(19)日本国特許庁(JP)　(12)公開特許公報(A)　(11)特許出願公開番号
特開2014-119964
(P2014-119964A)
(43)公開日　平成26年6月30日(2014.6.30)

(51)Int.Cl.		F I			テーマコード（参考）
G06F 11/30	(2006.01)	G06F	11/30	305G	5B042
G06F 9/48	(2006.01)	G06F	9/46	452J	
G06F 9/52	(2006.01)	G06F	9/46	472A	

審査請求 未請求　請求項の数 5　ＯＬ　（全 15 頁）

(21)出願番号　特願2012-274665 (P2012-274665)
(22)出願日　平成24年12月17日 (2012.12.17)

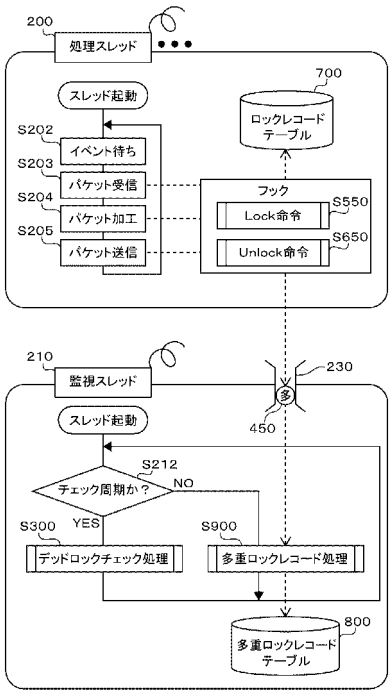
(71)出願人　000005108
株式会社日立製作所
東京都千代田区丸の内一丁目６番６号
(74)代理人　110000350
ポレール特許業務法人
(72)発明者　今野　功
神奈川県横浜市戸塚区戸塚町２１６番地
株式会社日立製作所通信ネットワーク事業部内
(72)発明者　松木　譲介
神奈川県横浜市戸塚区戸塚町２１６番地
株式会社日立製作所通信ネットワーク事業部内

Ｆターム(参考)　5B042 GA23　JJ06　JJ14　KK05　MC07

(54)【発明の名称】計算機システムおよびプログラム

(57)【要約】
【課題】マルチスレッドで動作するソフトウェアにおいて、デッドロック解消のためにカーネルの改造せずにデッドロックを自動検出し、フェールオーバなしでデッドロック状態を解消する方法を提供する。
【解決手段】スレッドの排他制御情報を収集、蓄積し、その情報からデッドロックを検出し、排他制御を追加することによりデッドロックを回避し、デッドロックが発生した場合はデッドロック発生前の状態に戻すことでデッドロックを解消する。
【選択図】図２

図２



【特許請求の範囲】
【請求項１】
複数の処理スレッドと、監視スレッドとを並列実行する計算機システムであって、
プロセッサと、メモリとを備え、
前記処理スレッドは、排他制御情報を収集および蓄積し、多重排他制御情報を前記監視スレッドに送信し、
前記監視スレッドは、前記多重排他制御情報を蓄積し、前記多重排他制御情報間にデッドロックを引き起こす組み合わせを検出したとき、当該組み合わせの多重排他制御情報の復帰場所に排他制御を追加することを特徴とする計算機システム。
【請求項２】
請求項１に記載の計算機システムであって、
前記処理スレッドは、前記排他制御情報を、ロック取得命令とロック解放命令へのフックを利用して収集することを特徴とする計算機システム。
【請求項３】
請求項１に記載の計算機システムであって、
前記監視スレッドは、デッドロック要因となるロック取得命令を含む関数の開始時と、ロック解放命令を含む関数の終了時とにフックして、復帰場所を指定することを特徴とする計算機システム。
【請求項４】
請求項１に記載の計算機システムであって、
デッドロック発生前にメモリ内容のスナップショットを取得し、
デッドロック発生後に、前記スナップショットからメモリ内容を復元し、
プログラムカウンタ位置を復帰場所に変更し、
デッドロックの要因となった排他制御を解放することによって、デッドロック状態から復元することを特徴とする計算機システム。
【請求項５】
コンピュータを、
排他制御情報を収集および蓄積し、多重排他制御情報を前記監視スレッドに送信する処理スレッド、
前記多重排他制御情報を蓄積し、前記多重排他制御情報間にデッドロックを引き起こす組み合わせを検出したとき、当該組み合わせの多重排他制御情報の復帰場所に排他制御を追加する監視スレッド、
として機能させるプログラム。
【発明の詳細な説明】
【技術分野】
【０００１】
本発明は、計算機システムおよびプログラムに係り、特に、マルチスレッドで動作する計算機システムにおいて、デッドロックを検出し、デッドロック発生前の回避とデッドロック発生後の解消を行う計算機システムおよびプログラムに関する。
【背景技術】
【０００２】
近年、様々なシステムにおいてマルチスレッド環境で稼働するソフトウェアが普及している。マルチスレッド環境では、ＯＳ（Operating System）がソフトエア内の実行単位をスレッドで管理する。ＯＳは、スレッドに対しＣＰＵ（Central Processing Unit）が時間割り当てのスケジューリングを行う。スレッドは、ＯＳからＣＰＵ時間を割り当てられた時間だけ動作することができる。
【０００３】
したがって、ＣＰＵを複数備えるマルチスレッド環境では、スレッドを並列に実行し高速化できる。しかし、処理を並列に実行することにより、デッドロックと呼ばれる問題が発生する。

【０００４】
ここで、デッドロックとは複数のスレッドまたはプロセスなどの処理単位がリソースを共有する場合において、リソース解放漏れやリソースを占有（Ｌｏｃｋ）中の処理単位が互いにリソースの解放（Ｕｎｌｏｃｋ）を待ち続け、処理を再開できずに停止している状態である。ここで、プロセスとはプログラムの実行単位であり、プログラム内で利用している変数や状態を保持し、１つ以上のスレッドから構成される。
【０００５】
デッドロックが発生したプロセスは、異常終了しない。このため、ソフトウェアの使用者、開発者が異常の発生直後に気付かない。また、ソフトウェアの開発者が異常に気付いたとしても、スレッドの挙動確認や解析に時間がかかる。そこで、デッドロックを防ぎ、解決する技術が重要となってくる。
【０００６】
特許文献１には、デッドロックが二重以上のロック取得するスレッド同士で発生する点に着目し、大域的なロックを追加している。つまり、二重以上のロックを取得するスレッドに対し、大域的なロックを取得する制約を設ける。これによって、二重以上のロックを取得するスレッドを制限し、デッドロックを防止する。さらに、大域的なロックを一律に取得するのではなく、条件を付けることで性能低下を抑えている。
【０００７】
また、特許文献２では、ロック取得および解除に対してＡＰＩ関数フックを行い、スレッド毎に共有リソースに対して「ロック中」および「ロック取得中」の情報を取得し、その組み合わせによってデッドロックを検出する計算機システムおよびプログラムが提案されている。ここで、ＡＰＩ関数とは、ＯＳやミドルウェアが提供するアプリケーションおよびソフトウェア開発向けのインタフェースであり、共通ライブラリの形で提供される。また、ＡＰＩ関数フックとは、ＡＰＩ関数の処理を横取りし、利用者が独自に定義した処理を行うことである。また、「ロック中」とは、スレッドがリソースを占有中により他のスレッドがリソースにアクセスできない状態である。一方、「ロック取得中」とは他のスレッドが占有中のリソースが解放されるのを待ってからロックしようとしている状態である。
【０００８】
特許文献２は、これらの情報を組み合わせ、具体的には、リソースＡを「ロック中」でリソースＢを「ロック取得中」のスレッド１と、リソースＡを「ロック取得中」でリソースＢを「ロック中」のスレッド２が同時に存在すると、互いにリソースが解放されるのを待ち続けるデッドロックとして検出できる。デッドロック検出後は、発生経路を記憶して冗長系にフェールオーバすることで、フェールオーバ先でデッドロックの再発を防止する。ここで、フェールオーバとは、正常時に冗長構成の計算機でデータの同期を取っておき、障害が発生した計算機から残りの正常な計算機にデータを引継ぎ、処理を継続する動作である。
【先行技術文献】
【特許文献】
【０００９】
【特許文献１】特開平０７－１９１９４４号公報
【特許文献２】特開２００９－２７１８５８号公報
【発明の概要】
【発明が解決しようとする課題】
【００１０】
特許文献１に記載の技術は、ロックの取得を連続で行う必要があり、多重ロックを取得する時点で予め必要なロックが全て既知であることが前提条件となる。実際のプログラムでは、経路によって必要なロックが異なるため、予め必要なロックを全て取得することは困難であり、必要なロックを全て取得できなかった場合、デッドロックが発生する。
【００１１】

特許文献２に記載の技術は、デッドロックの再発を防げる。しかし、別経路でデッドロックが発生する度にフェールオーバを行う必要がある。フェールオーバ実行中は、本来実行しなければならない処理を実行できず、本来提供したいサービスに影響が出てしまう。また、切替え可能な計算機が無ければフェールオーバできないため、可能な限りフェールオーバ実施を避けるべきである。

特許文献１と特許文献２を組み合わせたとしても、デッドロック発生後の復帰手段がないため、容易には解決できない。

【課題を解決するための手段】

【００１２】

本発明の代表的な一形態によると、複数のスレッドを実行する計算機システムであって、計算機システムは、少なくとも一つのプロセッサと、メモリとを備え、計算機システムで動作するソフトウェア内で第一のスレッドと第一のスレッドを監視するための監視スレッドを実行し、第一のスレッドの排他制御の情報を保持するための監視情報領域と、第一のスレッド用にデッドロック回避用の領域および情報と、メモリ内容を復元するための退避領域と、をメモリに保持し、第一のスレッドと監視スレッドは監視情報領域に排他制御の情報を格納し、第一のスレッドはメモリ内容を退避領域に格納し、監視スレッドが監視情報領域からデッドロックを検出し、監視スレッドはデッドロックが発生する可能性がある第一のスレッドに対し、デッドロック回避用領域に排他制御を追加し、第一のスレッドでデッドロックが発生した場合、第一のスレッドの処理を退避位置に戻し、かつ、退避領域からメモリ内容を復元することでデッドロックを解消する。

【００１３】

上述した課題は、複数の処理スレッドと、監視スレッドとを並列実行する計算機システムであって、プロセッサと、メモリとを備え、前記処理スレッドは、排他制御情報を収集および蓄積し、多重排他制御情報を前記監視スレッドに送信し、前記監視スレッドは、前記多重排他制御情報を蓄積し、前記多重排他制御情報間にデッドロックを引き起こす組み合わせを検出したとき、当該組み合わせの多重排他制御情報の復帰場所に排他制御を追加する計算機システムにより、達成できる。

【００１４】

また、コンピュータを、排他制御情報を収集および蓄積し、多重排他制御情報を前記監視スレッドに送信する処理スレッド、前記多重排他制御情報を蓄積し、前記多重排他制御情報間にデッドロックを引き起こす組み合わせを検出したとき、当該組み合わせの多重排他制御情報の復帰場所に排他制御を追加する監視スレッド、として機能させるプログラムにより、達成できる。

【発明の効果】

【００１５】

本発明によって、デッドロックを自動検出し、フェールオーバを実施せずにデッドロック状態から正常な状態に復帰できる。

【図面の簡単な説明】

【００１６】

【図１】通信装置のハードウェアブロック図である。

【図２】デッドロック監視を行う装置のスレッド構成と処理内容を説明する図である。

【図３】ロックレコード、多重ロックレコードを説明する図である。

【図４Ａ】Ｌｏｃｋ命令のフローチャートである。

【図４Ｂ】フック有りＬｏｃｋ命令のフローチャートである。

【図５Ａ】Ｕｎｌｏｃｋ命令のフローチャートである。

【図５Ｂ】フック有りＵｎｌｏｃｋ命令のフローチャートである。

【図６】ロックレコードテーブルを説明する図である。

【図７】多重ロックレコードテーブルを説明する図である。

【図８】多重ロックレコードの処理フローチャートである。

【図９】デッドロック検出、回避、解消の流れを示すフローチャートである。

【図１０】デッドロック検出のフローチャートである。
【図１１】デッドロック回避のフローチャートである。
【図１２】デッドロック解消のフローチャートである。
【図１３】メモリ内容をデッドロック発生前の状態に復元する処理を説明する図である。
【発明を実施するための形態】
【００１７】
以下、本発明の実施の形態について、実施例を用い図面を参照しながら詳細に説明する。なお、実質同一部位には同じ参照番号を振り、説明は繰り返さない。
【実施例１】
【００１８】
実施例１は、通信装置のように特定の処理を繰り返す計算機を説明する。
図１を参照して、最小構成のハードウェアの通信装置を説明する。図１において、通信装置１００は、プロセッサ１０１と、メモリ１０２と、ＮＩＦ（Network InterFace）１０３とを具備する。
【００１９】
プロセッサ１０１は、ＣＰＵ等の演算装置であり、ＯＳやアプリケーションプログラム等のソフトウェアを実行する。メモリ１０２は、主記憶装置であり、プロセッサ１０１がソフトウェア実行時に、プログラム実行バイナリおよびプログラムが使用するデータを格納する。ＮＩＦ１０３は、通信装置１００とは別の装置とパケットを送受信するためのインタフェースである。各ハードウェアブロックは、バス１１０－１、１１０－２によって相互に接続するため、互いに命令メッセージ及びデータを送信することが可能である。
【００２０】
なお、本実施例を適用するためには、プロセッサ１０１がマルチタスクに対応している必要がある。ここで、マルチタスクとは演算装置が複数の処理を切換えながら複数の処理を実行する方式であり、汎用計算機のプロセッサでは、そのほとんどがマルチタスクに対応している。
【００２１】
また、通信装置１００は、物理的に一つの計算機によって実装されてもよいし、少なくとも一つの計算機が提供する仮想的な計算機によって実装されてもよい。
【００２２】
図２を参照して、ソフトウェアのスレッド構成と、ロックを記録するためのフック処理を説明する。図２のサブルーチンに関しては、別途、後述の図を用いて詳述する。図２において、スレッドは、通信装置の呼制御などを行う複数の処理スレッド２００と、処理スレッド２００が正常に動作しているか否かを判定する監視スレッド２１０から成る。
【００２３】
処理スレッド２００は、起動後、パケット受信などのイベントを待つ（Ｓ２０２）。処理スレッド２００は、パケットを受信し（Ｓ２０３）、送信するパケットを加工し（Ｓ２０４）、送信する（Ｓ２０５）動作を繰返す。スレッド起動からパケット送信までは、ソフトウェアバイナリの変更は行わない。
【００２４】
処理スレッド２００は、フック処理２２０とロックレコードテーブル７００を用意しておく。フック処理２２０は、パケット処理のステップ２０３とパケット加工のステップ２０４とパケット送信のステップ２０５で呼び出されるすべてのＬｏｃｋ命令５００およびＵｎｌｏｃｋ命令６００を横取りするフック関数の集合である。フック処理２２０では、Ｌｏｃｋ命令５００が呼び出された際に、別途定義するサブルーチンＬｏｃｋ命令５５０を呼び出す。また、Ｕｎｌｏｃｋ命令６００が呼び出された際に、別途定義するサブルーチンＵｎｌｏｃｋ命令６５０を呼び出す。なお、サブルーチンＬｏｃｋ命令５５０は、図４で後述し、サブルーチンＵｎｌｏｃｋ命令６５０は、図５で後述する。フック処理２２０は、一時的にロックレコードをロックレコードテーブル７００に保存する。このうち、多重ロックレコードと判断した場合は、多重ロックレコード４５０を、監視スレッド２１

０のキュー２３０に追加する。
【００２５】
監視スレッド２１０は、プログラムカウンタが、サブルーチン多重ロックレコード処理９００に到達した際に、キュー２３０に多重ロックレコード４５０が入っているか調べ、多重ロックレコード４５０がキュー２３０に入っていた場合は、多重ロックレコードテーブル８００に、多重ロックレコード４５０を追加する。なお、サブルーチン多重ロックレコード処理９００は、図８を用いて詳述する。
【００２６】
上述した手順により、スレッド起動とイベント待ちのステップ２０２とパケット受信のステップ２０３とパケット加工のステップ２０４とパケット送信のステップ２０５を、変更することなく、ロック情報をロックレコードテーブル７００に記録することができる。
【００２７】
また、監視スレッド２１０は、起動後、チェック周期であるか否かを判定し、チェック周期であればデッドロックチェック処理（Ｓ３００）を行い、チェック周期でなければ多重ロックレコード処理（Ｓ９００）を行う動作を繰返す。ここで、チェック周期は開発者や使用者が指定する時間や回数以外にも、排他制御の統計から自動的に算出しても良い。デッドロックチェック処理は、図９を用いて、後述する。
【００２８】
本実施例では、監視スレッド２１０がデッドロックチェック処理を繰り返し実行する。これにより、デッドロックを検出し、デッドロック発生前に検出した場合は、デッドロックを回避し、デッドロック発生と同時にデッドロックを検出した場合はデッドロック状態からの解消処理を行い、デッドロックを自動的に解決する。なお、本実施例では監視スレッドを監視専用としているが、監視スレッドに別の役割を持たせることや、処理スレッド側に監視動作を追加することも可能である。このことにより、スレッド数を増やす必要が無く、メモリ量やＣＰＵリソースを節約する利点がある。
【００２９】
図３を参照して、デッドロックチェック動作の検出処理に利用するロックレコード４００および多重ロックレコード４５０を説明する。
【００３０】
図３（ａ）において、ロックレコード４００は、ネスト数４０１、ロックハンドル４０２、復帰場所４０３から成る。ロックレコード４００は、処理スレッド２００が個々に管理する排他制御の記録である。ここで、ネスト数４０１は、排他制御の深さである。ロックハンドル４０２は、排他制御に必要な識別子である。復帰場所４０３は、デッドロック退避用領域である。
【００３１】
図３（ｂ）において、多重ロックレコード４５０は、項番４５１、ロック順序４５２、復帰場所４０３、追加ロックハンドル４５３から成る。多重ロックレコード４５０は、監視スレッド２１０が管理する排他制御組合せパターンの記録である。ここで、項番４５１は、多重ロックレコードを管理するための識別子である。ロック順序４５２は、ロックハンドル４０２を取得した順序である。追加ロックハンドル４５３は、デッドロック回避用のロックハンドルである。
【００３２】
ロックレコード４００は、図６に示すロックレコードテーブル７００で管理する。多重ロックレコード４５０は、図７に示す多重ロックレコードテーブル８００で管理する。ロックレコードテーブル７００は、処理スレッド２００が個々に管理するテーブルである。ロックレコードテーブル７００は、処理スレッド２００がその時点で取得している排他制御を示す。ロックレコードテーブル７００は、ロック取得命令（以下、Ｌｏｃｋ命令、本実施例では、ｐｔｈｒｅａｄ＿ｍｕｔｅｘ＿ｌｏｃｋ）とロック解放命令（以下、Ｕｎｌｏｃｋ命令、本実施例では、ｐｔｈｒｅａｄ＿ｍｕｔｅｘ＿ｕｎｌｏｃｋ）にフックを行って取得する。ここで、フックとは処理を横取りし、使用者が定義した処理をプロセッサ

１０１に行わせる処理である。

【００３３】

具体的には、Ｌｉｎｕｘ（登録商標）系ＯＳのＬＤ＿ＰＲＥＬＯＡＤ環境変数と共有ライブラリを用いたＡＰＩ関数フックで実現できる。ＬＤ＿ＰＲＥＬＯＡＤ環境変数を指定しない状態では、ｐｔｈｒｅａｄ＿ｍｕｔｅｘ＿ｌｏｃｋ関数およびｐｔｈｒｅａｄ＿ｍｕｔｅｘ＿ｕｎｌｏｃｋ関数は、ｌｉｂｐｔｈｒｅａｄ．ｓｏという共有ライブラリの関数が実行される。しかし、ＬＤ＿ＰＲＥＬＯＡＤ環境変数で、共有ライブラリファイル名を指定すると、関数名から実行する関数へのアドレス解決順序を入れ替え、ＬＤ＿ＰＲＥＬＯＡＤ環境変数で指定した、共有ライブラリを優先的に検索する。指定する共有ライブラリに、ｐｔｈｒｅａｄ＿ｍｕｔｅｘ＿ｌｏｃｋ関数とｐｔｈｒｅａｄ＿ｍｕｔｅｘ＿ｕｎｌｏｃｋ関数を定義することにより、ｌｉｂｐｔｈｒｅａｄ．ｓｏのｐｔｈｒｅａｄ＿ｍｕｔｅｘ＿ｌｏｃｋ関数およびｐｔｈｒｅａｄ＿ｍｕｔｅｘ＿ｕｎｌｏｃｋ実行前に、共有ライブラリの関数を実行することができる。共有ライブラリのｐｔｈｒｅａｄ＿ｍｕｔｅｘ＿ｌｏｃｋ関数およびｐｔｈｒｅａｄ＿ｍｕｔｅｘ＿ｕｎｌｏｃｋ関数にて、ｌｉｂｐｔｈｒｅａｄ．ｓｏのｐｔｈｒｅａｄ＿ｍｕｔｅｘ＿ｌｏｃｋ関数およびｐｔｈｒｅａｄ＿ｍｕｔｅｘ＿ｕｎｌｏｃｋ関数を呼び出すことにより、Ｌｏｃｋ命令へのフックおよびＵｎｌｏｃｋ命令へのフックを実現することができる。

【００３４】

図４を参照して、Ｌｏｃｋ命令のフローチャートとフック有りＬｏｃｋ命令のフローチャートを説明する。また、図５を参照して、Ｕｎｌｏｃｋ命令のフローチャートとフック有りＵｎｌｏｃｋ命令のフローチャートを説明する。

【００３５】

図４Ａにおいて、Ｌｏｃｋ命令を開始すると、処理スレッド２００は、ロックハンドルが取得可能か判定する（Ｓ５０１）。取得可能であれば、処理スレッド２００は、ロックハンドルを取得し（Ｓ５０４）、処理を終了する。ステップ５０１でロックハンドルが取得不可であれば、ロックハンドルが解放されるまで待ち（Ｓ５０３）、ロックハンドルを取得する（Ｓ５０４）。

【００３６】

図４Ｂにおいて、フック有りＬｏｃｋ命令を開始すると、処理スレッド２００は、ロックレコードの記録追加処理を実施する（Ｓ５５１）。ロックレコードの記録追加処理は、取得しようとしているロックハンドルをロックレコードに登録し、ロックレコードテーブル７００に追加する処理である。追加後、処理スレッド２００は、ロックレコードテーブル７００のネスト数４０１が２以上か判定する（Ｓ５５２）。判定結果が２以上の場合、処理スレッド２００は、ロックレコードテーブル７００に登録中の全ロックレコード４００から多重ロックレコード４５０を作成し、監視スレッド２１０のキュー２３０にエンキューする（Ｓ５５３）。エンキュー完了後およびネスト数４０１が２未満の場合、処理スレッド２００は、Ｌｏｃｋ命令を実行し（Ｓ５００）、フック有りＬｏｃｋ関数を完了する。

【００３７】

図５Ａにおいて、Ｕｎｌｏｃｋ命令を開始すると、処理スレッド２００は、ロックハンドルを解放し（Ｓ６０１）、Ｕｎｌｏｃｋ命令を終了する。

図５Ｂにおいて、フック有りＵｎｌｏｃｋ命令６５０を開始すると、処理スレッド２００は、Ｕｎｌｏｃｋ命令を実行する（Ｓ６００）。処理スレッド２００は、処理完了後にロックレコードの記録削除処理を行い（Ｓ６５１）、フック有りＵｎｌｏｃｋ命令を完了する。

【００３８】

フック有りＬｏｃｋ命令のロックレコードの記録追加処理のステップ５５１と、フック有りＵｎｌｏｃｋ命令のロックレコードの記録削除処理のステップ６５１によって、処理スレッド２００は、取得しているロックハンドルを最新の状態として反映することができる。

【００３９】
図６を参照して、ロックハンドルテーブル７００の状態遷移を説明する。図６において、フック２２０によって、ロックハンドルＡを取得した状態７０１から、フック有りＬｏｃｋ（Ｂ）でロックハンドルＢを取得すると状態７０２になり、さらにフック有りＵｎｌｏｃｋ（Ｂ）でロックハンドルＢを解放すると状態７０３になる。
【００４０】
状態７０２ではネスト数４０１が２以上になるため、フック有りＬｏｃｋ命令において、処理スレッド２００は、多重ロックレコード４５０を作成する。ロックレコードテーブル７００が状態７０２であれば、作成する多重ロックレコード４５０のロック順序４５２はＡ→Ｂ、復帰場所４０３はＦｕｎｃＡＢとなり、作成した多重ロックレコード４５０を監視スレッド２１０のキュー２３０にエンキューする。なお、多重ロックレコード４５０の項番４５１と追加ロックハンドル４５３は、監視スレッド２１０が後で登録する。なお、デッドロックは、二重以上のロックハンドル４０２を取得した処理スレッド２００同士で発生するため、ネスト数４０１が２以上の場合のみ、多重ロックレコード４５０を作成すれば良い。
【００４１】
処理スレッド２００が多重ロックレコード４５０をキュー２３０にエンキュー後、監視スレッド２１０は、多重ロックレコード処理９００で多重ロックレコードテーブル８００へ登録する。多重ロックレコードテーブル８００は、図７に示す多重ロックレコード４５０を蓄積するテーブルであり、多重ロックレコード４５０を持たない状態８０１から始まる。なお、図７は、フローチャートの中で説明する。
【００４２】
図８を参照して、多重ロックレコード処理によって、未登録の多重ロックレコードを追加する処理を説明する。図８において、多重ロックレコード処理を開始すると、監視スレッド２１０は、キュー２３０に多重ロックレコードがあるか判定する（Ｓ９０１）。多重ロックレコードがある場合、監視スレッド２１０は、多重ロックレコードテーブル８００に未登録か判定する（Ｓ９０２）。未登録であれば、監視スレッド２１０は、多重ロックレコードテーブルに登録して（Ｓ９０３）、多重ロックレコード処理を終了する。なお、多重ロックレコードが未登録か否かの判定は、ロック順序および復帰場所が一致しているか否かで判定する。多重ロックレコードがキュー２３０に無い場合、または多重ロックレコードが登録済みの場合、監視スレッド２１０は、多重ロックレコード処理を終了する。この操作によって、図７において、多重ロックレコード４５０を保持していない状態８０１から、未登録の多重ロックレコード（Ａ→Ｂ）を受取り、受取った順に項番４５１を登録することで、状態８０２になる。
【００４３】
図２に示したように、監視スレッド２１０は、多重ロックレコードテーブル８００に対し、チェック周期以外では多重ロックレコード処理９００を行い、チェック周期であれば図９に示すデッドロックチェック処理３００を行う。
【００４４】
図９において、デッドロックチェック処理を開始すると、監視スレッド２１０は、デッドロック検出処理を実行する（Ｓ３１０）。監視スレッド２１０は、デッドロックを検出したか判定する（Ｓ３２０）。ＹＥＳのとき、監視スレッド２１０は、デッドロック回避処理を実行する（Ｓ３３０）。監視スレッド２１０は、検出したデッドロックが発生しているか判定する（Ｓ３４０）。ＹＥＳのとき、監視スレッド２１０は、デッドロック解消処理を実行して（Ｓ３５０）、終了する。ステップ３２０またはステップ３４０でＮＯのとき、監視スレッド２１０は、デッドロック検出処理を終了する。
【００４５】
図１０を参照して、デッドロック検出処理を説明する。図１０において、デッドロック検出処理を開始すると、監視スレッド２１０は、比較未実施の多重ロックレコードがあるか判定する（Ｓ３１１）。ここで、比較未実施の多重ロックレコードとは、多重ロックレ

コードテーブル８００に含まれる多重ロックレコードの組合せのうち、デッドロック検出の為の比較を実施していない多重ロックレコードの集合である。デッドロック検出処理の開始時点では全多重ロックレコードが比較未実施となる。未比較の多重ロックレコードがある場合、監視スレッド２１０は、その中から候補ＸとＹを選択する（Ｓ３１２）。監視スレッド２１０は、候補ＸとＹにデッドロックの可能性があるか判定する（Ｓ３１３）。デッドロックの可能性がある場合、監視スレッド２１０は、デッドロックパターンとして登録し（Ｓ３１４）、ステップ３１１へ戻る。全ての比較が完了後（Ｓ３１１：ＮＯ）、監視スレッド２１０は、終了する。

【００４６】

ステップ３１３でデッドロックの可能性あるか否かを判定する単純な方法としては、候補ＸとＹのロック順序４５２から一致するロックハンドルを抜出し、その順序が逆転しているか否かを判定する方法が挙げられる。

【００４７】

図７に戻って、多重ロックレコードテーブルが状態８０３であれば、ロックハンドルＡのみが一致し、順序逆転はないため、デッドロックは発生しないと判定する。また、多重ロックレコードテーブルが状態８０４であれば、３種類の多重ロックレコードの全組合せ３通りで比較し、項番１と項番３の多重ロックレコードでロックハンドルＡとＢが一致し、かつ順序逆転が発生しているため、デッドロックとして検出できる。

【００４８】

なお、多重ロックレコードテーブル８００は、多重ロックレコードを蓄積していくため、過去の多重ロックレコードから発生確率が低いデッドロックでも検出することが可能になる。しかし、多重ロックレコードを蓄積し続けると記憶領域を圧迫するため、必要に応じて古い情報を削除しても良い。また、未比較の多重ロックレコード４５０は、多重ロックレコードテーブル８００に新規登録する際に発生するため、多重ロックレコード処理９００のステップ９０３後にデッドロック検出処理を実行しても良い。

【００４９】

図１１を参照して、デッドロック回避処理を説明する。図１１において、監視スレッド２１０は、デッドロックを検出した多重ロックレコードの組合せ（最低でも２つ）を受取り、その多重ロックレコードの中で追加ロックハンドル４５３が登録されているかを確認する（Ｓ３３１）。追加ロックハンドル４５３が全ての多重ロックレコードに登録されていない場合（ＮＯ）、監視スレッド２１０は、追加ロックハンドル４５３を新規確保（ＧｌｏｂａｌＡＢとする）し、多重ロックレコードの追加ロックハンドル４５３にＧｌｏｂａｌＡＢを登録して（Ｓ３３２）、デッドロック回避処理を終了する。

【００５０】

ステップ３３１で追加ロックハンドル４５３を登録済みの場合（ＹＥＳ）、監視スレッド２１０は、登録している追加ロックハンドル４５３が１種類かを確認する（Ｓ３３３）。登録済みの追加ロックハンドル４５３が１種類（ＧｌｏｂａｌＸＸとする）の場合（ＹＥＳ）、監視スレッド２１０は、未登録の追加ロックハンドル４５３にＧｌｏｂａｌＸＸを登録して（Ｓ３３４）、デッドロック回避処理を終了する。

【００５１】

ステップ３３３で登録されている追加ロックハンドル４５３が複数（ＧｌｏｂａｌＸＸ、ＧｌｏｂａｌＹＹとする）の場合（ＮＯ）、監視スレッド２１０は、ロックハンドルを新規確保（ＧｌｏｂａｌＸＹとする）し、未登録の追加ロックハンドル４５３と、受取った多重ロックレコードのＧｌｏｂａｌＸＸとＧｌｏｂａｌＹＹをＧｌｏｂａｌＸＹで登録または上書き登録する（Ｓ３３５）。さらに、監視スレッド２１０は、多重ロックテーブル８００の中から、追加ロックハンドル４５３にＧｌｏｂａｌＸＸまたはＧｌｏｂａｌＹＹを登録しているレコードをＧｌｏｂａｌＸＹで上書き登録する（Ｓ３３６）。監視スレッド２１０は、上書き登録によって不要となったＧｌｏｂａｌＸＸとＧｌｏｂａｌＹＹのメモリ領域を削除する（Ｓ３３７）。追加ロックハンドル４５３の登録が完了後、監視スレッド２１０は、デッドロック回避処理を終了する。

なお、追加ロックハンドル４５３が複数となるのは、ネスト数が３以上の場合である。
【００５２】
デッドロック回避処理により、具体的には、図７において、多重ロックレコードテーブル８００が状態８０４であれば、項番１と項番３の多重ロックレコードを検出し、両方とも追加ロックハンドル４５３を登録していないため、追加ロックハンドル（ＧｌｏｂａｌＡＢ）を新規確保し、登録することで状態８０５となる。
【００５３】
デッドロック回避処理で登録した追加ロックハンドル４５３は、多重ロックレコードに登録している復帰場所にてスコープドロックし、デッドロック発生の可能性がある箇所を追加した排他制御で覆い込む。状態８０４では（Ａ）→（Ｂ）と（Ｂ）→（Ａ）の順番でロックハンドルを取得していたが、状態８０５で追加ロックハンドルにより（ＧｌｏｂａｌＡＢ）→（Ａ）→（Ｂ）と（ＧｌｏｂａｌＡＢ）→（Ｂ）→（Ａ）という順番になるため、ＧｌｏｂａｌＡＢによってデッドロックを防止できる。なお、スコープドロックとは、Ｌｏｃｋ命令を開始後、関数などの有効範囲（スコープ）終了時に自動でＵｎｌｏｃｋ命令を実行するための仕組みであり、リソース解放漏れを防ぐことが可能になる。
【００５４】
また、追加ロックハンドル４５３を上書き登録することにより、過去に検出したデッドロックパターン（具体的には、Ａ→Ｂ→Ｃ、Ｄ→Ｃ→Ｂ）と新たに検出したデッドロックパターン（Ｅ→Ｃ→Ｂ）で同一の追加ロックハンドル４５３を指定するため、デッドロックを全て防止できる。また、追加ロックハンドル４５３を一種類だけとし、全てのデッドロックパターンで同一の追加ロックハンドルを利用してすることも可能である。
【００５５】
ロックレコードテーブル７００や多重ロックレコードテーブル８００の復帰場所４０３は、デッドロックを防止できる箇所（プログラム上のアドレス）を指定する必要がある。その指定する方法は、チェックポイント方式と関数フック方式がある。チェックポイント方式は、復帰場所をソフトウェアの開発者がチェックポイントとして明示的に指定する方式である。一方、関数フック方式は、Ｃ言語やＣ＋＋言語で記述したソフトウェアのソースコードビルド時に「－ｆｉｎｓｔｒｕｍｅｎｔ－ｆｕｎｃｔｉｏｎｓ」オプションを指定することにより、関数の開始時と終了時にフックできるため、デッドロック要因となるＬｏｃｋ命令を含む関数の開始箇所を、自動的に指定する方式である。
【００５６】
デッドロック検出と同時にデッドロックが発生している場合は回避できないため、デッドロック解消処理を実施する。なお、デッドロックが発生しているか判定するには、チェック周期の際に処理スレッド２００に応答があるか確認し、応答がない、かつ処理スレッド２００のロックレコードテーブル７００がデッドロックの可能性を検出した多重ロックレコードを含む場合に、デッドロック状態と判定する。
【００５７】
図１２を参照して、デッドロック解消処理を説明する。デッドロック解消処理を開始すると、監視スレッド２１０は、プログラムカウンタに復帰場所を設定する（Ｓ３５１）。監視スレッド２１０は、メモリ内容をデッドロック発生前の状態に復元する（Ｓ３５２）。監視スレッド２１０は、デッドロック要因のロックハンドルを解放して（Ｓ３５３）、デッドロック解消処理を終了する。
【００５８】
プログラムカウンタは、プロセッサ１０１がプログラム上で次に実行する命令のアドレスを指す。各スレッドは、各々プログラムカウンタを保持しているため、プログラムカウンタを書き換えることにより、任意のスレッドが次に実行する命令のアドレスを任意のアドレスに変更することができる。したがって、プログラムカウンタを多重ロックレコードテーブル８００に登録した復帰場所４０３を指定する。処理スレッド２００は、追加ロックハンドル（ＧｌｏｂａｌＡＢ）取得から処理を再開し、デッドロック再発を防止する。
【００５９】

次に、デッドロック発生時点で既にメモリ内容が書き換わり、メモリに更新途中のデータが保持されている可能性がある。このため、メモリ内容をデッドロック発生前の状態に戻すことで、データの整合性を維持する。

【００６０】

図１３を参照して、メモリ復元を説明する。メモリ復元では、まず、メモリ１０２内でデッドロック発生前メモリ内容１３００のスナップショットを取り、メモリ内容をコピーする（状態１３０１）。ここで、スナップショットの際に名前付きメモリマップを利用することで、デッドロック発生後に必要なメモリを検索することができる。その後、デッドロック発生を検出した時点で、メモリ内容が状態１３０２のように値が書き換わっていたとしても、スナップショット１３０１をメモリ状態１３０２に上書きすることで、メモリ内容を復元できる。なお、スナップショットは、フック有りＬｏｃｋ命令内で実行すれば、デッドロック発生前のメモリ内容を保持することができる。

【００６１】

図１２に戻って、ステップ３５３において、処理スレッド２００の取得済みロックハンドルの内、デッドロック要因となっているロックハンドル（Ａ）および（Ｂ）と、その間に取得したロックハンドルを解放することで、Ｕｎｌｏｃｋ待ちで停止している処理スレッド２００が復帰場所ＦｕｎｃＡＢから処理を再開することが可能になる。

【００６２】

以上により、デッドロック検出、回避、解消が可能となった。

本実施例によれば、多重ロックを取得する時点で予め必要なロックが全て既知である前提条件が不要となり、どのようなソフトウェアにも汎用的に適用可能となる。また、デッドロックが発生しても、フェールオーバを実施することなく処理を再開できるため、ソフトウェアの信頼性および稼働率を向上することができる。

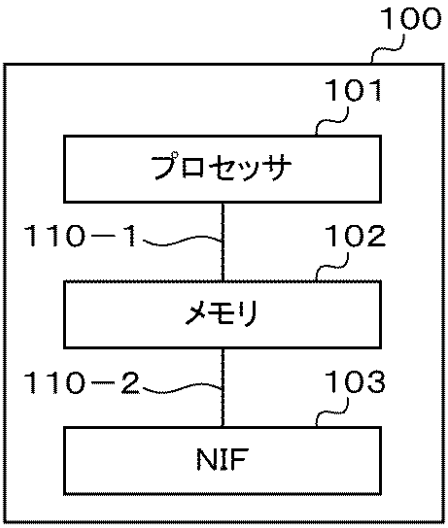
【符号の説明】

【００６３】

１００…通信装置、１０１…プロセッサ、１０２…メモリ、１０３…ＮＩＦ、１１０…バス、２００…処理スレッド、２１０…監視スレッド、２２０…フック、２３０…キュー、４００…ロックレコード、４５０…多重ロックレコード、５００…Ｌｏｃｋ命令、５５０…フック有りＬｏｃｋ命令、６００…Ｕｎｌｏｃｋ命令、６５０…フック有りＵｎｌｏｃｋ命令、７００…ロックレコードテーブル、８００…多重ロックレコードテーブル、１３０１…スナップショット

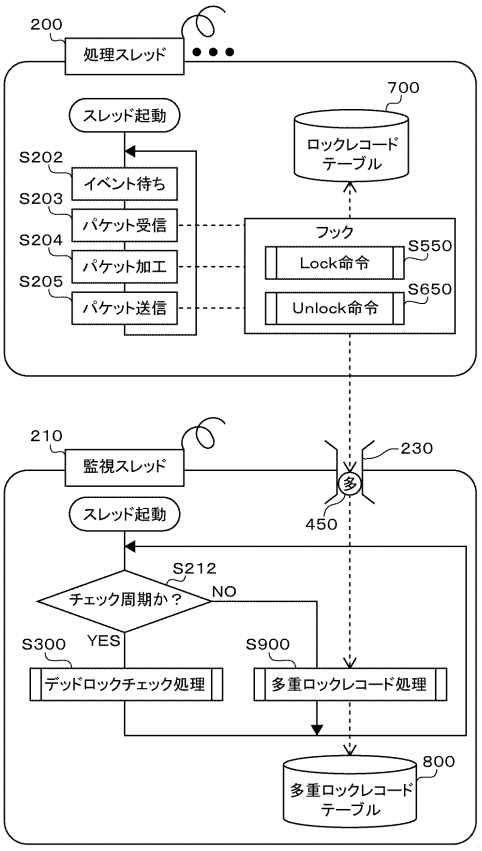
【図１】

図１



【図２】

図２



【図３】

図３

(a)

400

401	402	403
ネスト数	ロックハンドル	復帰場所
－	－	－

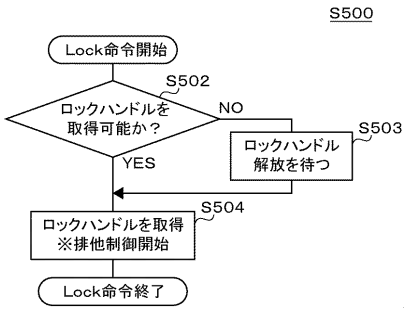
(b)

450

451	452	403	453
項番	ロック順序	復帰場所	追加ロックハンドル
－	－	－	－

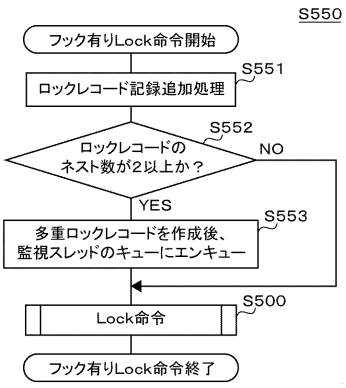
【図４Ａ】

図４Ａ



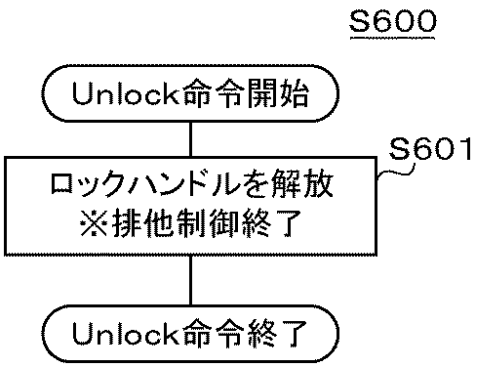
【図４Ｂ】

図４Ｂ



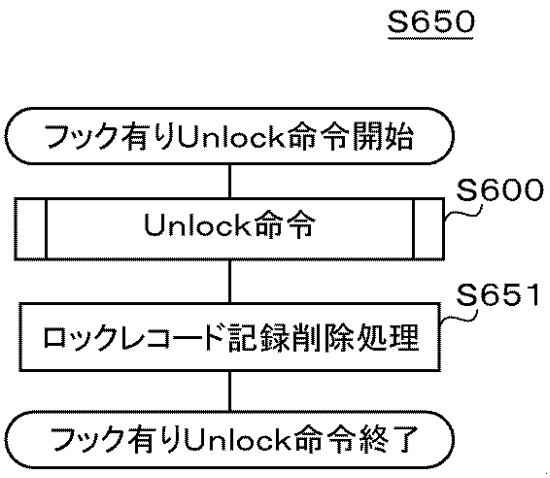
【図５Ａ】

図５Ａ



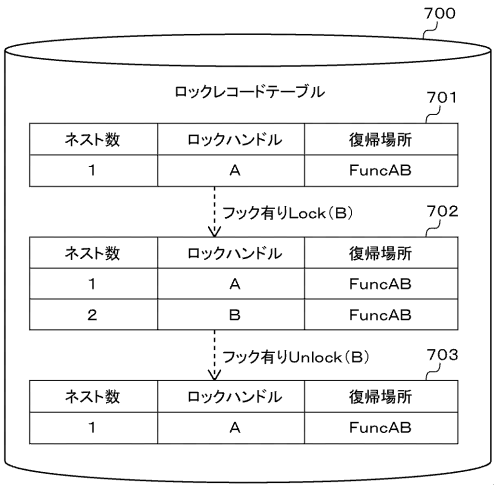
【図５Ｂ】

図５Ｂ



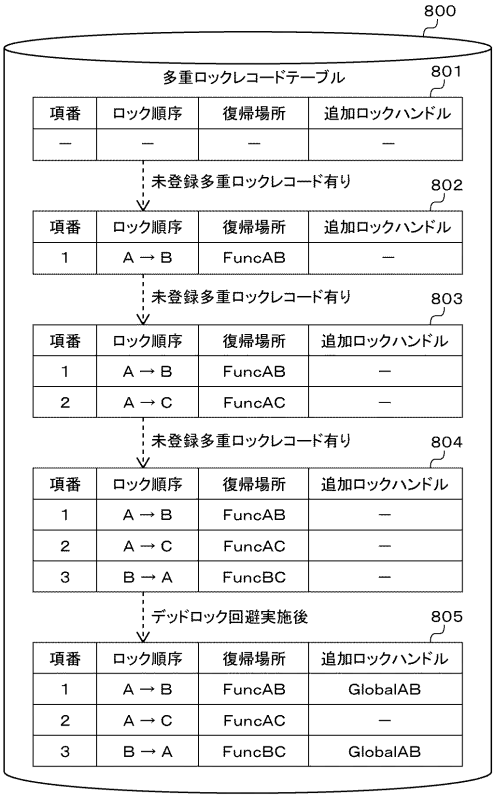
【図６】

図６



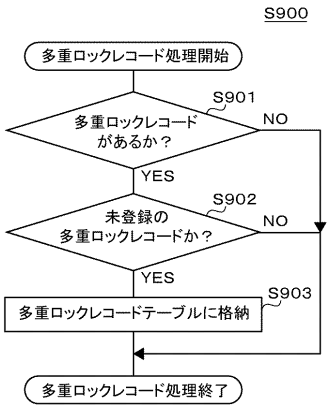
【図７】

図７



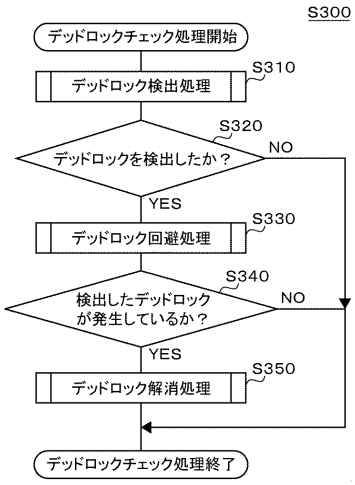
【図８】

図８



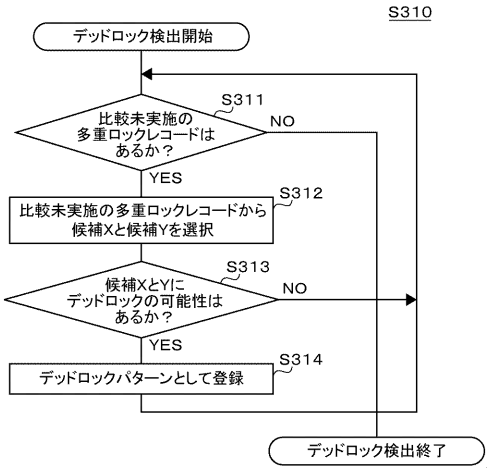
【図９】

図９



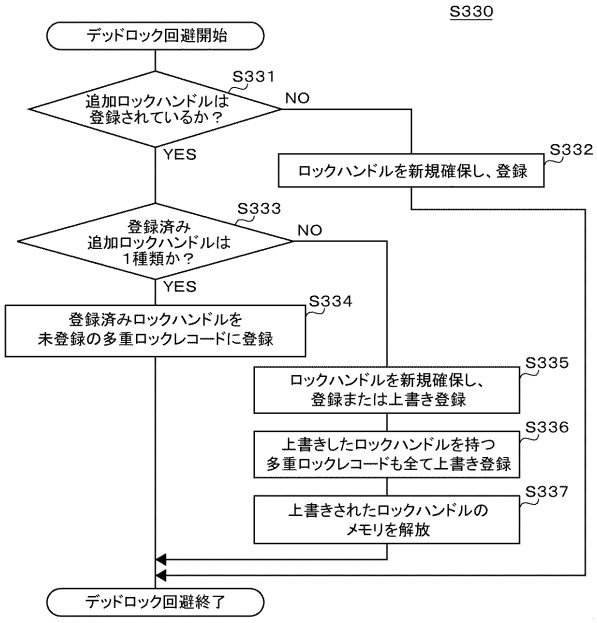
【図１０】

図１０



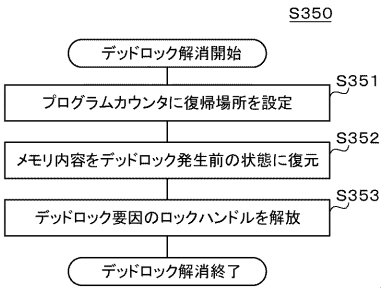
【図１１】

図１１



【図１２】

図１２



【図１３】

図１３