LẬP TRÌNH ĐỒNG THỜI & PHÂN TÁN

BÀI 6:
BÀI TOÁN TRUY CẬP TÀI NGUYÊN CHIA SẺ

Giảng viên: Lê Nguyễn Tuấn Thành
Email: thanhlnt@tlu.edu.vn

NỘI DUNG

- Bài toán loại trừ lẫn nhau trong hệ thống phân tán
- Những thuật toán dựa trên *timestamp*
- Những thuật toán dựa trên *token*

3 thuộc tính:
- an toàn
- tiến triển/ sự sống : không bị lặp vô hạn
- công bằng: tiến trình nào yêu cầu trước phải truy cập trước

Bài toán loại trừ lẫn nhau trong hệ thống phân tán

- Xét hệ thống phân tán bao gồm *một số lượng cố định tiến trình* và *một **tài nguyên** chia sẻ*
  - Việc truy cập đến tài nguyên chia sẻ được coi là *khu vực quan trọng CS*
- **Yêu cầu**: Đưa ra thuật toán để phối hợp truy cập tới tài nguyên chia sẻ thỏa mãn 3 thuộc tính sau:
  1. **Safety**: hai tiến trình không có quyền truy cập đồng thời vào CS
  2. **Liveness**: bất kỳ yêu cầu nào tới CS cuối cùng phải được cấp quyền
  3. **Fairness**: những yêu cầu khác nhau phải được cấp quyền đi vào CS theo thứ tự mà chúng được tạo ra
- Giả sử rằng không có lỗi trong hệ thống phân tán, các bộ xử lý và liên kết giao tiếp là tin cậy

Giao diện Xử lý thông điệp và Khoá

```
import java.io.*;
public interface MsgHandler {
    public void handleMsg(Msg m, int srcId, String tag);
    public Msg receiveMsg(int fromId) throws IOException;
}

public interface Lock extends MsgHandler {
    public void requestCS(); //may block
    public void releaseCS();
}
```

5 Những thuật toán dựa trên timestamp

Thuật toán mutex của Lamport (1)

- Trong thuật toán này, mỗi tiến trình sẽ lưu giữ:
  1. Một đồng hồ vector V (dùng để lưu dấu thời gian)
  2. Một hàng đợi Q (dùng để lưu các yêu cầu đi vào CS của các tiến trình trong hệ thống phân tán)
- Thuật toán này đảm bảo: các tiến trình đi vào CS theo thứ tự dấu thời gian của yêu cầu ở phía tiến trình gửi
  - Chứ không phải thứ tự nhận được của yêu cầu bên phía tiến trình nhận !
- Giả sử các thông điệp truyền đi theo thứ tự FIFO

Thuật toán mutex của Lamport (2)

- Nếu hai yêu cầu có cùng một dấu thời gian, thì yêu cầu của tiến trình có số hiệu nhỏ hơn được coi là nhỏ hơn
- Một cách chính thức, P_i có thể đi vào CS nếu:

$$\forall j : j \neq i : (q[i], i) < (v[j], j) \ \wedge \ (q[i], i) < (q[j], j)$$

  - **q[i], q[j]**: dấu thời gian của yêu cầu đi vào CS của hai tiến trình P_i và P_j
  - **v[j]**: dấu thời gian của thông điệp xác nhận từ tiến tình P_j được ghi nhận ở tiến trình P_i

Các bước thực hiện (1)

1. Khi tiến trình P_i muốn đi vào CS
   - P_i gửi thông điệp *request* có gắn dấu thời gian tới **tất cả tiến trình khác**
   - Đồng thời, P_i thêm yêu cầu có gắn dấu thời gian này vào trong hàng đợi của nó
2. Khi một tiến trình P_k nhận được thông điệp *request* từ tiến trình P_i
   - P_k lưu yêu cầu này và dấu thời gian của yêu cầu trong hàng đợi của nó
   - P_k gửi ngược lại thông điệp *ack* (xác nhận) có gắn dấu thời gian cho P_i

Các bước thực hiện (2)

3. Một tiến trình P_j nhận thấy nó có thể đi vào CS khi và chỉ khi thoả mãn các điều kiện sau:
   - ✔ P_j có một yêu cầu trong hàng đợi của nó với dấu thời gian *t* nhỏ hơn tất cả các yêu cầu khác đang trong hàng đợi của nó
   - ✔ P_j đã nhận thông điệp *ack* (xác nhận) từ tất cả tiến trình khác với dấu thời gian lớn hơn *t*
4. Để giải phóng CS, tiến trình P_j gửi một thông điệp *release* tới tất cả tiến trình khác
   - Khi một tiến trình P_m nhận được thông điệp *release*, P_m xoá yêu cầu tương ứng của P_j khỏi hàng đợi của nó

```
public class LamportMutex extends Process implements Lock {
    public synchronized void requestCS() {
        v.tick();
        q[myId] = v.getValue(myId);
        broadcastMsg("request", q[myId]);
        while (!okayCS())  myWait();
    }
    public synchronized void releaseCS() {
        q[myId] = Symbols.Infinity;
        broadcastMsg("release", v.getValue(myId));
    }
    boolean okayCS() {
        for (int j = 0; j < N; j++){
//REQ//    if(isGreater(q[myId], myId, q[j], j))          return false;
//ACK//    if(isGreater(q[myId], myId, v.getValue(j), j))return false;
        }
        return true;
    }
    public synchronized void handleMsg(Msg m, int src, String tag) {
        int timeStamp = m.getMessageInt();
        v.receiveAction(src, timeStamp);
        if (tag.equals("request")) {
            q[src] = timeStamp;  sendMsg(src,"ack", v.getValue(myId));
        } else if (tag.equals("release")) q[src] = Symbols.Infinity;
     else if (tag.equals("ack")) v[src] = timeStamp;
        notify(); // okayCS() may be true now
    }
}
```

Đánh giá thuật toán mutex của Lamport

Sử dụng **3*(N-1)** thông điệp cho mỗi lần yêu cầu CS

- N - 1 thông điệp *request*
- N - 1 thông điệp *ack* (xác nhận)
- N - 1 thông điệp *release*

Thuật toán của Ricart và Agrawala

Biến thể của thuật toán Lamport

- Sử dụng đồng hồ logic **C** và một hàng đợi **pendingQ**
- Kết hợp các chức năng của các thông điệp *ack* (xác nhận) và thông điệp *release* thành thông điệp *okay*
- Trong thuật toán này, tiến trình P_k không phải lúc nào cũng gửi thông điệp *okay* ngược lại khi nhận được một thông điệp *request* từ tiến trình P_i
  - Nó có thể trì hoãn xác nhận sau một khoảng thời gian
- Thuật toán chỉ sử dụng **2*(N-1)** thông điệp cho mỗi lần yêu cầu CS
  - Thay vì 3*(N-1) thông điệp như thuật toán của Lamport

Các bước thực hiện (1)

1. Khi tiến trình P_i muốn yêu cầu CS (để sử dụng tài nguyên chia sẻ)
   - P_i gửi một thông điệp *request* gắn dấu thời gian tới tất cả tiến trình khác
2. Khi tiến trình P_k nhận một thông điệp *request* từ tiến trình gửi P_i
   - P_k gửi một thông điệp *okay* nếu:
     - P_k không quan tâm đến việc vào CS, hoặc
     - Yêu cầu CS của P_k có dấu thời gian lớn hơn so với P_i
   - Nếu không, yêu cầu của tiến trình gửi P_i sẽ được lưu trong hàng đợi của P_k

Các bước thực hiện (2)

3. Một tiến trình P_j được đi vào CS khi:
   - ✔ P_j đã yêu cầu tài nguyên chia sẻ bằng cách gửi thông điệp *request* tới tất cả tiến trình khác, và
   - ✔ P_j đã nhận được N-1 thông điệp *okay* từ N-1 tiến trình khác xác nhận cho thông điệp *request* của nó
4. Khi tiến trình P_j giải phóng tài nguyên
   - P_j gửi thông điệp *okay* cho các tiến trình đang trong hàng đợi của P_j

```
public class RAMutex extends Process implements Lock {
    public synchronized void requestCS() {
        c.tick();
        myts = c.getValue();
        broadcastMsg("request", myts);
        numOkay = 0;
        while (numOkay < N-1)    myWait();
    }
    public synchronized void releaseCS() {
        myts = Symbols.Infinity;
        while (!pendingQ.isEmpty()) {
            int pid = pendingQ.removeHead();
            sendMsg(pid, "okay", c.getValue());
        }
    }
    public synchronized void handleMsg(Msg m, int src, String tag) {
        int timeStamp = m.getMessageInt();
        c.receiveAction(src, timeStamp);
        if (tag.equals("request")) {
            if ((myts == Symbols.Infinity ) || (timeStamp < myts)
               ||((timeStamp == myts)&&(src < myId)))//not interested in CS
                sendMsg(src, "okay", c.getValue());
            else    pendingQ.add(src);
        } else if (tag.equals("okay")) {
            numOkay++;
            if (numOkay == N - 1)  notify(); // okayCS() may be true now
}   }   }
```

16 Những thuật toán dựa trên token

Thuật toán dựa trên Token

- Sử dụng một tài nguyên phụ, *token*, cho những hệ thống phân tán với tài nguyên chia sẻ
- Nhiệm vụ: tạo, lưu giữ và luân chuyển yêu cầu token giữa các tiến trình trong hệ thống phân tán

Thuật toán mutex tập trung

- Thuật toán ít tốn kém nhất cho bài toán loại trừ lẫn nhau, dựa trên hàng đợi
  - Thuật toán chỉ thoả mãn hai thuộc tính safety và liveness !
- Một trong số các tiến trình sẽ đóng vai trò là *Người lãnh đạo* (*Leader*), hoặc *Người điều phối* (*Coordinator*) cho việc đi vào CS
- Biến *haveToken* sẽ có giá trị là *True* cho tiến trình có quyền truy cập tới CS
  - Lúc đầu chỉ có *haveToken* của *Leader* là True
  - *haveToken* của tất cả tiến trình khác là False
  - Trong một thời điểm, chỉ có 1 tiến trình có giá trị *haveToken* là True

Các bước thực hiện

1. Khi một tiến trình P_i muốn đi vào CS, nó sẽ gửi thông điệp *request* đến tiến trình *Leader*
2. Khi nhận được các thông điệp *request*, tiến trình *Leader* đặt những *request* này vào hàng đợi *pendingQ* của nó
3. *Leader* cấp quyền cho tiến trình P_k ở đầu hàng đợi bằng cách gửi thông điệp *okay* cho P_k
4. Khi tiến trình P_k hoàn thành công việc trong CS của nó, P_k gửi thông điệp *release* tới *Leader*
5. Khi nhận được thông điệp *release*, *Leader* gửi thông điệp *okay* tới tiến trình tiếp theo trong hàng đợi *pendingQ*, nếu hàng đợi không rỗng
   - Nếu không, *Leader* đặt giá trị *haveToken* của nó thành True

```
public class CentMutex extends Process implements Lock {
    public synchronized void requestCS() {
        sendMsg(leader, "request");
        while (!haveToken) myWait();
    }
    public synchronized void releaseCS() {
        sendMsg(leader, "release");
        haveToken = false;
    }
    public synchronized void handleMsg(Msg m, int src, String tag) {
        if (tag.equals("request")) {
            if (haveToken){
                sendMsg(src, "okay");
                haveToken = false;
            }
            else  pendingQ.add(src);
        } else if (tag.equals("release")) {
            if (!pendingQ.isEmpty()) {
                int pid = pendingQ.removeHead();
                sendMsg(pid, "okay");
            } else haveToken = true;
        } else if (tag.equals("okay")) {
            haveToken = true;
         notify();
        }
    }
}
```

Đánh giá thuật toán mutex tập trung

- Thuật toán mutex tập trung không thoả mãn thuộc tính công bằng!
- Các yêu cầu NÊN được cấp quyền đi vào CS theo thứ tự mà chúng được tạo ra chứ không phải theo thứ tự mà chúng nhận được
  - Giả sử rằng tiến trình P_i gửi yêu cầu đi vào CS cho tiến trình Leader
  - Sau đó một tiến trình P_j cũng gửi yêu cầu đi vào CS tới Leader và yêu cầu của P_j đến tiến trình Leader sớm hơn yêu cầu được tạo bởi tiến trình P_i
  - Như vậy, thứ tự yêu cầu mà tiến trình Leader nhận được có thể sẽ khác với thứ tự chúng được tạo ra !

Bài tập

- Đề xuất cải tiến thuật toán mutex tập trung để thoả mãn thuộc tính công bằng

Thuật toán vòng tròn token

- Giả sử tất cả tiến trình được tổ chức theo một hình tròn
- Token lưu thông quanh vòng tròn
- Tiến trình P_i muốn vào CS phải chờ đến khi token được luân chuyển đến nó
  - Khi đó P_i sẽ bắt lấy token và đi vào CS

```
public class CircToken extends Process implements Lock {
    public synchronized void initiate() {
        if (haveToken) sendToken();
    }
    public synchronized void requestCS() {
        wantCS = true;
        while (!haveToken) myWait();
    }
    public synchronized void releaseCS() {
        wantCS = false;
        sendToken();
    }
    void sendToken() {
        . . .
    }
    public synchronized void handleMsg(Msg m, int src, String tag) {
        if (tag.equals("token")) {
            haveToken = true;
            if (wantCS)     notify();
            else {
                Util.mySleep(1000);
                sendToken();
            }
        }
    }
}
```

haveToken = false
sendMsg(neighbour, “token”)

công bằng : 0
an toàn:
sự sống:

Tài liệu tham khảo

- *Concurrent and Distributed Computing in Java*, Vijay K. Garg, University of Texas, John Wiley & Sons, 2005
- Tham khảo:
  - *Principles of Concurrent and Distributed Programming*, M. Ben-Ari, Second edition, 2006
  - *Foundations of Multithreaded, Parallel, and Distributed Programming*, Gregory R. Andrews, University of Arizona, Addison-Wesley, 2000
  - *The SR Programming Language: Concurrency in Practice*, Benjamin/Cummings, 1993
  - *Xử lý song song và phân tán*, Đoàn văn Ban, Nguyễn Mậu Hân, Nhà xuất bản Khoa học và Kỹ thuật, 2009