TELKOMNIKA, Vol. 11, No. 9, September 2013, pp. 5395~5401
ISSN: 2302-4046

A Framework of Concurrent Mechanism Based on Java Multithread

Wuxue Jiang*[1], Qi Li[2], Zhiming Wang[3], Jianfeng Luo[1]
[1]Department of Computer Engineering, Dongguan Polytechnic, Dongguan 523808, Guangdong, China
[2]Department of Computer and Information, Huaihua Vocational and Technical College, Huaihua 418000, Hunan, China
[3]Educational Technology Center, Dongguan Polytechnic, Dongguan 523808, Guangdong, China
*Corresponding author, e-mail: 44284585@qq.com

Abstract

The continuously increased demand for paralleling multitask in domains such as grid computing and cloud computing has significantly promoted research on concurrent mechanism and concurrent programming. The Java programming language supports multithread mechanism for developing paralleling programs, however, it is difficult to apply Java concurrent primitives to specific problems. Thus, for the development of high reliable and qualitative Java concurrent programs, this paper analyses Java multithread mechanism and it's realization, studies the concurrent mechanism based on Java synchronization and interactive communication mechanism, compares the concurrent structure based on operating system and based on Java multithread, sums up some concurrent programming rules and strategies to prevent deadlock. A frame instance based on entire synchronization is presented, which can help to develop concurrent programs quickly.

Keywords: *java multithread, concurrent mechanism, synchronization mechanism, deadlock precaution*

1. Introduction

With the constantly advancement of the modern operating system employed a kernel-level multithreading structure and architecture of the processor, concurrent programming techniques also sprang up and occupied a pivotal position in the real-world applications. Currently, the language development tools like C++, Delphi, Java etc. are all supporting the concurrent multi-tasking program [1, 2].

Java was one of the first languages to make multithreading easily available to developers and provided built-in primitives for threads, such as wait(), notify(), and synchronized() etc. [3, 4], but those are not sufficient for some concurrent programming, and even can not solve some sophisticated concurrency issues.

The Java language and Java virtual machine have provided a completely sense of the multi-thread mechanism, its built-in language-level multi-thread mechanism can easily implement development of parallel programs of multiple-task [5, 6]. Java multithread mechanism has achieved simultaneous execution of multi-task in macro and provided synchronous mechanisms and communication mechanisms for achieving critical resource protection during the concurrent process, also provided preventive measures of deadlock due to synchronization.

2. Process and Thread

2.1. Concept Comparison

The process is a static code that the program conducts a dynamic running process on the processor, it's a process of generation, development and extinction. Thread is a smaller execution unit than the process, there may have one or more threads in one process, thread is also a dynamic process of creation, existence and demise.

2.2. Functions Comparison

The thread is an important part of the process. The traditional process wears two hats: as the basic unit of resource allocation and CPU scheduling, separate its function and share it

Received March 5, 2013; Revised June 14, 2013; Accepted June 24, 2013

with entities called threads in order to better carry out concurrency of the development program and allow the process to get rid of the burdensome task, then thread is come into being. Process is a basic unit for resource allocation, and thread is a basic unit of the CPU scheduling. Processes apply and obtain the required resources, the corresponding threads active in these resources and use them.

2.3. System Overhead Comparison

Overhead of conversion and changing-over between threads is smaller than the overhead needed for the processes doing the same amount of work because of the difference between and threads in the above-mentioned functions, and it's easier to implement multi-thread synchronization and communication.

3. Thread State and Control Method

Java threads have a dynamic life cycle, a complete cycle will go through five states [7], conversion between threads and the corresponding thread control method shown in Figure 1.

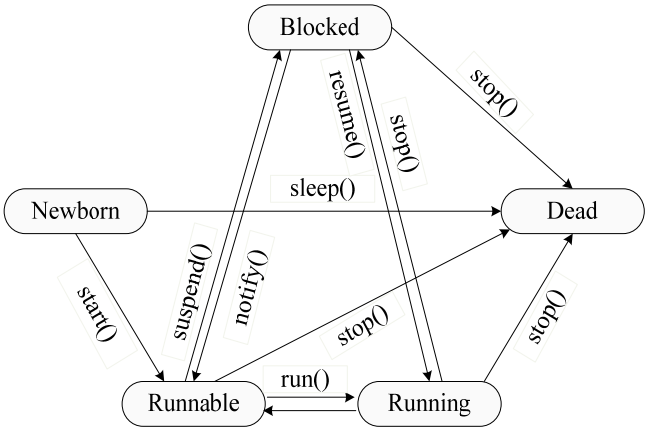

Figure 1. Thread's State Convention and Control Method

3.1. New Born State

The object is in a new born state after creating the instance of a subclass of the thread class and initialization, corresponding storage space and process resources are created by this time.

3.2. Runnable State

The thread is in runnable state after starting the new born thread, it is in the runnable queue to wait for the arrival of the CPU time slice. Running condition is existed now.

3.3. Running State

The threads in runnable state get CPU time slice and come into a running state, then perform the run () method code of corresponding thread to complete the appropriate action.

3.4. Blocked State

The threads in running or runnable state abdicate the CPU and suspend the execution that is to enter a blocked state for some reason, convert to runnable or running state only when the blocked cause are removed.

3.5. Dead State

A thread completed all of its operations or forcibly terminated is in a dead state, the thread can not be restored and executed.

4. Thread Creation based on the Thread Subclass and Runnable Interface

It usually requires more than one task simultaneously perform in macro, also needs to take full advantage of the system resources and improves the program execution efficiency in large utility software development process, multi-thread can provide excellent solution. Java is a complete sense of programming language oriented to the object, provides abundant class libraries and application interfaces to achieve user-friendly development and utilization. There are two ways to create threads in Java [8].

4.1. Inherit the Thread Class

Java. lang. Thread is a class used to represent the process provided by the system, many methods in Thread class provide a complete multi-thread processing function. Thread can define the subclass and construct users thread.

Such as:

```
public MyThread ex tends Thread
{
/ / Related properties and methods definition
public void run()
{
, / / Thread body code
}
public static void main (String args[])
{Thread t = new MyThread();
And
t. start(); / / Start the thread t
,
}
}
```

Run() method is essential and it is the core of the whole thread, the code is the content to be executed by the thread, start() method calls the run() method to start the thread.

4.2. Complete Runnable Interface

Java provides a runnable interface which can achieve the goal of multithread, this interface only has one run() method, uses can overload this method to complete related thread operations, runnable interface can be automatically identified and performed by the system.

For example:

```
public class My Thread implements Runnable
{
, / / Related attribute definitions
public void run()
{
, / / Code to be executed by the thread
}
}
```

There are two ways to start, but the essence is of the same, as follows:

```
MyThread my = new MyThread();
Thread t = new Thread (my);
t. start();
```

or

```
MyThread my = new MyThread();
new Thread(my).start.
```

4.3. Comparison of Two Methods of Thread Creation

The first method is distinctive nuance, logic clear and easy to use, it achieves Runnable interface in essence; the second method can make up the shortage of the first method, that can achieve multiple inheritance (Java classes only can achieve single inheritance), such as applications in the small program of Applet. The Runnable interface also is achieved based on the Thread class framework.

5. Analysis and Study of the Java Concurrent Mechanism

5.1. Mutually Exclusive Mechanism based on Synchronized and Monitor

The introduction of concurrency, in order to achieve some kind of real objective in multi-programming, there always have some program segment that can't be "simultaneously" (the essence is simultaneously in macro) accessed by more than two threads, that is critical resource. Java defines the critical resources using synchronized to lock the mark, critical resources here may be a method or a code segment. The method or code segments marked by synchronized in the program can only be accessed by one thread at any time, that mutually exclusive access to critical resources is achieved. It has introduced Monitor similar to semaphore mechanism in the realization principle, Java assigned each object a Monitor, and its role was responsible for managing the thread's access to critical resources. A thread getting access to a certain critical resource will obtain the Monitor of the subordinate object of the critical resource, and the Monitor will "lock", other threads who would like to have access to the critical resources must wait till the Monitor "unlocked" (see Figure 2).

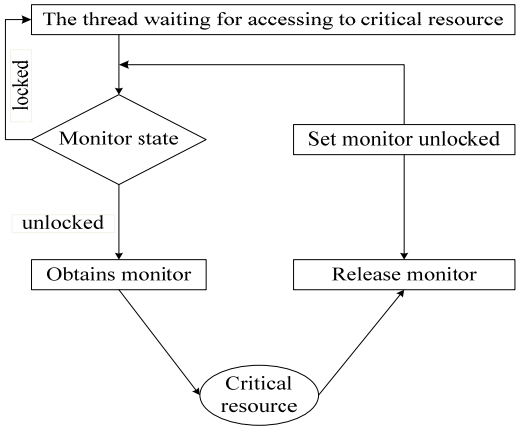

Figure 2. Realization Schematic of Locked-unlocked Mutex Mechanism

5.2. Interactive Communication Mechanism Sased on wait() and notify()

Synchronized has solved the mutually exclusive access issues for the critical resources, it also requires collaborative work between threads in real-world applications, that conducts mutually exclusive access to the critical resources, meanwhile, the threads also communicate with each other. Such as classic producer-consumer issues, writers - readers issues, sender–receiver issues in network transmission and so on. Java provides three standard Object class methods in order to solve the interaction problems between the multi-thread: wait(), notify() and notifyAll(). Wait() method is to convert the running state to the blocked state for the currently running thread, wait() entered the waiting queue concentrates and releases the Monitor; notify() is to wake up the first thread in the queue of the wait() concentration (or wake up a thread according to certain algorithms), so that convert the thread's blocked state to running state; notifyAll() is to wake up all the threads centralized by wait(). Co-use of wait() and notify() can easily achieve the mutually exclusive communication mechanism, so that to effectively solve the above cited problems.

5.3. Comparison of Synchronization based on Mutex and Java Synchronized

Concurrency means multiple threads wait for each other and communicate with each other in some of the key points. Java's synchronized has achieved a mutually exclusive mechanism of the thread, wait() and notify() have achieved a message communication mechanism between the threads, they solved the problem of data consistency and the collaboration between the multi-thread respectively, and realized complete sense of concurrency. Figure 3 makes a simple comparison between Java's synchronization mechanism and synchronization mechanism of signal lamp and P, V primitives' realization at operating system-level.

Synchronized(this)	
{,	P(empty)
Wait();	P(mutex)
,	CS;
Notify() ;	V(mutex)
,	P(full)

Figure 3. Comparison of Synchronization Mechanism between Java and the Operating System

6. A Routine of Concurrent Multitask based on Full Synchronization

6.1. Practical Routines

Construct a shared buffer zone class SharedBuffer, there is related data attributes within it, that is one send() method and one receive() method, both are synchronized methods, and achieve interactive communication between them through wait() and notify(). Construct another two sub-class Sender and Receiver which were inherited from the Thread class, to achieve specific sending and receiving operation by generating two threads of t1 and t2. Routines are as follows:

```
public class FrameSample
{
public static void main( String args[ ] )
{
SharedBuffer buffer= new SharedBuffer( ) ;
Sender t1= new Sender( buffer) ;
Receiver t2= new Receiver( buffer) ;
t1. start( ) ;
t2. start( ) ;
}
} / / Main class FrameSample, buffer is shared object instance
class Sender extends Thread
{ SharedBuffer theBuffer;
public Sender( SharedBuffer s)
{ theBuffer= s; }
public void run( )
{ char c;
for( int i= 0; i< 5; i+ + )
{ c= ( char) ( Math. random ( ) * 26+ -A. ) ;
theBuffer. send( c) ;
System . out. println(/ Sender: 0+ c) ;
}
}
} / / Sender class
class Receiver ex tends Thread
{ SharedBuffer theBuffer;
public Receiver( SharedBuffer s)
{ theBuffer= s; }
public void run( )
{ char c;
for( int i= 0; i< 5; i+ + )
{ c= theBuffer. receive( ) ;
System . out. println(/ Receiver: 0 + c) ;
}
}
} / / Receiver class
class SharedBuffer
{ private int index= 0;
```

```
private char buf[ ] = new char[ 5] ;
public synchronized void send( char c)
{ while( index = = buf. length)
{ try { this. wait( ) ; }
catch( InterruptedEx ception e) { }
}
this. notify( ) ; buf[ index] = c; index+ + ;
}
public synchronized char receive( )
{ while( index = = 0)
{ try { this. wait( ) ; }
catch( InterruptedException e) { }
}
this. notify( ) ; index - - ; return buf[ index] ;
}
} / / Shared buffer zone class.
```

6.2. Some Explanations

a) Program instance owns a complete sense of concurrency

For the mutually exclusive access to the shared buffer zone, it is through wait() and notify() to communicate with each other that whether you can send data to the buffer zone and fetch data from the buffer zone. Complete routine code was based on Windows 2003 and JDK1.5 environment; if you do not use the synchronized to achieve mutually exclusion in the example, there will be errors of data loss and duplication, abnormal situation will appear and the efficiency of the CPU will significantly reduce without the use of wait() and notify().

b) Deep understanding to critical resources

The critical resources refer to some shared equipment, shared data segment and data structure etc. in the real world, and critical resource in the routine is the buffer. The computer world is an abstraction and packaging of the real world, it's determined by specific mechanism, the corresponding critical resources are code segments of the program, and they are send() and receive() methods corresponding to the routines. Substances in two worlds are interconnected and consistent in essence, the reason why the buffer is critical resource is manifested and realized through operations of send() and receive() methods.

7. Precaution of Deadlock

Such a phenomenon may occur in thread synchronization mechanism that several threads each owns some of the resources required, but still need some of the resources of other threads at the same time, and they wait for the release of other threads, otherwise they are unable to push ahead, such a stalemate phenomenon due to competition and allocation of resources is called deadlock [9, 10]. The Java concurrency mechanism based on the JVM level is similar to the synchronization mechanism of the operating system, deadlock is of great hidden, it may easily lead to deadlock failures when users use it, and Java can not avoid deadlock and recover deadlock, the only resolution is to take precautions against deadlock when use it. A number of methods and strategies provided below can be used to prevent deadlocks.

7.1. Partition of Thread Tasks must be Clear and Reasonable

Multithread concurrency has become the root cause of deadlock, so you must combine with the actual needs and Java's multithread mechanism characteristics to reasonably arrange the tasks of each thread; ensure balanced resources; try to reduce the number of threads [10].

7.2. Properly use the Synchronization Mechanism

The use of synchronized is the direct reason for generating user-level deadlock, carefully consider the call between the synchronization methods; try to avoid nested call between the synchronization methods.

7.3. Properly use the Communication Mechanism

We usually make paring use of wait() and notify() for the two methods are able to solve certain deadlock problems, and consider whether to use notifyAll() method when necessary.

7.4. Ensure the Critical Resources to be Refined

Critical resource is the material base causing deadlock, try to refine the critical resources in order to reduce conflict due to competition of resources [11].

8. Conclusion

The powerful multithread technology provides terrific solving strategies of realistic problems for the users' multitask programming. The paper discussed the implementation technique of the Java multithread mechanism from the point of view of the principles and applications, analyzed the synchronous communication problems in multithread technology based on the operating system level, summarized some deadlock precaution methods and strategies, and constructed a framework instance of a fully synchronized concurrent multitask. A high-reliability and high-quality concurrent program can be developed quickly in real software development process according to the framework.

Acknowledgements

This work was supported by Science and Technological Program for Dongguan's Higher Education, Science and Research, and Health Care Institutions(No.2011108101010), and by Guangdong Province High-tech Industrial Projects(No. 2012B010100050).

References

[1] Hongwei Liao, Yin Wang, Hyoun Kyu, et al. Concurrency bugs in multithreaded software: modeling and analysis using Petir nets. *Discrete Event Dynamic Systems: Theory and Applications*. 2013; 23(2): 157-195.
[2] Doug Lea. The java.util.concurrent synchronizer framework. *Science of Computer Programming.* 2005; 58(3): 293-309.
[3] Brad Long, Paul Strooper, Luke Wildman. A method for verifying concurrent Java components based on an analysis of concurrency failures. *Concurrency and Computation: Practice and Experience*. 2007; 19(3): 281-294.
[4] Brad Long. A Framework for Model Checking Concurrent Java Components. *Journal of Software.* 2009; 4(8): 867-874.
[5] Yaniv Eytani. Concurretn Java Test Generation as a Search Problem. *Electronic Notes in Theoretical Computer Science.* 2006; 144(4): 57-72.
[6] Pavel G Zaykov, Georgi Kuzmanov. Multithreading on redconfigurable hardware: An architectureal approach. *Microprocessors and Microsystems*. 2012; 36(8): 695-704.
[7] Jonathan Aldrich, Emin Gun Sirer, Craig Chambers, Susan J Eggers. Comprehensive synchronization elimination for Java. *Science of Computer Programming*. 2003; 47(2): 91-120.
[8] Seetharami Seelam, Yanbin Liu, Parijat Dube, et al. Experiences in building and scaling an enterprise applicaiton on multicore systems. *Concurrency and Computation: Practice and Experience*. 2012; 24(2): 111-123.
[9] V Rafe, AT Rahmani, L Baresi, P Spoletini. Towards Automated Verification of Layered Graph Transformation Specifications. *Journal of IET Software*. 2009; 3(4): 276-291.
[10] Jyotirmoy V, Dshmukh E, Allen Emerson, Sriram Sankaranarayanan. Symbolic modular deadlock analysis. *Automated Software Engineering*. 2011; 18(3): 325-362.
[11] Mariagrazia Dotoli, Maria Pia Fanti. Deadlock Detection and Avoidance Strategies for Automated Storage and Retrieval Systems. *IEEE Transaction on Systems, MAN, and Cybernetics-Part C: Applications and Reviews*. 2007; 37(4): 541-552.