システムソフトウェアと
オペレーティング・システム 61−12
（１９９３．８．１８）

リライアブルなマイクロカーネルの実現について

加藤 暢敬† 明石 創† 大久保 英嗣† 大野 豊† 白川 洋充††

†立命館大学理工学部情報工学科

††近畿大学理工学部経営工学科

リライアブルなマイクロカーネルの設計方法について述べる．一般に，分散システムでは故障からの回復をチェックポインティングとロールバックで行う手法が用いられる．この手法ではステータスは安定記憶に保存される．しかし，リアルタイム性を考慮したオペレーティングシステムでは安定記憶への保存は，効率の点で最適とはいえない．そこで，本稿では安定記憶の代りに他ノード上の揮発メモリにステータスを保存する弱い永続性を提案する．また，グローバルに一貫性を保つチェックポイントの取得方法についても述べる．

A Design of a Reliable Microkernel.

Nobutaka Kato† Hajime Akashi† Eiji Okubo† Yutaka Ohno†
Hiromitsu Shirakawa††

†Department of Computer Science and Systems Engineering,
Faculty of Science and Engineering, Ritsumeikan University
56-1 Tojiin Kita-machi, Kita-ku, Kyoto 603, Japan

††Department of Industrial Engineering,
Faculty of Science and Engineering, Kinki University
3-4-1 Kowakae, Higashi-Osaka, 577, Japan

This paper describes a design method of a reliable microkernel. Usually distributed system employs the checkpointing and rollback recovery for error recovery. States of the system are saved in the stable storage. This method is suitable only if application programs are considered, however, in the case of time critical operating system stable storage is not suitable for efficiency. This paper proposes weak persistence in that volatile storage of other node is used instead of stable storage. We also present how consistent global checkpoint is taken.

1 はじめに

近年分散処理への関心が高まっている．このような中で分散システムの実現に関する諸課題は徐々に解決されてきてきた．これに伴い，研究の関心は分散システムの本質的な重要課題である信頼性向上の問題に移ってきた [1][2]．分散システムでは信頼性向上のために，チェックポイントを安定記憶にとり，故障の場合には前のチェックポイントの位置にロールバックして処理を再開するという方法が一般的にとられている．従来，分散システムの信頼性向上ということは，応用プログラムに限って議論しており，分散システムのあるノードがクラッシュしても，ノードが立ち直れば応用プログラムが再開できることを意味している．それでは，オペレーティングシステム（以降，OS と略す）の信頼性向上は応用プログラムの信頼性向上とは独立したものであるのかという疑問が生じる．

さて，近年，OS の実現方法はマイクロカーネルの手法を取り入れることにより，カーネルの肥大化を防ぐとともに，従来はカーネルに存在した機能の多くをサーバで実現することが考えられている．このことは OS を応用プログラムとあまり区別しないということを意図している．それでは OS の信頼性向上を応用プログラムの信頼性向上と同じ概念で扱えないかということが考えられる [3][4]．その場合，最大の課題は，チェックポイントのシステムに与えるオーバヘッドと前のチェックポイントの位置にロールバックして処理を再開するまでの時間遅れである．これは，リアルタイムを指向した OS では重要な問題である．チェックポイントよってステータスを安定記憶に保存する手法を採用する限り，オーバヘッドの軽減と時間遅れを減少させることは望めないであろう．ここで我々は OS のステータスを安定記憶に保存する，いわゆる強い永続性を OS に持たせるのではなく，チェックポイントによるステータスの記憶場所を他ノード上の揮発メモリ上とする弱い永続性の提案を行う．

強い永続性では一度保存したステータスは失われることはない．しかし，保存したステータスの一貫性を保つためのオーバヘッドは無視できない．一方，他ノードにステータスの保存を行う弱い永続性では，ステータスの保存を行っているノードがクラッシュするとステータスも同時に失われてしまうという恐れがある．しかし，ステータスの保守・管理は他ノードが行うのでオーバヘッドは少なくてすむという利点がある．また，ステータスを他のノード上に保存することにより，このステータスをメッセージとして交換するときに相手のクラッシュを検出できるということが考えられる．さらにクラッシュが生じた場合においても，チェックポイントの一貫性を保持する機構によりメッセージ交換中あるいは交換後にシステムがクラッシュしても直ちに元のステータスに回復できる．

本稿では，上記の弱い永続性実現のためのモデルついて述べた後に，各機構の詳細について述べる．

2 システムモデル

OS の信頼性を向上するために，他ノード上へステータスの保存を行う弱い永続性の導入を以下の前提をもとに行った．

- ノード同士がメモリを共有することはない．他ノードとの通信はメッセージパッシングで行う．
- 通信網自体の故障は生じない．
- ノードの故障は他のノードに影響を与えない．

使用した OS は分散 OS Theta[5] である．Theta はカーネルの機能を最小限にとどめたマイクロカーネルと，システムの大部分の機能を実現するシステムサーバとライブラリ関数で構成される．Theta で用いられている実行単位はスレッドである．

Theta に以下の機能を組み込むことにより，弱い永続性の実現が可能となる．

- ステータスの保存を行うチェックポイント機能.
- クラッシュの検出機能.
- ステータスの保守・管理機能.
- クラッシュを検出した後のシステムの一貫性・永続性を保持する機能.
- ロールバックによるステータスの回復を行う機能.

3 チェックポインティング

ノードは，スレッドごとのステータスを管理することで管理できる．したがって，ステータスを保存しておけば，ノードがクラッシュしてもその保存してある時点から再開することができる．この保存されるステータスがチェックポイントである．

3.1 弱い永続性

分散システムでは通常チェックポイントは安定記憶に保存される．しかし，分散OSではチェックポイントを他ノードの揮発メモリ上に保存しておけば，そのノードがクラッシュするまではチェックポイントは失われることはない．そこで我々はこの他ノードの揮発メモリ上をチェックポイントの保存場所とする弱い永続性の提案を行う．

弱い永続性ではチェックポイントを他ノードへ保存するのでチェックポイントを送信するだけでよい．したがって，チェックポイント管理のオーバヘッドを軽減することができる．

3.2 グローバルな一貫性

各プロセスが個々にチェックポイントを保存した場合には全システムの一貫性を保つことはできない．したがって，全システムの一貫性を保つようなチェックポイントの集合（グローバルなチェックポイント）が必要となる．文献[6]によると，グローバルなチェックポイントがグローバルに一貫性を保つのは以下の条件を満たす場合である．

(1) プロセス P_iからプロセス P_jにメッセージが送られる場合，プロセス P_jのチェックポイントの前に受理されるすべてのメッセージはプロセス P_iのチェックポイントの前に送信されなければならない（図1 (a)参照）．

(2) プロセス P_iからプロセス P_jにメッセージが送られる場合，プロセス P_iのチェックポイントの前に送信され，プロセス P_jのチェックポイントの後に受信されるすべてのメッセージを考える．もしプロセスがグローバルなチェックポイントにロールバックされなければならないのならメッセージのログをとっておかなければならない（図1 (b)参照）．

(3) プロセスが次のチェックポイントを取ることができるのは前のグローバルなチェックポイントが完了している場合に限る．

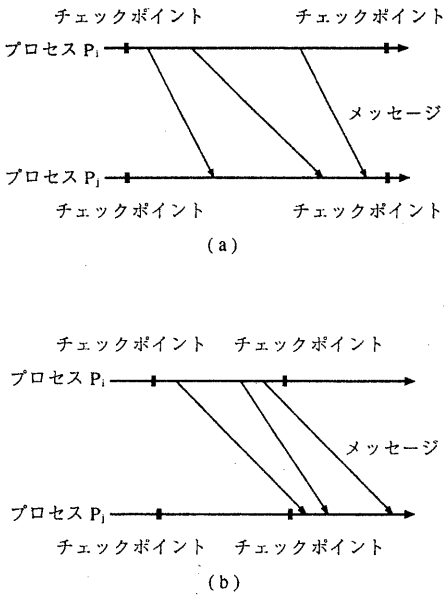

図1 一貫性を保持したチェックポイントの組

3.3 チェックポイントの更新

チェックポイントの更新時には必ずグローバルな一貫性を保持しなければならない．そこで，グローバルなチェックポイントには単調増加の数が割り当てられる（図2参照）[6]．そして，ノードがクラッシュした場合には，グローバルな一貫性が保持できるところからロールバックされることになる．

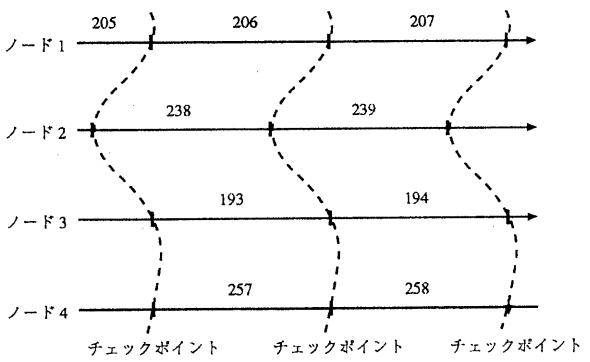

図2　一貫性を保持したチェックポイントの更新

チェックポイントの更新は以下の順で行われる．他ノードのチェックポイントを保持するノードをマスタノード，保持を依頼するノードをスレーブノードと呼ぶ．チェックポイントの更新を要求するのは，クラッシュしない限り必ず同じマスタノードである．この最初に要求を行うマスタノードをイニシエータノードと呼ぶ．

(1) イニシエータノードはチェックポイントを取得後，ブロードキャストで他の全ノードにチェックポイントの取得を要求．各ノードはチェックポイントを取得後イニシエータノードへ応答．

(2) イニシエータノードは各ノードに他ノードへのチェックポイントの保存を要求．各ノードはマスタノードへチェックポイントを保存する．マスタノードはすべてのチェックポイントが揃った後にイニシエータノードに応答．

(3) イニシエータノードは各ノードにグローバルなチェックポイントの終了を通達し，古いチェックポイントの破棄を要求．各ノードは古いチェックポイントを破棄後イニシエータに応答．

図3に一貫性を保持したチェックポイントの更新の例を示す（図では確認は省略）．ここではノード2がイニシエータノードである．

3.4 チェックポイントの間隔

チェックポイントの間隔は信頼性の上からは短いほうが望ましい．しかし，間隔が短くなるとチェックポイントの処理にかかるオーバヘッドが大きくなる．間隔が長くなるとクラッシュから回復後のオーバヘッドが大きい．また，チェックポイントが不定期であると，上記のトレードオフを図ることはできないのでチェックポイントは周期的に行う必要がある．

4 チェックポイントの管理

取得されたチェックポイントには更新，保存等の処理が行われる．これらの処理の間に一貫性が失われてはならない．したがって，チェックポイントは管理する機構によって，一貫性を保つ必要がある．本システムでは以下の2種類の機構で管理される．

- ノードマネージャ
- システムマネージャ

ノードマネージャは各ノードに存在し，自ノードのチェックポイントの管理を行う．システムマネージャは他ノードから依頼されたチェックポイントの管理を行うものであり，全システム中に最低2個存在する．

図4に図3の場合のそれぞれのマネージャの配置の例を示す．ノード2がイニシエータノードであり，ノード3がマスタノードであるのでどちらのノードにもシステムマネージャが存在する．

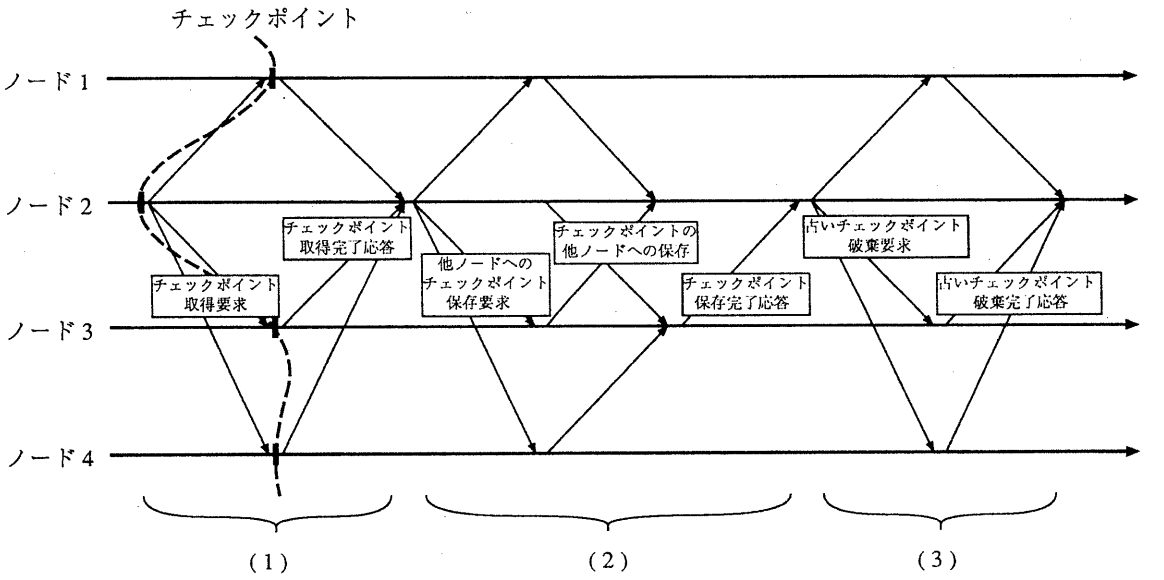

図 3　一貫性を保持したチェックポイントの更新

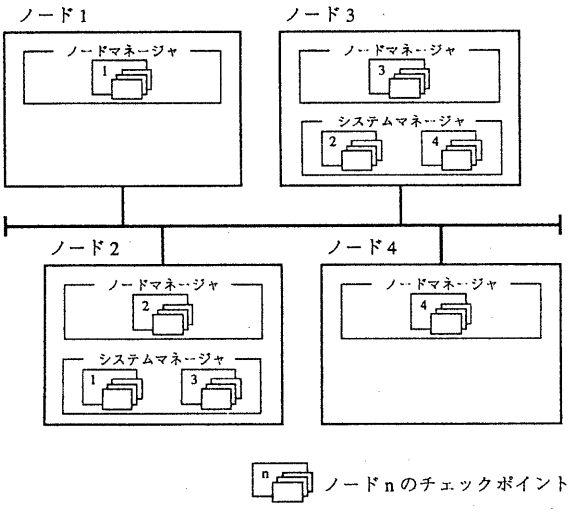

図 4　マネージャの配置例

4.1　ノードマネージャ

各ノードに存在するノードマネージャでは以下のような処理を行う．

(1) **管理**　自ノードのチェックポイントの更新を行う．また，それらのチェックポイントの管理も行う．必ず自ノード内のチェックポイントの一貫性は保持する．

(2) **スレッドの登録，抹消**　スレッドが生成，消去された場合にはそのスレッドを管理する管理テーブルへの登録，抹消を行う．

(3) **システムマネージャへの要求**　他ノード上に存在するシステムマネージャに対して通信機構を用いて，チェックポイントの管理の依頼や，チェックポイントの送信を行う．

(4) **ロールバック**　他ノードのノードクラッシュを知った場合に，前回のチェックポイントから後にメッセージを受信していたらチェックポイントをロールバックして再スタートする．

4.2 システムマネージャ

他ノードのシステムステータスを保守・管理するシステムマネージャでは以下のような処理を行う.

(1) **チェックポイントの登録, 抹消** 他ノードから管理を依頼された各チェックポイントの登録, 抹消を行う.

(2) **チェックポイントの更新** 他ノードから送信される各チェックポイントの更新. この場合も必ず一貫性は保持する.

(3) **チェックポイントのロールバック** クラッシュしていたノードが復帰したときにそのノードのチェックポイントをロールバックする.

(4) **更新要求** イニシエータノードのシステムマネージャは全ノードへのチェックポイントの更新要求を周期的に行い, 各ノードに指示を与える.

5 ノードクラッシュの検出

ノードのクラッシュの検出は他ノードによって行われる. 現在のところ, ノードのクラッシュの検出はタイムアウトで判断する方法しかない. クラッシュとみなされるのは以下の場合である.

(1) **要求に対する応答がない場合** 他ノードへ処理の依頼の要求を行ったにも関わらず応答がなく, 要求の再送 (タイムアウト) が一定回数を越えた場合, そのノードはクラッシュしているとみなす.

(2) **チェックポイント更新の要求がない場合** 各ノードはチェックポイントを周期的に他ノードに保存している. しかし, イニシエータノードがクラッシュした場合にはチェックポイントの更新要求が発行されないので更新することはできなくなる. したがって, 他のマスタノードによって, イニシエータノードから長い間チェックポイントの更新要求が発行されなければクラッシュとみなされる.

6 ロールバックによる回復処理

計算機が単独で稼動している場合と違い, ネットワーク上につながっているとノードがクラッシュした場合には様々な処理が必要となる. 永続性を導入した場合には, クラッシュしたノードが回復しても一貫性を保持できるように, チェックポイントをロールバックしてその時点から再スタートする必要がある. また, 弱い永続性では他ノードにチェックポイントを定期的に保存しているので, ノードのクラッシュの検出時, 回復時にはその他のノードの永続性も考慮した処理が行われなければならない.

ノードクラッシュ検出後の処理は, 各ノードの一貫性を保つロールバック及び永続性を保持する処理と, ノード自身の回復処理に分けられる.

6.1 一貫性・永続性保持

ノードクラッシュが検出された場合には, 全システムの一貫性と永続性を保つ処理を行わなければならない. この処理を行うのはクラッシュしたノードのチェックポイントを管理しているノードである. したがって, ノードクラッシュを検出したのがチェックポイントを管理しているノードでなければ, ノードクラッシュの検出を行ったノードはチェックポイントを管理しているノードから応答があるまで全ノードに対してノードクラッシュを通知する.

その後に, チェックポイントを管理していたノードの行う処理は以下のものである.

(1) ノードクラッシュを検出したノードが通知したノードクラッシュのメッセージが全てのノードに届いている保証はない. そこで, 改めて全ノードに対して全ノードから応答があるまでクラッシュしたノードへのメッセージ送信の中断の通知を行う.

(2) クラッシュしたノードにシステムマネージャが存在していた場合には, 永続性保持の処理を行う. 全システム上にシステムマネージャが1個しか存在しなくなっていた時には, ど

こか他のノードにその代行となる新たなシステムマネージャの依頼を行う．まだ2個以上存在する時には，現存のシステムマネージャに依頼を行ってもよい．ただし，1個のシステムマネージャにあまりに多くのノードの管理を要求すると処理しきれない恐れがあるので，その時の状況によって判断しなければならない．

(3) 代行のシステムマネージャの存在するノードを通知する．これも必ず応答を得る．

(4) メッセージのログを保存．

(5) これらの処理の後，自ノード内のチェックポイントをロールバックして，再スタートする．

ノードクラッシュを知らされたその他のノードは上記の通知にしたがって，以下の処理を行う．

(1) クラッシュしたノードに対するメッセージ送信の中断．

(2) クラッシュを通知しているノードへの応答．

(3) メッセージのログの保存．

(4) 代行のシステムマネージャを依頼された場合にはその実行．

(5) クラッシュしたノードにチェックポイントを保存していたノードは，チェックポイントの保存ノードを新たに代行システムマネージャを起動したノードに変更する．

(6) これらの処理を行った後，自ノード内のチェックポイントをロールバックして，再スタートする．

図4のマネージャの配置でノード3がクラッシュした場合にはノード1に代行システムマネージャを起動する（図5参照）．これはノード3がノード4のステータスも保持しているためである．ノード4に代行システムマネージャを立ち上げた場合には，ノード4のステータスの保存をノード2で行わせる処理が必要となる．

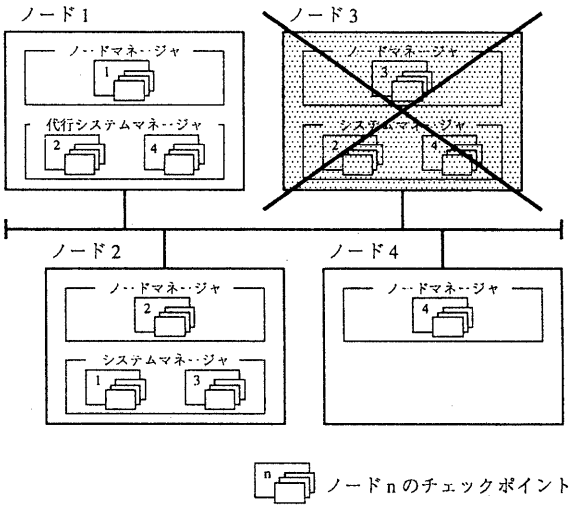

図5　代行システムマネージャの起動

6.2　ノードの回復

クラッシュしていたノードが再起動されてクラッシュする前の状態から再開するには，他ノードからチェックポイントのロールバックを行う．ノードの回復には以下の処理を自分自身で行う必要がある．

(1) クラッシュする前のチェックポイントを保存しておいたノードからロールバックする．

(2) 全ノードにノード回復の通知を行う．これは全ノードから応答を得るまで続ける．この通知を得たノードはこのノードに対してのメッセージの送信を再開する．

(3) 回復したノードにシステムマネージャが存在していて，システムマネージャも回復する場合には該当のノードに通知を行う．システムマネージャの回復をしない場合には，存在したシステムマネージャを削除する．この場合，システムマネージャの処理は代行のシステムマネージャに任せたままになる．

(4) ロールバックした時点から再スタートし，通常の処理に戻る．

チェックポイントの保存してある時点から再起動しない場合には，起動時に初期化を行う．ただし，チェックポイントの保存してある時点から再開する場合でも，システムマネージャの再開については，再開する，しないの選択が必要である．

7 おわりに

本稿では，OSへの弱い永続性の導入について述べた．チェックポイントを他ノードへ保存することにより，信頼性を向上させることができる．また，クラッシュした場合でも，システムの一貫性を素早く保つことができる．しかし，ノードと同時にそのノードのチェックポイントを管理しているノードもクラッシュした場合には永続性は失われてしまう．したがって，信頼性とオーバヘッドとのトレードオフを図りながら，安定記憶装置へチェックポイントを保存する，強い永続性の導入についても検討を行う必要がある．

参考文献

[1] R. Koo and S. Toueg: *Checkpointing and Rollback-Recovery for Distributed Systems*, IEEE Transactions os Software Engineering, SE-13(1), pp. 23-31, January 1987.

[2] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel: *The Performance of Consistent Checkpointing*, in Proc. 11th Symposium on Reliable Distributed Systems, pp. 39-47, 1992.

[3] T. P. Ng: *The Design and Implementation of a Reliable Distributed Operating System-ROSE*, in Proc. 9th Symposium on Reliable Distributed Systems, pp. 2-11, 1990.

[4] C. R. Landau: *The Checkpoint Mechanism in KeyKOS*, in Proc. 2nd International Workshop on Object Orientation in Operating Systems, pp. 86-91, 1992.

[5] H. Shirakawa and E. Okubo: *When Object-Oriented Operating System is Time Critical*, in Proc. 1992 EUROMICRO Workshop on Real Time System, pp. 54-59, June 1992.

[6] L. M. Silva and J. G. Silva: *Global Checkpointing for Distributed Programs*, in Proc. 11th Symposium on Reliable Distributed Systems, pp. 155-162, 1992.