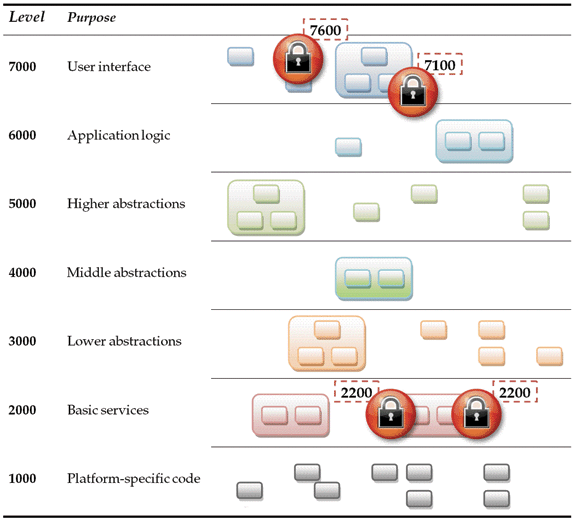
**Luật 1**: Khi đang nắm mutex ở tầng N, thread chỉ được phép khóa thêm mutex cho tầng dưới N.

**Luật 2**: Nếu muốn khóa nhiều mutex, các mutex cần phải khóa đồng loạt với **std::lock**. Đảm bảo các quá trình khóa phải tuân theo trật tự nhất định. Ví dụ: phương pháp thông thường để khóa các mutex cùng lúc là theo thứ tự tăng địa chỉ.

Tuân đúng 2 luật trên, sẽ không bao giờ deadlock vì không bao giờ tồn tại 2 thread đang cố khóa mutex A và B theo 2 chiều khác nhau:

* A và B nằm khác tầng: cái nào tầng cao hơn sẽ bị khóa trước.
* A và B cùng tầng: cả 2 sẽ bị khóa cùng lúc

Hình bên dưới minh họa cơ chế phần tầng, kỹ thuật kiểm tra định thời để kiểm soát tính lệ thuộc trong ứng dụng. Code phải chia thành module – hay tầng, các đoạn code chỉ được phép truy xuất module cùng hoặc dưới tầng.



Mục đích phân tầng và mutex sau hết vẫn là bảo vệ và kiểm soát truy xuất tới dữ liệu – trong từng module code. Ý nghĩa của luật 1 là Mutex tầng dưới thường duy trì ngắn hơn so với tầng cao hơn.

Trong thiết kế phân tầng, cần lưu ý về backdoor: như callback funtion, chúng có thể nhảy lên tầng cao hơn, làm chương trình bị crash dù là trong 1 thread.

Nếu 1 module code ở tầng giữa làm thay đổi trạng thái ổn định hệ thống, nó gọi xuống tầng dưới, vấn đề xảy ra nếu có callback trở lại tầng trên và gặp phải trạng thái không ổn đó.

Kiến trúc phần tầng giúp giải quyết vấn đề của đơn thread, đồng thời cho cả nhiều thread.

1. Frameworks và khóa phân tầng

Rất ít frameworks sữ dụng mutex lại không có sự hỗ trợ nào về khóa phân tầng. Các nhà phát triển sẽ triển khai trong nay mai.

Còn chúng ta nên triển khai wrapper mutex phân tầng cho dự án của mình. (ví dụ như đóng gói thao tác mutex thành phương thức hay phân lớp):

* Triển khai wrapper class, có hàm dựng nhận tham số là mã số tầng “**myLevel**”, lưu nó lại như một thành viên và dùng wraper này mọi nơi.
* Cung cấp wrapper một biến static thread nội bộ gọi là “**currentLevel**”, khởi tạo giá trị cao hơn bất kì tầng khóa nào
* Trong phương thức khóa của wrapper, kiểm tra xem **currentLevel** có cao hơn **myLevel** không? Nhớ rằng nếu giá trị trước của **currentLevel** đang dùng là của 1 biến thành viên khác, thì thiết lập **currentLevel = myLevel,** và yêu cầu khóa.
* Trong phương thức mở khóa của wrapper, trả về giá trị trước cho **currentLevel.**
* Khi cần thiết, chúng ta cũng có thể wrap phương thức cần dùng như **std::try\_lock** chẳng hạn. Bất kì phương thức cố gắng để khóa cũng nên làm tương tự như **std::lock**.
* Cuối cùng viết phương thức khóa **std::lock**(m1,m2…) nhận đủ các kiểu tham số của đối tượng khóa, kiểm tra chúng có cùng tầng hay không, và khóa chúng theo thứ tự địa chỉ (hay theo ID, thứ tự globall…) nhưng phải nhất quán.

Chúng ta nên dùng **std:assert** trong phương thức khóa, để ép mọi lỗi vi phạm luật phân cấp xuất ra ngay lần chạy. Lý do code path không hoàn hảo có thể vì chưa bao quát hết module hay sơ sót vài trường hợp. Nên tạo file log hay dump để xem lại.

1. Lưu ý khi thiết kế

Khóa phân tầng cũng có những điểm yếu của nó: như khóa chính chúng. Đòi hỏi sự rèn luyện và nỗ lực của bạn trong việc kiểm soát module code.

Với plug-in code, bạn nên cẩn thận khi sử dụng trong mô hình phân cấp khóa.

Ví dụ: trong hình minh họa phân tầng, nếu ứng dụng cho phép plugin được gọi từ tầng 5000s. Đầu tiên chương trình và tất cả plug-ins phải tránh gọi các module code không rõ danh tính trong khi đang cầm khóa.

Tuy nhiên, thực tế đã có trường hợp dù plug-in không cầm khóa nội bộ của nó nhưng callback về main() và tạo code path tới tầng cao hơn và nhận khóa.

Trường hợp này, plug-in phải nhận biết được sự phân tầng trong ứng dụng – nói với chúng hoạt động ở tầng 4999 và chỉ được quyền gọi tầng dưới 4999.

1. Tiển khai khóa phân tầng

Ví dụ 2 thread đang dùng cơ chết mutex phân cấp:

**Dùng khóa phân cấp tránh Deadlock**

class hierarchi\_lock;

hierarchi\_lock HiLevel(10000); //#4

hierarchi\_lock LoLevel(5000); //#7

int do\_loLevel\_func();

int LoLevel\_func(){

std::lock\_guard<hierarchi\_lock> lk(LoLevel); //#6

return do\_loLevel\_func();

}

void do\_hiLevel\_func(int some\_param);

void HiLevel\_func(){

std::lock\_guard<hierarchi\_lock> lk(HiLevel); //#3

do\_hiLevel\_func(LoLevel\_func()); //#5

}

void thread\_a(){ //#1

HiLevel\_func();

}

hierarchi\_lock other\_mutex(100); //#9

void do\_other\_stuff();

void other\_stuff(){

HiLevel\_func(); //#10

do\_other\_stuff();

}

void thread\_b(){

std::lock\_guard<hierarchi\_lock> lk(other\_mutex); //#8

other\_stuff();

}

thread\_a() (#1) vẫn tuân theo luật, nên chạy tốt. Ngược lại, thread\_b() (#2) không tuân theo luật nên thất bại trong runtime.

1. Tips tránh deadlock

Deadlock không chỉ xuất hiện khi bạn khóa mutex: bạn có thể tạo 2 thread, không có mutex, rồi thread này gọi join cho thread kia. Cả 2 không thực thi vì nó đợi thread kia hoàn thành, giống như những đứa trẻ tranh nhau đồ chơi.

Vòng lặp kì lạ này có thể xảy ra bất kì chỗ nào, khi mà 1 thread đợi thread khác làm gì đó, nếu nhiều thread: ta có deadlock cho 1 chuỗi thread.

Kim chỉ nam chủ yếu xoay quanh ý tưởng: không đợi thread khác nếu có chút khả năng nó đang đợi bạn.

1. Tránh khóa lồng mutex

Ý tưởng đầu tiên khá đơn giản là không khóa thêm mutex nếu bạn đã cầm 1 cái rồi. Không thể nào deadlock xảy ra khi mà mỗi thread chỉ được phép cầm 1 khóa.

Deadlock chỉ xảy ra ở trường khợp khác như thread đợi lẫn nhau. Nếu bạn cần khóa nhiều mutex cùng lúc, nhớ dùng tới **std::lock** để tránh Deadlock

1. Tránh gọi hàm khi đang giữ mutex

Chúng ta không lường trước được cái gì sắp xảy ra khi gọi hàm trong phạm vi khóa mutex, bao gồm việc khóa thêm mutex.

Như vậy là vi phạm nguyên tắc đầu tiên: khóa lồng mutex.

Thỉnh thoảng điều này cũng khó tránh: khi chúng ta triển khai teamplate thread safe dùng chung như **std::stack**, thì tất cả thao tác trên kiểu của tham số hay kiểu có sẵn nên để người dùng định nghĩa. (Bạn cần hướng dẫn thêm)

1. Khóa theo trật tự cố định

Trường hợp không thể dùng **std::lock** để khóa nhiều mutex cùng lúc, phương án thứ 2 là khóa theo cùng trật tự với mọi thread.

Chúng ta xem lại ví dụ về **std::stack**, mutex triển khai nội bộ của lớp, phần xử lí dữ liệu vẫn do người dùng định nghĩa. Nhưng chúng ta có thể thiết kế lại để không xảy ra bất kì thao tác trực tiếp lên dữ liệu của stack, nó tạo gánh nặng về cách truy xuất cho người dùng, những tính cách của bộ chứa dữ liệu như vậy khá lạ lẫm.

Trường hợp về danh sách liên kết, chỉ có một cách để các thread cùng truy xuất 1 lúc là triển khai mutex cho từng node.

Thread phải khóa từng node nó đi qua, nếu xóa node, nó phải khóa cả 3 node (node xóa và 2 node liền kề).

Khi duyệt DSLK, trong khi đang giữ khóa node hiện tại, thread sẽ yêu cầu khóa node tiếp theo, đảm bảo rằng node tiếp theo không bị thao tác trong lúc đó. Khi node tiếp theo bi khóa, thread sẽ unlock node hiện tại khi không cần.

Tuy nhiên phải đảm bảo các thread duyệt DSLK theo một chiều. Nếu 2 thread duyệt ngược chiều, giả sử chúng đi tới 2 node A, B kế cận. Thread #1 nắm mutex node A, thread #2 nắm mutex node B, cả 2 đợi nhau để khóa mutex node tiếp theo, hệ quả là deadlock.

Tương tự, nếu thao tác xóa node B giữa 2 node A và C, thread khóa mutex trên node A, B sau đó là C, chẳng may có thread đang duyệt DSLK ngược chiều, đang khóa mutex C. Có một cách để tránh deadlock ở đây là quy định chiều duyệt

Các quy tắc trên DSLK đều có thể áp dụng cho các cấu trúc dữ liệu khác.

1. Chuyển quyền sở hữu mutex

Vì đối tượng **std::unique\_lock** không sở hữu cố định mutex, nên nó có thể chuyển quyền sở hữu qua lại với nhau khi hàm trả về đối tượng hay dùng **std::move()**.

Theo lý thuyết, nó phụ thuộc vào đối tượng là Lvalue (biến, tham chiếu ) hay Rvalue (đối tượng tạm). Quyền sở hữu được trao tự động từ đối tượng Rvalue, và phải tường minh dùng **std::move()** đối với đối tượng Lvalue để tránh các lỗi có thể xảy ra.

**std::unique\_lock** có triển khai **move semantic** nhưng không hỗ trợ sao chép, giống như con trỏ **unique\_ptr** vậy.

Đoạn code dưới dùng kỹ thuật “gateway” để truy xuất gián tiếp tới dữ liệu của lớp đích: tất cả truy xuất phải thông qua phương thức thành viên của lớp gateway này (phương thức **get**\_**lock**()), để lấy được khóa.

Khóa sau đó được return về cho **process\_data**() tại (#2), và gọi do\_something() để xử lí tiếp dữ liệu đã chuẩn bị trước từ prepare\_data()

**Triển khai phương thức gateway lấy khóa**

std::unique\_lock<std::mutex> get\_lock(){

extern std::mutex some\_mutex;

std::unique\_lock<std::mutex> lk(some\_mutex);

prepare\_data();

return lk; //#1

}

void process\_data(){

std::unique\_lock<std::mutex> lk(get\_lock()); //#2

do\_something();

}

Khi (#1) return về, **std::unique\_lock** sẽ gọi hàm dựng **move** tại (#2) nên không cần gọi tới **std::move()**.

Mô hình triển khai này thường dùng khi mutex bị khóa lệ thuộc vào trạng thái hiện tại của chương trình, hay các phương thức có trình tự thực thi liên quan tới nhau .

Sau khi xong việc, đối tượng gateway bị hủy, khóa sẽ được mở và cho phép thread khác truy xuất tới dữ liệu bảo vệ.

Lớp gateway như vậy thường được triển khai **move semantic** và cả đối tượng khóa cũng cần triển khai **move.**

1. Sử dụng khóa hiệu quả
2. Mở khóa ngay khi có thể

Một khóa được đánh giá theo các tiêu chí sau:

* Phạm vi dữ liệu mà khóa bảo vệ
* Phạm vi của khóa có phù hợp (chỉ áp dụng cho các tiền trình thực sự cần)

Hình dung tới ví dụ thực tế trong hàng đợi xe trolly trên một quầy tính tiền trong siêu thị, chúng ta thường thấy một vài khách hay quên mua một vài món đồ nào đó và bỏ xe đi lấy hàng, khiến cho cả đoàn xe phải chờ.

Hay trường hợp thu ngân đã ra hóa đơn, nhưng khách còn mải lục kiếm bóp trong túi,...

Mọi thủ tục sẽ đơn giản đi rất nhiều khi mọi người có sự chuẩn bị tốt mọi thứ.

Nguyên tắc này cũng áp dụng cho threads: nếu nhiều thread đang đợi cùng một tài nguyên, tổng thời gian chờ sẽ rất lớn nếu bất kì thread cũng nắm khóa lâu hơn cần thiết.

Tính linh hoạt của **std::unique\_lock** còn nằm ở phương thức thành viên **unlock().** Giống như mutex, **std::unique\_lock** hỗ trợ **unlock**() và **lock**(), để nó được dùng với các hàm generic như **std::lock().**

Chúng ta có thể mở khóa trước khi hủy đối tượng của **std::unique\_lock ,** khi mà khóa không còn cần nữa. Khóa mở càng sớm thì thread khác sớm có cơ hội thực thi, gián tiếp tăng hiệu suất ứng dụng.

void get\_and\_process\_data()

{

std::unique\_lock<std::mutex> my\_lock(the\_mutex);

some\_class data\_to\_process = get\_next\_data\_chunk();

my\_lock.unlock(); //#1

result\_type result = process(data\_to\_process);

my\_lock.lock(); //#2

write\_result(data\_to\_process,result);

}

//#1 Không nhất thiết phải khóa mutex cho process()

//#2 Khóa mutex lại khi cần ghi dữ liệu

Nhớ lại cách triển khai một mutex cho toàn bộ cấu trúc dữ liệu, chúng ta có thể tối ưu hơn thời gian giữ khóa cho từng tác vụ thực thi.

Chỉ nên khóa mutex khi thực sự truy xuất tới dữ liệu dùng chung, đừng làm những thao tác tốn thời gian như xử lí I/O khi đang nắm khóa, tiến trình I/O thường chậm hơn hàng ngàn lần so với trên vùng nhớ, trừ khi có lý do chính đáng.

1. Tips

Cùng lúc với việc giảm thời gian giữ khóa, triệt tiêu khả năng deadlock, chúng ta phải lưu ý về cách triển khai.

Xem Listing 3.10 lớp Y triển khai toán tử so sánh:

**Listing 3.10 Khóa một mutex tại thời điểm trong toán tử so sánh**

class Y

{

private:

int some\_detail;

mutable std::mutex m;

int get\_detail() const

{

std::lock\_guard<std::mutex> lock\_a(m); //#3

return some\_detail;

}

public:

Y(int sd):some\_detail(sd){}

friend bool operator==(Y const& lhs, Y const& rhs)

{

if(&lhs==&rhs)

return true;

int const lhs\_value=lhs.get\_detail(); //#1

int const rhs\_value=rhs.get\_detail(); //#2

return (lhs\_value==rhs\_value); //#4

}

};

**Giải thích**

Phép toán so sánh gọi tới phương thức thành viên get\_detail() tại (#1 và #2) để lấy dữ liệu thành viên có khóa bảo vệ(#3). Giá trị trả về tại (#4).

Nếu dữ liệu thành viên có dữ liệu lớn, thì thao tác sao chép dữ liệu là không nên, làm giảm hiệu suất, nên dùng tham chiếu.

Phép so sánh returns true tức thời điểm đó giá trị lhs.some\_detail == rhs.some\_detail.

Tuy nhiên, vẫn có khả năng hai giá trị có thể bị thay đổi theo nhiều cách giữa hai thao tác đọc: giữa #1 và #2, khiến việc so sánh là vô nghĩa.

Chúng ta cần cẩn thận khi triển khai theo mô hình khá mơ hồ thế này, nếu bạn không giữ khóa cho toàn bộ tiến trình thì bạn đang tự mình dẫn tới race conditions.

Đôi khi chúng ta không thể định nghĩa được mức độ phù hợp cho việc đặt khóa đối với truy xuất tới cấu trúc dữ liệu, phải dùng khóa cùng level. Lúc này cần phải có cơ chế thay thế phù hợp thay cho **std::mutex**.

##### Khởi tạo dữ liệu dùng chung

Mutex là cơ chế chung nhất nhưng không phải duy nhất khi cần bảo vệ dữ liệu dùng chung. Có trường hợp khi dữ liệu dùng chung chỉ cần bảo vệ khi nó đang được khởi tạo, giai đoạn sau đó không cần đồng bộ và tối ưu được hiệu suất.

Đây chính là lý do mà Standard C++ cung cấp cơ chế chuyên bảo vệ dữ liệu dùng chung trong quá trình khởi tạo

1. Lazy initialization

Xét tình huống bạn có tài nguyên chia sẻ, khá tốn chi phí để khởi tạo như việc mở một kết nối tới cơ sở dữ liệu, hay cấp phát vùng nhớ lớn.

Lazy-initialization là kỹ thuật dùng code đơn thread khi khởi tạo tài nguyên dùng chung.

Mỗi tiến trình cần tài nguyên phải kiểm tra trước xem tài nguyên đã khởi tạo chưa, nếu chưa thì phải khởi tạo trước khi dùng. Đoạn code sau được dùng rất phổ biến:

**Lazy Initialization**

#include <memory>

std::shared\_ptr<some\_resource> resource\_ptr;

void foo(){

if(!resource\_ptr) {

resource\_ptr.reset(new some\_resource); //#1

}

resource\_ptr->do\_something();

}

Nếu tài nguyên chia sẻ bản thân nó đã an toàn, thì phần cần bảo vệ khi chuyển nó sang code đa luồng là quá trình khởi tạo (#1) như trong listing 3.11 dưới đây:

**Listing 3.11: Thread-safe Lazy Initialization dùng Mutex**

std::shared\_ptr<some\_resource> resource\_ptr;

std::mutex resource\_mutex;

void foo(){

std::unique\_lock<std::mutex> lk(resource\_mutex); //#1

if(!resource\_ptr) {

resource\_ptr.reset(new some\_resource); //#2

}

lk.unlock();

resource\_ptr->do\_something();

}

//#1 Tất cả threads phải đợi ở đây

//#2 Chỉ có quá trình khởi tạo mới cần bảo vệ

Triển khai trên được dùng nhiều vì nó an toàn.

1. Data race

Tuy nhiên nhiều người lại triển khai theo cách khác, bao gồm cả mô hình không phổ biến Double-checked Locking:

void undefined\_behaviour\_with\_double\_checked\_locking(){

if(!resource\_ptr) //#1 đọc con trỏ không dùng khóa

{

std::lock\_guard<std::mutex> lk(resource\_mutex); //Chỉ khóa khi con trỏ NULL

if(!resource\_ptr) //#2 Kiểm tra lần thứ 2

{

resource\_ptr.reset(new some\_resource); //#3 Khởi tạo dữ liệu không đồng bộ với kiểm tra dữ liệu tại #1

}

}

resource\_ptr->do\_something(); //#4

}

Sẽ có trường hợp có một thread khác đã hoàn thành khởi tạo giữa lần kiểm tra đầu tiên và thao tác nắm khóa.

Nguy cơ xảy ra race condition khi việc đọc con trỏ nằm ngoài khóa (#1) không được đồng bộ với thao tác ghi vào con trỏ bên trong khóa (#3).

#4: Dù một thread sử dụng con trỏ đang được ghi bởi thread khác, nó vẫn không nhìn thấy đối tượng mới được khởi tạo “some\_resource”.

Kết quả là việc gọi hàm do\_something() diễn ra trên giá trị sai. Đây là trường hợp về data race trong Standard C++.

1. Dùng std::once\_flag và std::call\_once

Ủy ban Standard C++ đã nhận thấy điều này là nghiêm trọng, nên đã cung cấp **std::once\_flag** và **std::call\_once()** để xử lí trường hợp này.

template <class Fn, class... Args>

void call\_once (once\_flag& flag, Fn&& fn, Args&&... args);

**call\_once** sẽ thực thi hàm khởi tạo tài nguyên **fn**

Nếu đang có thread khác gọi tới **std::call\_once** với cùng **flag**, sẽ gây ra tình trạng thực thi thụ động: tức không gọi tới hàm **fn** nhưng sẽ không return tới khi thread khác đã thực thi xong **call\_once** và trả về, hàm có tác dụng đồng bộ cho tất cả thread gọi tới hàm này với cùng một **flag.**

Nếu **call\_once** xảy ra ngoại lệ và đang xảy ra thực thi thụ động, thì một thread được chọn để thực thi lệnh gọi mới vào hàm **call\_once**.

Lưu ý rằng khi **call\_once** return, tất cả thực thi thụ động gọi tới của **call\_once** (cùng **flag**) cũng return mà không thực thi **fn**.

Nếu dùng đúng cách, **std::call\_once** tốn ít chi phí hơn là mutex

Đoạn code bên dưới triển khai phương thức giống như listing 3.11, sử dụng **std::call\_once**.

std::shared\_ptr<some\_resource> resource\_ptr;

std::once\_flag resource\_flag;

void init\_resource(){

resource\_ptr.reset(new some\_resource);

}

void foo(){

std::call\_once(resource\_flag,init\_resource); //#1

resource\_ptr->do\_something();

}

//#1 init\_resource() được gọi chính xác duy nhất 1 lần

Quá trình khởi tạo thực hiện bởi **std::call\_one** với tham số địa chỉ hàm gọi khởi tạo tài nguyên và đối tượng **std::once\_flag**.

Ở ví dụ trên, **std::one\_flag** và dữ liệu được khởi tạo nằm trong tên miền của đối tượng, nhưng **std::call\_once**() có thể dùng khởi tạo lazy initialization cho thành viên của đối tượng, như trong listing 3.12.

**Listing 3.12: Thread-safe Lazy Initialization of a Class Member dùng std::call\_once**

class X{

private:

connection\_info connection\_details;

connection\_handle connection;

std::once\_flag connection\_init\_flag;

void open\_connection(){

connection=connection\_manager.open(connection\_details);

}

public:

X(connection\_info const& connection\_details\_):

connection\_details(connection\_details\_)

{}

void send\_data(data\_packet const& data) //#1

{

std::call\_once(connection\_init\_flag,&X::open\_connection,this); //#3

connection.send\_data(data);

}

data\_packet receive\_data() //#2

{

std::call\_once(connection\_init\_flag,&X::open\_connection,this); //#3

return connection.receive\_data();

}

};

Quá trình khởi tạo sẽ được gọi tới cho lần gọi hàm đầu tiên send\_data() (#1), hay receive\_data() (#2).

Phương thức thành viên open\_connection() để khởi tạo dữ liệu connection cần thêm tham số con trỏ this\* tại #3 để **call\_once** biết đối tượng nào đang gọi, cũng giống như các phương thức khác trong thư viện chuẩn (**std::thread** và **std::bind**).

Một ví dụ khác về race condition trong quá trình khởi tạo liên quan tới biến **static local**. Khởi tạo cho biến static local được định nghĩa là cho lần thực thi đầu tiên đi qua.

Với nhiều thread cùng gọi tới hàm vào cùng thời điểm có thể xảy ra race condition với khái niệm lần đầu này hoặc một thread khác đang sử dụng biến trong khi thread khác chưa khởi tạo xong. Tình huống này xảy ra phổ biến trên các trình biên dịch trước C++0x.

Vấn đề này trên C++0x về sau đã được giải quyết, luôn chỉ có một thread khởi tạo, các thread khác chỉ được thực thi khi quá trình khởi tạo hoàn tất. Với các trường hợp chỉ có một biến global thì không cần phải dùng **call\_once**

class my\_class;

my\_class& get\_my\_class\_instance()

{

static my\_class instance; //#1

return instance;

}

//#1 khởi tạo đối tượng được đảm bảo thread-safe.

Các thread có thể gọi get\_my\_class\_instance() an toàn.

Bảo vệ dữ liệu trong quá trình khởi tạo là một trường hợp riêng khi bảo vệ dữ liệu.

Trường hợp cấu trúc dữ liệu cần cập nhật, cần phải có cơ chế bảo vệ.

##### Cập nhật dữ liệu dùng chung

Xét ví dụ bảng DNS entries lưu trữ tên miền cùng địa chỉ IP tương ứng. Một DNS entry thường không thay đổi trong một giai đoạn nhất định.

Các entry sẽ được thêm vào bảng liên tục, nhưng đồng thời chúng được kiểm tra định kỳ để xác minh tính hợp lệ hay site đó còn alive hay không.

Race condition có thể xảy ra, nếu cache bị truy xuất bởi nhiều thread, bảng DNS cần cơ chế bảo vệ khi cập nhật dữ liệu nhằm đảm bảo không có thread nào đọc cache ở trạng thái chưa sẵn sàng.

Thao tác cập nhật cần không gian truy xuất dành riêng tới dữ liệu cho tới khi nó hoàn thành, sau đó dữ liệu lại có thể duyệt tự do bởi các thread khác.

1. boost::shared\_mutex

Việc sử dụng **std::mutex** tỏ ra hạn chế, khiến dữ liệu khó có thể duyệt đa luồng được khi không ở trạng thái cập nhật.

Chúng ta cần một kiểu mutex cho phép hai kiểu truy xuất:

* Truy xuất dành riêng cho một thread ghi
* Truy xuất đa luồng bởi nhiều thread đọc.

Tuy nhiên, Standard C++ chưa hỗ trợ kiểu mutex như vậy, dù có đề xuất tới Ủy ban C++, tạm thời chúng ta sử dụng đối tượng **boost**::**shared**\_**mutex** trong thư viện của **Boost**

Tiến trình cập nhật sẽ dùng **std::lock\_guard**<**boost::shared\_mutex**> hay **std::unique\_lock**<**boost::shared\_mutex**> để khóa.

Tiến trình duyệt dùng **boost::shared\_lock**<**boost::shared\_mutex**> để lấy quyền truy xuất chung.

Cơ chế **shared\_lock** khá giống với **unique\_lock**, ngoài trừ **shared\_mutex** có thể sở hữu bởi nhiều thread.

Listing 3.13 minh họa quá trình update DNS cache dùng std::map

**Listing 3.13 Bảo vệ cấu trúc dữ liệu chung với boost::shared\_mutex**

#include <map>

#include <string>

#include <mutex>

#include <boost/thread/shared\_mutex.hpp>

class dns\_entry;

class dns\_cache{

std::map<std::string,dns\_entry> entries;

boost::shared\_mutex entry\_mutex;

public:

dns\_entry find\_entry(std::string const& domain) {

boost::shared\_lock<boost::shared\_mutex> lk(entry\_mutex); //#1

std::map<std::string,dns\_entry>::const\_iterator const it = entries.find(domain);

return (it==entries.end())? dns\_entry(): \*it;

}

void update\_or\_add\_entry(std::string const& domain,dns\_entry const& dns\_details) {

std::lock\_guard<boost::shared\_mutex> lk(entry\_mutex); //#2

entries[domain]=dns\_details;

}

};

Phương thức find\_entry() sử dụng boost::shared\_lock<> tại #1: có thể duyệt dữ liệu đa luồng.

Khi một thread cập nhật dữ liệu DNS entry đang gọi update\_or\_add\_entry() và nắm khóa std::lock\_guard<> tại #2, thì tất cả thread khác đang gọi tới find\_entry() hay update\_or\_add\_entry() đều bị khóa.

1. boost::upgrade\_lock và boost::upgrade\_to\_unique\_lock

Ngoài ra, chúng ta có thể linh hoạt trong thao tác cập nhật dữ liệu với **boost::upgrade\_lock<>** và **boost::upgrade\_to\_unique\_lock<>**

#include "boost\thread\thread.hpp"

boost::shared\_mutex \_access;

void reader()

{

boost::shared\_lock< boost::shared\_mutex > lock(\_access); //#1

// do work here, without anyone having exclusive access

}

void conditional\_writer()

{

boost::upgrade\_lock< boost::shared\_mutex > lock(\_access); //#2

// do work here, without anyone having exclusive access

if (something) {

boost::upgrade\_to\_unique\_lock< boost::shared\_mutex > uniqueLock(lock); //#3

// do work here, but now you have exclusive access

}

// do more work here, without anyone having exclusive access

}

void unconditional\_writer()

{

boost::unique\_lock< boost::shared\_mutex > lock(\_access); //#4

// do work here, with exclusive access

}

Tại một thời điểm chỉ có một thread nhận khóa tại #2 nhưng không phải truy xuất dành riêng. Khi thread vào #3, sẽ là truy xuất dành riêng, mọi thread đang cố lấy khóa tại #1 , #2 và #4 đều phải đợi cho tới khi thread này mở khóa

##### Kết luận

Chương này giới thiệu cách sử dụng **std::mutex** trong triển khai hàm và interface để tránh race condition và sử dụng **std::lock**() để tránh dead lock.

Ngoài ra, chúng ta có thể dùng các kỹ thuật dựa trên khái niệm về quyền sở hữu cho khóa

Các phương pháp bảo vệ dữ liệu dùng chung như **std::call\_once**(), và cơ chế đồng bộ read-write dữ liệu dùng chung đa luồng với **boost::shared\_mutex**.

### Đồng bộ các tiến trình

Trong triển khai Thread-safe stack, ngoại lệ chỉ ném ra khi stack rỗng, nếu một thread muốn đợi thread khác đẩy dữ liệu vào stack, nó sẽ phải đợi hay kiểm tra stack liên tục. Việc này gây lãng phí tài nguyên và hiệu suất hệ thống.

Tình huống một thread cần đợi thread khác hoàn thành một tác vụ nào đó trước rất phổ biến.

Vậy làm sao để thông báo cho thread khác biết được là thread kia đã hoàn thành xong hoặc không thể hoàn thành vì xảy ra lỗi.

Chương này đề cấp các phương pháp đồng bộ khi một thread phải đợi một sự kiện với biến điều kiện và futures, cũng như cách dùng.

##### Tình huống

Chúng ta đang xây dựng ứng dụng network, gồm các tác vụ:

* Thực hiện bước handshaking với server
* Đọc dữ liệu từ XML files.
* Xử lí dữ liệu XML vừa đọc.

Tác vụ 1 có thể chạy độc lập với 2 tác vụ sau, nhưng tác vụ 3 phụ thuộc vào tác vụ 2.

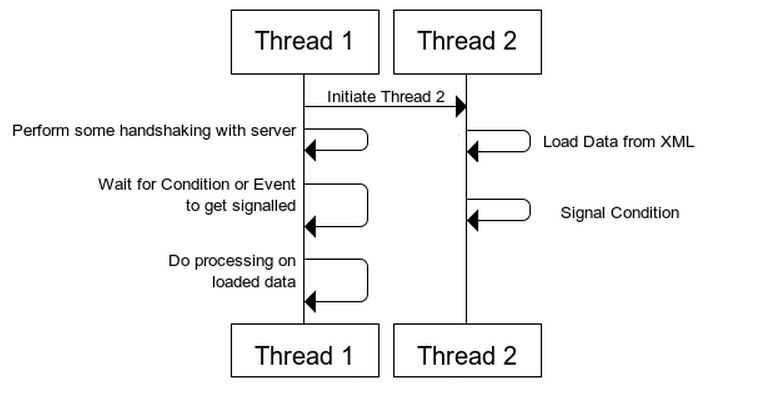
Như vậy có thể chạy song song tác vụ 1 và 2.

Thiết kế ứng dụng này sẽ có 2 thread, thread 1 :

* handshaking với server.
* Đợi dữ liệu đọc từ XML bởi Thread 2
* Sau đó xử lí dữ liệu XML.

Thread 2:

* Đọc dữ liệu từ XML file
* Báo hiệu cho các thread khác đang đợi.

[](http://thispointer.com/wp-content/uploads/2015/06/first.png)

Theo thiết kế trên, Thread 1 sau khi handshaking sẽ đợi một sự kiện/điều kiện xảy ra, đó là: dữ liệu XML đã được nạp thành công chưa?

Trong khi Thread 1 đang tiến hành 3-way handshake với server, Thread 2 tiến hành song song đọc dữ liệu từ file XML.

Thread 2 đọc dữ liệu XML thành công sẽ báo lại cho Thread 1 bằng cách phát tín hiệu sự kiện.

Sau khi nhận được phản hồi thành công từ Thread 2, Thread 1 sẽ tiến hành xử lí dữ liệu XML.

Phần tiếp theo sẽ đề cập các phương tiện để Thread 2 phát tín hiệu tới Thread 1.

##### Giải pháp

1. Dùng biến chia sẻ với std::this\_thread::sleep

Tạo một biến global flag dùng chung mặc định là false. Khi Thread 2 hoàn thành sẽ set giá trị sang true, còn Thread 1 sẽ tiếp tục kiểm tra giá trị trong vòng lặp, ngay khi thấy flag thỏa điều kiện, Thread 1 sẽ xử lí dử liệu.

Biến flag này dùng chung nên cần đồng bộ với mutex.

Tuy nhiên việc bắt các thread phải đợi nhau và check flag liên tục là không cần thiết, gây tốn hiệu suất.

Giải pháp là để thread “nghỉ”, không làm gì cả trong một khoảng thời gian đủ ngắn giữa các lần kiểm tra flag, bằng cách sử dụng **std::this\_thread::sleep\_for**()

Cú pháp:

template< class Rep, class Period >

void sleep\_for( const std::chrono::duration<Rep, Period>& sleep\_duration )

Tham số:

* sleep\_duration : thời gian ngủ

Khóa thực thi của thread ít nhất trong khoảng thời gian qui định bởi sleep\_duration, thời gian khóa có thể lâu hơn do độ trễ của tài nguyên.

Chúng ta nên sử dụng đồng hồ an toàn từ thư viện chuẩn **std**::**chrono** để đo lường quá trình thực thi. Nếu dùng đồng hồ hệ thống, thời gian đợi có thể không đúng với điều chỉnh dự định.

|  |  |
| --- | --- |
| **Class Helper** | **Tham số** |
| std::chrono::nanoseconds | Nano giây, kiểu std::nano, số nguyên có dấu dài ít nhất 64-bits |
| std::chrono::microseconds | Micro giây, kiểu std::micro, số nguyên có dấu dài ít nhất 55-bits |
| std::chrono::milliseconds | Mili giây, kiểu std::mili, số nguyên có dấu dài ít nhất 45-bits |
| std::chrono::seconds | Giây , số nguyên có dấu dài ít nhất 35-bits |
| std::chrono::minutes | Phút, std::ratio<60>, số nguyên có dấu dài ít nhất 29-bits |
| std::chrono::hours | Phút, std::ratio<3600>, số nguyên có dấu dài ít nhất 23-bits |

Lưu ý: Mỗi kiểu thời gian định nghĩa trước chỉ được nằm trong khoảng ±292 years.

Sẽ không có bất kì ngoại lệ nào ném ra bởi đồng hồ, time\_point hay giai đoạn thực thi của thư viện chuẩn

#include<thread>

#include<mutex>

class Application{

private:

std::mutex m\_mutex;

bool m\_bDataLoaded;

public:

Application(){m\_bDataLoaded = false;}

void loadData()

{

// Để Thread này sleep trong 1 Second

std::this\_thread::sleep\_for(std::chrono::milliseconds(1000));

std::cout<<"Loading Data from XML"<<std::endl;

// Khóa trước khi truy xuất dữ liệu XML dùng chung

std::lock\_guard<std::mutex> guard(m\_mutex);

// Set flag = true khi dữ liệu đã nạp thành công

m\_bDataLoaded = true;

}

void mainTask()

{

std::cout<<"Do Some Handshaking"<<std::endl;

// Lấy khóa

m\_mutex.lock();

// Kiểm tra flag trong vòng lặp

while(m\_bDataLoaded != true)

{

// Mở khóa trước khi sleep #1

m\_mutex.unlock();

//sleep trong 100 milli giây

std::this\_thread::sleep\_for(std::chrono::milliseconds(100));

// Lấy khóa để bắt đầu check flag lại #2

m\_mutex.lock();

}

// Mở khóa

m\_mutex.unlock();

//Bước xử lí dữ liệu XML

std::cout<<"Do Processing On loaded Data"<<std::endl;

}

};

int main()

{

Application app;

std::thread thread\_1(&Application::mainTask, &app);

std::thread thread\_2(&Application::loadData, &app);

thread\_2.join();

thread\_1.join();

return 0;

}

Nguyên tắc là phải mở khóa tại #1 trước khi sleep là để các Thread khác có thể lấy khóa để kiểm tra flag hay đọc dữ liệu.

Điều khó khăn là phải canh thời gian ngủ vừa đủ, không quá ngắn gây tốn hiệu suất hay quá lâu gây ra độ trễ không cần thiết.

Khi sleep quá lâu gây ảnh hưởng tới thực thi ứng dụng thì không nghiêm trọng, nhưng đối với ứng dụng game, việc để rớt khung hình là không cấp nhận được hay đối với các ứng dụng real-time.

Việc Thread liên tục khóa và mở khóa chỉ để check flag sẽ tốn tài nguyên hệ thống và khiến các Thread chậm lại vì phải cùng lấy một khóa duy nhất.

Chúng ta cần một cơ chế tốt hơn, để Thread mainTask chỉ tạm thời ngưng lại và đợi cho một sự kiện từ Thread loadData phát ra cảnh báo tới, giúp tiết kiệm tài nguyên hệ thống và tăng hiệu suất.

1. Dùng biến điều kiện

Nếu chúng ta dùng flag chia sẻ (đồng bộ bởi mutex), và thread thứ hai sẽ set flag khi nó hoàn tất tác vụ thì có 2 điểm bất cập là:

* Hiệu suất kém
* mutex bị khóa cứng bởi thread đợi, có thể gây ra deadlock cho các thread khác cũng đang cần check flag.

Deadlock còn xảy ra khi thread thực thi đang cần set flag cũng không thể làm được do mutex đang bị khóa bởi thread kiểm tra flag.

1. Khái niệm

Thư viện C++ chuẩn cung cấp phương tiện giúp các thread giao tiếp với nhau, một trong số đó là biến điều kiện.

Biến điều kiện là một kiểu sự kiện dùng để báo hiệu giữa 2 hay nhiều threads. Một hay nhiều thread có thể đợi biến điều kiện để nhận thông báo, trong khi thread khác phát thông báo tới biến này.

Để dùng biến điều kiện chúng ta include header **<condition\_variable>**

Thư viện chuẩn C++ hai lớp điều kiện : **std**::**condition**\_**variable**, và **std**::**condition**\_**variable**\_**any**

Cả hai đều cần mutex cho tiến trình đồng bộ:

* **std::condition\_variable**: là lớp đồng bộ chuẩn có thể dùng để khóa một hay nhiều thread cùng lúc, và đợi cho tới khi thread khác hoàn thành xử lí xong đối với dữ liệu dùng chung (điều kiện) và thông báo tới biến điều kiện. **std::condition\_variable** chỉ làm việc với **std::unique\_lock**<**std**::**mutex**>
* **std**::**condition**\_**variable**\_**any:** làlớp teamplate triển khai từ **std::condition\_variable**. **condition\_variable\_any** tỏ ra linh hoạt hơn do có thể làm việc với bất kì loại khóa **BasicLockable** nào (đó là lý do có hậu tố \_any ở cuối).

**std::condition\_variable\_any** lại tốn chi phí hệ thống hơn, chúng ta nên ưu tiên dùng **std::condition\_variable**.

1. Wait()

Phương thức thành viên wait() sẽ khóa thread hiện tại tới khi biến điều kiện nhận được phản hồi hay bị kích hoạt ngẫu nhiên vì lý do nào đó.

Nó tự động mở khóa mutex, ngưng thread hiện tại lại và đăng ký thread này vào danh sách đợi của biến điều kiện.

Để đảm bảo mọi thứ diễn ra đúng, sẽ có tác vụ kiểm tra lại biến điều kiện trước khi kích hoạt thread.

Hàm gọi callback dùng để kiểm tra điều kiện được truyền như tham số của wait().Khi có thread khác phản hồi thông qua hàm gọi notify\_one() hay notify\_all() trên cùng một biến điều kiện, Thread đợi sẽ được kích hoạt trở lại.

wait() sẽ khóa mutex, sau đó gọi hàm callback để xác thực rằng điều kiện đã thỏa.

Nếu điều kiện chưa thỏa, nó tự động mở khóa mutex, ngưng thread đợi.

Phương thức wait hỗ trợ hai tác vụ cốt lõi lock() và unlock() mutex.

1. notify\_one()

Hàm thành viên notify\_one() sẽ kích hoạt một trong các thread đợi của cùng biến điều kiện.

1. notify\_all()

Hàm thành viên notify\_all() sẽ kích hoạt tất cả các thread đợi của cùng biến điều kiện.

1. Triển khai

Hình dung lại tình huống trước, Thread mainTask sẽ gọi thành viên wait() của biến điều kiện, nó sẽ khóa mutex nội bộ và kiểm xem điều kiện có thỏa hay chưa.

Nếu chưa thì mở khóa và đợi tới khi có phản hồi từ bên ngoài, thread mainTask tiếp tục bị block.

Thread loadData sẽ phản hồi cho biến điều kiện khi điều kiện thỏa mãn. Khi biến điều kiện nhận được phản hồi, nó kích hoạt Thread mainTask, khóa mutex và kiểm tra lần nữa để xác thực không phải tín hiệu giả mạo.

Nếu có nhiều thread đợi thì notify\_once() chỉ unblock 1 thread mà thôi.

Nếu chỉ là tín hiệu giả, tức điều kiện chưa thỏa thì Thread đợi sẽ gọi lại wait().

#include <iostream>

#include <thread>

#include <functional>

#include <mutex>

#include <condition\_variable>

using namespace std::placeholders;

class Application

{

std::mutex m\_mutex;

std::condition\_variable m\_condVar;

bool m\_bDataLoaded;

public:

Application(){m\_bDataLoaded = false;}

void loadData()

{

// Cho Thread sleep 1 Second

std::this\_thread::sleep\_for(std::chrono::milliseconds(1000));

std::cout<<"Loading Data from XML"<<std::endl;

// Khóa cấu trúc dữ liệu

std::lock\_guard<std::mutex> guard(m\_mutex);

// Set flag = true, khi dữ liệu nạp thành công

m\_bDataLoaded = true;

// Phản hồi tới biến điều kiện

m\_condVar.notify\_one();

}

bool isDataLoaded() {return m\_bDataLoaded;}

void mainTask()

{

std::cout<<"Do Some Handshaking"<<std::endl;

// khóa mutex dùng đối tượng BasicLockable

std::unique\_lock<std::mutex> mlock(m\_mutex);

// Bắt đầu đợi cho biến điều kiện nhận được phản hồi

// Wait() sẽ mở khóa và làm thread hiện tại bị block

// Ngay khi biến điều kiện nhận được phản hồi, kích hoạt thread lại

// khóa mutex. Sau đó kiểm tra điều kiện

// Tiếp tục xử lí nếu điều kiện thỏa, ngược lại gọi wait().

m\_condVar.wait(mlock, std::bind(&Application::isDataLoaded, this));

std::cout<<"Do Processing On loaded Data"<<std::endl;

}

};

int main()

{

Application app;

std::thread thread\_1(&Application::mainTask, &app);

std::thread thread\_2(&Application::loadData, &app);

thread\_2.join();

thread\_1.join();

return 0;

}

1. Thread-safe Queue với biến điều kiện

Việc sử dụng queue làm bộ chứa dữ liệu dùng chung giữa các threads rất phổ biến, nếu triển khai tốt thì ngay bản thân queue cũng giúp rất nhiều cho quá trình đồng bộ, giảm nguy cơ race condition.

1. Sử dụng queue

Listing 4.1 biểu diễn cách dùng **std::condition\_variable** để xử lí trường hợp một thread đang sleep đợi cho tới khi có dữ liệu sẵn sàng để làm việc.

**Listing 4.1 Cách dùng biến điều kiện**

std::mutex mut;

std::queue<data\_chunk> data\_queue; //#1 queue chứa dữ liệu dùng chung giữa các threads

std::condition\_variable data\_cond;

// threads chuẩn bị dữ liệu

void data\_preparation\_thread(){

while(more\_data\_to\_prepare())

{

data\_chunk const data = prepare\_data();

std::lock\_guard<std::mutex> lk(mut); //dùng std::lock\_guard bảo vệ flag dùng chung

data\_queue.push(data); //#2 nạp dữ liệu vào queue

data\_cond.notify\_one(); //#3 gọi phương thức member notify\_one() thông báo cho threads đang đợi rằng dữ liệu đã sẵn sàng

}

}

// threads xử lí dữ liệu

void data\_processing\_thread(){

while(true) {

std::unique\_lock<std::mutex> lk(mut); //#4 dùng std:: unique\_lock

data\_cond.wait(lk,[]{return !data\_queue.empty();}); //#5 gọi phương thức member wait() nhận khóa và biểu thức lambda mô tả điều kiện để đợi

data\_chunk data = data\_queue.front();

data\_queue.pop();

lk.unlock(); //#6 khi đã lấy được dữ liệu thì mở khóa mutex

process(data);

if(is\_last\_chunk(data))

break;

}

}

**Giải thích**

Hàm lambda cho phép viết các hàm vô danh, rất phù hợp để dùng cho các hàm chuẩn như wait(). Trong trường hợp này biểu thức lambda sẽ kiểm tra xem queue có rỗng hay không

wait() sẽ khóa mutex trước, sau đó kiểm tra điều kiện (bằng cách gọi hàm lambda) và trả về ngay nếu phương thức điều kiện trả về true.

Nếu điều kiện không thỏa, wait() mở khóa mutex và đặt threads vào trạng thái “đợi”.

Khi biến điều kiện được thông báo ở #3 từ threads chuẩn bị dữ liệu, threads xử lí “tỉnh” lại, khóa mutex và kiểm tra điều kiện lần nữa

* Khi điều kiện thỏa thì return từ wait() với mutex đã được lock.
* Khi điều kiện vẫn chưa thỏa thì threads unlock mutex và tiếp tục nằm đợi.

#4: Do trạng thái khóa phải liên tục lock và unlock trong wait(), nên chúng ta phải dùng **std::unique\_lock**, hơn là **std::lock\_guard**, do có hỗ trợ 2 tác vụ này. Nếu dùng **std::lock\_guard**, threads xử lí dữ liệu vẫn nắm khóa kể cả khi nó đang ở trạng thái “đợi”, khiến threads chuẩn bị không thể nạp dữ liệu vào queue, gây ra trình trạng deadlock.

#5: Ngoài biểu thức lambda, bất kì đối tượng có thể gọi nào đều truyền vào được.

Trong quá trình gọi wait(), biến điều kiện có thể kiểm tra điều kiện nhiều lần, do đó hàm điều kiện nên chỉ đơn giản, đúng mục đích.

#6: Để tối ưu quá trình xử lí dữ liệu có thể tốn thời gian, chúng ta nên mở khóa ngay khi có dữ liệu để tối ưu hiệu suất cho các threads khác. Đây là lí do tại sao phải dùng **std::unique\_lock**, hỗ trợ mở khóa và khóa mutex linh hoạt

1. Thiết kế thread-safe queue

Chúng ta sẽ bắt đầu thiết kế queue template, trước hết là các phương thức cần thiết. Triển khai sẽ dựa theo thư viện chuẩn C++.

**Listing 4.2: std::queue interface**

template <class T, class Container = std::deque<T>>

class queue {

public:

explicit queue(const Container&);

explicit queue(Container&& = Container());

queue(queue&& q);

template <class Alloc> explicit queue(const Alloc&);

template <class Alloc> queue(const Container&, const Alloc&);

template <class Alloc> queue(Container&&, const Alloc&);

template <class Alloc> queue(queue&&, const Alloc&);

queue& operator=(queue&& q);

void swap(queue&& q);

bool empty() const;

size\_type size() const;

T& front();

const T& front() const;

T& back();

const T& back() const;

void push(const T& x);

void push(T&& x);

void pop();

};

Bỏ qua hàm dựng, phép gán và phép hoán vị, còn lại chia thành 3 nhóm:

* Nhóm trạng thái: empty(), size()
* Nhóm truy vấn: front(), back()
* Nhóm truy xuất: push(), pop()

Tương tự như stack, queue có cùng vấn đề race condition trong triển khai.

Cần kết hợp front() và pop() thành một phương thức, giống với trường hợp top() và pop() của stack

Listing 4.1 đưa ra một ý tưởng: dùng queue truyền dữ liệu giữa các threads, chúng ta cung cấp phương thức

* try\_pop() : để lấy dữ liệu, nhưng luôn trả về ngay dù có hay không có dữ liệu trong queue.

Và phương thức:

* wait\_and\_pop() : sẽ đợi cho tới khi có dữ liệu để lấy.

**Listing 4.3: The interface cho thread-safe queue**

template<typename T>

class queue

{

public:

queue();

queue(const queue&);

queue& operator=(const queue&) = delete; //#1 Bỏ qua phép gán để đơn giản hơn

void push(T new\_value);

bool try\_pop(T& value); //#2

std::shared\_ptr<T> try\_pop(); //#3

void wait\_and\_pop(T& value);

std::shared\_ptr<T> wait\_and\_pop();

bool empty() const;

};

**Giải thích**

(#2) : phiên bản quá tải try\_pop trả về dữ liệu vào biến tham chiếu truyền vào, return true nếu lấy được và false nếu ngược lại.

(#3) : phiên bản quá tải try\_pop trả về dữ liệu trực tiếp. Con trỏ trả về có thể NULL nếu không có dữ liệu để lấy.

Triển khai cho push() và wait\_and\_pop() sẽ sử dụng biến điều kiện

**Listing 4.4: push() và wait\_and\_pop() theo listing 4.1**

template<typename T>

class queue

{

private:

std::mutex mut;

std::queue<T> data\_queue;

std::condition\_variable data\_cond;

public:

void push(T new\_value) {

std::lock\_guard<std::mutex> lk(mut);

data\_queue.push(data);

data\_cond.notify\_one();

}

void wait\_and\_pop(T& value) {

std::unique\_lock<std::mutex> lk(mut);

data\_cond.wait(lk,[]{return !data\_queue.empty();});

value=data\_queue.front();

data\_queue.pop();

}

};

queue<data\_chunk> data\_queue; //#1

void data\_preparation\_thread() {

while(more\_data\_to\_prepare()){

data\_chunk const data = prepare\_data();

data\_queue.push(data); //#2

}

}

void data\_processing\_thread() {

while(true) {

data\_chunk data;

data\_queue.wait\_and\_pop(data); //#3

process(data);

if(is\_last\_chunk(data))

break;

}

}

//#1 Khai báo queue mới dùng chung

//#2 không cần phải đồng bộ cho push() vì queue đã đồng bộ bên trong rồi

//#3 wait\_and\_pop() sẽ lo việc đồng bộ với hàm wait() bên trong

Chúng ta sẽ quá tải wait\_and\_pop() để có thể ghi dữ liệu.

Phiên bản đầy đủ của queue :

**Listing 4.5: Triển khai đầy đủ của thread-safe queue dùng biến điều kiện**

template<typename T>

class queue{

private:

mutable std::mutex mut; //#1 mutex phải là mutable để có thể khóa trong empty() và copy constructor

std::queue<T> data\_queue;

std::condition\_variable data\_cond;

public:

queue()

{}

queue(queue const& other)

{

std::lock(this->m, other.m);

std::lock\_guard<std::mutex> lk\_a(this->m,std::adopt\_lock);

std::lock\_guard<std::mutex> lk\_b(other.m,std::adopt\_lock);

data\_queue=other.data\_queue;

}

void push(T new\_value)

{

std::lock\_guard<std::mutex> lk(mut);

data\_queue.push(data);

data\_cond.notify\_one();

}

void wait\_and\_pop(T& value)

{

std::unique\_lock<std::mutex> lk(mut);

data\_cond.wait(lk,[]{return !data\_queue.empty();});

value=data\_queue.front();

data\_queue.pop();

}

std::shared\_ptr<T> wait\_and\_pop()

{

std::unique\_lock<std::mutex> lk(mut);

data\_cond.wait(lk,[]{return !data\_queue.empty();});

std::shared\_ptr<T> res(new T(data\_queue.front()));

data\_queue.pop();

return res;

}

bool try\_pop(T& value) {

std::lock\_guard<std::mutex> lk(mut);

if(data\_queue.empty)

return false;

value=data\_queue.front();

data\_queue.pop();

}

std::shared\_ptr<T> try\_pop(){

std::lock\_guard<std::mutex> lk(mut);

if(data\_queue.empty())

return std::shared\_ptr<T>();

std::shared\_ptr<T> res(new T(data\_queue.front()));

data\_queue.pop();

return res;

}

bool empty() const {

std::lock\_guard<std::mutex> lk(mut);

return data\_queue.empty();

}

};

1. Kết luận

Biến điều kiện rất hữu ích khi có nhiều thread cùng đợi một sự kiện, khi đó chỉ có một thread phản hồi tới cảnh báo mà thôi.

Triển khai trong Listing 4.1: cho phép gọi nhiều tiến trình xử lí dữ liệu cùng lúc.

Khi dữ liệu mới được thêm vào, việc gọi hàm notify\_once() sẽ đánh tín hiệu tới thread đang thực thi hàm gọi wait().

Tuy nhiên, không có sự gì đảm bảo là thread đợi sẽ được cảnh báo về sự kiện dữ liệu đã sẵn sàng, dù thực tế đang có thread nằm đợi cảnh báo. Nếu điều trên thực sự xảy ra, chúng ta cũng khó nhận biết được .

Trường hợp nhiều thread cùng đợi sự kiện và tất cả thread này đều cần phải được phản hồi khi

* Dữ liệu dùng chung cần được khởi tạo, các thread đợi sẽ dùng chung một dữ liệu.
* Dữ liệu dùng chung được cập nhật

Thread chuẩn bị dữ liệu có thể gọi tới notify\_all() của biến điều kiện, tất cả các thread đang thực thi hàm wait() sẽ kiểm tra điều kiện ngay lập tức.

Nếu thread đợi chỉ chờ đúng một sự kiện duy nhất thì nó sẽ không đợi biến điều kiện này lần thứ 2, cơ chế đồng bộ dùng biến điều kiện chưa phải tốt nhất nếu điều kiện đang đợi chỉ dành cho một đơn vị dữ liệu riêng biệt.

Trong tình huống này thì dùng biến Future sẽ phù hợp hơn.

1. std::future , std::promise và return từ ThreadFutures

Đối tượng std::future có thể dùng với asych, std::packaged\_task và std::promise. Trong chương nay chủ yếu đề cập tới cách dùng std::future với đối tượng std::promise.

Rất nhiều trường hợp chúng ta muốn thread trả về kết quả, ví dụ tác vụ nén một thư mục, thread trả về một file nén và kích thước của file là kết quả.

Sẽ có 2 cách để thực hiện:

1. Chia sẻ dữ liệu giữa các threads dùng con trỏ

Truyền con trỏ tới thread mới khởi tạo và thread này sẽ set dữ liệu cho con trỏ. Thread chính sẽ đợi thread nén dùng biến điều kiện.

Khi thread mới set dữ liệu và phát tín hiệu cho biến điều kiện, thread chính sẽ tỉnh lại và lấy dữ liệu từ con trỏ.

Tất cả phương tiện chúng ta cần để lấy kết quả trả về là:

* Biến điều kiện
* Mutex
* Con trỏ hay dữ liệu dùng chung.

Giả sử chúng ta cần 3 giá trị trả về khác nhau vào các thời điểm không cố định, phải có một giải pháp đơn giản hơn để trả về.

Đó là dùng đối tượng std::future.

1. C++11: Dùng std::future và std::promise

Đối tượng std::future là lớp teamplate, để chứa giá trị future.

Giá trị future ở đây nghĩa là gì? Thực ra là giá trị sẽ được đăng kí sau này và std::future cung cấp cơ chế để truy xuất giá trị này, bằng cách sử dụng phương thức thành viên get()

Nhưng nếu có bất kì truy xuất get() nào tới giá trị tương ứng với đối tượng std::future khi giá trị này chưa sẵn sàng, thì phương thức get() sẽ block thực thi

Đối tượng std::promise cũng là một lớp tempate và nhiệm vụ của nó để đảm bảo set giá trị future sau này. Mỗi đối tượng std::promise có đối tượng std::future tương ứng.

Đối tượng std::promise chia sẻ dữ liệu cùng với đối tượng std::future.

Tình huống:

Tạo đối tượng std::promise trong Thread1.

std::promise<int> promiseObj;

Đối tượng promise hiện tại chưa có liên kết với bất kì giá trị nào. Nhưng nếu đưa cho nó một giá trị và khi giá trị này được set, bạn có thể lấy giá trị này sau đó thông qua đối tượng std::future tương ứng.

Giả sử Thread 1 tạo đối tượng promise và truyền tới Thread 2, làm sao để Thread 1 biết khi nào Thread 2 định set giá trị trong đối tượng promise?

Câu trả lời là dùng đối tượng std::future.

Mọi đối tượng std::promise có 1 đối tượng std::future tương ứng, chúng ta dùng std::promise để set hay get giá trị future.

Đầu tiên Thread 1 sẽ tạo đối tượng std::promise, sau đó lấy đối tượng std::future từ promise

std::future<int> futureObj = promiseObj.get\_future();

Thread 1 truyền tham chiếu **promiseObj** vào hàm callback của Thread 2.

Để lấy giá trị dùng phương thức std::future:: get()

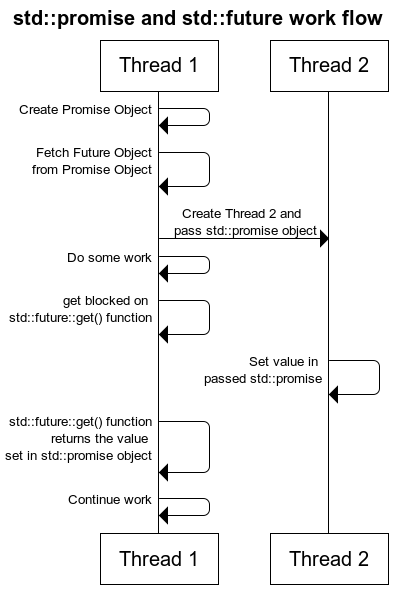
int val = futureObj.get();

Nhưng nếu giá trị chưa được set bởi Thread 2 thì việc gọi hàm get() sẽ bị block cho tới khi Thread 2 set giá trị trong đối tượng promise.

Để set giá trị dùng phương thức std::promise::**set\_value**()

promiseObj.set\_value(45);

Luồng xử lí minh họa theo biểu đồ bên dưới:

[](http://thispointer.com/wp-content/uploads/2015/06/promise.png)

Ví dụ về std::future và std::promise

#include <iostream>

#include <thread>

#include <future>

void initiazer(std::promise<int> \* promObj)

{

    std::cout<<"Inside Thread"<<std::endl;

   promObj->set\_value(35);

}

int main()

{

    std::promise<int> promiseObj;

    std::future<int> futureObj = promiseObj.get\_future();

    std::thread th(initiazer, &promiseObj);

    std::cout<<futureObj.get()<<std::endl;

    th.join();

    return 0;

}

Nếu đối tượng std::promise bị hủy trước khi set giá trị thì việc gọi hàm std::future::get() sẽ ném ra ngoại lệ.

Nếu muốn thread trả về nhiều giá trị tại nhiều thời điểm thì chỉ cần truyền nhiều đối tượng promise vào và lấy nhiều giá trị trả về từ đối tượng std::future tương ứng.

Phần tiếp theo sẽ hướng dẫn sử dụng std::future với std::asych và std::packaged\_task.

1. std::async

std::async() is a function template that accepts a callback(i.e. function or function object) as an argument and potentially executes them asynchronously.

template <class Fn, class... Args>

future<typename result\_of<Fn(Args...)>::type> async (launch policy, Fn&& fn, Args&&... args);

**std::async** returns a **std::future<T>,** that stores the value returned by function object executed by **std::async()**. Arguments expected by function can be passed to std::async() as arguments after the function pointer argument.

First argument in std::async is launch policy, it control the asynchronous behaviour of std::async. We can create std::async with 3 different launch policies i.e.

* **std::launch::async** It guarantees the asynchronous behaviour i.e. passed function will be executed in seperate thread.
* **std::launch::deferred** Non asynchronous behaviour i.e. Function will be called when other thread will call get() on future to access the shared state.
* **std::launch::async | std::launch::deferred** Its the default behaviour. With this launch policy it can run asynchronously or not depending on the load on system. But we have no control over it.

If we do not specify an launch policy. Its behaviour will be similar to **std::launch::async | std::launch::deferred**.

We are going to use std::launch::async launch policy in this article.

We can pass any callback in std::async i.e.

* Function Pointer
* Function Object
* Lambda Function

Let’s understand the need of std::async by an example,

1. Need of std::async()

Suppose we have to fetch some data (string) from DB and some from files in file-system. Then I need to merge both the strings and print.

In a single thread we will do like this,

#include <iostream>

#include <string>

#include <chrono>

#include <thread>

using namespace std::chrono;

std::string fetchDataFromDB(std::string recvdData)

{

// Make sure that function takes 5 seconds to complete

std::this\_thread::sleep\_for(seconds(5));

//Do stuff like creating DB Connection and fetching Data

return "DB\_" + recvdData;

}

std::string fetchDataFromFile(std::string recvdData)

{

// Make sure that function takes 5 seconds to complete

std::this\_thread::sleep\_for(seconds(5));

//Do stuff like fetching Data File

return "File\_" + recvdData;

}

int main()

{

// Get Start Time

system\_clock::time\_point start = system\_clock::now();

//Fetch Data from DB

std::string dbData = fetchDataFromDB("Data");

//Fetch Data from File

std::string fileData = fetchDataFromFile("Data");

// Get End Time

auto end = system\_clock::now();

auto diff = duration\_cast < std::chrono::seconds > (end - start).count();

std::cout << "Total Time Taken = " << diff << " Seconds" << std::endl;

//Combine The Data

std::string data = dbData + " :: " + fileData;

//Printing the combined Data

std::cout << "Data = " << data << std::endl;

return 0;

}

**Output:**

Total Time Taken = 10 Seconds

Data = DB\_Data :: File\_Data

As both the functions fetchDataFromDB() & fetchDataFromFile() takes 5 seconds each and are running in a single thread so, total time consumed will be 10 seconds.

Now as fetching data from DB and file are independent of each other and also time consuming. So, we can run them in parallel.

One way to do is create a new thread pass a promise as an argument to thread function and fetch data from associated std::future object in calling thread.

The other easy way is using std::async.

1. Calling std::async with function pointer as callback

Now let’s modify the above code and call fetchDataFromDB() asynchronously using std::async() i.e.

#include <thread>

#include <future>

std::future<std::string> resultFromDB = std::async(std::launch::async, fetchDataFromDB, "Data");

// Do Some Stuff

//Fetch Data from DB

// Will block till data is available in future<std::string> object.

std::string dbData = resultFromDB.get();

std::async() does following things:

* It automatically creates a thread (Or picks from internal thread pool) and a promise object for us.
* Then passes the std::promise object to thread function and returns the associated std::future object.
* When our passed argument function exits then its value will be set in this promise object, so eventually return value will be available in std::future object.

Now change the above example and use std::async to read data from DB asyncronously i.e.

Checkout the compete example as follows,

#include <iostream>

#include <string>

#include <chrono>

#include <thread>

#include <future>

using namespace std::chrono;

std::string fetchDataFromDB(std::string recvdData)

{

// Make sure that function takes 5 seconds to complete

std::this\_thread::sleep\_for(seconds(5));

//Do stuff like creating DB Connection and fetching Data

return "DB\_" + recvdData;

}

std::string fetchDataFromFile(std::string recvdData)

{

// Make sure that function takes 5 seconds to complete

std::this\_thread::sleep\_for(seconds(5));

//Do stuff like fetching Data File

return "File\_" + recvdData;

}

int main()

{

// Get Start Time

system\_clock::time\_point start = system\_clock::now();

std::future<std::string> resultFromDB = std::async(std::launch::async, fetchDataFromDB, "Data");

//Fetch Data from File

std::string fileData = fetchDataFromFile("Data");

//Fetch Data from DB

// Will block till data is available in future<std::string> object.

std::string dbData = resultFromDB.get();

// Get End Time

auto end = system\_clock::now();

auto diff = duration\_cast < std::chrono::seconds > (end - start).count();

std::cout << "Total Time Taken = " << diff << " Seconds" << std::endl;

//Combine The Data

std::string data = dbData + " :: " + fileData;

//Printing the combined Data

std::cout << "Data = " << data << std::endl;

return 0;

}

Output:

Total Time Taken = 5 Seconds

Data = DB\_Data :: File\_Data

1. Calling std::async with Function Object as callback

#include <thread>

#include <future>

/\*

\* Function Object

\*/

struct DataFetcher

{

std::string operator()(std::string recvdData)

{

// Make sure that function takes 5 seconds to complete

std::this\_thread::sleep\_for (seconds(5));

//Do stuff like fetching Data File

return "File\_" + recvdData;

}

};

//Calling std::async with function object

std::future<std::string> fileResult = std::async(DataFetcher(), "Data");

1. Calling std::async with Lambda function as callback

#include <thread>

#include <future>

//Calling std::async with lambda function

std::future<std::string> resultFromDB = std::async([](std::string recvdData){

std::this\_thread::sleep\_for (seconds(5));

//Do stuff like creating DB Connection and fetching Data

return "DB\_" + recvdData;

}, "Data");

Bạn đang trong kỳ nghỉ và phải đi bằng máy bay, sau khi làm đủ mọi thủ tục cần thiết, chỉ còn việc ngồi đợi thông báo lên tàu bay. Trường hợp bị delay, sẽ mất một vài tiếng, trong thời gian này bạn có thể đọc sách, lướt net hay dùng café tại sân bay…

Tình trạng này có thể tái diễn cho lần sau khi bạn cần đi công tác hay du lịch.

1. std::future và std::unique\_future

Thư viện chuẩn C++ đã làm mẫu cho kiểu sự kiện thế này gọi là future.

Nếu một thread cần đợi duy nhất một kiểu sự kiện riêng biệt, thì nó nên có future cho sự kiện đó.

Thread có thể thăm dò future xem sự kiện có tồn tại (kiểm tra tàu bay), trong khi đang thực thi tác vụ khác (uống café) hoặc đang ngồi chơi chờ sự kiện.

Một future luôn có dữ liệu tương ứng với nó (thông báo cổng lên tàu bay), hoặc không có cũng được.

Khi sự kiện xảy ra (future đã sẳn sàng), future không thể reset. Có 2 loại template class future trong thư viện chuẩn:

* std::unique\_future<> : chỉ cho phép duy nhất một đối tượng future tương ứng với sự kiện
* std::shared\_future<> : cho phép nhiều đối tượng future tương ứng với cùng một sự kiện. Tất cả đối tượng xem như sẵng sàng cùng lúc, và tất cả thread có thể truy xuất dữ liệu tương ứng với sự kiện.

Hai loại template class này được lấy cảm hứng từ std::unique\_ptr và std::shared\_ptr.

Chuyên hóa std::unique\_future<void> và std::shared\_future<void> nên dùng khi không có dữ liệu tương ứng nào.

Listing 4.6 minh họa hai hành khách đang đợi tàu bay.

**Listing 4.6: Đợi sự kiện với std::share\_future**

#include <future>

void wait\_for\_flight1(flight\_number flight)

{

std::shared\_future<boarding\_information> boarding\_info = get\_boarding\_info(flight); //#1 Lấy future

board\_flight(boarding\_info.get());//#1 Lấy thông tin tương ứng

}

void wait\_for\_flight2(flight\_number flight)

{

std::shared\_future<boarding\_information> boarding\_info = get\_boarding\_info(flight); //#2 Lấy future

while(!boarding\_info.is\_ready()) //#4 kiểm tra chuyến bay đã sẵn sàng

{

eat\_in\_cafe(); //#3 khách còn dùng café và mua đồ trong khi chờ

buy\_duty\_free\_goods();

}

board\_flight(boarding\_info.get());

}

**Giải thích**

#1: Thread của hành khách đầu tiên đang chạy wait\_for\_flight1(), lấy std::shared\_future<boarding\_information> kèm theo thông tin tàu bay. Hành khách này chỉ ngồi đợi thông báo nên chỉ gọi phương thức get() để đợi future sẵn sàng trước khi trả về dữ liệu tương ứng.

#2: Thread hành khách thứ hai đang chạy wait\_for\_flight2(), khi lấy thông tin tàu bay từ std::shared\_future<boarding\_information>, khách hàng này còn dùng café và mua đồ ăn (#3), trong khi đợi thông báo chuyến bay calling is\_ready() tại (#4).

Đoạn code sử dụng std::shared\_future hơn là std::unique\_future, vì có nhiều hành khách (thread) đang cùng đợi chuyến bay.

Ngược lại, nếu chỉ có một thread đợi future – điều này khá phổ biến khi future này gồm nhiều tác vụ con cần hoàn tất – thì chỉ cần dùng std::unique\_future

std::unique\_future đơn giản hơn nên tốn ít chi phí hơn std::shared\_future. std::shared\_future chia sẻ (sao chép) dữ liệu tương ứng của future, còn std::unique\_future move dữ liệu (chuyển quyền sở hữu) sang đối tượng khác.

Khi dữ liệu quá lớn, việc sao chép trở nên kém hiệu quả, nhưng move thì lại rất dễ (tương tự như std::vector với hàng triệu dữ liệu bên trong), và điều quan trọng ở đây là dữ liệu không thể so chép, chỉ move mà thôi (như std::thread hay std::ifstream).

Trong Listing 4.7: future giữ phản hồi yêu cầu tải file từ thread chạy ngầm #1

Tác dụng của future cho phép thread này tiếp tục xử lí tin nhắn GUI đúng giờ #3.

Khi future đã sẵn sàng #2, dữ liệu file có thể tách khỏi future tại #4 và xử lí tại #5.

Vì đây là tác vụ con,

The use of the

future allows this thread to continue processing GUI messages in the mean time (#3). Once

the future is ready (#2), the file data can be extracted from the future (#4) and then

processed (#5). Because this is a subsidiary task, we know there's only the one thread

waiting, so std::unique\_future is appropriate. unique\_future<>::get() returns an

rvalue, so we can also avoid copying the data when we extract it from the future if

file\_contents has a move constructor.

**listing 4.7: Dùng std::unique\_future để chuyển một block dữ liệu**

void process\_remote\_file(std::string const& filename)

{

std::unique\_future<file\_contents> contents=server.download\_file\_in\_background(filename); //#1

while(!contents.is\_ready()) //#2

{

process\_gui\_events(); //#3

}

file\_contents file\_data(contents.get()); //#4

process\_file(file\_data); //#5

}

**Giải thích**

So, that's futures from the point of view of the waiting thread, but what about the thread

that triggers the event? How do we make a future ready? How do we store the associated

data? The C++ Standard Library provides the answers to these questions in the form of two

function templates: std::packaged\_task<> and std::promise<>.

1. std::packaged\_task

std::packaged\_task<> ties a future to the result of a function call: when the function

completes, the future is ready, and the associated data is the value returned by the function.

This is ideal for overall operations that can be subdivided into a set of self-contained tasks:

these tasks can be written as functions, packaged up so each task is associated with a future

using std::packaged\_task, and then executed on separate threads. The driving function

can then wait for the futures before processing the results of these subtasks.

1. std::promise

std::promise<> is a slightly lower-level tool that provides explicit functions for setting the

data associated with a future, and making the future ready. This is good for the case where

the data may come from multiple sources: in a parallel search algorithm, for example, the

search space may be divided between threads, but only one result is required. A

std::unique\_future/std::promise pair can be used to manage the return value, and the

first thread to find a match can set the future value through the std::promise instance,

and instruct the other threads to stop searching.

1. Kết hợp Task với Future

Let's look back at the example in listing 4.7. The details of creating the

std::unique\_future that will contain the contents of the downloaded file are wrapped up

in the server member function download\_file\_in\_background(). If it was not a

background task but a foreground task, this function would probably be something along the

lines of the listing 4.8: open a connection, request the file, receive chunks of data until done,

and return the result to the caller.

**Listing 4.8: Downloading a file from a remote server**

file\_contents download\_file(std::string const& filename)

{

Connection con(remote\_server);

con.send\_file\_request(filename);

file\_contents contents;

while(!con.eof())

{

std::vector<unsigned char> chunk = con.receive\_data\_chunk();

contents.append(chunk);

}

return contents;

}

If we already have a function such as download\_file(), or we're willing to write one,

writing download\_file\_in\_background() becomes remarkably easy to write using

std::packaged\_task: wrap download\_file() in a std::packaged\_task (#1), using

std::bind() to pass in the name of the file to download, get the associated future (#2),

start a thread to run the packaged task and finally transfer the future back to the caller.

std::unique\_future<file\_contents> download\_file\_in\_background(std::string const& filename)

{

std::packaged\_task<file\_contents()> //#1

task(std::bind(download\_file,filename)));

std::unique\_future<file\_contents> //#2

res(task.get\_future());

std::thread(std::move(task)); //#3

return res; //#4

}

Cueballs in code and preceding and following text

Note the use of std::move() when starting the thread (#3): std::packaged\_task cannot

be copied, so the ownership of the task must be explicitly transferred into the std::thread

object. Even though std::unique\_future cannot be copied either, since the value returned

(#4) is a local variable (#2), the ownership of the result is implicitly transferred into the

function return value.

What about those tasks that can't be expressed as a simple function call, or those tasks

where the result may come from more than one place? As already mentioned in the

introduction to section 4.2, std::promise provides a better fit for these cases than

std::packaged\_task.

1. Sử dụng std::promise