Research and Design of Java Timing Processing Method

Weiwen He*
School of Information Engineering, Guangzhou Nanyang Polytechnic, Guangzhou 510000, China
*Corresponding author e-mail: heweiwen@gznylg.edu.cn

Abstract. There are differences among programming languages. In contrast to C/C++ in Linux, standard Java timer has some issues, such as uncontrollable execution order, unpredictable start moment and running duration, when it's applied to handle dense timing tasks. So it could not be directly used by some military applications with strict requirements on time sequence. In this paper, its root cause is analyzed, and then an improved method based on queue and delay feedback is presented, as well as its implementation flow chart. The test results show that, this method is able to not only ensure the correct execution order, but also reduce the uncontrollable delay. So it could satisfy the time sequence demand of military applications.

Keywords: Standard Java Timer, Queue, Delay Feedback, Dense Timing Task

1. Introduction
As Java could run on many operating systems, it has been widely used in civil field. In military field, due to the history of technology development and people's doubt about Java's poor real-time ability, the primary programming language is still C/C++. However migrations from C/C++ to Java have been tried in several countries gradually [1]. Considering the different design ideas between C/C++ and Java, some characteristics of Java really don't meet the requirements of military applications. So issues may occur when application is programmed with standard Java directly, especially in cases which are strict on the time sequence of timing task processing. Timing task processing is a common requirement of application, which means application asks the specified task to perform at expected moment. This paper focuses on the issues happening when timing task processing is implemented with standard Java timer, and finds a way to fix the issues. The rest of this paper is organized as following. Section 2 introduces the issues and gives its root cause. Then section 3 presents an improved method based on queue and delay feedback. Tests are done in section 4 and conclusion is drawn in section 5.
2. Issue Analysis

2.1 Requirements on Timing Task Processing
In military field, applications have special restrictions on timing task processing [2]. Here are three typical restrictions.

1) Controllable execution order. For all tasks, the actual execution order should consist with the order of expected start moments. If one time period is applied for by several tasks, the rule is first come first served.

2) Controllable start moment. The error between actual start moment and expected start moment could be controlled in a certain range.

3) Controllable running duration. The time from start to end of a task should be controllable. Timing task is generally small task, which is expected to complete in a certain time, so its running duration is basically fixed and evaluable.

Due to above three 'under control' requirements, timing task is often called time-controllable task in some military applications.

2.2 Issues with Standard Java Timer
Standard Java provides pre-defined classes to support timing task processing [3-5], such as Timer, Timer Task, and Scheduled Executor Service, etc. Timer tries to trigger the execution of timing task at expected moment. Some military applications often create dozens of timing tasks in a very short time, part of which may have very close or even identical expected start moments. Such tasks are called Dense Timing Tasks in the paper. Firstly tests show that, when handling dense timing tasks Java runtime environment (shorten as runtime in the following) is unable to ensure the correct execution order, i.e., the actual order does not consist with the expected order. Secondly the error between actual start moment and expected start moment enlarges several times than normal runtime system error, which means timing accuracy decreases. Finally, the running duration of some tasks sometimes becomes several times of normal as well. In this case, the abnormal task starts earlier but ends later than other tasks. Above issues, especially the disorder issue, lead to the malfunction of application's business logic, and further cause the wrong processing outputs.

2.3 Root Cause Analysis
To root cause above issues, the principle of how timing task processing is implemented in standard Java is analyzed. At the beginning, a timer thread is created by runtime, which continuously monitors the time change. When expected start moment arrives, runtime creates a new task thread or pick up an available thread from threads pool to carry out the timing task. If the time period is occupied by several tasks, these tasks need to run concurrently. Java uses multi-thread mechanism to handle concurrent tasks, but obviously multi-thread mechanism could not ensure which task thread will start first, to say nothing of the execution order of all the timing tasks. And since all threads are scheduled by operating system, the actual start moment of a specific task is unpredictable and varies randomly in a range, which causes additional start moment error. In addition, if a task thread could not finish in one time-slot assigned by operating system, then this thread will be suspended until its next scheduled time-slot. This explains why at times some task's running duration is prolonged several times. All above bring the uncertainness to timing task processing, and deviate from the requirements.

In a word, handling timing task with multi-thread mechanism is the root cause, which explains why standard Java could not satisfy some applications with strict time sequence requirements. However for traditional C based applications in Linux, there are no such issue. The reason is that, C does not implement timing function by itself, but depends on the timing function in the kernel of operating system.
Linux kernel uses signal technique for timing task processing, which does not allow concurrency, and does not make any scheduling if a task is still running. In addition, the timer of in Linux kernel is much more accurate than the Java's timer implemented in application layer [6]. So all the three 'under control' requirements for timing task processing could be met.

3. Solution

3.1 Idea of Improvement

Since multi-thread behind timer introduces the uncertainness of time sequence, standard Java timer is not suitable for dense timing task processing [7-8]. The solution proposed in this paper is to wrap Java timer, and construct a timing task manager, in which queue and delay feedback is used for eliminating uncertainness [9-10].

The timing task manager maintains a queue composed of all the unexecuted timing tasks. The queue is ordered by expected start moments of tasks, and its head is the first one to start. Only the head owns Java timer resource. Others in the queue have no timer assigned, and will not be automatically triggered to run by runtime. The expected start moment of the head is the only timing moment managed by runtime. When the moment arrives, the head is trigger to run. It's removed from queue after it's finished, and the Java timer resource is passed to the new head. When a new task is added, it should be inserted to proper position. Since only the head owns timer resource, at most one timing task could be running at any time. In this way disorder and abnormal running duration issues are solved.

When application requests a new timing task via the timing task manager, the manager firstly checks if the expected time period has been occupied by other tasks. If not, the new task will be inserted into queue immediately. If occupied, the manager will figure out the latest available period after the original expected start moment, and feed the start moment of the available period back to application as the new expected start moment. If application accepts it, the task will be inserted into queue according to the new moment. Otherwise application may re-select a new expected start moment on the basis of its business logic, and above steps will be repeated. In this way, application could update the expectation of the task's start moment according to the feedback, and evaluate the effect brought by the predicted delay. The delay caused by conflict is unavoidable, but its length is basically knowable, effect is evaluable as well. So this kind of delay is controllable. While for delay introduced by multi threads racing, evidently it is random and uncontrollable, so makes a negative impact on application.

The precondition of delay feedback is the assumption that task duration is evaluable. If all the timing tasks of the application are short-time tasks with same type, then the timing task manager would be able to evaluate the duration accurately after several iterations of leaning. This assumption is true in some data processing military applications.

3.2 Implementation Flow Chart

According to above idea, Fig. 1 shows how it could be implemented. Fig. 1(a) shows the procedure of adding a new timing task, whose principle has been introduced in section 3.1. Fig. 1(b) describes how Java timer thread and timing task thread work with each to accomplish the task when the start moment arrives. Runtime changes the timer thread's state from blocked to ready when it's time for the queue head to start. After running, timer thread picks up an available task thread from threads pool, assigns it with the timing task of the head, and makes the task thread ready. After task thread is scheduled to run, it firstly finishes the timing task. Then the head is removed from the queue. If the just finished head task is a periodic task, a new task will be created and inserted into queue's proper position. Afterwards the timer of runtime is set according to the new head,
and task thread changes to blocked state. At last timer thread puts task thread to threads pool, and changes itself to blocked state as well, waiting for the next start moment.

Figure 1. (a) Flow chart of adding a task; (b) Flow chart of task processing when the starting moment arrives

4. Test Results
Test environment is designed as below. An application continuously creates 5 identical timing tasks at the beginning of a 200ms-long cycle, and the expected start moment of each task is just 20ms after application raises the request of that task. Timing task is composed of simulated data processing code. VMware is used to simulate two hardware configurations. Both are 2GHz CPU with 2G RAM, but one is single-core, and the other is dual-core.

Standard Java timer and the improved method proposed above are compared. Three performance indexes are cared about, i.e., execution order, running duration, and uncontrollable delay of start moment. The actual start moment and end moment of timing tasks are recorded in the log during test. And the log will be analyzed offline after test for index comparison.

The results of execution order test are shown in Table 1. As mentioned previously, multi-thread mechanism of standard Java timer could not ensure the execution order. For single-core case, the correct rate is 18%. For dual-core case, it falls further to only 2%, because dual-core increases the uncertainty
of threads racing. On the contrary, proposed method could achieve 100% correct rate in both single-core case and dual-core case.

The results of running duration test are shown in Table 2. With the proposed method, the variances of task running duration are 2.19ms in single-core case and 0.39 in duo-core case respectively, which means the task running duration is almost fixed on a certain platform. So the precondition of delay feedback could be satisfied. While for standard Java time, its variances are quite larger.

Table 1. Results of Execution Order

|                | Single-core |           | Dual-core |           |
|----------------|-------------|-----------|-----------|-----------|
|                | Standard Java | Proposed method | Standard Java | Proposed method |
| Correct rate   | 18%         | 100%      | 2%        | 100%      |

Table 2. Results of Running Duration

|                | Single-core |           | Dual-core |           |
|----------------|-------------|-----------|-----------|-----------|
|                | Standard Java | Proposed | Standard Java | Proposed |
| Average        | 16.0        | 16.1      | 11.7      | 10.1      |
| Variance       | 8.6         | 2.1       | 4.0       | 0.39      |

Table 3. Results of Uncontrollable Delay

|                | Single-core |           | Dual-core |           |
|----------------|-------------|-----------|-----------|-----------|
|                | Standard | Proposed | Standard | Proposed |
| Min            | 1         | 1         | 1         | 1         |
| Max            | 62        | 15        | 37        | 15        |
| Average        | 32.9      | 8.7       | 15.466    | 8.8       |

The results of uncontrollable delay test are shown in Table 3. The proposed method could achieve about 9ms average delay in both single-core and dual-core cases, which is less the usual 10ms threshold. For standard Java timer, in respect that tasks could be carried out concurrently, dual-core could save half time than single-core case, but its delay is still more than two times of proposed method.

Finally the proposed method is applied to a real military prototype to verify if it could fix issues in practice. The result shows timing caused data processing error doesn't occur any more.

5. Conclusion

To fix the issues in dense timing task processing with standard Java timer, an improved method is proposed. Queue is used to ensure the correct execution order, and delay feedback is used to reduce uncontrollable delay. The tests show that, proposed method could not only ensure tasks are executed in expected order, but also reduce uncontrollable delay evidently. So it meets the time sequence requirement of military applications.

References

[1] W.M. Zhang, H.H. Chen, X.S. Luo, and D.K. Guo. 2013.7. “Developing Integrated Electronics Information System,” National University of Defense Technology Press: 158-162.

[2] Z.G. Zhao, B.F. Jiang, K. Wang, Y. Xu. 2012.11. “Concept, Method and Application of Information Fusion,” National Defence Industry Press: 120-131.

[3] G. Li. 2012.6. “Crazy Java Lecture (Second Edition),” Publishing House of Electronics Industry:
530-548.

[4] Bruce Eckel. 2007.6. “Thinking in Java (Fourth Edition),” China Machine Press: 324-331.

[5] Geolo. “The Principle of Java Timer,” [Online] Available: http://blog.csdn.net/geolo/article/details/6455503.

[6] Wolfgang Mauerer. 2010.6. “Professional Linux Kernel Architecture,” Posts & Telecom Press: 289-312.

[7] Amei Zhang. The Design and Application of Paperless Examination System. International Information and Engineering Association. Proceedings of 2018 2nd International Conference on Systems, Computing, and Applications(SYSTCA 2018) 2018:4.

[8] Biqing Li, Zhao Li, and Suping Jiang."Realization of Music Fountain Control system based on single Chip Microcomputer".Proceedings of 2018 International Conference on Network, Communication, Computer Engineering (NCCE 2018).Ed.. Atlantis Press, 2018, 320-326.

[9] Weining Kang, Hongyuan Lu, Guodong Pang. Fluid approximations of many-server queues with delayed feedback and heterogeneous service and patience times. ACM SIGMETRICS Performance Evaluation Review, 2014,41(3).

[10] P Rajadurai, K Indhira, M C Saravanarajan, V M Chandrasekaran. Analysis of an M[X]/G/1 Feedback Retrial Queue with Two Phase Service, Bernoulli Vacation, Delaying Repair and Orbit Search. Advances in Physics Theories and Applications, 2015,40.