TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI

VIỆN ĐÀO TẠO SAU ĐẠI HỌC

=====o0o=====

TIỂU LUẬN

ĐỀ TÀI: TÌM HIỂU CÁC KỸ THUẬT ĐIỀU KHIỂN TƯƠNG TRANH TRONG LẬP TRÌNH SONG SONG CHIA SẺ BỘ NHỚ CHUNG

Giáo viên giảng dạy:	TS. Nguyễn Hữu Đức
Sinh viên thực hiện:	Vũ Thị Thùy Như
	Phạm Thị Nhung
	Nguyễn Thị Thi
Lớp:	CH2012B

Hà Nội, tháng 12/2012

Mục lục

MỞ ĐẦU

Trong môi trường lập trình song song các câu lệnh của chương trình có thể thực hiện đan xen lẫn nhau, ở cùng một thời điểm có thể có nhiều hơn một lệnh được thực hiện, nghĩa là mỗi chương trình sẽ tự chủ thực hiện các tiến trình của mình. Các chương trình phải tương tác với nhau và việc thực hiện của chúng ảnh hưởng tới nhịp độ thực hiện của nhau.

Trong lập trình song song, người lập trình không chỉ viết chương trình, dữ liệu như trong môi trường tuần tự mà còn phải sử dụng các công cụ để đồng bộ hoá(synchronize) và điều khiển sự tương tác giữa các tiến trình. Người lập trình cần tạo ra và lập lịch cho các tiến trình, nghĩa là sự thực hiện chương trình có thể nhìn thấy được bởi người lập trình.

Các tình huống thường gặp:

- Tại một thời điểm có một số tiến trình muốn truy cập vào một tài nguyên chung hoặc cập nhật vào một biến chia sẻ. Mà những tài nguyên đó chỉ cho phép một tiến trình truy cập tại mỗi thời điểm.

- Khi một tiến trình được quyền truy cập vào tài nguyên chung thì nó sử dụng tài nguyên đó nhưng không được ngăn cản hoạt động của những tiến trình khác.

- Khi một số tiến trình cùng kết hợp để thực hiện một số phép toán trên cơ sở quan sát hành động của nhauthì người lập trình phải lập lịch cho những tiến trình đó.

Các mô hình về lập trình song song

- Lập trình chia sẻ bộ nhớ chung
- Lập trình song song dựa vào các tiến trình
- Lập trình song song dựa vào các luồng
- Lập trình theo mô hình truyền thông điệp
- Lập trình trên cụm máy tính PVM

Trong khuôn khổ của tiểu luận sẽ tìm hiểu về các kỹ thuật điều khiển tương tranh trong lập trình song song chia sẻ bộ nhớ dùng chung theo các phần sau:

Phần 1: Lập trình chia sẻ bộ nhớ chung

Phần 2: Lập trình song song dựa vào các luồng

Phần 3: Các kỹ thuật giải quyết tương tranh

Sau đây chúng ta sẽ đi tìm hiểu từng phần

PHẦN 1: LẬP TRÌNH CHIA SẺ BỘ NHỚ CHUNG

Cả tiến trình và luồng đều hữu ích cho lập trình song song và truy cập đồng thời:

- Ẩn độ trễ
- Tối đa hóa việc sử dụng CPU
- Xử lý nhiều sự kiện không đồng bộ

1.1 Tiến trình(Process)

- Tiến trình: thực hiện nội dung (PC, thanh ghi) + không gian địa chỉ, tập tin
- Trong môi trường lập trình chia sẻ bộ nhớ có hai ràng buộc quan trọng:
    - Một tiến trình có thể chờ một khoảng thời gian bất kỳ giữa hai lệnh cần thực hiện. Giả sử bộ xử lý P thực hiện một chương trình có một 100 lệnh, bộ xử lý Q thực hiện chương trình có 10 lệnh và cùng bắt đầu thực hiện đồng thời. Thậm chí, tất cả các lệnh có tốc độ thực hiện như nhau cũng không thể nói rằng Q sẽ kết thúc trước P.
    - Không thể xem các lệnh thực hiện là nguyên tố ở mức các ngôn ngữ lập trình. Ví dụ, một lệnh đơn giản như: a = a + 1 sẽ là một dãy bốn lệnh trong ngôn ngữ máy. Mà ta cũng biết rằng các tiến trình và hệ điều hành chỉ nhận biết được các câu lệnh của ngôn ngữ máy.
- Trong lập trình bộ nhớ chia sẻ:
    - Các tác vụ (tasks) sẽ đọc/ghi dữ liệu từ bộ nhớ chia sẻ qua không gian địa chỉ bộ nhớ chung (common address space)
    - Có những cơ chế khác nhau như (locks/semaphores) để điều khiển việc truy nhập đến bộ nhớ chia sẻ
    - Người lập trình không cần mô tả việc truyền thông dữ liệu, do đó việc viết chương trình sẽ đơn giản.
    - Khó quản lý dữ liệu vì không can thiệp trực tiếp quá trình truyền dữ liệu.

1.2 Phương pháp lập trình song song chia sẻ bộ nhớ

- Khi muốn sử dụng bộ nhớ chung, người lập trình cần phải xin cấp phát bộ nhớ và sau khi sử dụng xong phải giải phóng chúng.
- Nếu có một tiến trình truy cập vào một vùng nhớ với ý định cập nhật thì nó phải được đảm bảo rằng không một tiến trình nào khác đọc dữ liệu ở vùng đó cho đến khi việc cập nhật đó kết thúc.

- Để giải quyết được vấn đề trên thì phải có cơ chế đảm bảo rằng, tại mỗi thời điểm các khối lệnh của chương trình được thực thi chỉ bởi một tiến trình.
- Nếu có một tiến trình bắt đầu vào thực hiện một khối lệnh thì những tiến trình khác không được vào khối lệnh đó.
- Khi một tiến trình vào một vùng lệnh nào đó thì nó sẽ gài khoá (lock).
- Ngược lại, khi ra khỏi vùng đó thì thực hiện cơ chế mở khoá (unlock) để cho tiến trình khác có nhu cầu sử dụng.

1.3 Ví dụ minh họa

- Các câu lệnh để thực hiện các yêu cầu trên:
    - init_lock(Id): Khởi động bộ khoá vùng nhớ chia sẻ, trong đó Id là tên của vùng nhớ sử dụng chung.
    - lock(Id): khoá lại vùng nhớ Id. Nếu một tiến trình đã khoá một vùng nhớ chung thì những tiến trình khác muốn truy cập vào đó sẽ phải chờ.
    - unlock(Id): mở khoá vùng đã bị khoá và trả lại cho tiến trình khác.
- Sử dụng cơ chế gài khoá để viết một đoạn chương trình thể hiện chia sẻ bộ nhớ dùng chung theo hướng lập trình song song

```
main(){
        int *lock1,id, sid1, sid2, *i, j;
        lock1 = (int*)shared(sizeof(int), &sid1)
        init_lock(lock1);
        i = (int*)shared(sizeof(int), &sid2);
        *i = 100; j = 100;
        printf(“Before fork: %d, %d\n”, *i, j);
        id = create_process(2);
        lock(lock1);
        *i = id; j = id * 2;
        printf(“After fork: &d, %d\n”, *i, j);
        unlock(lock1);
        join_process(3, id);
        printf(“After join: &d, %d\n”, *i, j);
        free_shm(sid1); free_shm(sid2);
}
```

PHẦN 2: LẬP TRÌNH BỘ NHỚ CHIA SẺ DỰA VÀO LUỒNG

2.1 *Định nghĩa luồng (thread)*

- Mỗi tiến trình bao gồm một hoặc nhiều luồng. Các luồng có thể xem như các tập con của một tiến trình.
- Các luồng của một tiến trình có thể chia sẻ với nhau về không gian địa chỉ, các đoạn dữ liệu và môi trường xử lý, đồng thời cũng có vùng dữ liệu riêng để thao tác.
- Các tiến trình và các luồng trong hệ thống song song cần phải được đồng bộ, nhưng việc đồng bộ các luồng được thực hiện hiệu quả hơn đối với các tiến trình.
- Đồng bộ các tiến trình đòi hỏi tốn thời gian hoạt động của hệ thống, trong khi đối với các luồng thì việc đồng bộ chủ yếu tập trung vào sự truy cập các biến chung của chương trình.
- Trong lập trình song song Thread, chia sẻ tất cả mọi thứ ngoại trừ: ngăn xếp, thanh ghi & dữ liệu thread cụ thể.
- Khi đó nhiều luồng / tiến trình truy cập chia sẻ tài nguyên đồng thời sẽ dẫn đến tương tranh

Nó được ví như vấn đề mua quá nhiều sữa

Vì vậy cần phải đồng bộ hóa để đảm bảo 2 mục đích:

- Đảm bảo an toàn cho việc cập nhật dữ liệu chia sẻ: tránh điều kiện race
- Phối hợp hành động của các luồng (Threads)

+ Song song tính toán

+ Thông báo sự kiện

2.2 Lập trình luồng trong java

- Trong Java, luồng là một loại "đối tượng hệ thống":
    - Các phương thức trên đối tượng luồng
    - Mỗi đối tượng là một đơn vị song song có thể được thực hiện một cách độc lập
- Java là ngôn ngữ lập trình hướng đối tượng hỗ trợ đa luồng, tiện lợi cho các ứng dụng web.
- Trong mô hình hướng đối tượng, tiến trình và thủ tục là thứ yếu, mọi chức năng của chương trình được xác định thông qua các đối tượng.
- Cũng giống như tiến trình, luồng được tạo lập, sau đó thực hiện một số công việc và kết thúc hoạt động khi không còn vai trò sử dụng.
- Hai hướng tiếp cận Threads trong JAVA
    - Xây dựng lớp con của lớp Thread.
    - Cài đặt giao diện Runnable.
- Các trạng thái của Thread
    - new: khi một luồng mới được tạo ra với toán tử new().
    - ready: khi chúng ta gọi phương thức start() để bắt đầu của một luồng.
    - blocked: từ trạng thái runnable chuyển sang trạng thái "bị chặn" khi gọi một trong các phương thức: sleep(), suspend(), wait(),hay bị chặn lại ở Input/output.
    - dead: luồng chuyển sang trạng thái "chết" khi nó kết thúc hoạt động bình thường, hoặc gặp phải ngoại lệ không thực hiện tiếp được.
    - Phương thức yield() sẽ ngưng luồng hiện hành để cho một luồng khác có cùng độ ưu tiên chạy
    - wait(): giống yield(), nhưng yêu cầu một luồng khác phải đánh thức nó while (!condition) wait();
    - notify(): Luồng nào có thể ảnh hưởng đến condition sẽ gọi notify() để phục hồi luồng đang chờ
- Điều gì xảy ra khi có Luồng mới:
    - Luồng chính vẫn tiếp tục

- Luồng mới thi hành phương thức run() và “kết thúc” khi phương thức kết thúc.
- Nếu có bất kỳ luồng nào gọi System.exit(0) thì nó sẽ “giải phóng” tất cả mọi luồng.
- Có thể xem run() như là phương thức chính của riêng mỗi luồng

- Chấm dứt một Luồng
  - Luồng kết thúc khi phương thức run() kết thúc
  - Tuy nhiên, điều gì xảy ra khi luồng đang “nghỉ” (sleeping) hoặc đang bị “khóa” (blocked)?
  - Đây là lúc mà vai trò của phương thức interrupt() được thể hiện. Khi interrupt() được gọi trên một Luồng đang bị “khóa”, luồng sẽ bị chấm dứt.
- Các vấn đề khác về Luồng
  - Chia sẻ và đồng bộ hóa
  - Lập lịch

PHẦN 3: CÁC KỸ THUẬT GIẢI QUYẾT TƯƠNG TRANH

Như đã trình bày trong phần 2 để giải quyết tương tranh khi sử dụng Thread phải dùng kỹ thuật đồng bộ hóa và lập lịch. Với mỗi mục đích của đồng bộ hóa chúng ta sử dụng kỹ thuật tương ứng.

3.1 Đảm bảo an toàn dữ liệu

Để đảm bảo an toàn cho việc truy cập dữ liệu chia sẻ. Ta hiểu rằng dữ liệu chia sẻ chỉ an toàn khi:

- Tất cả các truy cập không có hiệu lực về tài nguyên
  Ví dụ : đọc một biến, hoặc
- Tất cả truy cập không thay đổi giá trị
  Ví dụ: a = abs(x), a = highbit (a)
- Chỉ có một truy cập tại một thời điểm: loại trừ lẫn nhau

Để ngăn chặn nhiều hơn một luồng truy cập vào vùng giới hạn ta sử dụng kỹ thuật khóa: khóa, cập nhật, mở khóa

```
Lock (&1);
Update data;/*Critical section*/
Unlock  (&1)
```

Ví dụ: để giải quyết bài toán mua sữa quá nhiều ta dùng khóa

Luồng A	**Luồng B**
lock (&1)	lock (&1)
if (no milk)	if (no milk)
buy milk	buy milk
unlock (&1)	unlock (&1)

Nhưng để giải quyết khóa cũng biến, cập nhật đồng thời bởi nhiều luồng và để trả lời câu hỏi khi nào khóa chúng ta sử dụng phần cứng cấp độ nguyên tử hoạt động với 2 giải thuật cơ bản: Test and Set và Compare-and –swap

a. Giải thuật Test and Set

- Thực tế có rất nhiều tình huống xấu: sự thực hiện không đúng theo trật tự, truy nhập lại bộ nhớ với những trình biên dịch tối ưu hóa

- Bộ xử lý Test and Set là đặc tính của phần cứng. được sử dụng bởi hệ điều hành
    - Viết vào địa chỉ bộ nhớ và trả lại giá trị như một vi lệnh
    - Ý tưởng: quá trình ghi 1 vào các ô nhớ sẽ kết thúc khi các giá trị cũ là 0
        - Giữa việc viết và kiểm tra, không có phép toán nào có thể định nghĩa giá trị
    - Có thể thực hiện bởi việc hoán đổi **atomic** (hoặc với các thao tác đọc – sửa – ghi) của hoạt động phần cứng

```
Int testandset (int&v) {
        Int old = v;
        v= 1;
        return old;
  }
```

Pseudo-code: red = atomic

Ảnh hưởng của testandset (value)

Khi:

Value =0?

(mở khóa)

Value = 1?

(khóa)

b. Compare-and –swap

Để trả lời câu hỏi các khóa sẽ được sử dụng như thế nào ta có 3 kỹ thuật khóa:

- Blocking locks
- Spin locks
- Hybrids

c. Blocking locks

Hoãn các luồng ngay lập tức

Cho phép lập lịch thực hiện một luồng

Giảm thiểu thời gian chờ đợi

Nhưng: luôn luôn gây ra bối cảnh chuyển đổi

Thuật toán:

```
Void blokinglock (lock& 1) {
        While (tesandset (1.v) ==1) {
        Sched_yield ();
        }
 }
```

d. Spin locks

Thay vì ngăn chặn, vòng lặp thực hiện đến khi một khóa phát hành

Thuật toán:

```
Void spinlock (Lock & 1) {
                While (testandset (1.v) ==1) {
                ;
                }
 }
Void spinloc2 (Lock &1) {
        While (testandset (1.v) ==1){
                While (1.v ==1);
        }
   }
```

e. Other variants

Quay một thời gian, sau đó tiến hành

Thời gian quay cố định

Khóa truy vấn

Đảm bảo sự công bằng và khả năng mở rộng

Khi thực hiện kỹ thuật khóa thì các khóa có thể thực thi loại trừ lẫn nhau và nó sẽ phát sinh những lỗi phổ biến:

- Không thể mở khóa (Failure to unlock)
- Đôi khóa (Double locking)
- Bế tắc (Deadlock)
- Ưu tiên đảo ngược

Để xử lý các lỗi này ta tìm hiểu hiểu cơ chế gây lỗi và kỹ thuật giải quyết các lỗi:

❖ ***Không thể mở khóa***

```
pthread_mutex_t 1;
void square (void){
        pthread_mutex_lock (&1);
        // acquires lock
        //do stuff
        If (x==0) {
        return;
        } else {
                x= x*x;
                }
        pthread_mutex_unlock (&1);
        }
```

Điều gì sẽ xảy ra khi chúng ta gọi square () hai lần khi x= =0?

Vùng khóa với RAI

Tiếp nhận dự liệu vào và xuất dữ liệu ra

C++: Tài nguyên tiếp nhận là khởi tạo

```
class Guard {
        public:
        Guard (pthread_mutex_t&1)
        :_lock (1)
        {pthread_mutex_lock (&_lock);}
        ~Guard (void) {
                Pthread_mutex_unlock (&_lock);
                }
        private:
        pthread_mutex_t_lock;
};
```

Thuật toán ngăn chặn không thể mở khóa

```
pthread_mutex_t 1;
void square (void) {
        Guard lockIt (&1);
        // acquires lock
```

```
        // do stuff
        If (x==0){
                return; // releases lock
                } else {
                        x = x*x;
        }
       //releases lock
  }
```

❖ ***Hai khóa (Double locking)***

```
pthread_mutex_lock (&1)
//do stuff
//now unlock (or not …)
pthread_mutex_lock (&1);
```

để giải quyết vấn đề ***Double locking*** ta sử dụng khóa đệ quy

Giải pháp: khóa đệ quy

+ if mở khóa:

threadID = pthread_self ()

count = 1

+ cùng một luồng khóa ➔ increment count

Nếu không, khối

+ mở khóa ➔ decrement count

Thực sự mở khóa khi count ==0

❖ ***Tránh bế tắc (deadlock)***

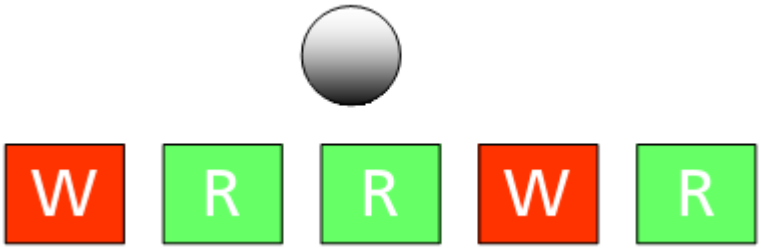
- Chu kỳ trong đồ thị khóa = bế tắc
- Giải pháp tiêu chuẩn:

  Quy tắc thứ tự co các khóa

  + Tiếp nhận theo thứ tự tăng dần

  + Xuất theo thứ tự giảm dần
- Đảm bảo deadlock-freedom, nhưng không phải luôn luôn dễ dàng để làm
- Truy cập đồng thời tăng dần

Một đối tượng, chia sẻ giữa các luồng



- Mỗi luồng sẽ đọc hoặc ghi

  + đọc – chỉ đọc dữ liệu mà không sửa

  + Ghi – đọc và sửa dữ liệu

❖ ***Giải pháp một khóa - Single lock***

thread A	**thread B**	**thread C**
lock (&1)	lock (&1)	lock (&1)
read data	modify data	read data
unlock (&1)	unlock (&1)	unlock (&1)
thread D	**thread E**	**thread C**
lock (&1)	lock (&1)	lock (&1)
read data	read data	modify data
unlock (&1)	unlock (&1)	unlock (&1)

Một khóa thì an toàn, nhưng giới hạn truy cập đồng thời chỉ một luồng thực hiện tại một thời điểm.

Nhận thức: An toàn để đọc đồng thời, phải đảm bảo loại trừ lẫn nhau để viết

- Đọc/ghi

thread A	**thread B**	**thread C**
lock (&rw)	lock (&rw)	lock (&rw)
read data	modify data	read data
unlock (&rw)	unlock (&rw)	unlock (&rw)
thread D	**thread E**	**thread C**
lock (&rw)	lock (&rw)	lock (&rw)
read data	read data	modify data
unlock (&rw)	unlock (&rw)	unlock (&rw)

- Các vấn đề - khóa đọc/Ghi
  Khi đọc/ghi xếp hàng thì ai được khóa?
  + Ưu tiên đọc : cải tiến đồng thời, có thể ghi chết đói
  + Ưu tiên ghi
  + Luân phiên : Tránh đói

3. 2 Đảm bảo sự phối hợp

a. Kỹ thuật đèn báo:

Kỹ thuật đèn báo là gì?

Một tín hiệu hình ảnh bộ máy với cờ, đèn, hoặc cánh tay cơ học di chuyển, như là một sử dụng trên đường sắt. Điều chỉnh lưu lượng truy cập tại phần quan trọng

Để sử dụng đèn báo trong CS: không truy cập số nguyên âm với nguyên tử tăng & giảm đi. Khối thay vì đi tiêu cực

Thuật toán

- P (sem), a.k.a.wait = decrement counter
  + If sem = 0, block until greater than zero
  + P = “prolagen”
    (proberen te verlagen, “try to decrease”)
- V (sem), a.k.a.signal
  = increment counter
  + Wake 1 waiting process
  + V = “verhogen”
    “Increase

Bằng cách khởi tạo semaphore 0, Luồng có thể chờ đợi một sự kiện xảy ra

Luồng A	**Luồng B**
//waiting for thread b	//do stuff, then
sem.wait ();	//wake up A
//do stuff ….	Sem.signal ();

Ta sử dụng kỹ thuật đếm đèn báo để kiểm soát các nguồn tài nguyên

Ví dụ như cho phép luồng sử dụng tối đa 5 tập tin cùng một lúc

Luồng A	**Luồng B**
sem.wait ();	sem.wait ();
// use a file	// use a file
sem.signal ();	sem.signal ();

Khi sử dụng kỹ thuật đèn báo sẽ nảy sinh vấn đề chờ

- Giả sử chúng ta có một hàng đợi luồng an toàn
  + Insert (item), remove ()

- Tùy chọn cho loại bỏ khi hàng đợi rỗng
  + Trả lại giá trị lỗi (VD: rỗng)
  + Ném và ngoại lệ
  + Chờ cho một cái gì đó để xuất hiện trong hàng đợi
- Chờ = sleep()
  + Nhưng ngủ khi giữ khóa và đi ngủ không bao giờ thức dậy

Khi đó ta sẽ phải dùng đến kỹ thuật điều kiện biến:

Chờ cho 1 sự kiện, nguyên tử lấy khóa

+ wait (lock &1)

Nếu hàng đợi rỗng, chờ đợi

Nguyên tử phát hành khóa, đi vào giấc ngủ

Nhập lại khóa khi đánh thức

+ Notify ()

Chèn mục trong hàng đợi

Tỉnh dậy một luồng chờ đợi, nếu có

+ NotifyAll ()

Tỉnh dậy tất cả các luồng chờ đợi

Kết luận: Nội dung trên đã trình bày một phần các phương pháp xử lý song song nhờ vào việc lập trình. Nhằm tăng được khả năng tính toán của các hệ thống máy tính để giải được những bài toán đáp ứng yêu cầu thực tế thì không còn cách nào khác là phải khai thác được những khả năng xử lý song song của hệ thống máy tính hiện đại.

Mục đích của xử lý song song là tận dụng các khả năng tính toán của các hệ đa bộ xử lý, của các máy tính song song để thực hiện những tính toán nhanh hơn trên cơ sở sử dụng nhiều bộ xử lý đồng thời. Cùng với tốc độ xử lý nhanh hơn,việc xử lý song song và phân tán sẽ giải quyết được những bài toán lớn hơn, phức tạp hơn của thực tế.

Tham khảo

http://people.cs.umass.edu/~emery/classes/cmpsci691w-spring2006/