Java VM の実行時情報に基づくデッドロック検出ツール*

4M−1

野中 裕介†　牛島 和夫‡
九州大学大学院システム情報科学研究科§　(財) 九州システム情報技術研究所¶

1　はじめに

デッドロック問題は並行システムにおいて深刻かつ複雑な問題である。本研究では、対象となる Java 並行プログラムを実際に走行させることによって得られる実行時情報を用いた、デッドロック動的検出法を確立することを目的とする。

実用性という観点から、理想的なデッドロック動的検出法が満たすべき要件は、(1) 完全性: 任意のデッドロックを検出できること、(2) 健全性: 存在しないデッドロックを検出しないこと、(3) 効率性: 現実的な時間と記憶域の量で検出できること、であると言える。

デッドロックの検出法に関する限り、Java の並行処理機構は、Ada 95 のそれと類似している点が多い。本論文では、Ada 95 を対象とした、完全性、健全性の点で最も優れているデッドロック動的検出法 [2] を、Java 特有の問題に対応することによって、Java に適用させたものを示す。また、Java VM から実行時情報を取得するという実装方法が、効率性の面で優れていることを示す。

2　デッドロック検出原理

Java 並行プログラムにおけるデッドロックとは、実行がブロックされたスレッド間の同期待ち関係が循環して、それらのスレッドがいずれも制御を進めることができない状態のことである。ここで、実行がブロックされたスレッドとは、1 つ以上の他のスレッドとの同期を待っていて、待ち状態が同期が発生するまで続くスレッドのことである。

スレッドの実行をブロックする原因となる、スレッド間の同期待ち関係は、以下の 4 種類に分類することができる。

通知待ち　オブジェクト B の wait() メソッドを、引数無し、もしくは 0 の引数で呼び出したスレッド A は、他のスレッドがオブジェクト B に対する notifyAll() メソッドを呼び出すか、notify() メソッドを呼び出して対象にスレッド A が選択されるまでブロックされる。このとき、スレッド A は、オブジェクト B に対して notifyAll() もしくは notify() メソッドを呼び出す可能性のあるスレッド群を通知待ちしている、と言う。

*A Deadlock Detector Based on Run-time Information in Java VM.
†Yusuke Nonaka, ‡Kazuo Ushijima
§Graduate School of Information Science and Electrical Engineering, Kyushu University
¶Institute of Systems and Information Technologies/KYUSHU

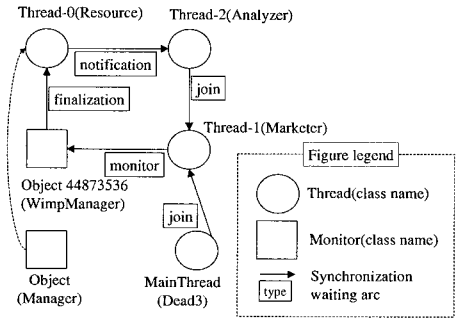

図 1　Java スレッド待ちグラフの例

合流待ち　スレッド A は、別のスレッド B に対する join() メソッドを呼び出したとき、スレッド B の実行が終了するまでブロックされる。このとき、スレッド A はスレッド B を合流待ちしている、と言う。

終了待ち　スレッド B によってロックが獲得されているオブジェクト A は、スレッド B がロックを解放するまで、他のスレッドからのメソッド呼び出しを受け付けない。このとき、オブジェクト A は、スレッド B を終了待ちしている、と言う。

モニタ待ち　他のスレッドによって既にロックが獲得されているオブジェクト B に対するメソッド呼び出しを行ったスレッド A は、オブジェクト B のロックを獲得するまでブロックされる。このとき、スレッド A はオブジェクト B をモニタ待ちしている、と言う。

Java 並行プログラムの実行中における、ある瞬間の実行状態、特にスレッド間の同期待ち関係を明示的に表現するモデルとして、Java スレッド待ちグラフ (Java Thread-Wait-For Graph, 以下 JTWFG) を提案する。

JTWFG は、枝が分類された有向グラフであり、各頂点は、スレッドもしくはロックが獲得されているオブジェクトに対応し、各枝は、同期待ち関係に対応する。枝は同期待ち関係と同様に 4 種類あり、同期待ち関係の種類と対応している。

図 1 は、あるプログラムの一実行状態における JTWFG の例を示している。この実行状態ではデッドロックが発生しており、図中、枝が循環している部分があることが確認できる。JTWFG 中に枝が循環している部分があることはデッドロック状態であるための必要条件であるが、十分条件ではない。サイクル中に通知待ち状態であるスレッドが含まれていて、そのスレッドが通知待ちを

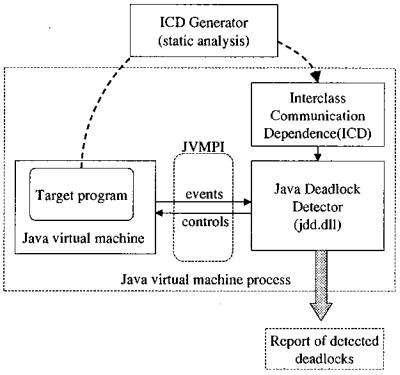

図 2 デッドロック動的検出ツールの構成

```
$java -Xrunjdd Dead3
JDD> initializing .....
JDD> .... ok

Deadlock will occur in this program!
(0.220000 seconds after starting.)
Elements of the deadlock :
Thread Thread-1(generated by Marketer) is waiting for
acquiring Monitor 44873536 (generated by WimpManager)
(monitor waiting).
Monitor 44873536(generated by WimpManager) is waiting
for finalization of Thread Thread-0(generated by
Resource) (finalization waiting).
Thread Thread-0(generated by Resource) is waiting for
notification from Thread Thread-2(generated by
Analyzer) (notification waiting).
Thread Thread-2(generated by Analyzer) is waiting for
completion of Thread Thread-1(generated by Marketer)
(join waiting).
(Now 0.220000 seconds after starting.)
```

図 3 デッドロック報告の例

しているサイクル外のスレッドが制御を進める可能性を残しているのであれば、そのサイクルの存在がデッドロックの存在を決定することはない。

3 デッドロック動的検出ツールの設計と実装

前節に示した原理に基づいて、Java 並行プログラムの実行時、デッドロックが発生する直前にその旨を通知して、デッドロックに巻き込まれているスレッドやロックが獲得されているオブジェクト、およびそれらの間の同期待ち関係を出力するツールの実装を行った。

図 2 がツールの概要を示したものである。JVMPI は Java VM 内の実行時情報を取得するためのインタフェースである。本ツールは JVMPI を利用しており、Java VM と同じプロセス内で動作する。JVMPI によって取得が可能な実行時情報は、前節で示した原理に基づくデッドロック検出法にとって十分なものであることを明らかにした。ただし、通知待ち関係を決定するためには、JVMPI から得られる実行時情報だけでは十分ではない。詳細は以下に述べる。

図 2 中の Interclass Communication Dependence(ICD) とは、各スレッドの性質を決定するクラス (Runnable インタフェースを実装したクラス、Thread クラスのサブクラス、main() を持つクラス) と、それらのスレッドによって notify() もしくは notifyAll() メソッドが呼び出される可能性のあるオブジェクトをインスタンスとするクラスとの対応関係を明示的に表現したものである。同期待ち関係のうちの 1 つである通知待ち関係を決定するためには、あるオブジェクトに対して wait() メソッドを実行して待ち状態になったときに、どのスレッドがその待ち状態を解放できる可能性があるのかを知る必要がある。そのために、本ツールでは、対象プログラムのソースコードに対する、実行前の解析を行う。図 2 中の ICD Generator がソースコードを解析して ICD を抽出する部分である。

図 3 は、図 1 に示したデッドロックが発生したときに本ツールが出力するデッドロック報告である。本例題について、プロセッサ Celeron 500MHz, メモリ 192MB, Windows Me, Java HotSpot Client VM (build 1.3.1-rc2-b23, mixed mode) の環境で、本ツールを使用した場合とそうでない場合の実行に要した CPU 時間を各 10 回測定し、平均を求めた。本ツールを使用した場合が 0.13 秒、使用しない場合が 0.10 秒であったので、本ツールを使用することによってこの場合には 1.3 倍の CPU 時間を要するようになることが分かった。自己計測原理 [1] によると、正確にデッドロック検出を行うためには、デバッグ時のみではなくリリース後も自己計測を続けなければならない。つまり性能低下の程度は重要であり、1.3 倍程度の増加であればシステムの信頼性を得るためであれば十分許容範囲であると我々は考えている。

4 おわりに

有向グラフ JTWFG に基づく検出法によって、Java 並行プログラムから実行時にデッドロックを検出するツールを開発した。

ただし、ICD を用いて求めた通知待ち関係が必ずしも正確ではないという問題がある。静的に知ることができる、クラス間の関係のみからでは、インスタンス間で動的に決定する関係を知ることができない。ICD を用いて単純に通知待ち関係を決定した場合、同期が発生する可能性の無いスレッド間の通知待ち関係を検出してしまう危険性がある。今後は、静的に解析した ICD に加えて、動的な実行時情報も利用して、通知待ち関係の、より精度の高い検出を行う方法を検討したい。

参考文献

[1] Cheng, J. : The Self-Measurement Principle: A Design Principle for Large-scale, Long-lived, and Highly Reliable Concurrent Systems, Proc. 1998 IEEE-SMC Annual Int. Conf. on Systems, Man, and Cybernetics, vol.4 (1998) 4010–4015

[2] Nonaka, Y., Cheng, J., Ushijima, K. : A Tasking Deadlock Detector for Ada 95 Programs, Ada User Journal, Vol. 20, No. 1 (1999) 79–92