法政大学学術機関リポジトリ
HOSEI UNIVERSITY REPOSITORY

PDF issue: 2025-07-06

ソフトウェアトランザクショナルメモリを使用した予約処理の並行化

加藤, 公一 / KATO, Koichi

(発行年 / Year)
2013-03-24
(学位授与年月日 / Date of Granted)
2013-03-24
(学位名 / Degree Name)
修士(理学)
(学位授与機関 / Degree Grantor)
法政大学 (Hosei University)

ソフトウェアトランザクショナルメモリを使用した予約処理の並行化

Parallel Reservation Processing Using Software Transactional Memory

加藤　公一

Koichi Kato

法政大学情報科学研究科情報科学専攻

E-mail: koichi.kato.8z@cis.k.hosei.ac.jp

Supervisor: Prof. Kenji Ohmori

目次

1. 序論 4

2. Software Transactional Memory

2.1. Clojure 5

2.2. Clojure の **STM** 6

3. 追い越し 8

3.1. 追い越しが予約処理へ与える影響 9

3.2. 追い越しについての結論

4. 座席予約処理の仕様

5. 設計指針と実装方法 10

5.1. ログ機能の付加

5.2. トランザクションの適用範囲

5.3. 中断処理 12

6. 実験概要 15

6.1. ツール 16

6.2. 実験

6.2.1. 並行処理関数による動作実験

6.2.2. web アプリケーションとしての動作実験 22

6.3. 実験結果 23

6.4. 考察 24

7. 結論

8. おわりに 25

Abstract

Although the necessity for parallel processing is increasing by spreading of multicore processors or Cloud, the problem peculiar to parallel processing has still faced us greatly. There are dangers of competition among data, deadlock, and livelock when we build parallel processing using low-level-locks. However, Communicating Sequential Processes(CSP) and Software Transactional Memory(STM) are available as technologies to resolve the problems. As CSP's description is simpler, we should only describe what event will happen and which event will be interconnected. Using JCSP, we can implement a CSP model in Java. The description will be also simpler. But we will face complex code in other part of a CSP model using Java. Moreover, CSP takes a message-passing form that makes us wait when many processes access to one process. In contrast, STM takes an optimistic non-blocking parallelism form that does not make us wait. Instead of wait, a retry will occur. Therefore, we have chosen STM and we consider implementing parallel processing using Clojure. Clojure has STM as language specification, and makes our description simpler than Java, and also can use huge Java library. By explaining seat reservation processing as an example the guideline for constructing parallel processing using Clojure STM is shown.

1. 序論

マルチコア化やクラウドの普及により並行処理の必要性は高まって来てはいるものの，並行処理特有の問題は未だ大きく立ち塞がっている．抽象度の低いレベルでロックを使用して並行処理を構築しようとすると，データの競合やデッドロック，ライブロックなどが発生する危険がある．しかし，それらの問題は，Communicating sequential processes(CSP)[1]や Software transactional memory(STM）[3,4]を利用することで抽象度の高いレベルで解決することができる．CSP の記述は，相互作用を起こすプロセスの同士の接続構造とそれぞれのプロセスの状態遷移から構成され，簡明にできている．CSP の実装には Java で CSP の動作を実現するためのライブラリである JCSP を使用することが考えられ[5]，CSP モデルに関する部分では CSP での記述と同じような記述を行うことが可能であり，簡明に記述することができる．　しかし，それ以外の部分は Java で記述することになり，コードは複雑になる．また，CSP はメッセージパッシング方式なので，複数のプロセスが同じプロセスに対して読み取り，書き込みを行う場合，どちらを行ったとしても何らかの待ちが発生する．それに対して，STM は楽観的な並行性制御方式を取っていて，競合が起こればどちらかの処理をリトライするが待ちは起こらない．また，STM を言語仕様として持ち，同じ処理を Java での記述より短い記述で実現可能であり，さらに，膨大な Java のライブラリも使用可能な関数型言語 Clojure[2]がある．

そこで本論文では Clojure を利用した予約処理を検討する．座席予約処理を例にとって Clojure の STM の特徴を説明し，またその特徴により考えられる設計指針，また，Clojure による簡明な実装を示す．

本論文で予約処理を対象としたのは，複数の並列に動作する処理が共有する資源を獲得する処理で，並行性制御を必要とすること．また，図書館への書籍の予約，航空機の予約，公共施設の予約などさまざまな場所で必要とされる処理であり，適用範囲が広いためである．また，航空機の予約システムでよく知られる並行性制御を必要とする問題として，ダブルブッキング，オーバーブッキングがある．ダブルブッキングは，ある座席に対して同時に 2 人が予約処理を行おうとした場合に，並行性制御を行わずに処理を行うと 2 人の予約のどちらもが成功してしまう可能性があるという問題である．

2. Software Transactional Memory

STM はデータベースで使用されるトランザクションをメモリに対して適用した技術である．STM のトランザクションは，データベースのトランザクションに持たせる性質である

ACID のうち，ACI までを保証する．D が保証されないのはメモリに対しての処理のみに限定されているからである[2]．また，STM は楽観的な並行性制御方式を取っている．競合が起こらない限り他の処理によって自分の処理がブロックされることはなく，並列性は向上する．競合が起きた場合，競合するいずれかの処理がリトライされる．競合する処理が多いほど STM のパフォーマンスは悪くなるため[4]，STM を使用する場合には，競合のより少ない問題が適している．予約処理はその種類にもよるが，比較的競合の起こりにくい問題であり，また，ダブルブッキング，オーバーブッキングなどの，並行性制御を必要とする問題をはらんでいるため，STM を使用するのに適した問題であるといえる．

2.1. Clojure

Clojure は，同じコードを記述する場合に java による記述より簡明な記述が可能であり，また，STM を仕様として持つ関数型言語である．Clojure のコンパイラは Java で実装されていて，Clojure は JVM 上で動作する．したがって Java との親和性は非常に高い．膨大な Java のライブラリも独自の記述で簡単に使用することが可能である．さらに，Clojure の重要な特徴として変更不可能なデータ構造が挙げられる．変更不可能なデータ構造を使用して並行処理を行った場合，並行動作する他の処理によってデータが変更されることはないため，並行処理を記述するプログラマへの負担は軽減される．また，変更可能なデータを扱うことも可能で，変更可能なデータは ref として明示的に表される．ref への書き込みはトランザクション内でのみ行われ，STM により，同期的，協調的な変更が行われる．STM により並行性制御が行われるため，変更可能なデータを同時並行的に扱う処理を記述する場合にもプログラマへの負担は軽減される[2]．以下に変更可能なデータに対しての処理を Clojure で記述した例を示す．

```
(def data (ref 0))
(defn change [n]
  (dosync (ref-set data n)))
```

まず 1 行目で，初期値 0 の変更可能なデータを data として定義している．2，3 行目では，そのデータに対しての更新関数を change として定義している．実際の更新処理の記述は (ref-set data n)の部分であり，data に対して引数として取った n を新しい値としてセットする処理を記述している．これを包んでいる dosync はトランザクションによる処理を表している．これだけの短い記述で並行性制御によって守られた処理を実現できている．

2.2. Clojure の STM

まずは，Clojure の STM が行う更新処理の概要を述べる．Clojure の STM は多版型同時実行制御(Multiversion Concurrency Control，MVCC)と呼ばれる技法を使用している[2]．データへの書き込みを行う場合には，トランザクションによって更新されるデータの本体には変更は行われない．トランザクションは更新対象のデータのコピーをそのトランザクション内での固有の値として受け取り変更を行う．コピーのバージョンは受け取るトランザクションの開始時点のバージョンである．更新の完了が可能な場合には，更新対象であるデータの本体は更新が行われる前のバージョンとして保持され，変更を行ったコピーを最新のバージョンとして置き換える．したがってデータを読み取る場合には，競合は起こらない．最新のバージョンのデータを更新するトランザクションが並列に動作していたとしてもデータ本体は変更されず保持されているためである．競合は起こらないが，並列に動作するトランザクションによって更新が行われている場合でも更新前の値を読み取ってしまうため情報の伝播にはデメリットとなる可能性がある．

次に，競合が起こった場合のリトライに関して優先付けがどのように起こるか，また，リトライの起こるタイミングについて述べる．競合が起こった場合にどのトランザクションがリトライされるかは，タイムスタンプの順によって決まる．タイムスタンプはそれぞれのトランザクションの開始時に取得される．トランザクションにリトライが起こるタイミングは，トランザクションのコミット時，変更を加える ref にアクセスする時で，変更を加える対象のデータが他のトランザクションによって変更されている場合である．競合が発生するとリトライを繰り返すためオーバーヘッドが生じるが，競合を検知するまでは互いの処理をブロックすることがないので並行性は向上し，ある処理で何らかの中断が起こった場合には他の処理は並行に進行しているため，競合による影響は少なくなる．

図 1 に，Clojure の STM の動作例を示す．

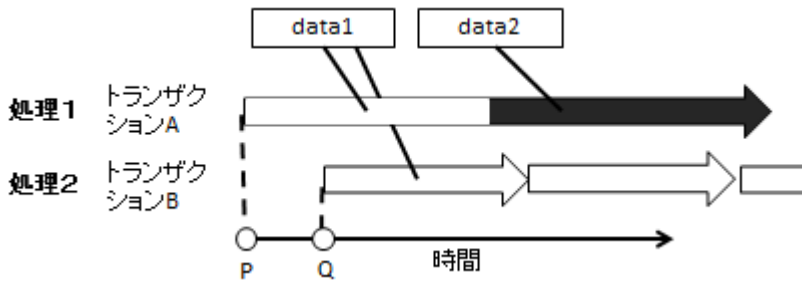

図 1　Clojure の STM の動作例.

図 1 の処理 1 の右側に位置している白と黒の混じった矢印は，2 つのデータに対して連続して処理を行うトランザクションを表している．白い部分は data1 に対しての処理を表し，黒い部分は data2 に対しての処理を表している．処理 2 の右側に位置している複数の白矢印は，それぞれ一つのデータに対して処理を行うトランザクションを表している．白矢印が複数並んでいる状況はリトライが複数回実行されていることを表現している．

図 1 ではトランザクション A が時間 P から開始している．つまり，P の時点でタイムスタンプが押される．トランザクション B は A に少し遅れた Q の時点でタイムスタンプが押される．トランザクション A は data1 から処理を開始し，data1 を更新した後に data2 の更新を行う．トランザクション B は data1 のみに対して処理を行う．ここで，トランザクション A は data1 に対し先行して処理を完了するため，トランザクション B はリトライすることになる．その後トランザクション A が data2 に対し更新を開始するが，トランザクション A が data2 への変更を完了しコミットするまで B はリトライし続けることになる．これは，タイムスタンプの順で先行しているトランザクション A が優先されているためである．

タイムスタンプの順で先行するトランザクションがデータを変更した場合，そのデータに対して変更を加えようとする他のトランザクションは，先行するトランザクションがコミットもしくはアボートされるまでリトライを繰り返すことになる．トランザクションを開始した時間が更新の順序を決める指標となる．

この例に関して注意しなければならない箇所がある．それはリトライ時の処理内容である．トランザクション B はリトライを繰り返しているが，トランザクションとして記述されたコードの内，data1 への変更を行う処理までのコードについて，リトライを繰り返している．data1 への変更を行う処理からトランザクションの終了，つまりコミットされるまでのコードの部分についてはリトライ時の処理には含まれない．Clojure の STM では，トランザクション内での処理結果が，コミットするまで他のトランザクションからは見えない仕組みになっているが，競合の検知はコミットする時点だけでなく，トランザクションの実行途中での ref へのアクセス時にも行われる．つまり，リトライ時の処理の単位がトランザクション処理の全体だけでなく，ref の実行箇所によって変動してしまうため，副作用を持つコードを記述する必要がある場合，記述する箇所によってリトライ時に再処理される箇所，されない箇所が出てくる．

たとえば，図 1 の例では，トランザクション A が先に開始され，トランザクション B が後に開始されているが，この順番が逆になった場合を考える．また，コードの区間について，トランザクション A の開始から data1 への変更を行う処理までのコードの区間を区間

P，data1 への変更を行う処理から data2 への処理を行うまでの区間を区間 Q，data2 への変更を行う処理からトランザクション A の終了までの区間を区間 R とする．この場合，トランザクション B が処理を完了し，トランザクション A が処理をリトライすることになるが，リトライが行われるのは区間 P を処理し，data1 への変更を行う処理の時点である．トランザクション A の処理内容は，区間 P を複数回行った後，区間 Q，区間 R を処理し処理の完了となる．この例から，一つのトランザクション内で複数の ref に対して変更処理を行う場合，区間 P のようにトランザクションの開始から ref に対しての変更を行う処理に始めて遭遇するまでの区間は，リトライが起こる場合，そのリトライがいずれの ref への変更処理に対して起こったものでも必ず処理される区間であること．区間 Q のように異なる ref への変更処理に挟まれた区間は，その区間の終わりに位置する ref への変更処理に対して起こるリトライ，またその後の区間があれば，そのそれぞれの区間の終わりに位置する ref への変更処理に対して起こるリトライを合わせた分リトライが起こる区間であること．区間 R のように最後に位置する ref への変更処理からトランザクションの終了までの区間は，一度だけ処理される区間であることが分かる．これらのそれぞれの区間の性質を考慮したコード記述が必要である．

3. 追い越し

Clojure の STM では，複数のデータをまとめて処理するトランザクションがあった時，そのトランザクションが変更の対象としているデータの内のあるデータに対して処理を行う前に，他のトランザクションがそのデータに対して処理を行うことで，複数のデータをまとめて処理するトランザクションがリトライする場合ある．これを追い越しと呼ぶことにする．たとえば，データ A，B に処理を行うトランザクション X，Y，Z があり，X は A，B の順で処理を行い，Y は B にのみ処理を行う．また，トランザクションの開始順は X，Y の順で行うとする．この時，Y が B への処理を X が B へ処理を行うよりも早く終えると，Y はコミットを成功して終了し，その後で X が B に対して処理を行おうとすると，STM が競合を検知してリトライが起こる．さらにこの時点で Z が発生し，B への処理を X が B に対して処理を行うよりも早く終えた場合，X にはさらにリトライが起こることになる．同じような状況がさらに続くと X はリトライ回数が上限に達するまで処理をリトライすることになる．

図 2 に，上述した例を示す．

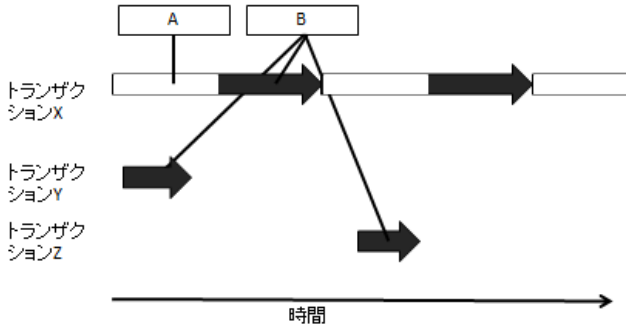

図 2　追い越しの例.

3.1. 追い越しが予約処理へ与える影響

予約システムの中にも，予約対象を複数選択することが可能なものがある．たとえば，航空機の座席予約システムでは，複数の座席を選択し，それらの席をまとめて予約することが可能である．より複数のデータにアクセスするトランザクションほど，そのアクセスするデータの数だけ，より複数のトランザクションと競合を起こす可能性が高まる．したがって，終了までの間に割り込まれやすく，また，追い越しを受けやすい．予約処理では，追い越された場合にはそのデータに対しての予約処理は不可能になってしまうためトランザクションを中断し，別のデータに対してアクセスを実行することを考える．この場合は，影響は薄いと考えられる．

3.2. 追い越しについての結論

予約処理は中断処理を行うため，予約処理での影響は比較的少ないと考えられるものの，不利益は発生する．また，リトライを繰り返し，変更処理を実行しなければならないような処理では，処理が長引けば長引くほど，よりトランザクション同士の競合が起こる可能性が高まると考えられるため，単位時間当たりの同時実行者数がより少ない問題を対象とするのが望ましいと考えられる．

4. 座席予約処理の仕様

本論文では，座席データとその更新関数，座席データを格納するテーブルを持つ座席予約処理について考える．

座席データは，座席データを識別するための id，座席が空き状態，予約状態のいずれの

状態であるのかを示す state を持つ．座席データは STM による並行処理により処理されることを想定しているため ref として定義し ref の内容は id，state，resid を持つマップである．また，複数選択時の処理を考慮することにする．すべての座席データは allchair に格納される．

更新関数として，座席の作成を行う関数，座席の状態を切り替える関数，座席に対してログを付加する関数，また，座席データの状態を確認するために，座席が予約状態かを判別する関数を用意する．

5. 設計指針と実装方法

5.1. ログ機能の付加

ログは並行処理の実行状況を観測する以外に問題領域に関する分析にも有用である．ここでは，ログ機能を付加する方法を説明する．Clojure には ref に対してその更新を検知する機能を付加することができる add-watch という関数がある．これを利用して ref の更新に連動して変更前の値を取り出しログとして保存する機能を作ることができる．以下にその例を示す.

```
(defn setlog [mref log]
  (add-watch mref nil
    (fn [_key mref _old _new]
      (swap! log conj _old))))
```

mref はログ機能を付加する対象の ref，log はログを保存するための変数，_old は ref に更新が起こった場合の更新前の値である．mref として入力された ref が更新された時，log として入力された変数に入っていた値に ref の更新前の値をつなげて log に保存している．

5.2. トランザクションの適用範囲

本論文での座席予約処理では複数選択時の処理を考慮する．ここでは，複数データ処理時のトランザクションの適用範囲について説明する．複数のデータを処理する場合にトランザクションでデータを包み処理を行うが，すべてを一つのトランザクションで包むか，それぞれ別のトランザクションで包むかの 2 通りが考えられる．どちらを選択するかはデータ間の関連性と予約処理の方式によって決められる．

まずは，データ間に関連がある場合について考える．会計処理のようにデータが階層構造を成していて，ある勘定科目を更新する時にその上位の勘定科目を更新する必要があるようなデータ間の関連性がある処理では，複数のデータを一つのトランザクション内で更新する一貫性の保たれた処理が必要である．つまり，関係性のある複数のデータを更新する場合には一つのトランザクションにまとめて行う方法が必要である．

次に，データ間に関連がない場合について考える．本論文での座席予約処理では座席データ同士に関連性はない．他の座席データに依存せず，それぞれの座席データごとに状態の変更，ログの追加が行われる．このような関連性のない問題に対しては，上述した追い越しによる複数データを扱うトランザクションへの不利益を防ぐため，それぞれのデータに対して一つずつトランザクションを割り当て，STM の衝突検知能力のみを利用する方法も考えられる． データ間に関連性のない場合には，要求される予約処理の方式によってトランザクションによってデータを包む範囲を選択することができる．たとえば，座席を複数指定し，決定ボタンを押した時に座席のすべてが予約できなければすべての予約を取り消す方式を要求される場合は，一貫した処理が必要となるので，複数のデータを一つのトランザクションで包む方法を選択する．席を選択するごとに予約を確定する方式を要求される場合はそれぞれのデータに対して一つずつトランザクションを割り当て，STM の衝突検知能力のみを利用する方法を選択する．

図 3 に，一つのトランザクションによって複数のデータを扱う場合の例を示す．

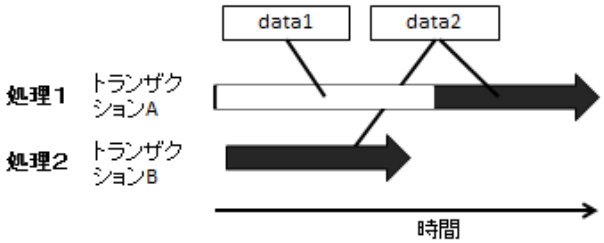

図 3　一つのトランザクションによって複数のデータを扱う場合の例.

図 3 では追い越しが起こる．処理 1 のトランザクション A が data2 にアクセスする以前にトランザクション B の処理は完了し，data2 に対しての更新が適用される．その後でトランザクション A が data2 更新したとしても，トランザクション A のコミット時に競合が検知されリトライが起こる．リトライにより，トランザクション A は data2 だけでなく data1 の分も再び処理を行うことになる．

図 4 に，複数のデータごとに別個のトランザクションによって処理を行う場合の例を示

す.

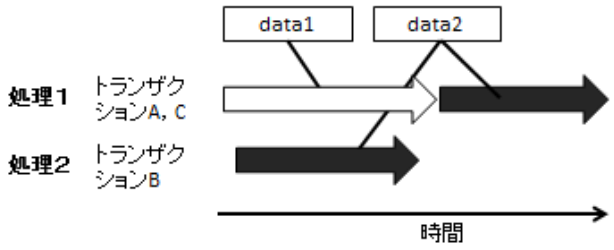

図 4　複数のデータごとに別個のトランザクションによって処理を行う場合の例.

図 4 以前の図では，複数の矢印記号が並んでいる場合はリトライを表していたが，今回は白い矢印がトランザクション A，黒い矢印がトランザクション C を表していて，2 本の矢印記号によって 2 つのトランザクションを表している．図 4 では追い越しが起こった場合のリトライ箇所を削減している．トランザクションは分割されていて，トランザクション C が data2 にアクセスした時にはトランザクション B の処理は終了している．トランザクション B の処理の終了がトランザクション C の開始時間よりも後に行われた場合にはリトライが起こるが，処理は data2 から始まり，data1 についての変更に関しては確定しているため，data1 に対するリトライ分が削減されると考えられる．しかし，その代わりにアトミック性を失うことになる.

5.3. 中断処理

予約処理では他のトランザクションによって予約処理が成功した場合，リトライ中のトランザクションを中断させる必要がある．たとえば，ある座席に対して 2 つのトランザクションが予約処理を行おうとした場合，中断処理を行わなければ，1 つ目のトランザクションが更新を終えた後，リトライにより待たされていたもう一つのトランザクションによって更新処理が行われる．これにより，予約された状態のデータに対してさらに予約処理を行ってしまう．つまり，1 つのデータに対して複数回同じ種類の更新処理が行われてしまう可能性がある．今回の予約処理では予約処理は空席の座席データに対して 1 回のみ行われるべきであり，複数回の同じ種類の処理の実行は許されず，同時に起きた並列に動作する処理については中断されるべきである．中断処理を行うことにするが，処理の配置位置について考えなければならない．Clojure の STM では 2.2 章で述べたようにリトライ時の処理内容がトランザクション内の区間によって違ってくると考えられるためである．中断処理の配置位置を考慮，つまり，どのタイミングで中断処理を行うかを考慮しつつ，トラ

ンザクションの中断方法としてそれぞれタイミングの異なる 3 通りの方法を考えた.

まず考えたのは，他のトランザクションの処理の成功をリトライしながら待ち，更新を確認してから中断を行う方法である．この方法は，Clojure の STM の実装内の機能を使用することで ref へのアクセス時に行われるため，もっとも更新処理に近いタイミングで中断を行う方法であり，また，中断条件が ref の値に依存している中断方法である．この方法のメリットは，他のトランザクションの処理が失敗した場合に即座に対応することが可能な点である．逆にデメリットとして，他のトランザクションの処理が成功した場合，リトライをしながら待たされた分の処理が無駄になる場合がある．この中断処理は ref に対してバリデーション関数を付加することで実現できる． バリデーション関数自体は真偽値を返す関数としてプログラマが定義する．そして，ref の生成時に引数として渡すか，set-validator!関数によって ref に付加する．付加されたバリデーション関数は STM 内部で処理を中断するかどうかの判断基準として使用され，ref が新しい値によって更新される場合に，その更新される新しい値が ref に対して妥当な値であるかどうか評価し，もし妥当でないと判断される場合にはエラーを投げる．以下にバリデーション関数によって中断処理を実現した例を示す.

```
(def before (atom -1))
(defn same? [n]
  (not= @before (:state n)))
(def data (ref {:state 0} :validator same?))
(defn setlog [mref log]
  (add-watch mref nil
    (fn [_key mref _old _new]
      (reset! log (_new :state)))))
(setlog data before)
```

same?は ref の現在の値と入力された n が同じ値であるかどうかを調べる関数である．before には data が更新される度にその値がログとして保持される．同じ値であれば false を返し，違う値であれば true を返す．data として定義された ref に対して same?をバリデーション関数として渡している．これにより，ref が更新される時，same?によって更新後の値と更新前の値が比較され，同じ値であればトランザクションは中断される．setlog はログ機能を付加する時に使用したものと構造は同じで，log を更新する関数部分で更新前の値である_old ではなく更新された新しい値である_new から値を取り出し，log に保存してい

る．このバリデーション関数を付加したことにより，ref をトランザクション内で実行した場合，他のトランザクションの処理の成功をリトライしながら待ち，更新を確認してから中断処理を行うことができる．バリデーション関数に対して不適格な値を渡した時には，エラーとして即座にトランザクションが中断されるため，トランザクションが中断した場合の終了処理をトランザクションの外に記述する必要がある．

次に考えたのは，他のトランザクションのアクセスを検知した瞬間，つまり，リトライが行われた瞬間に中断を行う方法である．この方法の中断タイミングはトランザクションの先頭でリトライが起きた場合必ず処理される．また，中断の条件として評価される変数をトランザクションの外部に配置し，トランザクション内でその更新を行うことにより中断を実現するため，中断条件が ref の値に依存していない中断方法である．また，この方法は，他のトランザクションの処理内容や中断を考慮していない方法であるが，本論文での座席予約処理に関しては，リトライが行われるということが他のトランザクションによって座席が確保されたということに等しいのでこの方法も考えられる．この方法でのメリットは，他のトランザクションをリトライしなが待つことは無く，リトライが起これば即座に他の処理を行うことのできる状態に移ることができる点．この中断処理はトランザクションの実行回数を保存する変数を用意し，分岐によって終了処理を行うことで実現できる．以下にトランザクションの実行回数を保存する変数として atom を使用した一例を示す．atom は ref のように変更可能な値であるが，非強調的な更新を行う．atom を使用した理由は，他の処理と強調して動く必要がないためである．

```
(defn oneretry []
  (let [retry (atom 0)]
    (dosync
      (reset! retry (inc @retry))
      (if (> @retry 1)
       終了処理
        更新処理))))
```

初回の処理では ref の更新処理が実行され，リトライによる 2 回目の処理では終了処理が実行される．リトライの実行数を変更することで複数回リトライした後に中断することも可能である．局所変数として retry を atom として定義し，トランザクション内の処理の実行回数を加算している．dosync 以下の部分がトランザクションで包まれている部分で，ロールバックによりリトライが行われる処理である． 上に示したコードでは，if 分岐でトラ

ンザクションの実行回数を確認していて，一度でもトランザクションによるリトライが起こると実行上限を上回り，終了処理に分岐するようになっている．

3 つ目に考えたのは，一つ目の中断処理のようにリトライした後変更を行わずに処理を中断する方法である．この方法は，リトライを繰り返すため，一つ目の中断処理と同じく複数データを一つのトランザクション内で処理する場合に，変更が行われた処理に対してもリトライによって再度処理し直すことで余分に処理を行う可能性がある．この中断処理は更新処理の内容によってさまざまな実現方法が考えられ，また，その中断タイミングもその実現方法によって違ってくる．本論文での座席予約処理での中断すべき状況は，直前に更新された値が予約状態である場合であり，単純な if 分岐と ref の値を参照することで以下のように実現した．

```
(if (= "res" (@chair :state))
  終了処理
  更新処理)
```

chair は座席データである．座席が空席の場合，更新処理に分岐し，予約済みの場合は終了処理を行い更新を中断する．

このような短い記述で並行処理を行うことができる理由は，衝突検知とリトライのタイミングにある．トランザクションは自分が開始した時点での更新対象のデータのコピーの値によって if で分岐する．他のトランザクションによって更新対象のデータが更新されていてもトランザクションの途中では見えない．衝突検知とリトライのタイミングは共にトランザクションがコミットされる時点，もしくは更新する ref へアクセスする時点で，外部からの影響はそれらの時点に集約されているため，開始からコミット，もしくは更新する ref へのアクセスまでの一連のトランザクション処理の流れの中で外部からの変更に影響されることはない．

6. 実験概要

ここまで説明してきた中断処理を Clojure を使用して実装し，実際に並行処理を正しく実現できるのか確認する．また，Clojure で記述された web アプリケーションフレームワークである noir，html の要素を Clojure のベクタで表現できるライブラリである hiccup を利用し，STM に対する処理をブラウザから起動する簡易な web アプリケーションとして動作させる．

6.1. ツール

Leiningen

Clojure でのビルドツール．プロジェクトや依存関係を管理し，必要なライブラリのダウンロードなども行うことができる．noir を利用するプロジェクトの雛形をコマンドを一つ実行するのみで構築することができる．

Noir

Clojure で記述された web アプリケーションフレームワーク．html を構成する view の部分とデータ処理を行う model の部分を記述する程度で web アプリケーションを構築できる．

6.2. 実験

一つのトランザクションで複数のデータを更新する場合，複数のデータのそれぞれについて別個のトランザクションで処理する場合，他のトランザクションの更新の完了を待ってトランザクション自体に中断させる場合，リトライが起こった瞬間に中断する場合，他のトランザクションの更新を待って中断処理を行う場合，これらそれぞれについて web アプリケーションの model として記述し実験する．

テスト方法は，それぞれの中断処理方法を実装したコードに対して Clojure の並列処理関数を使用し，遅延関数と組み合わせることで競合状態を作り出し，その状況下で構築した中断処理がどのように動作するかを確認する．また，実際に web アプリケーションとして実行される形式での動作も確認する．動作を確認したコードのそれぞれを model として web アプリケーションに組み込み，2 つのブラウザから，それぞれの中断処理を実現した web アプリケーションにアクセスし，並行性制御によってそれぞれのデータに対しての処理が期待通り行われるかを確認する．確認方法は，トランザクションに対して遅延を起こす関数，ref の更新の途中経過を出力する処理を更新処理に混ぜ込み，コンソールに出力することで確認する．テスト内容は，別々の席に対する予約処理を行った場合に並列に行われるか，同じ席に対する予約処理を行った場合に片方が予約処理を行い片方は中断されるか，複数選択を行い追い越しが起こる場合，追い越しが起こらない場合で，中断処理がどのように動作するかについてそれぞれ確認する．

6.2.1. 並行処理関数による動作実験

ここではこれまで説明してきた中断処理が実装されたコードについて説明する．

まず，コード間で共通する部分について説明する．座席データについて，座席を判別するための id を初期値 0 の ref として定義している．また，この id はすべての座席データを保持する allchair から座席データを引き出すためのインデックスであり，また，allchair 内の座席データの総数としても見ることができる．id の下で定義されているのは座席データの状態を表すパラメータである．座席が空席の場合は emp_c，座席が予約されている場合は res_c によって各座席の状態が表現される．allchair は新しい座席データを追加されていくため，ref として定義している．sorted-map として座席データを保持している理由は，それぞれの座席データに対してユニークな id を割り振っているためである．

座席データの下からは更新関数が記述されている．setlog 関数は前述したログを付加する関数である．res?関数は引数として与えられた id を与えられた座席データのその時点での状態が予約された状態かどうかを判別し，予約された状態であれば真を返す．reserve 関数は取る引数によって異なる予約処理を実行する．引数が 2 つの場合，一つの座席に対する予約処理を実行する．引数が 3 つの場合，複数の座席に対する予約処理を別々のトランザクションによって実行する．引数が 4 つの場合，複数の座席に対する予約処理を一つのトランザクションにまとめて実行する．

mkchair 関数は初期状態が空席の新しい座席データを作成し allchair に追加する関数である．実装した中断処理の内，2 つ目と 3 つ目の中断処理では同じ mkchair 関数を使用しているが．一つ目の中断処理に関しては，バリデーション関数を ref に対して付加する必要があったため，一部内容が異なっている．先に，2 つ目と 3 つ目の中断処理で記述されている mkchair 関数について説明する．allchair，id はどちらも ref として定義されているため，トランザクションによって包んで扱われる．作成される新しい座席データは空席の状態，ログを保持するためのフィールドをマップとして持ち，id とひも付けられて allchair に追加される．座席データが作成された次の行で setlog によってその座席データにログが付加される．次の行では次に作成される座席データの id を表すため，また，allchair の総数が加算されたことを表すために id の値がインクリメントされている．

次に，それぞれの中断処理によって異なる部分について説明していく．

まずは，1 つ目の中断処理の実装について説明する．1 つ目の中断処理ではバリデーション関数を使用するため，直前の値を保持しておく setbacklog 関数が用意されている．この関数は setlog 関数とほとんど同じで，保存されるログの形態と更新による最新の値を取得している所が違う．そして，setbacklog により更新されるログを保持する:backlog フィールドが mkchair 関数での座席データ作成時に追加されている．mkchair 関数では，バックログの付加に加えてバリデーション関数の付加も行っている．1 つ目の中断処理の説明ではバ

リデーション関数を ref を定義する時点で同時に付加していたが，ここでは set-validator!関数によってバリデーション関数を付加している．付加されている関数は，新しく更新される ref の状態が変更される前の ref の状態と等しくないか判断する．更新により変更された状態は直前の値としてバックログに保持されるため最新の ref の値のみで変更前の状態との比較を可能にしている．この比較により空席の座席データに対してのキャンセル処理，予約された座席データに対しての予約処理を検知して中断することができる．1 つ目の中断処理では ref に付加されたバリデーション関数によって中断すべきかの判別，中断がすべて行われるため，更新処理の本体は非常に簡単に記述できている．トランザクション内では座席データの状態への更新のみ行っている．

次に，2 つ目の中断処理の実装について説明する．3 つ目の中断処理と異なっている部分は更新処理の本体のみである．2 つ目の中断処理の説明で述べた方法をほぼそのまま使用している．少し違う点は，if の分岐条件である．3 つ目の中断処理とのハイブリッドのような構成になっている．

最後に，3 つ目の中断処理の実装について説明する．2 つ目の中断処理と同じく異なる部分は更新処理の本体のみである．1 つ目の中断処理と同じように更新処理の本体は非常に簡単に記述できていて，違いは if 分岐の有無のみである．

ここまで説明してきたコードを使用し，それぞれの設計によって並行処理が正しく実行されるかどうかをテストする．それぞれの中断処理に対して上述したように，

① 別々の席に対して予約処理を行った場合

(ア)`(pvalues (reserve 1 2) (reserve 2 1))`

② 同じ席に対して予約処理を行った場合

(ア)`(pvalues (reserve 1 2) (reserve 1 1))`

③ 複数のデータ更新をそれぞれ別のトランザクションで行い，追い越しが起こらない場合

(ア)`(pvalues (reserve 1 2) (reserve 1 2 1))`

④ 複数のデータ更新をそれぞれ別のトランザクションで行い，追い越しが起こる場合

(ア)`(pvalues (reserve 1 1) (reserve 2 1 3))`

⑤ 複数のデータ更新を一つのトランザクションで行い，追い越しが起こらない場合

(ア)`(pvalues (reserve 1 2) (reserve 1 2 1 0))`

⑥　複数のデータ更新を一つのトランザクションで行い，追い越しが起こる場合

(ア)(pvalues (reserve 1 1) (reserve 2 1 3 0))

これらの場合に対して，中断処理がどのように振舞うかについて確認する．中断処理の構造はそれぞれ違っているが，同じパラメータによる処理を実行して行う． pvalures は並列処理関数である．

遅延処理を行うために秒数を表す sec を引数として取っているが，sec はそれ以外にもトランザクションの名前付けにも使用している．flag は一つのトランザクションで複数のデータの更新をまとめて行う場合の処理を呼び出すために使用しているのみで値に意味はない．

まずは，1 つ目の中断処理の結果から説明する．

別々の席に対して予約処理を行った場合では，それぞれが並列に動いている．遅延時間を 1 秒に設定し，2 番目の座席に対し予約処理を行ったトランザクションが先に処理を終了し，遅延時間を 2 秒に設定し，1 番目の座席に対して予約処理を行ったトランザクションが次に処理を終了している．

同じ席に対して予約処理を行った場合では，片方の処理は処理を終了し，もう片方の処理はエラーを出して処理を中断している．このエラーはバリデート関数が出していて，バリデート関数によって同じ state への変更を行う処理がはじかれていることが分かる．遅延時間を 1 秒に設定した処理が先に処理を終え，その処理によって更新が行われたため 1 番目の席は予約状態になり，遅延時間を 2 秒に設定した処理は予約状態の席に対して予約をしようとしたため処理の中断が起きた．

複数のデータ更新をそれぞれ別のトランザクションで行い，追い越しが起こらない場合では，遅延時間を 1 秒に設定し，1 番目の座席と 2 番目の座席に対して順番に予約処理を行ったトランザクションは処理を終了している．遅延時間を 2 秒に設定し，1 番目の座席に対して予約処理を行ったトランザクションは中断されている．

複数のデータ更新をそれぞれ別のトランザクションで行い，追い越しが起こる場合では，遅延時間を 3 秒に設定し，2 番目の座席と 1 番目の座席に対して順番に予約処理を行ったトランザクションは，遅延時間を 1 秒に設定し，1 番目の座席に対して予約処理を行った

トランザクションに追い越しを受け，1 番目の座席に対しての予約処理を行うことができず中断したが，座席 2 に対しての処理は完了することができている．

複数のデータ更新を一つのトランザクションで行い，追い越しが起こらない場合では，遅延時間を 1 秒に設定し，1 番目の座席と 2 番目の座席に対して順番に予約処理を行ったトランザクションは 1 番目と 2 番目の座席への予約処理を終了している．遅延時間を 2 秒に設定し，1 番目の座席に対して予約処理を行ったトランザクションは中断している．

複数のデータ更新を一つのトランザクションで行い，追い越しが起こる場合，遅延時間を 3 秒に設定し，2 番目の座席と 1 番目の座席に対して順番に予約処理を行ったトランザクションは，遅延時間を 1 秒に設定し，1 番目の座席に対して予約処理を行ったトランザクションに追い越しを受け，1 番目の座席に対しての予約処理を行うことができず中断している．座席 2 に対しての処理もこちらの場合では完了することができていない．

次に，2 つ目の中断処理の結果について説明する．ここで，コロンの左側の数字は，トランザクション処理の実行回数を表している．つまり，数字が 2 となった場合にはリトライしたことを意味し，処理の中断が予想される．

別々の席に対して予約処理を行った場合では，1 つ目の中断処理と同様にそれぞれが並列に動いている．遅延時間を 1 秒に設定し，2 番目の座席に対し予約処理を行ったトランザクションが先に処理を終了し，遅延時間を 2 秒に設定し，1 番目の座席に対して予約処理を行ったトランザクションが次に処理を終了している．

同じ席に対して予約処理を行った場合では，片方の処理は処理を終了し，もう片方の処理は終了処理を行い，処理を中断している．遅延時間を 1 秒に設定し，1 番目の座席に対して予約処理を行ったトランザクションは終了している．遅延時間を 2 秒に設定し，1 番目の座席に対して予約処理を行ったトランザクションは，予約状態の席に対して予約をしようとしたため，終了処理に分岐し，処理を中断している．

複数のデータ更新をそれぞれ別のトランザクションで行い，追い越しが起こらない場合では，遅延時間を 1 秒に設定し，1 番目の座席と 2 番目の座席に対して順番に予約処理を行ったトランザクションは処理を終了している．遅延時間を 2 秒に設定し，1 番目の座席に対して予約処理を行ったトランザクションは終了処理に分岐し，中断している．

複数のデータ更新をそれぞれ別のトランザクションで行い，追い越しが起こる場合では，遅延時間を 3 秒に設定し，2 番目の座席と 1 番目の座席に対して順番に予約処理を行った

トランザクションは，遅延時間を 1 秒に設定し，1 番目の座席に対して予約処理を行ったトランザクションに追い越しを受け，1 番目の座席に対しての予約処理を行うことができず中断したが，座席 2 に対しての処理は完了することができている．

複数のデータ更新を一つのトランザクションで行い，追い越しが起こらない場合では，遅延時間を 1 秒に設定し，1 番目の座席と 2 番目の座席に対して順番に予約処理を行ったトランザクションは 1 番目と 2 番目の座席への予約処理を終了している．遅延時間を 2 秒に設定し，1 番目の座席に対して予約処理を行ったトランザクションは中断している．

複数のデータ更新を一つのトランザクションで行い，追い越しが起こる場合，遅延時間を 3 秒に設定し，2 番目の座席と 1 番目の座席に対して順番に予約処理を行ったトランザクションは，遅延時間を 1 秒に設定し，1 番目の座席に対して予約処理を行ったトランザクションに追い越しを受け，リトライすることにより終了処理に分岐し，処理を中断している．座席 2 に対しての処理はこの場合でも完了することができていない．

最後に，3 つ目の中断処理の結果について説明する．2 つ目の中断処理とほとんど同じ結果となった．

別々の席に対して予約処理を行った場合では，上の 2 つに同じく 1 つ目の中断処理と同様にそれぞれが並列に動いている．遅延時間を 1 秒に設定し，2 番目の座席に対し予約処理を行ったトランザクションが先に処理を終了し，遅延時間を 2 秒に設定し，1 番目の座席に対して予約処理を行ったトランザクションが次に処理を終了している．

同じ席に対して予約処理を行った場合では，片方の処理は処理を終了し，もう片方の処理は終了処理を行い，処理を中断している．遅延時間を 1 秒に設定し，1 番目の座席に対して予約処理を行ったトランザクションは終了している．遅延時間を 2 秒に設定し，1 番目の座席に対して予約処理を行ったトランザクションは，予約状態の席に対して予約をしようとしたため終了処理に分岐し，処理を中断している．

複数のデータ更新をそれぞれ別のトランザクションで行い，追い越しが起こらない場合では，遅延時間を 1 秒に設定し，1 番目の座席と 2 番目の座席に対して順番に予約処理を行ったトランザクションは処理を終了している．遅延時間を 2 秒に設定し，1 番目の座席に対して予約処理を行ったトランザクションは終了処理に分岐し，処理を中断している．

複数のデータ更新をそれぞれ別のトランザクションで行い，追い越しが起こる場合では，遅延時間を 3 秒に設定し，2 番目の座席と 1 番目の座席に対して順番に予約処理を行った

トランザクションは，遅延時間を 1 秒に設定し，1 番目の座席に対して予約処理を行ったトランザクションに追い越しを受け，1 番目の座席に対しての予約処理を行うことができず中断したが，座席 2 に対しての処理は完了することができている．

複数のデータ更新を一つのトランザクションで行い，追い越しが起こらない場合では，遅延時間を 1 秒に設定し，1 番目の座席と 2 番目の座席に対して順番に予約処理を行ったトランザクションは 1 番目と 2 番目の座席への予約処理を終了している．遅延時間を 2 秒に設定し，1 番目の座席に対して予約処理を行ったトランザクションは中断している．

複数のデータ更新を一つのトランザクションで行い，追い越しが起こる場合，遅延時間を 3 秒に設定し，2 番目の座席と 1 番目の座席に対して順番に予約処理を行ったトランザクションは，遅延時間を 1 秒に設定し，1 番目の座席に対して予約処理を行ったトランザクションに追い越しを受け，リトライすることにより終了処理に分岐し，処理を中断している．座席 2 に対しての処理はこの場合でも完了することができていない．

6.2.2. web アプリケーションとしての動作実験

Leiningen によって noir を利用した web アプリケーションの雛形を作成するために，コマンドプロンプトで以下のコマンドを実行する．

lein new noir projectname

作成された web アプリケーションの雛形にある src フォルダ内には，view，model を設置するフォルダがある．model フォルダについては，初めは空となっているはずなので，これまでテストしてきた異なる中断方式によって行われる予約処理を実装したコードのそれぞれを clj ファイルとして設置する．view フォルダについては，web ページの基礎構造を記述してある common.clj と view の雛形として welcome.clj が作成されているため，welcome.clj を改変するか，置き換えて web ページを構築する．この view ファイルに model 部分をインポートし，座席データとその更新処理を組み込む．また，必要があれば hiccup をインポートする．hiccup は html 表現を Clojure での木構造のベクタによって記述することを可能にするライブラリである．

付録の model と view は 1 つ目の中断方式によって行われる予約処理をテストするためのコードである．これらのコードを noir を利用した web アプリケーションの雛形に設置すると web アプリケーションとしての動作を試すことができる．まず，model として予約処理のコードを設置した時に変更された点について説明する．まず，名前空間をファイルのパスに合わせて変更している．更新関数である interrupt1，interrupt1t では，遅延関数を仕込

み，また，実行時，処理の終了時に引数としてとった id を出力するようにしている．下部では mkchair を 3 回よんでいて，web アプリケーションの起動と同時に id が 0，1，2 の 3 つの座席データを生成するようになっている．次に，view について説明する．web ページは，初期ページとして welcome，座席 1 への処理完了を表示するページとして chair1finish，座席 2 への処理完了を表示するページとして chair2finish，の 3 つのページで構成されている．welcome ページは 2 つのリンクを持つ．id が 1 の座席データとして扱われる座席 1 への予約処理を実行するためのリンクと，id が 2 の座席データとして扱われる座席 2 への予約処理を実行するためのリンクである．リンクをクリックすると，それぞれの座席データに対しての予約処理が実行され，処理が正常に終了した場合に処理が終了した旨のメッセージを表示するようになっている．今回の model では例外処理は実装していないので，処理が中断された場合はエラー表示画面に飛ぶようになっている．その場合コマンドプロンプトを見ると，処理の終了前に中断処理によるエラーが起こっているのを確認できる．これらのリンクによって予約処理を行うが，より確実に競合を確認するために，トランザクションが開始してから更新処理を行うまでの遅延は 3 秒に設定してある．(stmdata/reserve 1 3)，(stmdata/reserve 2 3)の引数の最後の部分で設定しているので，さらに余裕を持たせたい場合はより大きな値を設定して実行する．

6.3. 実験結果

一つのトランザクションで複数のデータを更新する場合，複数のデータのそれぞれについて別個のトランザクションで処理する場合，他のトランザクションの更新の完了を待ってトランザクション自体に中断させる場合，リトライが起こった瞬間に中断する場合，他のトランザクションの更新を待って中断処理を行う場合，これらのいずれについても，これまで説明してきた通りの動作が起こることを確認した．また，複数のデータに対して処理を行う時，一つのトランザクションでまとめて処理を行う場合に比べ，細かな処理ごとに小さなトランザクションで処理を行った場合の方が，中断される処理を減らすことができる．さらに，web プリケーション形式で，複数のブラウザからそれぞれの予約処理を実行した場合にも，並列関数を使用した場合と同じく並行性制御が働いていることが確認できた．それぞれの予約処理を実装したモデル部分の最終的なコード量は，1 つ目の予約処理で 865 文字，2 つ目の予約処理で 808 文字，3 つ目の予約処理で 698 文字である．Clojure のコードでは略記なども多用されていて，逐次処理のように明確な行数の基準がしにくいため純粋なタイプ量でコードを量っている．ワード単位で量った場合は 1 つ目の予約処理で 144 ワード，2 つ目の予約処理で 144 ワード，3 つ目の予約処理で 121 ワードである．

6.4. 考察

今回 3 通りの中断処理を考え実装したが，それぞれの実行結果にほとんど差異は現れなかった．状態が少なく中断を起こすような処理は，単純な if 分岐と STM によって非常に短い記述で構築できてしまうとも考えられる．中断処理の実装コード量に関して，1 つ目の中断処理で 865 文字，2 つ目の中断処理で 808 文字，3 つ目の中断処理で 698 文字となったが，今回の実験では，一つのトランザクションで複数のデータを更新する場合や，複数のデータそれぞれについて別個のトランザクションで処理する場合の複数の範囲のトランザクション処理を内包していることと，複数のデータの更新処理を静的な定義で記述していることから，動的に動作の内容が変化するようにそれらの処理を書き換えることで，それぞれの中断処理についてはさらに短いコードによって実現される．たとえば，1 つ目の予約処理を任意個の引数に対応させたものとして，以下のようなコードが考えられる．

```
(defn interrupt1
  [& ids]
  (dosync
    (apply #(alter (@allchair %) assoc :state res_c) ids)))
```

apply 関数によって，与えられた複数の id のそれぞれに対して更新関数が適用される．コード量は 76 文字で，実験で使用した更新処理部分のコードは 179 文字なので半分以下になっている．さらに，reserve 関数はトランザクションの適用範囲を切り替えるのみの関数なので，省くことができる．これらにより，1 つ目の予約処理の最終的なコード量としては，640 文字で構成することが可能である．

7. 結論

座席予約処理を例にとって Clojure の STM による並行処理に対する構築指針として中断処理についての 3 つの方法とそれらの特徴について示し，その指針によって構成される実装が期待した並行処理を行うことを確認した．noir や hiccup を使用することにより実装するべき部分は非常に小さくなり，また，Clojure による記述，プログラマへの並行処理による負担を削減する STM によって少ないコード量で並行処理を実現できることを示した．Clojure の STM の適用範囲に関して，関係性のあるデータ群に対しての処理は一つのトランザクション内で行い，一貫性を保つことが重要であるが，関係性のないデータ群に対しての処理ではトランザクションを分割し，別個のトランザクションによって処理を行った方がリトライによる無駄を削減できることを示した．

8. おわりに

今回の実験でテストとして行った競合パターンは，2 つのトランザクションによって行われる基本的なパターンで，その動作が確認できれば，より複数のトランザクション処理同士が絡み合った複雑なパターンでも，タイムスタンプによる実行順決定により，期待した並行性制御が行われると考えられるパターンであるが，より複数のトランザクション処理同士が絡み合った複雑なパターンについては実際にはテストを行っていないため，さらに多くの競合パターンを試す必要があると思われる．今回は Clojure を利用したが，STM を取り入れている他の言語では，STM の実装方法の違いによってトランザクションリトライ時の挙動に違いがある場合も考えられるため，言語ごとに STM 実装について深く理解する必要がある．

[1] C. A. R. Hoare, Communicating Sequential Processes, Prentice Hall International, 1985.

[2] Stuart Halloway(著), 川合史朗(訳), プログラミング Clojure, 株式会社オーム社, 2010.

[3] N. Shavit, and D. Touitou. Software Transactional Memory. Distributed Computing, Special Issue, 10 (1997), pages 99-116.

[4] Maurice Herlihy , Victor Luchangco , Mark Moir , William N. Scherer, III, Software transactional memory for dynamic-sized data structures, Proceedings of the twenty-second annual symposium on Principles of distributed computing, p.92-101, July 13-16, 2003, Boston, Massachusetts.

[5] 加藤公一，“クラウド環境での CSP 概念に基づく web アプリケーション設計法”，第７４回情報処理学会全国大会，2L-2.

謝辞

本論文をまとめるにあたり，多くの方にお世話になりました.
とくに，本研究に関して終始暖かくご指導頂きました大森健児先生に心より感謝致します.
本論文をご精読頂きました副査の佐々木先生，善甫先生に深く感謝致します．また，機材をお貸し頂いた廣津先生に深く感謝致します.
同じ研究室で相談に乗って頂いた林君，水谷君に深く感謝致します。

付録

並行処理関数を使用した実験での実行結果

Interrupt1

```
(pvalues (reserve 1 2) (reserve 2 1))
(doing: 1
finish : 1
doing: 2
finish : 2
nil nil)
```

```
(pvalues (reserve 1 2) (reserve 1 1))
(doing: 1
finish : 1
doing: 2
doing: 2
java.lang.IllegalStateException: Invalid reference state
```

```
(pvalues (reserve 1 2) (reserve 1 2 1))
(doing: 1
finish : 1
doing: 2
doing: 1
finish : 1
doing: 2
1:4 interrupt1=> java.lang.IllegalStateException: Invalid reference
```

state

```
(pvalues (reserve 1 1) (reserve 2 1 3))
(doing: 1
finish : 1
doing: 3
finish : 3
doing: 3
1:4 interrupt1=> java.lang.IllegalStateException: Invalid reference state
```

state

```
(pvalues (reserve 1 2) (reserve 1 2 1 0))
(doing: 1
finish : 1
doing: 2
doing: 2
1:4 interrupt1=> java.lang.IllegalStateException: Invalid reference state
```

```
(pvalues (reserve 1 1) (reserve 2 1 3 0))
(doing: 1
finish : 1
doing: 3
doing: 3
java.lang.IllegalStateException: Invalid reference state
```

interrupt2

```
(pvalues (reserve 1 2) (reserve 2 1))
(doing 1:1
finish : 1
doing 1:2
finish : 2
nil nil)


(pvalues (reserve 1 2) (reserve 1 1))
(doing 1:1
finish : 1
doing 1:2
doing 2:2
No.2: can't reserve.
finish : 2
nil nil)


(pvalues (reserve 1 2) (reserve 1 2 1))
(doing 1:1
finish : 1
doing 1:1
finish : 1
doing 1:2
doing 2:2
No.2: can't reserve.
finish : 2
nil nil)
```

```
(pvalues (reserve 1 1) (reserve 2 1 3))
(doing 1:1
finish : 1
doing 1:3
finish : 3
doing 1:3
No.3: can't reserve.
finish : 3
nil nil)
```

```
(pvalues (reserve 1 2) (reserve 1 2 1 0))
doing 1:1
(finish : 1
doing 1:2
doing 2:2
No.2: can't reserve.
finish : 2
nil nil)
```

```
(pvalues (reserve 1 1) (reserve 2 1 3 0))
(doing 1:3
doing 1:1
finish : 1
doing 2:3
No.3: can't reserve.
```

```
finish : 3
nil nil)


interrupt3


(pvalues (reserve 1 2) (reserve 2 1))
(doing: 1
finish : 1
doing: 2
finish : 2
nil nil)


(pvalues (reserve 1 2) (reserve 1 1))
(doing: 1
finish : 1
doing: 2
doing: 2
No.2: can't reserve.
finish : 2
nil nil)


(pvalues (reserve 1 2) (reserve 1 2 1))
(doing: 1
finish : 1
doing: 2
```

```
doing: 1
finish : 1
doing: 2
No.2: can't reserve.
finish : 2
nil nil)
```

```
(pvalues (reserve 1 1) (reserve 2 1 3))
(doing: 1
finish : 1
doing: 3
finish : 3
doing: 3
No.3: can't reserve.
finish : 3
nil nil)
```

```
(pvalues (reserve 1 2) (reserve 1 2 1 0))
(doing: 1
finish : 1
doing: 2
doing: 2
No.2: can't reserve.
finish : 2
nil nil)
```

```
(pvalues (reserve 1 1) (reserve 2 1 3 0))

(doing: 1

finish : 1

doing: 3

No.3: can't reserve.

finish : 3

nil nil)
```

並列処理関数を使用した実験で使用したコード

```
(ns interrupt1)

(defn waitsec
  [s]
  (Thread/sleep (* s 1000)))

;;;;;chairデータ周り
(def id (ref 0))      ;新規chairに割り振られるid，また，allchair内のchairの総数
(def emp_c 0)         ;chairの状態 empty
(def res_c 1)         ;chairの状態 reserved

(defn setlog [mref log]
  (add-watch mref nil
    (fn [_key mref _old _new]
      (swap! log conj (_old :state)))))

(defn setbacklog [mref log]
  (add-watch mref nil
    (fn [_key mref _old _new]
      (reset! log (_new :state)))))

(def allchair (ref (sorted-map)))    ;複数のchairをsorted-mapで保存

(defn mkchair
  "初期状態が空のchairを生成し，allchairに追加する．また，idを加算．"
  []
```

```
  (dosync
    (alter allchair assoc @id (ref {:state emp_c :cancel (ref 0) :log
(atom []) :backlog (atom [])}))
    (setlog (@allchair @id) (@(@allchair @id) :log))
    (setbacklog (@allchair @id) (@(@allchair @id) :backlog))
    (set-validator! (@allchair @id) #(not= @(% :backlog) (:state %)))
    (ref-set id (inc @id))))

(defn res?
  [id_c]
  (= (@(@allchair id_c) :state) res_c))

(defn interrupt1
  [id_c sec]
   (do
     (dosync
       (waitsec sec)
       (println (str "doing: " sec))
       (alter (@allchair id_c) assoc :state res_c))
     (println (str "finish : " sec))))

(defn interrupt1t
  [id_c id_d sec]
   (do
     (dosync
       (waitsec sec)
       (println (str "doing: " sec))
       (alter (@allchair id_c) assoc :state res_c)
       (alter (@allchair id_d) assoc :state res_c))
```

```
    (println (str "finish : " sec))))

(defn reserve
  ([id_c sec]
    (interrupt1 id_c sec))
  ([id_c id_d sec]
    (do (interrupt1 id_c sec) (interrupt1 id_d sec)))
  ([id_c id_d sec flag]
    (interrupt1t id_c id_d sec))
  )

(mkchair)
(mkchair)
(mkchair)




(ns interrupt2)

(defn waitsec
  [s]
  (Thread/sleep (* s 1000)))

;;;;;chairデータ周り
(def id (ref 0))      ;新規chairに割り振られるid，また，allchair内のchairの総数
(def emp_c 0)         ;chairの状態 empty
(def res_c 1)         ;chairの状態 reserved
```

```
(defn setlog [mref log]
  (add-watch mref nil
    (fn [_key mref _old _new]
      (swap! log conj (_old :state)))))

(def allchair (ref (sorted-map)))    ;複数のchairをsorted-mapで保存

(defn mkchair
  "初期状態が空のchairを生成し，allchairに追加する．また，idを加算．"
  []
  (dosync
    (alter allchair assoc @id (ref {:state emp_c :cancel (ref 0) :log
(atom [])}))
    (setlog (@allchair @id) (@(@allchair @id) :log))
    (ref-set id (inc @id))))

(defn res?
  [id_c]
  (= (@(@allchair id_c) :state) res_c))

(defn interrupt2
  [id_c sec]
  (let [retry (atom 0)]
    (do (waitsec sec)
      (dosync
        (waitsec sec)
        (reset! retry (inc @retry))
        (println (str "doing "@retry ":" sec))
        (if (or (> @retry 1) (res? id_c))
```

```
          (println (str "No." sec ": can't reserve."))
          (alter (@allchair id_c) assoc :state res_c)))
      (println (str "finish : " sec))))))

(defn interrupt2t
  [id_c id_d sec]
  (let [retry (atom 0)]
    (do
      (dosync
        (reset! retry (inc @retry))
        (println (str "doing "@retry ":" sec))
        (if (or (> @retry 1) (res? id_c) (res? id_d))
          (println (str "No." sec ": can't reserve."))
          (do
            (waitsec sec)
            (alter (@allchair id_c) assoc :state res_c)
            (alter (@allchair id_d) assoc :state res_c))))
      (println (str "finish : " sec))))))


(defn reserve
  ([id_c sec]
    (interrupt2 id_c sec))
  ([id_c id_d sec]
    (do (interrupt2 id_c sec) (interrupt2 id_d sec)))
  ([id_c id_d sec flag]
    (interrupt2t id_c id_d sec))
  )
```

```
(mkchair)
(mkchair)
(mkchair)
```

```
(ns interrupt3)

(defn waitsec
  [s]
  (Thread/sleep (* s 1000)))

;;;;;chairデータ周り
(def id (ref 0))      ;新規chairに割り振られるid，また，allchair内のchairの総数
(def emp_c 0)         ;chairの状態 empty
(def res_c 1)         ;chairの状態 reserved

(defn setlog [mref log]
  (add-watch mref nil
    (fn [_key mref _old _new]
      (swap! log conj (_old :state)))))

(def allchair (ref (sorted-map)))                ;複数のchairをsorted-mapで保存

(defn mkchair
  "初期状態が空のchairを生成し，allchairに追加する．また，idを加算．"
  []
```

```
  (dosync
    (alter allchair assoc @id (ref {:state emp_c :cancel (ref 0) :log
(atom [])}))
    (setlog (@allchair @id) (@(@allchair @id) :log))
    (ref-set id (inc @id))))

(defn res?
  [id_c]
  (= (@(@allchair id_c) :state) res_c))

(defn interrupt3
  [id_c sec]
  (do
    (waitsec sec)
    (dosync
      (waitsec sec)
      (println (str "doing: " sec))
      (if (res? id_c)
        (println (str "No." sec ": can't reserve."))
        (alter (@allchair id_c) assoc :state res_c)))
    (println (str "finish : " sec))))

(defn interrupt3t
  [id_c id_d sec]
  (do
    (dosync
      (if (or (res? id_c) (res? id_d))
        (println (str "No." sec ": can't reserve."))
        (do
```

```
          (waitsec sec)
          (println (str "doing: " sec))
          (alter (@allchair id_c) assoc :state res_c)
          (alter (@allchair id_d) assoc :state res_c))))
    (println (str "finish : " sec))))


(defn reserve
  ([id_c sec]
    (interrupt3 id_c sec))
  ([id_c id_d sec]
    (do (interrupt3 id_c sec) (interrupt3 id_d sec)))
  ([id_c id_d sec flag]
    (interrupt3t id_c id_d sec))
  )

(mkchair)
(mkchair)
(mkchair)
```

テスト用の関数や引数を省いた純粋なコード

```
(ns interrupt1)

(def id (ref 0))
(def emp_c 0)
(def res_c 1)

(defn setlog [mref log]
  (add-watch mref nil
    (fn [_key mref _old _new]
      (swap! log conj (_old :state)))))

(defn setbacklog [mref log]
  (add-watch mref nil
    (fn [_key mref _old _new]
      (reset! log (_new :state)))))

(def allchair (ref (sorted-map)))

(defn mkchair
  []
  (dosync
    (alter allchair assoc @id (ref {:state emp_c :cancel (ref 0) :log
(atom []) :backlog (atom [])}))
    (setlog (@allchair @id) (@(@allchair @id) :log))
    (setbacklog (@allchair @id) (@(@allchair @id) :backlog))
    (set-validator! (@allchair @id) #(not= @(% :backlog) (:state %)))
    (ref-set id (inc @id))))
```

```
(defn res?
  [id_c]
  (= (@(@allchair id_c) :state) res_c))

(defn interrupt1
  [id_c]
  (dosync
    (alter (@allchair id_c) assoc :state res_c)))

(defn interrupt1t
  [id_c id_d]
  (dosync
    (alter (@allchair id_c) assoc :state res_c)
    (alter (@allchair id_d) assoc :state res_c)))

(defn reserve
  ([id_c]
    (interrupt1 id_c))
  ([id_c id_d]
    (do (interrupt1 id_c) (interrupt1 id_d)))
  ([id_c id_d flag]
    (interrupt1t id_c id_d)))
```

```
(ns interrupt2)
```

```
(def id (ref 0))
(def emp_c 0)
(def res_c 1)

(defn setlog [mref log]
  (add-watch mref nil
    (fn [_key mref _old _new]
      (swap! log conj (_old :state))))))

(def allchair (ref (sorted-map)))

(defn mkchair
  []
  (dosync
    (alter allchair assoc @id (ref {:state emp_c :cancel (ref 0) :log
(atom [])}))
    (setlog (@allchair @id) (@(@allchair @id) :log))
    (ref-set id (inc @id))))

(defn res?
  [id_c]
  (= (@(@allchair id_c) :state) res_c))

(defn interrupt2
  [id_c]
   (let [retry (atom 0)]
   (dosync
    (reset! retry (inc @retry))
    (if (or (> @retry 1) (res? id_c))
```

```
      (alter (@allchair id_c) assoc :state res_c)))))

(defn interrupt2t
  [id_c id_d]
   (let [retry (atom 0)]
   (dosync
    (reset! retry (inc @retry))
    (if (or (> @retry 1) (res? id_c) (res? id_d))
      (do
        (alter (@allchair id_c) assoc :state res_c)
        (alter (@allchair id_d) assoc :state res_c))))))

(defn reserve
  ([id_c]
    (interrupt2 id_c))
  ([id_c id_d]
    (do (interrupt2 id_c) (interrupt2 id_d)))
  ([id_c id_d flag]
    (interrupt2t id_c id_d)))


(ns interrupt3)

(def id (ref 0))
(def emp_c 0)
(def res_c 1)
```

```
(defn setlog [mref log]
  (add-watch mref nil
    (fn [_key mref _old _new]
      (swap! log conj (_old :state)))))

(def allchair (ref (sorted-map)))

(defn mkchair
  []
  (dosync
    (alter allchair assoc @id (ref {:state emp_c :cancel (ref 0) :log
(atom [])}))
    (setlog (@allchair @id) (@(@allchair @id) :log))
    (ref-set id (inc @id))))

(defn res?
  [id_c]
  (= (@(@allchair id_c) :state) res_c))

(defn interrupt3
  [id_c]
   (dosync
    (if (res? id_c)
      (alter (@allchair id_c) assoc :state res_c))))

(defn interrupt3t
  [id_c id_d]
   (dosync
    (if (or (res? id_c) (res? id_d))
```

```
      (do
        (alter (@allchair id_c) assoc :state res_c)
        (alter (@allchair id_d) assoc :state res_c))))))

(defn reserve
  ([id_c]
    (interrupt3 id_c))
  ([id_c id_d]
    (do (interrupt3 id_c) (interrupt3 id_d)))
  ([id_c id_d flag]
    (interrupt3t id_c id_d)))
```

model部分のコード

```
(ns stmtest4.models.stmdata)

(defn waitsec
  [s]
  (Thread/sleep (* s 1000)))

;;;;;chairデータ周り
(def id (ref 0))      ;新規chairに割り振られるid，また，allchair内のchairの総数
(def emp_c 0)         ;chairの状態 empty
(def res_c 1)         ;chairの状態 reserved

(defn setlog [mref log]
  (add-watch mref nil
    (fn [_key mref _old _new]
      (swap! log conj (_old :state)))))

(defn setbacklog [mref log]
  (add-watch mref nil
    (fn [_key mref _old _new]
      (reset! log (_new :state)))))

(def allchair (ref (sorted-map)))

(defn mkchair
  []
  (dosync
```

```
      (alter allchair assoc @id (ref {:state emp_c :cancel (ref
0) :log (atom []) :backlog (atom [])}))
      (setlog (@allchair @id) (@(@allchair @id) :log))
      (setbacklog (@allchair @id) (@(@allchair @id) :backlog))
      (set-validator! (@allchair @id) #(not= @(% :backlog)
(:state %)))
      (ref-set id (inc @id))
      ))

(defn res?
  [id_c]
  (= (@(@allchair id_c) :state) res_c))

(defn interrupt1
  [id_c sec]
   (do
     (dosync
       (waitsec sec)
       (println (str "doing: " id_c))
       (alter (@allchair id_c) assoc :state res_c))
     (println (str "finish : " id_c))))

(defn interrupt1t
  [id_c id_d sec]
   (do
     (dosync
       (waitsec sec)
       (println (str "doing: " id_c id_d))
       (alter (@allchair id_c) assoc :state res_c)
       (alter (@allchair id_d) assoc :state res_c))
```

```
     (println (str "finish : " id_c id_d))))

(defn reserve
  ([id_c sec]
    (interrupt1 id_c sec))
  ([id_c id_d sec]
    (do (interrupt1 id_c sec) (interrupt1 id_d sec)))
  ([id_c id_d sec flag]
    (interrupt1t id_c id_d sec)))

(mkchair)
(mkchair)
(mkchair)
```

view部分のコード

```
(ns stmtest4.views.welcome
  (:require [stmtest4.views.common :as common]
            [noir.content.getting-started]
            [stmtest4.models.stmdata :as stmdata])
  (:use [noir.core]
        hiccup.element))

(defpage "/welcome" []
         (common/layout
           [:p "Welcome to stmtest."]
           [:br][:br]
           [:p "basic act."]
```

```
            [:br]
            [:tr
            (link-to "/welcome/chair1finish" "Chair1")
            [:td "    "]
            (link-to "/welcome/chair2finish" "Chair2")]
            [:br][:br]
            ))

(defpage "/welcome/chair1finish" []
          (stmdata/reserve 1 3)
          (common/layout
            [:p "Chair1 reserved."]
            (link-to "javascript:history.back(-1)" "return")))

(defpage "/welcome/chair2finish" []
          (stmdata/reserve 2 3)
          (common/layout
            [:p "Chair2 reserved."]
            (link-to "javascript:history.back(-1)" "return")))
```