울산대학교 공학연구논문집 제30권 2호 pp. 505~518, 울산대학교.
Journal of Engineering Research Vol. 30, No. 2, pp. 505~518, 1999
University of Ulsan.

Java 병행프로그래밍 도구의 개발

박양수 · 김현규 · 문남두 · 이명준
컴퓨터정보통신공학부

<요 약>

분산응용시스템은 서비스의 효율향상을 위해 일반적으로 병행성을 지원하는 것이 바람직하다. 그러나 병행성을 지원하기 위한 프로그래밍 기법은 난이도가 높은 구현 기법이며, 대부분의 프로그래밍 언어에서 이를 직접적으로 지원하지 않으므로, 분산응용시스템을 위한 개발자의 부담을 가중시키고 있다.

본 논문에서는 현재 분산응용시스템의 작성에 널리 쓰이고 있는 Java에서 병행성을 체계적으로 지원하기 위한 기반으로서, 세마포어나 조건변수와 같은 고수준 동기화 도구의 개발에 대해 소개하였다. 이러한 Java고수준 동기화 도구는 순수한 Java로 개발되어 이식성을 지니고 있으며, 개발자가 쉽게 기술할 수 있는 인터페이스를 제공하고 있다. 또한 Java가 지닌 대중성으로 인해 대부분의 분산응용시스템의 구축에 널리 사용될 수 있을 것으로 기대된다.

A Development of Java Synchronization Tools

Yang-Su Park · Hyun-Gyu Kim · Nam-Doo Moon · Myung-Joon Lee
School of Computer Engineering & Imformation Technology, University of Ulsan

<Abstract>

In a development of distributed applications, it is desirable for a server to support concurrency in order to provide effective services. However, the programming mechanism for concurrency is difficult to implement correctly, so most programming languages do not support this facility directly. Therefore, the fact imposes extra

burdens to the developers for distributed applications.

The high-level synchronization tools such as Semaphores and Condition Variables in Java are suggested as a basis for systematically supporting concurrency control. The suggested high-level synchronization tools also have a good portability since it has developed in pure Java. In addition, due to the popularity of Java, these tools will go a long way with developers in building distributed systems.

1.서 론

세마포어와 같은 고수준 동기화 도구에 대한 제안과 사용예는 1960년대 Dijkstra[4]로부터 시작되었으며, 그 후 오랜 기간동안 이들 도구에 대한 사용예와 그에 따른 정확성에 대한 증명이 있었다[7,8]. 그러나 이들 도구에 대한 구현은 Fortran, Pascal, C와 같은 기존의 범용언어에서 지원하지 않았으며, 기존의 개발자들은 필요할 때마다 수작업으로 시스템 함수를 사용하여 동기화 문제를 해결해야 했다. 또한 이들 범용언어 수준에서 고수준 동기화 도구를 구현하기 위한 방법론이 Ada언어에서 제시되기는 하였으나[9] 널리 범용화되지는 않았다.

Java언어는 이러한 동기화 문제를 처음으로 범용언어 단계의 수준까지 소개한 언어로서, 네트워크나 예외처리 등의 고수준 개념을 도입하여 분산환경의 특성을 가장 잘 반영한 언어로 소개되었다. 그러나 Java에서 지원하는 도구 역시 병행성 처리에 있어서 가장 기본적인 부분만을 지원하고 있으며, 사용자의 응용프로그램 작성에는 다소 제한적이다. 따라서 분산시스템에서 효율적인 서버의 구현에 필수적인 병행성 처리를 위한 개발자의 부담을 가중시키고 있다.

본 논문에서는 Java에서 세마포어나 조건변수와 같은 고수준 동기화 도구를 지원하기 위한 방법을 소개함으로써, 분산시스템에서 효율적인 서버의 구현에 필요한 병행성을 체계적으로 지원하고자 한다. 이러한 고수준 동기화 도구는 순수한 Java로 개발되어 있으므로 쉬운 확장과 이식성을 기대할 수 있으며, 사용하기 쉬운 인터페이스를 제공하고 있다. 따라서 개발자는 본 논문에서 제공하는 Java 고수준 동기화 도구를 이용하여 보다 신뢰성 있고 안정적인 응용프로그램을 작성을 기대할 수 있다.

본 논문의 구성은 다음과 같다. 2장에서는 고수준 동기화 도구에 대한 설계와 그의 기초가 되는 Java의 여러 기능에 대해 소개한다. 그리고 3장에서는 고수준 동기화 도구의 구현 방법과 효율적인 사용예에 대해 제시하며, 마지막으로 4장에서는 고수준 동기화 도구에 대한 간결한 요약과 기대효과에 대해 기술한다.

2. Java 병행프로그래밍 도구의 설계

이 장에서는 Java가 지원하는 기본적인 동기화 기능에 대해 소개하고, 이를 바탕으로 설계된 Java 고수준 동기화 도구에 대해 소개한다.

2.1. Java 동기화 도구

Java는 최근 분산환경에서 가장 널리 사용되고 있는 프로그래밍 언어로서, 다른 프로그래밍 언어와는 달리 언어 자체에서 동기화를 지원하기 위한 기능을 포함하고 있다. Java 동기화 도구는 서로 독립적으로 실행되는 스레드를 제어하고 동기화시키기 위해 주로 사용되며, 주요 메쏘드(method)로는 wait(), notify(), suspend(), resume(), yield(), sleep() 등이 있다. 다음은 Java에서 스레드의 기본 사용방법과 스레드를 제어하기 위한 각 함수의 기능에 대해 소개하고 있다.

(1) Thread

스레드는 일반적으로 서비스의 병행적인 수행을 제공하기 위해 지원되는 기능으로서, 동일한 목적으로 사용되는 프로세스에 비해 효율적이다. Java는 스레드 기능을 기본적으로 제공하고 있으며, 그 쓰임은 [그림.1]과 같다.

```
class myThreadExample extends Thread {
    myThreadExample() {
        //.. Do something ..... Initialization
    }
    public void run() {
        //.. Do something ..... Running a service in a form of Thread
        aProcessing();
    }
    void aProcessing() {
        //.. Do something ..... Processing
    }
}
```

[그림.1] Java 스레드 클래스

Java에서 특정 서비스가 스레드로 동작하려면, Thread 클래스로부터 상속받거나 또는 Runnable 인터페이스를 구현(implements)하여야 한다. 그리고 이 서비스의 내용은 반드시 run() 메쏘드내에 정의되어야 스레드로서 병행적으로 처리된다. [그림.2]는 위의 서비스 스레드를 사용하기 위한 클래스의 예를 보이고 있다.

```
class myThreadUseExample {
    myThreadUseExample() {
        //.. Do something ..... Initialization
    }
    void aProcessing() {
        //.. Do something ..... Use the Thread
        myThreadExample ex = new myThreadExample();
        ex.start();
    }
}
```

[그림.2] Java 스레드 클래스의 사용예

Java 서비스 스레드를 사용하려면 해당 클래스의 인스턴스를 생성한 후 start() 메쏘드를 호출해야 한다. start() 메쏘드는 서비스 스레드의 run() 메쏘드를 호출하는 역할을 한다.

(2) 스레드를 제어하기 위한 메쏘드

스레드를 제어하기 위한 메쏘드로는 wait(), notify(), suspend(), resume(), yield(), sleep() 등이 있다. 먼저 wait()는 스레드 뿐만 아니라 모든 Java의 기본적인 객체의 수행을 지연시킬 수 있는 기능을 제공하며, 지연시킨 후에 수행의 제어권을 다른 객체로 넘겨주므로서 교착상태(Deadlock)를 방지할 수 있도록 지원한다. notify()은 wait()로 수행이 지연된 메쏘드를 재실행시키며, 메쏘드의 재수행 순서는 시스템이 결정하게 된다. 일반적으로 wait()와 notify()를 이용하여 수행을 제어하는 방법은 안전하고 사용하기 쉽지만, 세밀한 제어가 어렵다는 단점 역시 가질 수 있다. 다음은 wait()과 notify()의 사용예를 제시하고 있다[그림.3].

```
class myWaitSignalExample {
    void myProcess() {
        //... Do something ..... Processing
        ...
        notify();
    }
    void myConsumer() {
        if(...) wait();
        //.. Do something ..... Consuming
        ...
    }
}
```

[그림.3] wait()과 notify()의 사용예

다음으로 suspend()와 resume()은 wait()와 signal()의 기능과 유사하며, 스레드의 핸들을 인자로 하여 보다 정교한 제어를 가능하게 해준다. 그러나 suspend()는 지연시킨 후에 수행의 제어권을 자신이 계속 유지함으로서 교착상태(Deadlock)를 일으킬 수 있으며, 반드시 스레드에서만 사용될 수 있는 제약이 있다. 다음은 suspend()와 resume()의 사용예를 제시하고 있다.

```
class mySuspendResumeExample extends Thread {
    ThreadQueue q = new ThreadQueue();
    void myProcess() {
        //.. Do something ..... Processing
        Thread t = q.get();
        t.resume();
    }
    void myConsumer() {
        if(...) {
                q.put(this)
                suspend(this);
        }
        //.. Do something ..... Consuming.
    }
}
```

[그림.4] suspend()과 resume()의 사용예

마지막으로 yield()는 자신의 수행 제어권을 다른 스레드로 이양하는데 사용되며, sleep()은 스레드의 수행을 특정 시간 동안 지연시키는 데 사용된다. wait()와 notify()을 제외한 모든 메쏘드는 스레드에서만 사용될 수 있다.

현재 Java가 지원하는 동기화는 가장 기본적인 기능으로 한정되며, Java에서 서버의 병행성을 지원하기 위해서는 개발자의 별도의 노력을 필요로 한다. 이러한 병행성 구현은 필요시마다 임기응변식으로 이루어지며 난이도가 높은 프로그래밍 기법이므로, 응용프로그램의 신뢰성을 떨어뜨릴 뿐만 아니라 개발속도 역시 저하시킨다.

2.2. Java 병행프로그래밍 도구의 설계

이 절에서는 앞절에서 지적한 단점을 보완하기 위한 Java 고수준 동기화 도구에 대해 소개한다. 서버의 병행성 구현을 보다 쉽게 구현하기 위한 고수준 동기화 도구로는 조건변수, 세마포어등을 고려할 수 있으며, 이들 도구는 직관적인 인터페이스를 제공하여야 한다.

(1) 조건변수 (Condition Variable)

기존의 조건변수는 병행프로세스간의 동기화(synchronization)를 목적으로 한 도구이며 큐(queue)의 성격이 강하다[6]. 이는 특정조건이 만족되면 해당 프로세스를 지연하는 데 사용되며, 지연된 프로세스는 큐 형태로 관리된다. 역시 재수행을 위한 조건이 만족되면 조

건변수에서 지연된 프로세스를 FIFO(First-In, First-Out)의 순서대로 재수행시킬 수 있다.

Java 조건변수는 wait(), signal()의 두가지 메쏘드를 제공한다. wait() 메쏘드는 해당 스레드를 지연시키고 후에 재수행을 위해 스레드의 핸들을 필요로 하며, 이는 조건변수의 내부큐에 저장된다. signal() 메쏘드는 이와 반대로 큐에 저장된 스레드 핸들을 이용해 스레드를 FIFO 순서로 재수행시킨다. [그림.5]는 Java 조건변수의 사용예를 제시하고 있다.

```
class myThreadExample extends Thread {
    static ConditionVariable c = new ConditionVariable ();

    void myCheck() {
        if( ... ) c.wait(this);
        //.. Do something .....

    }
    void myRun() {
        //.. Do something .....

        c.signalAll();
    }
}
```

[그림.5] 조건변수의 사용예

(2) 세마포어 (Semaphore)

기존의 세마포어[4]는 병행 프로세스간의 동기화(synchronization)를 목적으로 한 도구로서 본 논문에서 소개하는 Java 세마포어 역시 같은 기능을 지원한다. Java 세마포어의 lock() 메쏘드는 하나 이상의 프로세스가 실행되고 있는 상태에서 현재 프로세스를 실행할 수 없도록 하며, 그렇지 않은 경우에는 현재 프로세스를 수행할 수 있도록 한다. 이에 반해 unlock() 메쏘드는 lock()에서 대기 상태인 프로세스를 재실행시키는 역할을 한다.

[그림.6]은 본 논문에서 설계된 Java 세마포어의 사용예에 대해 소개하고 있으며, lock() 메쏘드는 Java 조건변수와 마찬가지로 스레드의 핸들을 필요로 한다.

```
class mySemaphoreExample {
    static Semaphore s = new Semaphore ();

    void myProcess() {
        //.. Do something ..... Processing

        s.unlock(this);
    }
    void myConsumer() {
        s.signal();
        //.. Do something ..... Consuming
    }
}
```

[그림.6] 세마포어의 사용예 .1

또한 세마포어는 기존의 mutex[5]에 해당하는 병행 프로세스간의 완전한 상호배제(mutual exclusion)를 위한 기능 역시 제공한다. [그림.7]은 세마포어를 mutex의 용도로 사용한 예이다.

```
class myMutexExample {
    static Semaphore m = new Semaphore();
    void myOp1() {
        m.lock(this);
        //.. Do something .....

        m.unlock();
    }
    void myOp2() {
        m.lock(this);
        //.. Do something .....

        m.unlock();
    }
}
```

[그림.7] 세마포어의 사용예 .2

3. Java 병행프로그래밍 도구의 구현

Java 고수준 동기화 도구는 Java에서 지원하는 기본적인 동기화 기능으로 구현된다. 이 장에서는 이들 조건변수나 세마포어등의 고수준 동기화 도구가 어떠한 방식으로 구현되는지에 대해 살펴보고, 효과적인 사용예를 제시한다.

3.1. 조건변수(Condition Variable)의 구현

Java 고수준 동기화 도구의 조건변수는 스레드를 지연시키거나 재수행할 수 있도록 설계되었으며, 이를 구현하기 위해 Java의 wait(), notify() 메쏘드를 사용한다.

Java 조건변수의 구현은 다음과 같다. wait()가 호출되었을 때는 스레드의 wait() 함수를 호출해서 수행을 지연시킨다. 그리고 signal() 시에는 스레드의 notify() 함수를 통해 수행을 재개한다. 그러나 Java의 notify() 함수는 특정 스레드를 선택하여 수행을 재개할 수 있는 기능을 제공하지 않으며, CPU의 상태에 의해 재수행되는 스레드가 결정된다. 따라서 단순히 notify()의 기능에 의존할 경우 어떤 스레드는 기아상태(Starvation)가 될 수 있다. 이를 위하여, Java 조건변수는 지연된 스레드를 FIFO 순서로 재수행할 수 있도록 다음과 같이 구현되었다 [그림.8].

```
ThreadQueue q = new ThreadQueue();

void wait(Thread t)
{
    t.setPriority(2);
    q.put(t);
    wait();
}

void signal()
{
    if(q.isEmpty() == false) {
        Thread t = q.get();
        t.setPriority(6);
        notify();
    }
}
```

[그림.8] Java 조건변수의 구현

Java 조건변수는 큐(Queue)를 이용하여 지연된 스레드의 리스트를 유지하고 있으며, notify()를 호출할 때 큐에서 꺼내온 순서대로 스레드를 재수행하기 위해 스레드의 우선순위(Priority)를 이용한다. 스레드의 우선순위는 일반적으로 5의 값이 디폴트로 선정되며, 보다 낮은 값일수록 notify()를 할 때 늦게 호출된다.

3.2. 세마포어(Semaphore)의 구현

Java 세마포어는 주로 상호배제(Mutual exclusion)나 동기화를 위해 사용되며, 조건변수의 wait()과 signal()에 해당하는 lock()과 unlock()을 제공하고 있다. Java세마포어의 lock() 메쏘드는 하나 이상의 프로세스가 실행되고 있는 상태에서 현재 프로세스를 실행할 수 없도록 하며, 그렇지 않은 경우에는 현재 프로세스를 수행할 수 있도록 한다. 이에 반해 unlock() 메쏘드는 lock() 메쏘드에서 대기 상태인 프로세스를 재실행시키는 역할을 한다.

Java세마포어의 구현은 조건변수의 구현방법과 유사하며, 단지 임계영역(Critical region)에 처음 실행되는 프로세스는 지연되지 않아도 된다는 특성만 추가된다. 이를 위해 boolean타입의 내부변수인 b를 두어 처음 실행되는 프로세스인지 아닌지를 결정한다. 다음은 역시 wait()과 notify()를 이용한 세마포어의 구현에 대해 보여주고 있다[그림.9].

```
ThreadQueue q = new ThreadQueue;
boolean b = false;
 void synchronized lock(Thread t)
 {
    if(b==false) b = true;
    else {
   t.setPriority(2)
  q.put(t);
  wait();
    }
 }
```

```
void synchronized unlock()
{
    if(q.isEmpty()==false) b = false;
    else {
Thread t = q.get();
t.setPriority(6);
notify();
    }
}
```

[그림.9] Java 세마포어의 구현

3.3. Java 병행프로그래밍 도구의 사용예

Java 고수준 동기화 도구는 서버의 병행성 처리에 효과적으로 사용될 수 있다. 일반적으로 분산시스템에서 서버는 클라이언트로부터 요청이 올 때마다 해당 요청에 대한 스레드를 생성하여 서비스를 처리한다. 그러나 이들 서비스 스레드는 서로 병행적으로 수행되므로 서버측 데이터의 일관성 유지에 문제를 일으킬 수 있다. Java 고수준 동기화 도구는 이러한 서버의 병행성 처리에 효과적으로 사용될 수 있다.

다음은 분산시스템의 서버에서 서비스 스레드의 여러가지 병행수행의 형태를 소개하고 있으며, 이들 스레드의 동기화를 만족시키기 위해 필요한 Java 고수준 동기화 도구의 사용예를 보여주고 있다. 분산시스템에서 서비스 스레드는 일반적으로 다른 스레드와 상호배제적으로 동작하는 Procedure와 스레드간에 병행적으로 수행될 수 있는 Function, 그리고 선행조건이 만족될 때만 수행되는 Guard등으로 구분될 수 있다[10].

(1) Procedure

procedure로 정의된 서비스 메쏘드는 수행되는 모든 서비스와 상호배제적으로 동작하며, 이러한 특성 때문에 주로 쓰기 전용의 서비스를 처리하는데 사용된다. Procedure의 상호배제는 세마포어를 이용하여 구현될 수 있으며, 다른 타입의 서비스와의 완전한 상호배제를 위해 서비스의 도입부인 Entry와 후반부인 Exit에서 [그림.10]과 같이 정의한다.

```
static Semaphore sp = new Semaphore();

void setValue() {
    //.. Entry code
    sp.lock();

    //.. User code ...
    //.. ...

    //.. Exit code
    sp.unlock();
}

void getValue() {
    //.. Entry code
    sp.lock();

    //.. User code ...
    //.. ...

    //.. Exit code
    sp.unlock();
}

void awaitValue() {
    //.. Entry code
    sp.lock();

    //.. User code ...
    //.. ...

    //.. Exit code
    sp.unlock();
}
```

[그림.10] procedure, function, guard간의 완전한 상호배제를 위한 구현

세마포어 sp는 모든 서비스 스레드에서 상호배제를 위해 사용되며, 공유가능하도록 static으로 선언된다. sp의 lock() 메쏘드는 세마포어 고유의 정의에 의해 이미 수행되고 있는 서비스가 있을 경우 현재 자신의 수행을 지연시키며, 그렇지 않으면 다른 서비스가 수행될 수 없는 상태로 만들고 자신은 수행을 계속한다.

(2) Function

function으로 정의된 서비스 메쏘드는 procedure나 guard서비스와는 상호배제적이며, function서비스간에는 병행적으로 수행된다. 예를 들어 수행되고 있는 procedure나 guard 서비스가 존재할 경우, function을 포함한 모든 서비스는 도입부에서 대기상태가 된다. 이에 반해 수행되고 있는 서비스가 function일 경우에는 procedure, guard의 요청은 대기되며, function의 요청은 병행적으로 수행될 수 있다. 이러한 function의 특성으로 인해 function은 주로 읽기 전용의 서비스를 처리하는 데 사용된다.

위의 procedure의 구현에서는 function서비스의 도입부에서 무조건적으로 세마포어의 lock()을 호출하므로, function서비스간의 병행수행을 불가능하게 한다. 따라서 function서비스간의 병행수행을 가능하게 하기 위해서는 처음 수행되는 function서비스에서 세마포어의 lock()을 호출하도록 수정하며, 다음에 수행되는 function서비스에서는 세마포어의 lock()을 호출하지 않도록 한다. 이를 위해 수행되고 있는 스레드의 갯수에 대한 정보를 유지하기 위해 cntFunc을 선언하였다. 그러나 cntFunc에 대한 변형과 참조는 function 스레드간에 병행적으로 수행될 수 있으므로 임계영역으로서 보호되어야 하며, 이를 구현하기 위해 새로운 세마포어 sf를 도입하였다. function간의 병행 수행을 위한 전체적인 구현은 [그림.11]과 같다.

```
static int cntFunc = 0;
    static Semaphore sf = new Semaphore();

    void getValue()                    {
        //..  Entry
        sf.lock();
        if(cntFunc==0) sp.lock(this);
        cntFunc++;
        sf.unlock();

        //..  User code will be here ...
        //..  ...

        //..  Exit
        sf.lock();
        cntFunc--;
        if(cntFunc==0) sp.unlock();
        sf.unlock();
}
```

[그림.11] function의 병행수행을 위한 [그림.10]의 수정

function을 위해 수정된 구현에서는 모든 function서비스에서 lock()을 호출하지 않고, 처음 수행되는 function 서비스에서만 세마포어sp의 lock()을 호출하도록 한다. 따라서 그 후 수행되는 function서비스는 세마포어sp의 lock()을 만나지 않으므로 function서비스간의 병행수행이 가능해진다. 그리고 수행하려는 procedure나 guard서비스는 처음 수행되는 function서비스에서 이미 세마포어sp의 lock()을 호출함으로서 도입부에서 대기상태가 된다. 이미 수행되고 있는 procedure나 guard서비스가 존재한다면 모든 function서비스가 위 구현에서 if블럭이하의 조건을 만족하므로 역시 procedure나 guard서비스간의 상호배제가 이루어지게 된다.

3.3. Guard의 지원

guard로 정의된 오퍼레이션의 스레드는 모든 타입의 스레드와 상호 배제적이며, 추가적으로 조건부 수행을 제공한다. 조건부 수행에서는 오퍼레이션의 선행 조건이 만족되면 오퍼레이션을 수행하고 그렇지 않으면 만족될 때까지 대기 상태로 있게 된다.

```
static ConditionVariable c1;

void AwaitValue1()
{
        //..  Entry
        sp.lock(this);
        if(!precond1()) {
                sp.unlock();
                c1.wait();
        }
        //..  User code will be here ...
        //..  Exit
        sp.unlock();
}
```

[그림.12] guard의 조건부수행을 위한 수정 [1]

guard의 조건부 수행을 처리하기 위해 guard 오퍼레이션의 Entry에서는 선행 조건인 precond1()이 만족되지 않았을 때 다른 스레드가 수행될 수 있도록 sp.unlock()을 호출한 후, 자신은AwaitValue1()의 선행 조건을 위한 조건 변수 c1의 대기열에서 대기할 수 있도록 [그림.10]의 구현을 수정하였다[그림.12]. 각 guard 오퍼레이션은 자신의 조건 변수를 가지게 된다.

guard의 구현에서 주의할 점은 대기 상태로부터 재수행을 위해 조건에 대한 재검사가 반복적으로 이루어져야 한다는 점이다. 그렇지 않으면 guard 스레드는 영원히 대기 상태가 된다. 이 재검사는 객체의 내부 상태가 변경된 후에 이루어지는 것이 바람직하다. 따라서 쓰기 가능한 연산으로 쓰이는 procedure나 guard 스레드의 Exit에서 재검사를 수행하도록 [그림.13]처럼 작성한다.

```
void SetValue()                          void AwaitValue()
{                                        {
    //..  Entry                            //..  Entry
    //..  User code will be here ...       //..  User code will be here ...

    //..  Exit                             //..  Exit
    reevaluation();                          reevaluation();
}                                        }
```

[그림.13] guard의 조건부수행을 위한 수정 [2]

재검사에 대한 코드는 reevaluation()에서 구현하고 있다[그림.14]. reevaluation()에서는

procedure나 guard 스레드의 수행 이후 바뀐 상태를 만족하는 guard 스레드를 깨워 주어야 한다. 그리고 만족하는 guard 스레드가 없으면 다음 스레드가 수행될 수 있도록 sp.unlock()을 호출한다.

```
void reevaluation()
  {
        if(precond1())      c1.signal();
        else if(precond2()) c2.signal();
        else if(precond3()) c3.signal();
        ...
        else sp.unlock();
  }
```

[그림.14] guard의 재수행을 위한 오퍼레이션 reevaluation()

4. 결 론

Java에서는 세마포어(Semaphore), 조건변수(condition variable)등과 같은 고수준 동기화 도구를 직접적으로 제공하지 않으므로, 개발자는 응용프로그램 작성시에 필요할 때마다 이들을 직접 구현하여 사용하고 있다. 이러한 임기응변식(adhoc) 구현은 정확성이 떨어지며, 기존에 구현된 코드에 대한 재사용 역시 어려우므로, 병행 프로그래밍의 구현에 있어서 보다 체계적인 접근이 필요하게 된다.

본 논문에서는 분산응용프로그램의 개발에서 흔히 쓰일 수 있는 세마포어 등의 고수준 동기화 도구의 개발에 대해 소개함으로써, 개발자가 동기화 문제에 보다 쉽게 접근할 수 있도록 지원하고자 하였다. 본 논문에서 소개한 Java 동기화 도구는 Java의 wait( )이나 notify( )등과 같은 기본적인 동기화 기능을 이용하여 구현되었으며, 큐와 스레드의 우선순위 변경기능을 이용하여 지연된 스레드의 기아상태를 방지할 수 있도록 지원하였다. 그리고 분산응용시스템에서 각 서비스의 병행처리 방법을 소개함으로서, 이러한 동기화 도구의 사용예에 대한 구체적인 이해를 돕고자 하였다.

Java 동기화 도구는 직관적인 인터페이스를 제공하여 개발자의 편의를 도모하였으며, 순수한 Java 언어로 제작되었다는 점에서 Java가 지닌 이식성이나 대중성을 기대할 수 있다. 또한 본 논문에서 제시한 방법은 다른 범용언어에서의 고수준 동기화 도구의 개발에 대한 연구에도 충분히 도움을 줄 수 있을 것으로 예측된다.

5. 참고문헌

[1] Kenneth P. Birman, "The Process Group Approach to Reliable Distributed Computing", 1993

[2] "The Java Language Tutorial", Sun Microsystems, Inc., 1995

[3] F. Ranno, S.K. Shrivastava and S.M.Wheater, "A System for specifying and Coordinating the Execution of Reliable Distributed Applications" (DAIS'97), Cottbus, Germany, September 30 - October 2, 1997

[4] E.W. Dijkstra, "Coorperating Sequential Processes", Technical Report EWD-123, Technological University, Eindhoven, the Netherlands(1965)

[5] L. Lamport, "The Mutual Exclusion Problem", Journal of the ACM, Volume 33, Number 2(1986), pages 313-348

[6] M. Ben-Ari, "Principles of Concurrent and Distributed Programming", Prentice-Hall, Englewood Cliffs, NJ (1990)

[7] P.J. Courtois, F. Heymans, and D.L. Parnas, "Concurrent Control with 'Readers' and 'Writers'" Communications of the ACM, Volume 14, Number 10 (October 1971), pages 667-668

[8] S.Patil, "Limitations and Capabilities of Dijkstra's Semaphore Primitives for Coordination Among Processes", Technical Report, MIT (1971)

[9] Pascal Lebru, "Translation of the Proceted Type mechanism in Ada83", ACM Ada Letters, Vol 15, No.1, pp.64-69 (Jan/Feb 1995)

[10] 박양수, 김현규, 이명준, 한상영, CORBA 개방형 분산환경을 위한 공유객체 명세언어 시스템, 한국정보처리학회 논문지 5권 2호, pp.404-413 (Feb 1998)