B-021

# *Happened-Before* 関係による非分散マルチタスクプログラムに対する排他制御機構デッドロックの定義

# A Definition of Deadlock by *Happened-Before* Relation for Undistributed Multitask Programs

土江 晋哉† 大森 晃†
Shinya Tsuchie Akira Ohmori

## 1. はじめに

ソフトウェアにおける悩ましい不具合現象のひとつとして，デッドロックがある．デッドロックとは，ある複数のタスクが互いの動作の完了を待ち続けることによってプログラム自体の動作が停止してしまう現象である．

一般に，各モジュールによって実行されるタスク群が資源をロックする順序を設計書において決定していたにしても，設計書を基にして構築されるモジュール群によって実行されるタスク群については，以下の理由から，実際にデッドロックが発生するか否かを判断することは容易ではない．

(1)大規模なプログラムにおいては，各モジュールによって実行されるタスクがロックする資源をすべて把握することは容易ではないし，ましてや各タスクが資源をロックする順序をすべて把握することは容易ではない．

(2)一人の開発者が開発を担当するモジュール群によって実行されるタスク群がデッドロックを起こさないように，各タスクによって資源がロックされる順序が決められていたとしても，他の開発者たちが開発を担当するモジュール群によって実行されるタスク群との間でデッドロックに陥る可能性は否定できない．

こうしたことから，設計書において資源のロック順序が決められていても，実際にデッドロックが発生するか否かは，テストを行って確認する他はない．

デッドロックは極わずかなタイミングによって発生する．動的テストを行っても，デッドロックの発生する極わずかなタイミングが運よく発生することは稀である，何万時間もの長時間耐久テストを実施しても，運用を開始した日にデッドロックが発生することもある．動的テストの完全性のひとつの根拠としてコードカバレッジが利用される場合もある．しかしながら，100%のコードを実行したという実績ができたとしても，マルチタスクプログラムではそれぞれのタスクが時々刻々と状態を変化させるため，状態の網羅性については 100%に到達させることは難しい．

現在，プログラムの実行時にデッドロックを検出する手法（動的検出機構）が研究されており，実際に Linux カーネル等で実践されている．しかしながら，デッドロックの発生可能性を完全に排除したプログラムを作成すれば，動的検出機構は必要がない．

そこで有効に利用できるツールが静的コードチェックツールである．静的コードチェックツールはプログラムを実行することなく，プログラムの不具合を検出するツールである．シンタックスチェックやコードレビューによって検出される文法的な不具合はもちろんのこと，昨今ではセキュリティ不具合や競合状態等のような動的テストでも検出の難しい機能的不具合を検出することができるようになりつつある．精度はともあれデッドロックを検出できるツールも存在する．

静的チェックツールをテストに用いることの有用性は，不具合の検出が機械化されている部分にもある．テスト設計などの難しく時間のかかる作業を行うことなくデッドロックを機械によって検出できれば，人為的ミスも少なく品質を底上げすることが可能である．

デッドロックを静的に解析する（つまりプログラムを実行することなくデッドロックの発生可能性を解析する）にあたって求められることは，デッドロックが発生する可能性があるにもかかわらずデッドロックは発生しないとする検出漏れ，デッドロックが発生する可能性が無いにもかかわらずデッドロックは発生するとする誤検出など，デッドロックの検出ミスが無いことである．デッドロックの静的解析に対するこうした要求に応えるためには，まずデッドロックという現象を厳密に定義する必要がある．そこで本研究では，検出ミスのない静的解析を目指して，デッドロックの厳密な定義を与える．また，その定義に基づいて，既存の静的解析アルゴリズムの評価を行う．

## 2. デッドロックの種類

本研究では，デッドロックを同期・通信機構デッドロックと排他制御機構デッドロックの２種類に分類する．本節では，これらについて概説する．

### 2.1 同期・通信機構デッドロック

同期・通信機構とは，複数のタスク間で，動作の相互続行許可やメッセージの送受信を行う機構（フラグやメールボックス等）である．それぞれ，メッセージの送信・受信あるいは，動作の許可・許可待ちの処理が対になることが特徴といえる．

同期・通信機構によって発生するデッドロックとは，複数のタスクにおいて永久に来ることのないメッセージや続行許可を互いに待ち続けてしまい，プログラムが動作しない状況に陥ることである．

### 2.2 排他制御機構デッドロック

排他制御機構とは，マルチタスクプログラムにおける競合状態を回避するための制御機構（セマフォやミューテックス等）である．同期・通信機構と違い，どのタスクも排他制御機構を利用するときは，ロック処理を実行するのが特徴である．

†東京理科大学 Tokyo University of Science

```
1:void task1_main(void)
2:{
3: loc_mtx(r1);
4: loc_mtx(r2);
5: ;
6: unl_mtx(r2);
7: unl_mtx(r1);
```

```
1:void task2_main(void)
2:{
3: loc_mtx(r2);
4: loc_mtx(r1);
5: ;
6: unl_mtx(r1);
7: unl_mtx(r2);
```

図 1　デッドロックが発生するプログラム

排他制御機構によって発生するデッドロックとは，マルチタスクプログラムにおいて，複数のタスクがすでに確保されている資源をお互いに確保しようとして，当該の資源を確保できないままプログラムが動作しない状況に陥ることである．

図 1 は，マルチタスクのプログラム例である．本論文内に示すコードはμITRON4.0 仕様[1]に従うものである．

左側のコード列をタスク 1 が，右側のコード列をタスク 2 が実行するものとする．このプログラムでは，タスク 1 とタスク 2 の実行タイミングによっては，デッドロックが発生する．タスク 1 が 3 行目のコード“loc_mtx(r1)”を実行し資源 r1 をロックした後，制御がタスク 2 へ移行するとする．この時，タスク 2 は 3 行目のコード“loc_mtx(r2)”を実行し資源 r2 をロックすると，引き続き 4 行目のコード“loc_mtx(r1)”を実行し資源 r1 をロックしようとする．しかしながら，資源 r1 はタスク 1 によってすでにロックされているため，タスク 2 は待ち状態へ遷移する．タスク 2 が待ち状態へ遷移した後，実行可能状態にあるタスク 1 へ制御が移行するが，タスク 1 もまたタスク 2 がロックしている資源 r2 をロックすることができないため，タスク 1 もまた待ち状態へと遷移する．タスク 1 とタスク 2 は互いの処理の完了を待ち続けることとなり，プログラム全体が動作できない状況となる．このような状況が排他制御機構デッドロックである．

こうしたデッドロックが発生する可能性は，図 1 のプログラムを図 2 のように資源のロック順序を統一することによって排除することができる．

本研究で取り上げるデッドロックは，排他制御機構デッドロックに限定する．以降，特に断りなく「デッドロック」と表現する場合，排他制御機構におけるデッドロックのことを意味するものとする．

## 3. *Happened-Before* 関係を用いたデッドロックの定義

Lamport[2]は，分散システムにおいてメッセージの送信または受信をプロセス内でのイベントとして捉え，イベントの集合上に，以下の条件を満たす *happened-before* 関係”→”を定義している．

(1) もしイベント *a* と *b* が同じプロセスで発生し，イベント *a* がイベント *b* より前に発生するなら，*a*→*b*.
(2) もしイベント *a* があるプロセスにおけるメッセージ送信で，イベント *b* がその他のプロセスによる同じメッセージの受け取りであれば，*a*→*b*.
(3) 任意のイベント *a*，*b*，*c* について，もし *a*→*b* かつ *b*→*c* なら，*a*→*c*.

ここで，上記の *happened-before* 関係は分散システムに対する定義である．本研究で対象にしているのは非分散マルチタスクプログラムである．そのため，上記の定義における(1)と(2)を以下のように読み替える．

(1) もしイベント a と b が同じタスクで発生し，イベント a がイベント b より前に発生するなら，a→b.
(2) もしイベント *a* と *b* が異なるタスクで発生し，イベント *a* がイベント *b* より前に発生するなら，*a*→*b*.

このように読み替えた *happened-before* 関係を用いてデッドロックを定義する．ただし，本研究では最も基本的な 2 タスク 2 資源のプログラムにおけるデッドロックを定義する．デッドロックの定義に先立ち，以下の表記を導入する．

```
1:void task1_main(void)
2:{
3: loc_mtx(r1);
4: loc_mtx(r2);
5: ;
6: unl_mtx(r2);
7: unl_mtx(r1);
```

```
1:void task2_main(void)
2:{
3: loc_mtx(r1);
4: loc_mtx(r2);
5: ;
6: unl_mtx(r2);
7: unl_mtx(r1);
```

図 2　デッドロックが発生しないプログラム 1

- タスク $t_1$ による資源 $A$ のロック処理 $A_{Lock}$ をイベントとして，$e(t_1, A_{Lock})$と表記する．
- タスク $t_2$ による資源 $B$ のロック処理 $B_{Lock}$ をイベントとして，$e(t_2, B_{Lock})$と表記する．
- タスク $t_1$ による資源 $A$ のアンロック処理 $A_{Unlock}$ をイベントとして，$e(t_1, A_{Unlock})$と表記する．
- タスク $t_2$ による資源 $B$ のアンロック処理 $B_{Unlock}$ をイベントとして，$e(t_2, B_{Unlock})$と表記する．
- タスク $t_1$ による資源 $B$ のロック処理 $B_{Lock}$ をイベントとして，$e(t_1, B_{Lock})$と表記する．
- タスク $t_2$ による資源 $A$ のロック処理 $A_{Lock}$ をイベントとして，$e(t_2, A_{Lock})$と表記する．

**デッドロックの定義：**2 タスク 2 資源のプログラムにおいて，以下の条件が成立する状況をデッドロックと呼ぶ．

(1) $e(t_1, A_{Lock}) \rightarrow e(t_1, B_{Lock})$
(2) $e(t_1, B_{Lock}) \rightarrow e(t_1, A_{Unlock})$
(3) $e(t_2, B_{Lock}) \rightarrow e(t_2, A_{Lock})$
(4) $e(t_2, A_{Lock}) \rightarrow e(t_2, B_{Unlock})$
(5) $e(t_1, A_{Lock}) \rightarrow e(t_2, A_{Lock})$
(6) $e(t_2, B_{Lock}) \rightarrow e(t_1, B_{Lock})$

以上の条件を，条件番号に対応付けて図解したものが図

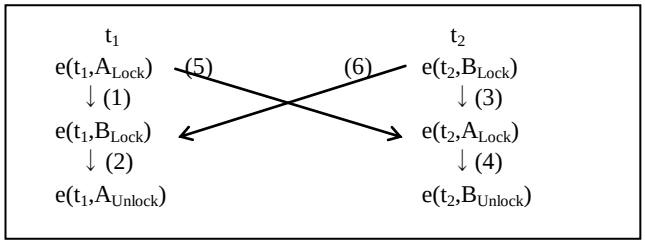


図 3　デッドロックが発生する *happened-before* 関係

3 である．これら(1)～(6)の条件が全て成立すると，タスク $t_1$，$t_2$ ともに，永久に互いが確保済みの資源を待ち続ける状況に陥る．

## 4. *lockset* アルゴリズムの評価

Savage[3]の *lockset* アルゴリズムを利用して Engler ら[4]は静的解析によって排他制御機構デッドロックの検出機能を実現している．しかしながら，未だ誤検出が避けられない状況である．誤検出があるだけで，そうした機能による検出結果は信憑性に欠けてしまう．ここでは，Engler ら[4]の *lockset* アルゴリズムを，前節で与えたデッドロックの定義に照らして評価する．

*lockset* アルゴリズムでは，まず各タスクについて，その出口点でのロック集合（以下，タスク t について Lockset(t)と記す）を作成する．これは，各タスクにおいて，各出口点に到達する経路上でロックされる資源を集めた集合の和集合である．例えば，図 4 における左側コード列を実行するタスク 1（これを t1 と記す）のロック集合は Lockset(t1)=｛r1, r2｝であり，右側コード列を実行するタスク 2（これを t2 と記す）のロック集合は Lockset(t2)=｛r1｝である．Engler ら[4]は，ロック集合上で，資源のロック順序関係を記号”→”によって表現し，例えば，(r1→r2)∧(r2→r1)といった閉路が形成される際にデッドロックが発生する可能性があるとしている．先に導入した *happened-before* 関係の表記との混乱を避けるため，以降，ロック集合上での資源のロック順序関係には記号” ➪”を用いる．

図 4 のプログラムにおいては，t1 はロック集合として Lockset(t1)={ r1, r2 }を有し，Lockset(t1)上では(r1➪r2)∧(r2➪r1)であることから，Lockset(t1)における資源のロック順序について，(r1➪r2)∧(r2➪r1)という閉路が形成され，*lockset* アルゴリズムによれば，デッドロックの可能性が検出される．しかしながら，図 4 のプログラムにおいては，実際にはデッドロックは発生することはなく，これは誤検出である．

ここで，前節で与えたデッドロックの定義におけるすべての条件が図 4 のプログラムにおいて満たされる可能性があるか否かを検討する．その際，当該定義における資源 A と B をそれぞれ r1 と r2 に読み替える．

(1) $e(t_1, A_{Lock}) \rightarrow e(t_1, B_{Lock})$：この関係はタスク 1 における 3 行目から 4 行目へのパスとして存在するので，満たされる．
(2) $e(t_1, B_{Lock}) \rightarrow e(t_1, A_{Unlock})$：この関係はタスク 1 における 4 行目から 6 行目へのパスとして存在するので，満た

```
1:void task1_main(void)
2:{
3:  loc_mtx(r1);
4:  loc_mtx(r2);
5:  ;
6:  unl_mtx(r1);
7:  ;
8:  loc_mtx(r1);
9:  unl_mtx(r2);
```

```
1:void task2_main(void)
2:{
3:  ;
4:  loc_mtx(r1);
5:  ;
6:  unl_mtx(r1);
7:
8:
9:
```

図 4　デッドロックが発生しないプログラム 2

される．

(3) $e(t_2, B_{Lock}) \rightarrow e(t_2, A_{Lock})$：タスク 2 では資源 r2 のロックが存在しないので，この関係は常に満たされない．

(4) $e(t_2, A_{Lock}) \rightarrow e(t_2, B_{Unlock})$：タスク 2 では資源 r2 のアンロックが存在しないので，この関係は常に満たされない．

(5) $e(t_1, A_{Lock}) \rightarrow e(t_2, A_{Lock})$：この関係は，タスク 1 における 3 行目からタスク 2 における 4 行目への制御の移行がタイミングによってはあり得るので，満たされる可能性がある．

(6) $e(t_2, B_{Lock}) \rightarrow e(t_1, B_{Lock})$：タスク 2 では資源 r2 のロックが存在しないので，この関係は常に満たされない．

以上のように前節で与えたデッドロックの定義に照らせば，図 4 のプログラムにおいては，常に満たされない条件が存在するため，デッドロックは発生しないと判断できる．このことは，*lockset* アルゴリズムが誤検出するデッドロックの発生可能性を，本研究で与えたデッドロックの定義によって排除できることを意味している．*lockset* アルゴリズムとの間でこうした差異が生じるのは，*lockset* アルゴリズムにおいては資源のロックがどのタスクで行われるかの区別が明示的になされていないことによる．*lockset* アルゴリズムによるデッドロックの誤検出は，これを原因として生じるものと考えられる．

## 5. おわりに

本研究では，2 タスク 2 資源のプログラムにおけるデッドロックを，もともとは分散システムに対して定義されている *happened-before* 関係を非分散マルチタスクプログラム用に読み替えた *happened-before* 関係によって，定義した．今後この定義を一般化する方向で，タスク数と資源数を拡張する必要がある．

さらに本研究では，簡単なプログラム例を対象にして，Engler ら[4]の *lockset* アルゴリズムを評価した．その結果，デッドロックが発生する可能性のないプログラム例について，*lockset* アルゴリズムはデッドロックの発生可能性を誤検出するが，本研究で与えたデッドロックの定義によれば，そうした誤検出を排除できることを示した．このことは，当該定義の有用性を示すものである．

また，本研究では対象とするデッドロックを排他制御機構デッドロックに限定した．しかしながら，実際には，排他制御機構と，同期・通信機構による複合的なデッドロックが発生する可能性もある．例として図 5 のようなプログラムにおいて右側のタスクによって資源 r1 が確保されてしまうと，どちらのタスクも動作しない状況に陥ってしまう．これらを設計時に把握することは非常に難しい問題である．同期・通信機構デッドロックについても検討を行い，こうした問題も含めたデッドロックの静的解析手法の検討が必要であると考える．

**謝辞**

東京理科大学大学院工学研究科経営工学専攻の仁木直人教授には，本研究について参考となる指摘を頂いたことにお礼申し上げます．

参考文献

[1] 高田広章，“μITRON4.0 仕様”，社団法人トロン協会 (2006).

[2] Lamport Leslie, “Time, clocks, and the ordering of events in a distributed system”, Communications of the ACM, Vol.21，No.7，pp.558-565 (1978).

[3] Savage Stefan., “Eraser: A Dynamic Data Race Detector for Multithreaded Programs”, ACM TOCS，Vol.15，No.4, pp.391-411 (1997).

[4] D. Engler and K. Ashcraft., “RacerX: Effective,Static Detection of Race Conditions and Deadlocks”, Proc. of the 19th ACM symposium on Operating systems principles，pp.237-252 (2003).

```
1:void task1_main(void)
2:{
3: loc_mtx(r1);
4: flag = 1;
5: unl_mtx(r1);
```

```
1:void task2_main(void)
2:{
3: loc_mtx(r1);
4: while(flag!=1);
5: unl_mtx(r1);
```

図 5　複合的なデッドロックが発生するプログラム