NAIST-IS-DT9761023

博士論文

無待機分散アルゴリズムに関する研究

守屋 宣

2000 年 2 月 7 日

奈良先端科学技術大学院大学
情報科学研究科 情報処理学専攻

本論文は奈良先端科学技術大学院大学情報科学研究科に
博士 (工学) 授与の要件として提出した博士論文である。

守屋 宣

審査委員：　藤原 秀雄 教授
福田 晃 教授
伊藤 実 教授
増澤 利光 助教授

無待機分散アルゴリズムに関する研究*

守屋 宣

内容梗概

自律的に動作する複数のプロセスからなる分散システムにおいて, プロセスを効率的に動作させるためのアルゴリズムを分散アルゴリズムとよぶ. 分散システムの特長の 1 つは, その潜在的な冗長性を活用することにより, 故障耐性を有するシステムを構築できるということである. そこで, 故障耐性を有する分散アルゴリズムが重要である. 特に, 停止故障に対する高度な故障耐性を有する分散アルゴリズムとして, 無待機分散アルゴリズムが注目されている. 無待機分散アルゴリズムとは, 各プロセスが他のプロセスの動作を待つことなく有限時間内にアルゴリズムを完了できるような分散アルゴリズムである.

本論文では, 基本的な無待機分散アルゴリズムとして, 無待機時計合わせアルゴリズム, 線形化可能性を保証する共有オブジェクト無待機実現アルゴリズムに関する考察を行う.

第 2章では, フェーズ内システムとよばれる同期式共有メモリシステムにおいて, 時計合わせ問題に対する無待機分散アルゴリズムについて考察する. ここで時計合わせ問題とは, システム内のプロセスが保持する局所時計の値を一致させる問題である. まず, フェーズ内システム上の無待機時計合わせアルゴリズムにおいて, 同期時間とよばれる評価尺度の下界がプロセス数 n に対し $n-2$ であることを示す. 次に, 同期時間がオーダー的に最適な無待機時計合わせアルゴリズムを提案する. また, 一時故障耐性を意味する自己安定性とよばれる性質を有し, さらに空間複雑度が有界であるような, 同期時間がオーダー的に最適な無待機時計合わせアルゴリズムの提案も行う.

*奈良先端科学技術大学院大学 情報科学研究科 情報処理学専攻 博士論文, NAIST-IS-DT9761023, 2000 年 2 月 7 日.

第 3章では, 同期式メッセージパッシングシステム上に線形化可能性を保証する共有オブジェクトを実現する無待機分散アルゴリズムの考察を行う. まず, 放送モデルとして信頼放送モデル, 無信頼放送モデルを導入し, 局所時計モデルとして非同期時計モデル, u-同期時計モデルを導入する. そして, 放送モデル, 局所時計モデルのすべての組合せに対して read/write-レジスタを無待機に実現するアルゴリズムを提案する. また, 信頼放送モデル上で先に挙げた 2 種類の局所時計モデルに対して一般オブジェクトを実現する無待機分散アルゴリズムも提案する.

キーワード

分散システム, 分散アルゴリズム, 共有メモリシステム, 故障耐性, 無待機性

Studies on Wait-Free Distributed Algorithms*

Sen Moriya

Abstract

A distributed system consists of multiple autonomous processes. A distributed algorithm makes processes cooperate to work efficiently in a distributed system. A distributed system has an advantage such that we can construct a fault-tolerant system by drawing out its potential redundancy. Therefore, a distributed algorithm with fault-tolerance is important. Especially, a wait-free distributed algorithm which has excellent fault-tolerance to crash fault is attractive. In a wait-free distributed algorithm, a process does not have to wait for other processes' work to complete a computation in finite time.

In this dissertation, we consider wait-free distributed algorithms for two fundamental problems: one is a clock synchronization problem and the other is linearizable implementation of shared objects.

In chapter 2, we consider a wait-free clock synchronization algorithm, which synchronizes local clocks of processes, on a synchronous shared-memory system called an in-phase system. First, we show the lower bound of synchronization time, which is an efficiency measure, is $n-2$ where n is the number of processes. Next, we propose a wait-free clock synchronization algorithm with asymptotically optimal synchronization time. Furthermore, we propose an asymptotically optimal wait-free clock synchronization algorithm with self-stabilization, which implies the system tolerates transient fault, and space-boundedness.

In chapter 3, we consider wait-free linearizable implementation of a shared object on a synchronous message-passing system. We introduce two kinds of models

*Doctor's Thesis, Department of Information Processing, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-DT9761023, February 7, 2000.

to a broadcast, a reliable broadcast model and an unreliable broadcast model. We introduce two kinds of models also to local clocks, an asynchronous clock model and a u-synchronous clock model. We propose wait-free linearizable implementations of read/write-registers for all combinations of a broadcast model and a clock model. Furthermore, we propose wait-free linearizable implementations of general objects for both clock models where we assume only a reliable broadcast model.

Keywords:

distributed system, distributed algorithm, shared-memory system, fault-tolerance, wait-freedom

関連発表一覧

- 学術論文誌

  1. 守屋 宣, 井上美智子, 増澤利光, 藤原秀雄, "共有メモリマルチプロセッサシステムにおける同期時間最適な無待機時計合せプロトコル," 電子情報通信学会論文誌 (D-I), Vol.J83–D–I, No.1, pp.99–109, 2000.
  2. S. Moriya, K. Suda, M. Inoue, T. Masuzawa and H. Fujiwara, "Wait-Free Linearizable Implementation of a Distributed Shared Memory," IEICE Transaction on Information and Systems, (条件付採録).

- 国際会議 (査読付き)

  1. M. Inoue, S. Moriya, T. Masuzawa and H. Fujiwara, "Optimal Wait-Free Clock Synchronization Protocol on a Shared-Memory Multi-Processor System," Proceedings of the 11th International Workshop on Distributed Algorithms (Lecture Notes in Computer Science 1320), pp.290–304, 1997.
  2. S. Moriya, M. Inoue, T. Masuzawa and H. Fujiwara, "Self-Stabilizing Wait-Free Clock Synchronization with Bounded Space," Proceedings of the 2nd International Conference on Principles of Distributed Systems, pp.129–143, 1998.
  3. S. Moriya, K. Suda, M. Inoue, T. Masuzawa and H. Fujiwara, "Wait-Free Linearizable Distributed Shared Memory," Proceedings of the 11th IASTED International Conference on Parallel and Distributed Computing and Systems, pp.335–342, 1999.

- 研究会報告

  1. 守屋 宣, 井上美智子, 増澤利光, 藤原秀雄, "共有メモリマルチプロセッサシステムにおける同期時間最適な無待機時計合わせプロトコル," 電子情報通信学会技術報告, COMP96–69, pp.85–92, 1997.

2. 守屋 宣, 井上美智子, 増澤利光, 藤原秀雄, ”共有メモリシステムにおける同期時間最適な自己安定無待機時計合わせプロトコル,” 電子情報通信学会技術報告, COMP98–18, pp.31–38, 1998.

3. 須田克朗, 守屋 宣, 井上美智子, 増澤利光, 藤原秀雄, ”線形化可能性を保証する共有オブジェクトの無待機な実現,” 電子情報通信学会技術報告, COMP98–73, pp.9–15, 1999.

4. S. Moriya, K. Suda, M. Inoue, T. Masuzawa and H. Fujiwara, ”Wait-Free Linearizable Implementation of a Distributed Shared Memory,” Technical Report of IEICE, COMP99–37, pp.45–52, 1999.

目 次

図 目 次

表 目 次

第 1 章

序論

1.1. 無待機分散アルゴリズム

自律的に動作する複数のプロセスとそれらが情報を交換するための通信媒体にからなるシステムを分散システム (*distributed system*) とよぶ. 近年, 局所ネットワークや広域ネットワークが整備されてくるに従って, 分散システム下で実行される計算の重要性が高まりつつある. 分散システムの長所として, 以下の点が挙げられる.

1. 高機能性：全体を制御する必要から生じるボトルネックなしに, 計算を実行可能.
2. 拡張性：環境の変化に従って, システムを段階的に充実することが可能.
3. 故障耐性：システムの冗長性を活用することにより, 故障耐性を有するように構築することが可能.
4. 資源共有：データベースなどの情報の共有が可能.

分散システムはプロセス間通信に使用される機構により区別される. 代表的な分散システムに, プロセスがメッセージを送受信することにより通信するメッセージパッシングシステム (*message-passing system*), 共有メモリを介して通信を行う

共有メモリシステム (*shared-memory system*) がある. また, 分散システムはプロセス, 局所時計, 通信の同期条件によっても区別される.

分散システムの下でプロセスを効率的に協調動作させるためのアルゴリズムを分散アルゴリズム (*distributed algorithm*) とよぶ. 先に述べたように, 分散システムは故障耐性を有するように構築することが可能である. そこで近年, さまざまな故障耐性を有するような分散アルゴリズムに関する研究が注目されている. 特に, プロセスの故障モデルとして, 故障するまでは正常に動作するが故障後は永遠に停止する停止故障 (*crash*), 一時的に任意にプロセスの状態が変化する一時故障 (*transient fault*), 同期式分散システムで同期パルスを受信しても動作しない居眠り故障 (*napping fault*) といったモデルがある.

高度な故障耐性を有する分散アルゴリズムの１つに, 無待機分散アルゴリズムがある. 無待機分散アルゴリズムとは, 各プロセスが他のプロセスの動作を待つことなくアルゴリズムを完了できるような分散アルゴリズムである. これは, プロセスの停止故障, あるいは居眠り故障を仮定した分散システムでは, 任意個のプロセスの故障に対する耐性を意味する. 近年, さまざまな無待機分散アルゴリズムが研究されているが, 本論文では基本的な無待機分散アルゴリズムとして次の２つの無待機アルゴリズムの考察を行う.

- 無待機時計合わせアルゴリズム (wait-free clock synchronization algorithm)
- 共有オブジェクト無待機実現アルゴリズム (wait-free implementation of shared objects)

1.2. 同期式共有メモリシステムにおける無待機時計合わせアルゴリズム

分散システムにおける重要な問題の１つに, プロセス間の同期を実現することがある. プロセス間の同期をとるための手段として, 大域時計がよく用いられる. しかし, 全プロセスが共通の１つの時計を参照する方法では, その時計が故障すればどのプロセスも大域時計を利用できなくなるという欠点があり, システム全体の信頼性は低い. そこで, 各プロセスが個別に時計を実現し, これらの時計を同期

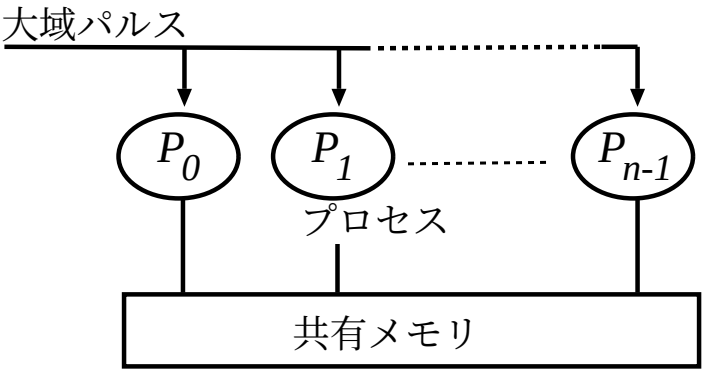


図 1.1 フェーズ内システム

させるという方法が提案されている ([5, 8, 7, 17] など). この方法では, 各プロセスが個別に時計を実現するため, 一部のプロセスが故障しても正常なプロセス間で時計を同期するように実現するといった故障耐性を持たせ, 信頼性を上げるということが考えられる. このような状況で, 各プロセスが管理する局所時計を同期させるアルゴリズムを時計合わせアルゴリズムとよぶ.

故障プロセスが存在するシステムにおける時計合わせ問題は多くのアプリケーションにとって重要であり, またそれ自体も興味深い問題である. 時計合わせ問題は, これまで様々なモデルの下で, 様々な結果が示されている. 特に, 共有メモリシステムにおいて各プロセスが共通の大域パルスを共有するというフェーズ内システム (*in-phase system*)(図 1.1) に関しては, Dolev らによって故障プロセッサが存在するフェーズ内システムの下での時計合わせプロトコルが提案された [7]. Dolev らは, プロセスの故障として任意時間動作を停止した後に動作停止に気づかずに動作を再開するという居眠り故障 (*napping fault*) を対象としている. 彼らのプロトコルは, ある定数 kに対し, 以下の 2 つの条件を保証する. (1) kパルスの間, 正常に動作し続けたプロセスは, それ以降正常に動作し続ける限り, 局所時計の値は各パルスで 1 ずつ増える. (2) kパルスの間, 正常に動作し続けた 2 つのプロセスの局所時計の値は一致する. このプロトコルは, 各プロセスは少なくとも k パルス動作し続ければ他のプロセスの動作に関わらず局所時計の値を一致させることができるという意味で, 無待機時計合わせアルゴリズムとよばれる. ここで,

定数 kを同期時間 ($synchronization\ time$) とよぶ. 無待機時計合わせアルゴリズムでは, 居眠り故障をおこすプロセスも kパルス正常に動作し続ければ局所時計を同期させることができ, また, 任意個のプロセスが居眠り故障をおこす場合でも故障プロセスの動作に影響されることなく局所時計を同期させることができる.

本論文では, 第 2章でフェーズ内システムにおける無待機時計合わせアルゴリズムについて考察する. まず, フェーズ内システム上の無待機時計合わせアルゴリズムにおいて, 同期時間とよばれる評価尺度の下界がプロセス数 n に対し $n-2$ であることを示す. 次に, 同期時間がオーダー的に最適な無待機時計合わせアルゴリズムを提案する. また, 一時故障耐性を意味する自己安定性とよばれる性質を有し, さらに空間複雑度が有界であるような, 同期時間がオーダー的に最適な無待機時計合わせアルゴリズムの提案も行う.

1.3. 線形化可能性を保証する共有オブジェクトの無待機な実現

メッセージパッシングシステムは, ハードウェアの実現やシステムの拡張が容易である反面, その複雑な構造によりプロセスのメッセージの送受信によるプログラミングは複雑になりがちである. ここで, メッセージパッシングシステム上に論理的な共有オブジェクトを実現することを考える. プロセスが共有オブジェクトを介して通信を行うようにすることで, プログラミングは単純化される. そこで, メッセージパッシングシステム上に仮想的に共有オブジェクトを実現する効率のよいアルゴリズムに関する研究が行われている.

実現された共有オブジェクトは, 各プロセスの並行なアクセスに対し, あるコンシステンシ ($consistency$) を保証する必要がある. コンシステンシ条件として, 線形化可能性 ($linearizability$)([10]), 逐次コンシステンシ ($sequencial\ consistency$)([13]) などが知られている. とくに線形化可能性は, 直観的には, 実現されたオブジェクトに対する操作が逐次的に実行されたようにみえること, さらに, ある操作とその操作の完了後に呼び出された別の操作に対しその 2つの操作は実時間での順序で実行されたようにみえることを保証する. 線形化可能性にはいくつかの優れた特性がある. それは, 局所性とノンブロッキング性である. 局所性は, 個々のオ

ブジェクトが線形化可能ならばシステム全体も線形化可能であるという性質である．これにより，線形化可能性を保証する複数の共有オブジェクトから線形化可能なシステムを設計，構築することが容易になる．ノンブロッキング性はあるアクセス中の操作が他のアクセス中の操作にブロックされないような実現が可能であるという性質である．この性質は実時間応答が重要なシステムのための適切な条件である.

本論文では, 第 3章でプロセスが停止故障をおこすようなメッセージパッシングシステムにおいて, 線形化可能性を保証する共有オブジェクトを無待機に実現するアルゴリズムの考察を行う. システム内に同期の仮定を全くおかない完全非同期式メッセージパッシングシステムに対しては, read/write-レジスタを無待機に実現することが不可能であることが示されている [12]. そこで本論文では, 各プロセスが実時間スピードの局所時計を保持し, すべてのメッセージ遅延が各プロセスに既知である定数 $d, u(o < u \leq d)$ に対し $[d-u, d]$ であるようなメッセージパッシングシステムを考える. また, 各プロセスのメッセージ放送, 局所時計に関して, それぞれ 2 通りのモデルを考える. メッセージ放送に関する 2 通りのモデルは, 放送中にプロセスが故障した場合の保証が異なる. 信頼放送 (*reliable broadcast*) モデルでは, 放送されたメッセージをすべてのプロセスが受信する, またはすべてのプロセスが受信しないことを保証する. 無信頼放送 (*unreliable broadcast*) モデルでは, 放送中にプロセスが故障した場合のメッセージの受信に対して何も仮定しない. 局所時計に関する 2 通りのモデルは, 局所時計の値の差に対する仮定が異なる. u-同期時計モデルでは任意の 2 つの局所時計の値の差が高々u であるのに対し, 非同期時計モデルでは局所時計の値の差に関して何も仮定しない.

第 3章ではまず, 信頼放送／無信頼放送, 非同期時計／ u-同期時計モデル上に read/write-レジスタを無待機に実現するアルゴリズムを提案する. 次に, 信頼放送, 非同期時計／ u-同期時計モデル上に一般オブジェクトを無待機に実現するアルゴリズムも提案する.

1.4. 本論文の構成

本論文の構成は以下の通りである. 第 2章では, フェース内システムにおける無待機時計合わせアルゴリズムについて考察する. 第 3章では, 同期式メッセージパッシングシステム上に線形化可能性を保証する共有オブジェクトを無待機に実現するアルゴリズムの考察を行う. 最後に第 4章で, 以上の研究成果の結論を述べるとともに, 今後の研究課題について述べる.

第 2 章

無待機時計合わせアルゴリズム

2.1. はじめに

本章では, フェーズ内システムにおける無待機時計合わせアルゴリズムについて述べる.

過去に提案されたフェーズ内システムにおける無待機時計合わせプロトコルを表 2.1に示す. Dolev らは, 同期時間 $16n^2 + n - 1$(n:プロセッサ数) の無待機時計合わせプロトコルを提案した [7]. また, 任意の初期システム状況から開始する実行においても大域時計を実現する自己安定無待機時計合わせプロトコルとして, Dolev らは同期時間 $8n^3 + n - 1$ の自己安定無待機時計合わせプロトコルを提案した [7]. また, Papatriantafilou らは, 同期時間 $4n^2 - 3n - 1$ の自己安定無待機時計合わせプロトコルを提案した [17].

本論文では, 同期時間 $12n$ の無待機時計合わせプロトコル WCS, 同期時間 $15n$? の自己安定時計合わせプロトコル $ssWCS$を提案をする. 特にプロトコル $ssWCS$ では, 空間複雑度 (アルゴリズムで用いる共有変数のサイズ) の有界性も考慮している. 一般の自己安定性を持たないアルゴリズムは決められた初期システム状況から開始するため, 非有界変数を含むようなアルゴリズムでも, 現実的にはアルゴリズムを終えるまでに用いられる十分な空間を推測し準備することができる. これに対し, 自己安定アルゴリズムは任意のシステム状況から開始するため, 空間複雑度が非有界ならばアルゴリズム終了までの十分な空間を準備することができな

表 2.1 フェーズ内システムにおける無待機時計合わせプロトコル

		同期時間		空間複雑度
		上界	下界	
Dolev ら [7]		$16n^2+n-1$		非有界
Papatriantafilou ら [17]$\star$		$4n^2-3n-1$		非有界
本論文	WCS	$12n$		非有界
	$ssWCS\star$	$15n$		有界
			$n-1$	

$\star$ 自己安定無待機時計合わせプロトコル

い. 例えば, 自己安定アルゴリズムで単調増加する非有界変数を使うならば, この変数に対しどんなに大きいサイズを用意しても, その変数の最大値から開始することが考えられる. 従って, 空間複雑度の有界性は, とくに自己安定アルゴリズムにおいて意味のある性質であるといえる. 本論文ではさらに, 同期時間の下界が $n-2$ であることを証明することにより, 提案した 2 つのプロトコルが同期時間に関してオーダー的に最適であることを示す.

以下, 2.2節で本章で提案するアルゴリズムに関する定義を行う. 2.3節では, フェーズ内システム上の無待機時計合わせアルゴリズムの同期時間の下界が $n-2$ であることを示す. 2.4節では同期時間 $12n$ の無待機時計合わせアルゴリズムを提案し, 2.5節では同期時間 $15n$, 空間複雑度が有界な自己安定無待機時計合わせアルゴリズムを提案する.

2.2. 諸定義

本節でシステムのモデル, 無待機時計合わせ問題を定義する.

2.2.1 システム

n 個のプロセスから成る共有メモリシステムを考える. プロセスは共有変数を介してのみ通信を行うことができる. プロセスは 0 から $n-1$ までの相異なる識別子を持つとし, 識別子 i のプロセスを P_iと表す. また, 各レジスタは, 所有者とよばれるある 1 プロセスのみが書き込みをできすべてのプロセスが読み出しをできる, single-writer multi-reader レジスタとする. プロセス P_iが書き込みできるレジスタを R_iと記す. プロセス P_iはレジスタ R_iを複数のフィールドに分割して使用し, 各フィールドに 1 つの変数を割り当てることにより, 1 つのレジスタに複数の変数の値を書き込む. 各プロセスは状態機械としてモデル化される. 状態 s のプロセス P_iは, 以下の (1) から (2) の動作を 1 ステップの操作として行い, 次の状態 s'に遷移する.

(1) 状態 s により決定するレジスタ R_jを読み出す.

(2) 状態 s と R_jの値を基に, R_iを更新し, 状態 s'に遷移する.

本論文では, フェーズ内システムと呼ばれる同期式共有メモリシステムを扱う. フェーズ内システムでは, 全プロセスが共通の大域パルスを共有し, 全プロセスは大域パルスを受けたときに 1 ステップの動作を同期して行う. ただし, 本論文では後で述べる居眠り故障を考慮し, 各パルスで全プロセスが動作するとは限らないとする. システムの大域状況を, 全プロセスと全共有変数の状態の組で表す. 以降, このシステムの大域状況のことを, 単に状況という. システムの実行は, 状況と, 各パルスで動作したプロセスの集合の無限または有限交替列 $E = c_0\pi_1c_1\cdots$ で表す. ここで, 各 $c_t(t \geq 0)$ は状況を表し, 特に c_0はアルゴリズム開始時のシステム状況 (以降, 開始状況とよぶ) を表す. 各プロセス P_iは c_0における R_iの値は知っているとする[a]. 自己安定性のないアルゴリズムに対しては, c_0は各プロセス, 共有変数の特定の初期状態からなる状況 c_{init}(以降, 初期状況とよぶ) を表す. 初期システム状況 c_{init}から始まる実行を初期化実行とよぶ. また, π_tは t 番目のパル

[a]任意の開始状況を想定する自己安定アルゴリズムにおいて, プロセス P_iがレジスタ R_iの値を知っているという仮定は不自然のようにもみえる. しかし, Dolev らにより P_iが R_iの更新を更新前の値を知らずに行うような自己安定無待機時計アルゴリズムは存在しないことが示されている [7]. よって, 自己安定無待機時計アルゴリズムの考察でも, P_iは開始状況における R_iの値を知っていると仮定する.

ス発生時に 1 ステップの動作したプロセッサの集合を表す. 実行 Eは, 有限であるときはある状況 c_{end}で終る. この実行 Eは, 各 t に対し, π_tに属する各プロセスが c_{t-1}を基に同期して 1 ステップの動作を行い, 状況が c_tになったことを意味する. また, t 番目のパルスが発生した (大域) 時刻を時刻 t とよぶ. また, 状況 c_tでの変数 $P_i.x$ の値を $P_i.x(t)$ で表す.

プロセス P_iについて, $P_i \notin \pi_t$のとき, P_iは時刻 t で居眠り故障した, または単に居眠りしたという. P_iが時刻 t で居眠りしたならば, P_iは時刻 t では全く動作をしない. すなわち, c_{t-1}と c_tにおける P_i, R_iの状態は変わらない. システムの実行 $E = c_0\pi_1c_1\cdots$に対して, プロセス P_iが時刻 t まで連続して動作したパルス数を $work(P_i, t)$ と表す. すなわち, $work(P_i, t) = \max\{l | P_i \in \bigcap_{t-(l-1)\leq t'\leq t} \pi_{t'}\}$ であり, $work(P_i, t) = u$ のとき P_iが区間 $[t-u+1, t]$ で居眠りすることなく動作し続けたことを意味する. ただし, $P_i \notin \pi_t$のときは, $work(P_i, t) = 0$ と定義する.

2.2.2 無待機時計合わせ問題

次に, フェーズ内システムにおける無待機時計合わせ問題を定義する. 各プロセス P_iは, P_iの局所時計を表す共有変数 $Clock$を持つとする. 有界サイズの局所時計を考えるならば, ある決められた整数 Mに対し, $Clock$は 0 から $M-1$ までの整数値をとるとする. 任意の時刻 t に対し, t まで連続して動作したパルス数がある定数以上であるような任意のプロセス P_i, P_jの局所時計の値が一致し, 以降 P_iが動作し続ける限り局所時計の値が 1 パルス毎に 1 ずつ増えることを保証するプロトコルを無待機時計合わせプロトコルという.

定義 1 ある正整数 kに対し, 任意の初期化実行が次の 2 つの条件を満たすようなプロトコルを同期時間 kの無待機時計合わせプロトコルという.

kに対する調整完了性 任意の $t \geq 1$, 任意のプロセッサ P_iに対し, $work(P_i, t) \geq k+1$ ならば, $P_i.Clock(t) = P_i.Clock(t-1)+1$ が成り立つ.

kに対する一致性 任意の $t \geq 1$, 任意のプロセス P_i, P_jに対し, $work(P_i, t) \geq k, work\ (P_j, t) \geq k$ならば, $P_i.Clock(t) = P_j.Clock(t)$ が成り立つ. ■

自己安定無待機時計合わせプロトコルは以下のように定義される.

定義 2 ある正整数 kに対し, 任意の実行が kに対する調整完了性, 一致性を満たすようなプロトコルを同期時間 kの自己安定無待機時計合わせプロトコルという.
■

また, 局所時計のとりうる値の数が有界であるようなシステムを考えるならば, 調整完了性の式 $P_i.Clock(t) = P_i.Clock(t-1) + 1$ は Mの剰余で成り立つと定義する.

2.3. 同期時間の下界

本節では, フェーズ内システムにおける無待機時計合わせプロトコルの同期時間の下界が$\Omega(n)$ であることを示す.

定理 1 同期時間が $n-2$ 以下の無待機時計合わせプロトコルは存在しない.

(証明) 同期時間 $k \leq n-2$ の時計合わせプロトコル Aが存在すると仮定し, 矛盾を導く. 次の 3 つの条件を満たす 2 つの実行 $E = c_0\pi_1c_1\cdots, E' = c'_0\pi'_1c'_1\cdots$を考える. 以下, $Clock_E$,$Clock_{E'}$と記すときは, それぞれ実行 E,E'における $Clock$の値を表す.

(条件 1) ある時刻t に対し, $c_0\pi_1c_1\cdots c_t = c'_0\pi'_1c'_1\cdots c'_t$が成り立ち, すべてのプロセッサ Pに対して $work(P,t) \geq k$ が成り立つ.

このとき, 実行 E, 実行 E'において, 一致性より, すべてのプロセッサ P_iに対し $P_i.Clock(t)$ が一致する. この値を $clock(t)$ とする.

(条件 2) 実行 Eに対し, $\pi_{t+1} = \pi_{t+2} = \cdots = \pi_{t+n-1} = \{P_0\}$ が成り立つ.

このとき, 調整完了性より, 実行 Eでは $P_0.Clock_E\ (t+n-2) = clock(t) + n - 2$ が成り立つ. ここで, P_0は区間 $[t+1, t+n-2]$ で $n-2$ ステップの動作をするので, その間に読み出しをしていない共有変数集合 $V_i(i \neq 0)$ が存在する.

(条件 3) 実行 E'に対し, $\pi'_{t+1} = \{P_i\}, \pi'_{t+2} = \cdots = \pi'_{t+n-1} = \{P_0, P_i\}$ が成り立つ.

このとき, 調整完了性より, 実行 E'において (1,1) $P_i.Clock_{E'}(t+n-1) = clock(t)$ $+n-1$ が成り立つ. また, 両実行 E, E'において P_0, P_i以外のプロセッサは区間 $[t+1, t+n-1]$ ではステップしないので, 実行 E'における区間 $[t+2, t+n-1]$ での P_0の動作は, 実行 Eにおける区間 $[t+1, t+n-2]$ での P_0の動作に一致する. よって, (1,2)$P_0.Clock_{E'}(t+n-1) = P_0.Clock_E(t+n-2) = clock(t)+n-2$ が成り立つ. しかし, 一致性より $P_i.Clock_{E'}(t+n-1) = P_0.Clock_{E'}(t+n-1)$ が成り立つはずであり, 式 (1, 1), (1,2) に矛盾する. ■

以上より, 無待機時計合わせプロトコルの同期時間の下界は$\Omega(n)$ である.

2.4. 無待機時計合わせアルゴリズム

本節で, フェーズ内システムにおける同期時間 $12n$ の無待機時計合わせアルゴリズム WCSを提案する. プロセス P_iのプログラムを図 2.1–2.3に示す. プログラム内では, 共有変数を $Clock$のように大文字で始まる変数名で表す. また, 各プロセスの状態を局所変数の集合で表し, 局所変数を curのように小文字で始まる変数名で表す. 特に変数 curは, 読み出すレジスタの所有者の識別子を表す変数である.

2.4.1 アルゴリズム WCS

概略

本アルゴリズムでは, 各プロセス P_iは 2 つのモード, 調整中モード ($adjusting$ mode), 調整済モード ($adjusted$ mode) を持つ. 初期状況では, 全プロセスが調整中モードで時計調整を開始する. 調整中モードにあるプロセス P_iは, 手続き $adjust$ に従って時計調整を行う. 調整中モードにおいて手続き $check_adjusted$により時計調整が終了したと判定したならば, 調整済モードへ移行する. 調整済モードにあるプロセスは, ステップ毎に局所時計の値を 1 ずつ増やす. また, プロセス P_iはすべてのステップにおいて P_i自身および他のプロセスの居眠りのチェックを行う (手続き $check\_nap$). 他のプロセス P_jの居眠りを検知したならば, 共有変数を通して

shared variables
$Clock$, 初期値 0
$Count$, 初期値 0
$Work\_count$, 初期値 0
Gen, 初期値 1 /* 世代番号 */
$Invalid_j (j = 0...n-1)$, 初期値 0
$Mode$, "$adjusting$" or "$adjusted$", 初期値 "$adjusting$"
$Sync$, $\subset \{P_0, ..., P_{n-1}\}$, 初期値$\emptyset$ /* 局所時計が一致するプロセッサの集合 */
$Async$, $\subset \{P_0, ..., P_{n-1}\}$, 初期値$\emptyset$ /* 局所時計を無視, 棄却したプロセッサの集合 */
local variables
cur, 初期値 0
$last\_count_j (j = 0...n-1)$, 初期値 0
$last\_gen_j (j = 0...n-1)$, 初期値 0
$last\_my\_count_j (j = 0...n-1)$, 初期値 0
$key_j (j = 0...n-1)$ /* 世代を始めた相対時刻 */
$list[0...n-1]$ /* 世代を始めた順番 */
next, sync, pos

```
1   repeat forever do on receipt of a pulse
2       read R_cur;
3       if (check_nap) then partial_reset; /* 居眠りリセット */
4       elseif Mode = "adjusting" then adjust;
5       else /* Mode = "adjusted" */
6           Clock := Clock + 1;
7           next := cur + 1 (mod n);
8       last_count_cur := P_cur.Count;
9       last_gen_cur := P_cur.Gen,;
10      last_my_count_cur := Count;
11      cur := next;
12      Count := Count + 1;
13      Work_count := Work_count + 1;
14  end
```

図 2.1 アルゴリズム WCS(P_iの変数, メインプログラム)

```
15  procedure check_nap;
16      diff := (Count − last_my_count_cur) − (P_cur.Count − last_count_cur);
17      if (diff > 0 and P_cur.Gen = last_gen_cur)
18          then Invalid_cur := P_cur.Gen;
19      return { [naps I]_i: Work_count ≥ n and diff < 0 } or
20              { [naps II]_i: Work_count ≥ n and P_cur.Invalid_i = Gen }
21  end procedure;

22  procedure partial_reset;
23      Gen := Gen + 1;
24      next := 0; Mode := "adjusting"; Work_count := −1; /* becomes 0 at line 13*/
25  end procedure;

26  procedure sort_list;
27      list[0..n − 1] := sort ids in the descending lexicographic order of (key_id, id);
28      Sync := ∅; Async := ∅; pos := 0; check_adjusted;
29  end procedure;

30  procedure check_adjusted;
31      if list[pos] = i then Mode := "adjusted"; next := 0; else next := list[pos];
32  end procedure;
```

図 2.2 アルゴリズム $WCS(P_i$の手続き１$)$

```
33  procedure adjust;
34     if 0 ≤ Work_count ≤ 4n − 1 then next := Work_count + 1(mod n);
35     elseif 4n ≤ Work_count ≤ 5n − 1 then
36        begin
37           if Invalid_cur ≠ P_cur.Gen
38              then key_cur := P_cur.Work_count − Work_count;
39              else key_cur := −1;
40           next := Work_count + 1(mod n);
41           if Work_count = 5n − 1 then sort_list;
42        end
43     else /* Work_count ≥ 5n */
44        begin
45           if Invalid_cur = P_cur.Gen or P_cur.Work_count < Work_count then
46              begin
47                 Async := Async ∪ {P_cur};    /* 無視 */
48                 pos := pos + 1; check_adjusted; Clock := Clock + 1;
49              end
50           elseif P_cur.Mode = "adjusted" then
51              begin
52                 if (Sync ⊈ P_cur.Async and P_cur.Sync ≠ ∅ and Clock ≠ P_cur.Clock)
53                    then partial_reset; /* 居眠りリセット */
54                 else
55                    begin
56                       Clock := P_cur.Clock + 1;
57                       if Sync ⊆ P_cur.Async
58                          then Async := Async ∪ Sync; Sync := ∅; /* 棄却 */
59                       Sync := Sync ∪ {P_cur}; pos := pos + 1; check_adjusted;
60                    end
61              end
62           if (Work_count = 6n − 1 and Mode = "adjusting")
63              then partial_reset; /* 時間超過リセット */
64        end
65  end procedure;
```

図 2.3 アルゴリズム $WCS(P_i$の手続き２)

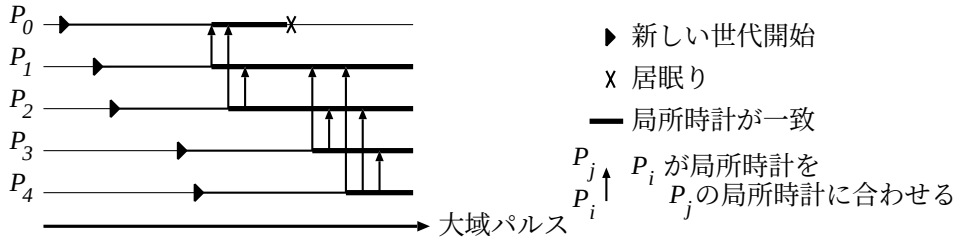


図 2.4 アルゴリズム WCSのアイデア

P_jに知らせる. 自分の居眠りを検知した場合, プロセス P_iは手続き $partial\_reset$ を行い, 時計調整を始めからやり直す.

本アルゴリズムにおける時計調整のアイデアを図 2.4に示す. プロセス P_iが調整中モードで時計調整を開始してから次に故障を検知して $partial_reset$を行うまでの区間を**世代**とよぶ. 各プロセスは, 現在の世代でのステップ数を表す共有変数 $Work\_count$ を使う. 時計調整を開始したプロセスはまず, 現在の世代を自分より早く始めたプロセスを $Work_count$により調べる. 図 2.4に示すように, 現在の世代を後から始めたプロセスは, 自分より早く始めたプロセスの局所時計だけを基にして時計調整を行う.

以下, 手続き $check_nap$による居眠りのチェックについて説明し, 次に手続き $adjust$による時計の調整の詳細について説明する.

居眠りのチェック (手続き $check\_nap$)

各プロセスは, 初期状態からのステップ数を表す共有変数 $Count$ (初期値 0) を使う. また, 新しい世代の時計調整を開始するたびに更新する共有変数 Gen(初期値 1) を持つ. $Gen = g$の間の時計調整を, 世代 gの時計調整とよぶ.

プロセス P_iが R_jを読み出すステップにおいて, 前回の R_jを読み出したステップからの $P_i.Count$, $P_j.Count$の増分の差 (図 2.2, 第 16 行,$diff$で表す) により, 図 2.5のように P_iは自身または P_jが居眠りしていたことを判定する. P_i が P_jの居眠りを検知したならば (第 17 行の条件式が成立), $P_j.Gen$の値を P_iの変数 $Invalid_j$ に代入することにより, P_iは P_jの居眠りを P_jに知らせる. P_iは自身の居眠りを検

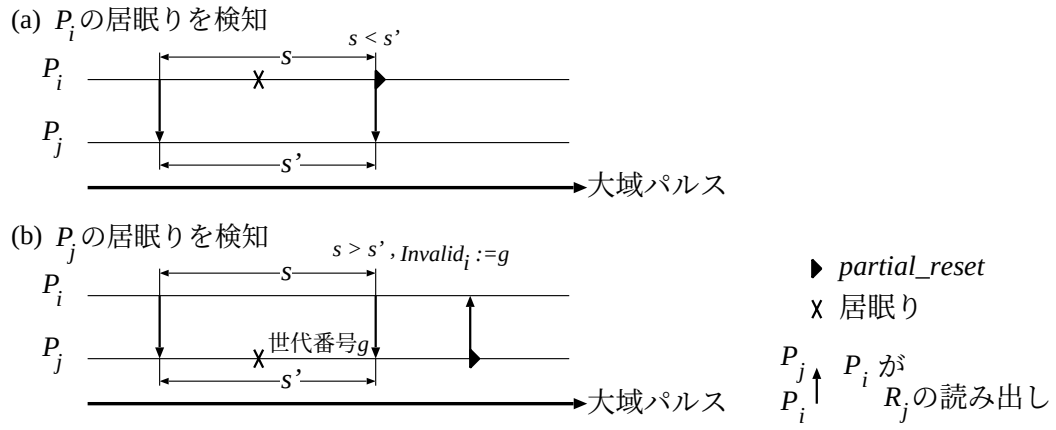


図 2.5 プロセス P_iの居眠り検知

知したときに $P_i.Work\_count \geq n$ ならば (第 19 行 $[\text{naps I}]_i$ または第 20 行 $[\text{naps II}]_i$ が成立), $partial_reset$を行う. プロセス P_iが居眠りをしたならば, 複数のプロセスとの $Count$の増分差の比較において P_iが P_i自身の居眠りを検知しうる. このような居眠り検知の重複を避けるため, $P_i.Work\_count \geq n$ を条件としている.

ここで述べた以外に調整中モード中に $partial_reset$を行う場合がある. それらについては, 手続き $adjust$の中で説明する.

時計の調整 (手続き $adjust$)

調整中モードにおける時計調整は, 3 つのピリオドから構成される. 図 2.6に示すように時刻 $t-1$ における $Work_count$の値に従って, 時刻 t にそれぞれ以下のピリオドのステップを行う.

- 準備ピリオド $(0 \leq Work\_count(t-1) \leq 4n-1)$
- 調査ピリオド $(4n \leq Work\_count(t-1) \leq 5n-1)$
- 調整ピリオド $(5n \leq Work\_count(t-1))$

プロセスは, 準備ピリオド, 調査ピリオドでは共有変数の読み出しがその所有プロセスの識別子が巡回する順序の繰り返しになるようにステップを続け, 調査ピリ

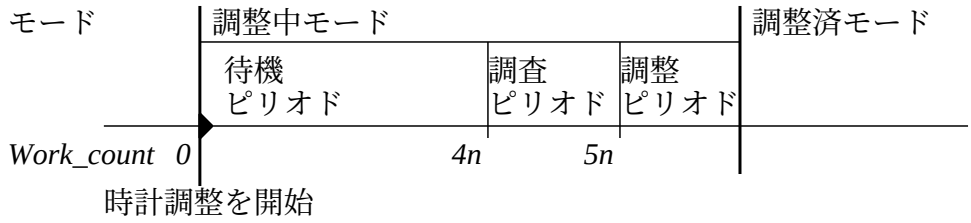


図 2.6 プロセス P_iのモード, フェーズ

オドではアルゴリズムに従って読み出しの順序を変更する. 調査ピリオドでは, 他のプロセスがそのときの世代を始めた時刻を調べる. 調整ピリオドでは, 自分より早く世代を始めた他のプロセスの局所時計と自分の局所時計を一致させる. また, $4n$ ステップの準備ピリオドは, 調査ピリオドで他のプロセスが時計調整を始めた時刻を正しく調べられることを保証するために必要とする (補題 2). 準備ピリオドでは, 居眠りのチェックのみを行う.

調査ピリオドで, プロセス P_iは $P_j.Work\_count$ により, P_iが世代を始めた時刻に対する, P_jがそのときの世代を始めた相対時刻を調べる. ただし, P_iが P_jの居眠りを検知したならば, P_jは遅くとも次に P_i の共有変数を読み出すステップで $partial_reset$を行うので, P_iは P_jを時計調整を遅く開始したプロセスとして扱う. 調整ピリオドでは, その所有者が時計調整を始めた時刻の順に共有変数を読み出す. その順序を知るため, 調査ピリオドの最後のステップで手続き $sort_list$により P_iより早く時計調整を始めたプロセスを時計調整を開始した時刻の順にソートし, その結果を配列 $list$に格納する. ただし, 同時刻に時計調整を開始したプロセスが複数存在するならば, 識別子により順序を付ける. P_iより早く時計調整を始めたプロセスがなければ, 手続き $check_adjust$により, このステップで P_iは調整ピリオドに入ることなく調整済モードへ移行する.

調整ピリオドでは, P_iはまず最も早く時計調整を始めたプロセス, すなわち $list$ の先頭にあるプロセスの局所時計に自分の時計を合わせる. その後, 原則として $list$の順に, P_iより早く時計調整を始めた他のプロセスの局所時計と自分の局所時計が一致していることを確認していく. 調整ピリオドのステップは以下のように行う. 以下では, そのステップで読み出す共有変数を R_jとする.

- P_iが P_jの居眠りを検知したとき, または P_jが自身の居眠りを検知したときは, P_jの局所時計に合わせない, または 一致確認をしない. 次のステップでは $list$の次のプロセスの共有変数を読み出す. このとき, P_iは P_jの時計を**無視**するとよぶ (第 47 行).

- P_jがまだ調整中モードのときは, 今回のステップでは P_jの局所時計に合わせない, または 一致確認をしない. 次のステップでも再び R_jの読み出しをする.

- P_iがこれまで局所時計を合わせた, または一致確認したプロセスが信頼できないと判断したら, 現在の局所時計を**棄却**し (第 58 行), P_jの局所時計に合わせる. このために, 局所時計を合わせた, または一致確認したプロセス集合を表す $Sync$, 無視または棄却をしたプロセスの集合 $Async$ を使う.

- P_iと P_jが, 現在の世代で, 共通のプロセス P_mの局所時計に合わせた, または一致確認しているにもかかわらず, 両者の局所時計が一致しないならば, P_iは $partial_reset$を行い時計調整をやり直す.

プロセス P_iは, P_iより早く時計調整を始めたすべてのプロセス, すなわち配列 $list$において P_iより前にあるすべてのプロセスの局所時計に対し, 合わせる, 一致確認をする, または無視したとき, 手続き $check_adjust$により P_iは調整中モードから調整済モードに移行する. アルゴリズム WCSは, 現在の世代の時計調整を始めてから P_iが居眠りすることなく動作し続けている場合は, $P_i.Work\_count = 6n$ に達する前に調整中モードから調整済モードへ移行することを保証する (補題 5 で証明). そこで, P_iが居眠りしたことにより調整済モードへ移行することなく $P_i.Work\_count = 6n$ に達するならば, $partial_reset$を行い時計調整をやり直す. この場合の $partial_reset$を**時間超過リセット**, 他の場合の $partial_reset$を**居眠りリセット**と呼んで区別する.

2.4.2 アルゴリズム WCSの正当性

アルゴリズム WCSが同期時間 $12n$ の無待機時計合わせアルゴリズムであることを示す.

同期時間を, プロセスが居眠りをしてから手続き $partial\_reset$ を行うまでのステップ数と, 時計調整を始めてから (初期状況または $partial_reset$を行ってから) 調整済モードへ移行するまでのステップ数に分けて考える. アルゴリズム WCS では, 調整ピリオドでの n 回のステップ後も調整中モードから調整済モードへ移行しないならば, $P_i.Work\_count(t'-1) = 6n-1$ なる時刻 t'で時間超過リセットを行う. 補題 5では, 調整中モードのプロセスが時計調整を始めてから居眠りをしていない, かつ居眠りリセットを行わないならば, $P_i.Work\_count(t'-1) = 6n-1$ なる時刻 t'までに調整済モードへ移行することを示す. 補題 8では, 調整済モードの 2 つのプロセスが十分長く居眠りすることなく動作し続けているならば, 両者の局所時計の値が一致することを示す. 補題 9では, プロセスが居眠りをしてから $partial_reset$を行うまでのステップ数が高々$6n$ であることを示す. 補題 10,11で, 任意の実行に対し調整完了性と一致性が成り立つことを示す.

第 27 行のソートの順序関係を表すため, 任意の 2 つの 2 項組 $(k_i, i), (k_j, j)$ に対し, $k_i < k_j$ または $k_i = k_j \wedge i < j$ならば, $(k_i, i) < (k_j, j)$ と定義する. また, 時計調整を始めた時刻の前後関係を表すため, $t_i < t_j$ または $t_i = t_j \wedge i > j$ ならば, $(t_i, i) \prec (t_j, j)$ と定義する. また, $(t_i, i) \prec (t_j, j)$ または $(t_i, i) = (t_j, j)$ ならば, $(t_i, i) \preceq (t_j, j)$ とする. プロセス P_iの時刻 t_iの手続き $sort_list$において, $list[p] = j$,$list[p'] = j'$なるプロセス P_j,$P_{j'}$に対し $p < p'$が成り立つならば, $P_j \longrightarrow^{i}_{t_i} P_{j'}$と記す.

手続き $sort_list$のソートの結果に関して以下の補題が成り立つ.

補題 2 任意の時刻を t とする. 区間 $[t+2n+1, t+5n]$ において, *3* つのプロセス $P_{j_1}, P_{j_2}, P_{j_3}$は正常に動作し続け, かつ $partial_reset$を行わないとする. また, $(t_1, j_1) \preceq (t_2, j_2) \prec (t_3, j_3)$, $t_1 \geq t+4n+1, t_3 \leq t+5n$ である時刻 t_1, t_2, t_3で, それぞれ $P_{j_1}, P_{j_2}, P_{j_3}$が手続き $sort_list$を行ったとする. このとき, $P_{j_2} \longrightarrow^{j_3}_{t_3} P_{j_3}$ ならば, $P_{j'} \longrightarrow^{j_1}_{t_1} P_{j_1}$ なるすべての j'に対し, $P_{j'} \longrightarrow^{j_3}_{t_3} P_{j_2}$ または $P_{j_3} \longrightarrow^{j_3}_{t_3} P_{j'}$ が成り立つ.

(証明) P_{j_1}は, 時刻 $t_1 - n + j'$のステップで $R_{j'}$を読み出し, P_{j_1}が世代を始めた時刻に対する $P_{j'}$が世代を始めた相対時刻 $P_{j_1}.key_{j'}$の値を決定し, 時刻 t_1で $sort_list$を行う. ここで, $P_{j_1}.key_{j'} = x$ とすると, $P_{j'} \longrightarrow^{j_1}_{t_1} P_{j_1}$ より $(x, j') > (0, j_1)$ である. 一方 P_{j_3}は, 時刻 $t_3 - n + j'$のステップで $R_{j'}$を読み出して $P_{j_3}.key_{j'}$の値を決

定し, 時刻 t_3で $sort_list$を行う. このとき, $t+4n+1 \leq t_1 \leq t_3 \leq t+5n$ より $t_3-2n+j' < t_1-n+j' \leq t_3-n+j'$が成り立つ. P_{j_3}は区間$[t_3-2n+j', t_3-n+j']$では正常に動作し続けるので, $P_{j'}$がこの区間で居眠りをしているならば, 時刻t_3-n+j'のステップで P_{j_3}はそれを検知する. 時刻 t_3-n+j'に P_{j_3}が $P_{j'}$の居眠りを検知しないなら以下の (a) が成り立ち, $P_{j'}$の居眠りを検知するなら以下の (b) が成り立つ.

(a) $(P_{j_3}.key_{j'}, j') = (x+t_3-t_1, j') > (t_3-t_1, j_1) \geq (t_3-t_2, j_2) = (P_{j_3}.key_{j_2}, j_2)$
($\because$ $P_{j_2} \longrightarrow_{t_3}^{j_3} P_{j_3}$ より)

(b) $(P_{j_3}.key_{j'}, j') = (-1, j') < (0, j_3) = (P_{j_3}.\ key_{j_3}, j_3)$

(a),(b) はそれぞれ $P_{j'} \longrightarrow_{t_3}^{j_3} P_{j_2}$, $P_{j_3} \longrightarrow_{t_3}^{j_3} P_{j'}$ を意味する. ■

$steps(P_i, t', t)$ を以下のように定義する. $t' < t$ のとき, 区間 $[t', t-1]$ でのステップ数とする. また, $t' = t$ のときは $steps(P_i, t', t) = 0$ とし, $t' > t$ のときは $-steps(P_i, t, t')$ と定義する. この定義より, 任意のt_1, t_2, t_3に対して $steps(P_i, t_1, t_2)$ $+steps(P_i, t_2, t_3) = steps(P_i, t_1, t_3)$が成り立つ. アルゴリズムより, $steps\ (P_i, t', t)$ に対し, 以下の事実が成り立つ.

事実 3 任意のプロセス P_i, P_jと時刻 t に対し, $steps\ (P_i, t', t)$ が以下を満たすような時刻 t'が存在するならば, P_iは区間 $[t', t-1]$ のある時刻で R_j を読み出すステップを行っている.

(a) $n \leq P_i.Work\_count(t-1) \leq 5n$ または $7n \leq P_i.Work\_count(t-1)$ ならば, $steps(P_i, t', t) = n$

(b) $5n < P_i.Work\_count(t-1) < 7n$ ならば, $steps(P_i, t', t) = 2n$

(c) $P_i.Work\_count(t-1) < n$ ならば, $steps(P_i, t',\ t) = 3n-1$ ■

P_iが世代 gの調整ピリオドでのステップ数, すなわち, 時刻 t で $sort_list$を行った後, $partial_reset$を実行するか調整済モードへ移行する時刻 t'までのステップ数 $steps(P_i, t+1, t'+1)$ を$\alpha(P_i, g)$ と記す.

補題 4,5では, 以下に示す条件 A を満たすプロセス P_iを考える.

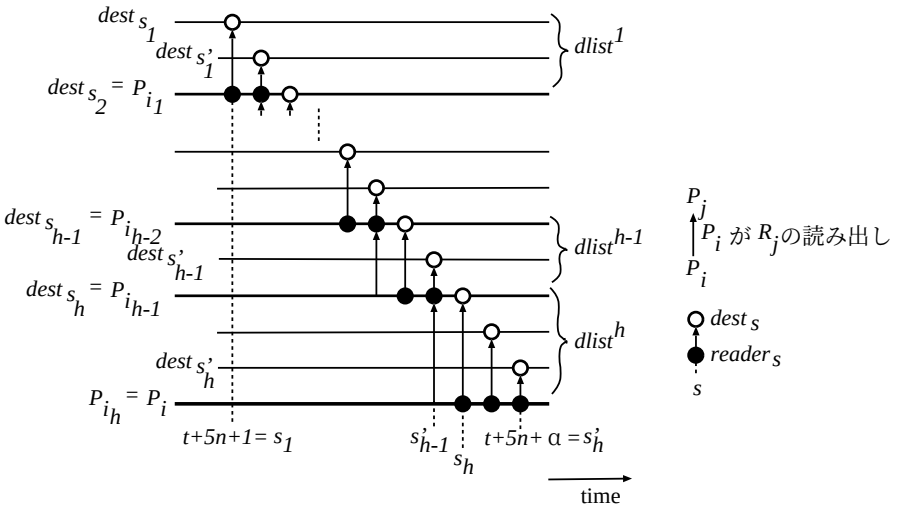


図 2.7 プロセス, 時刻の定義

条件 A： P_iが時刻tで $partial_reset$を実行し, $work\,(P_i, t+6n) \geq 6n$かつ $[t, t+6n]$ で居眠りリセットを行わない.

条件 A を満たす P_iに対し, $\bar{\alpha} = \alpha(P_i, P_i.Gen(t))$ とする. このとき, 世代 $P_i.Gen(t)$ の調整ピリオドで P_iが時間超過リセットを行うならば, 時刻 $t+6n$ で $partial\_reset$ を行うので$\bar{\alpha} = n$ が成り立つ. よって, $\bar{\alpha} < n$ が成り立つならば, P_iが時刻 t までに調整済モードへ移行する.

条件 A を満たすプロセス P_iが調整ピリオドにいる区間 $[t+5n+1, t+5n+\bar{\alpha}]$ に属する各時刻 s に対し, 2 種類のプロセス $reader_s, dest_s$を以下のように時刻 $t+5n+\bar{\alpha}$ から再帰的に定義する (図 2.7). まず, $reader_{t+5n+\bar{\alpha}} = P_i$とする. 時刻 s で $reader_s$ が読み出すレジスタの所有者を $dest_s$と定義する. $reader_s, dest_s (t+5n+2 \leq s \leq t+5n+\bar{\alpha})$ に対し, $reader_s$が時刻 s と $s-1$ で同じレジスタの読み出しをするならば $reader_{s-1} = reader_s$と定義し, 異なるレジスタの読み出しをするならば $reader_{s-1} = dest_s$ と定義する. ここで, $dest_s$ $(t+5n+1 \leq s < t+5n+\bar{\alpha})$ のリスト $dlist = dest_{t+5n+1}, dest_{t+5n+2}, \cdots, dest_{t+5n+\bar{\alpha}}$ を, $reader_s$が同一でありかつ連続するように分割した極大部分リスト $dlist^1, dlist^2, \cdots, dlist^h$を考える. 各 $x (1 \leq x \leq h)$ に対し, $dlist^x$で共通の $reader_s$を P_{i_x}, $dlist^x = dest_{s_x}, dest_{s_x+1}, \cdots, dest_{s'_x}$

とし, P_{i_x} が世代 $P_{i_x}.Gen(s'_x - 1)$ において $sort_list$を行うならば, その時刻を $sort_time^x$とする. ここで, 以下の補題が成り立つ.

補題 4 すべての $x(1 \leq x \leq h)$ に対し, 以下が成り立つ.

(a) $dlist^x$の要素はすべて異なる.

(b) P_{i_x}は区間 $[t + 2n + 1, s'_x - 1]$ で居眠りをしない.

(c) $x < h$ ならば, $(t + 4n + 1, 0) \preceq (sort\_time^x, i_x) \prec (sort\_time^{x+1}, i_{x+1})$

(証明) すべての $x(1 \leq x \leq h)$ に対して条件が成り立つことを $x = h$ の場合から帰納的に示す.

まず, $x = h$ の場合に条件 (a), (b) が成り立つことを示す. 定義より, $P_{i_x} = P_i$である. また, $work(P_i, t + 6n) \geq 6n$ より区間 $[t + 2n + 1, s'_h - 1]$ で居眠りしていない. P_iは $[s_h, s'_h]$ で調整ピリオドにあるので, この区間では $list$ の順にレジスタを読み出している. 従って, $dlist^h$は $dest_{s_h}$と $dest_{s_h} \longrightarrow^{i_h}_{sort\_time^h} P_m \longrightarrow^{i^h}_{sort\_time^h} P_i$ なるすべての P_mから成り, $dlist^h$の要素はすべて異なる.

次に, $x + 1, \cdots, h$ に対して条件 (a), (b), (c) が成り立つならば, x に対して条件 (a), (b), (c) が成り立つことを示す. まず, (b) が成り立つことを示す. 定義より, $P_{i_x} = dest_{s_{x+1}}$である. $P_{i_{x+1}}$は時刻 s'_x, s_{x+1}で連続して R_{i_x}を読み出す. ここで, $P_{i_{x+1}}$は時刻 s'_xで調整ピリオドにいるので $P_{i_{x+1}}.Work\_count(s'_x - 1) \geq 5n$ が成り立ち, $(sort\_time^{x+1}, i_{x+1}) \preceq (sort\_time^h, i_h)$ が成り立つので, $2n + 1 \leq P_{i_{x+1}}.Work\_count(t + 3n) \leq 3n$ が成り立つ. よって, 事実 3より区間 $[t + 2n + 1, t + 3n]$ に $P_{i_{x+1}}$が R_{i_x}を読み出すステップが存在する. $[t + 3n, s'_x - 1]$ で P_{i_x}が居眠りをしているならば, 遅くとも時刻 s'_xまでに $P_{i_{x+1}}$が P_{i_x}の居眠りを検知するか, 時刻 $s'_x - 1$ までに P_{i_x}が自身の居眠りに気付いて次の世代を始めている. また, $P_i.Work_count(t + 2n) = 2n$が成り立つので, 事実 3より区間 $[t + n + 1, t + 2n]$ に P_iが R_{i_x}を読み出すステップが存在する. よって, $[t + 2n + 1, t + 3n - 1]$ で P_{i_x}が居眠りをしているならば, P_iに必ず検知される. 時刻 $s'_x - 1$ で P_{i_x}は調整ピリオドにあるので, $P_{i_x}.Work\_count(t + 5n) \geq n$ が成り立ち, 事実 3より $[t + 3n + 1, t + 5n]$ で R_iを通して P_{i_x}が自身の居眠りを検知する. よって, 時刻 s_{x+1}では R_{i_x}以外のレ

ジスタを読み出すので, 矛盾する. よって, P_{i_x}は区間 $[t+2n+1, s'_x-1]$ で居眠りをしていない. すなわち, (b) が成り立つ.

また, 時刻 s'_x-1 で P_{i_x}は調整中モードにいることより $P_{i_x}.Work\_count(s'^x-1) < 6n$ が成り立つので $sort\_time^x \geq t+4n+1$ が成り立つ. 区間 $[t+4n+1, sort\_time^{x+1}]$ で P_{i_x}, $P_{i_{x+1}}$はともに居眠りしていないので, $P_{i_x} \longrightarrow^{i_{x+1}}_{sort_time^{x+1}} P_{i_{x+1}}$より, $(sort\_time^x, i_x) \prec (sort\_time^{x+1}, i_{x+1})$ が成り立つ. すなわち, (c) が成り立つ.

最後に (a) が成り立つことを示す. $(sort\_time^x, i_x) \preceq (sort\_time^h, i_h)$ が成り立つので, 区間 $[s_x, s'_x]$ で P_{i_x}は調整ピリオドにある. よって, $dlist^x$は $dest_{s_x}$と $dest_{s_x} \longrightarrow^{i_x}_{sort\_time^x} P_m \longrightarrow^{i_x}_{sort\_time^x} dest_{s'_x}$ なるプロセス P_mから成り, $dlist^x$の要素はすべて異なる. 従って, (a) が成り立つ. ■

補題 5 条件 A を満たすプロセス P_iは時刻 $t+6n$ までに調整済モードへ移行する.

(証明) すべての s $(t+5n+1 \leq s \leq t+5n+\bar{\alpha})$ の間で $dest_s$が異なり, かつ $dest_s$として P_iが現れないことを示すことにより, $\bar{\alpha} < n$ が成り立つことを示す.

$h=1$ の場合, すなわち $dlist = dlist^1$の場合, 補題 4(a) より $dlist$ の要素はすべて異なり, また $dlist$ に P_iが表れない. 従って, $\bar{\alpha} < n$ が成り立つ.

次に $h>1$ の場合を考える. 補題 4(a) より, 任意の $x(1 \leq x \leq h)$ に対し $dlist^x$ の要素がすべて異なる. 以下では, 任意の異なる $x, y(1 \leq x \leq h, 1 \leq y \leq h)$ に対し, P_iが $dlist^x$に現れず, $dlist^x$と $dlist^y$に共通要素が現れないことを示す. 補題 4(b) より, 任意の $x, y(1 \leq x < y \leq h)$ に対し, P_{i_x}は区間 $[t+2n+1, t+5n]$ を含む区間 $[t+2n+1, s'_x-1]$ で正常に動作し続けている. また, 補題 4(c) より, $(t+4n+1, 0) \preceq (sort\_time^x, i_x) \prec (sort\_time^y, i_y)$ が成り立つ. 従って, $P_{j_1} = P_{i_x}$, $P_{j_2} = P_{i_{y-1}}$, $P_{j_3} = P_{i_y}$, $t_1 = sort\_time^x$, $t_2 = sort\_time^{y-1}$, $t_3 = sort_time^y$と定めたときに補題 2が成り立つ. すなわち, $P_{j'} \longrightarrow^{j_1}_{sort\_time^x} P_{j_1}$ なるプロセス $P_{j'}$に対して, $P_{j'} \longrightarrow^{j_3}_{sort\_time^y} P_{j_2}$ または $P_{j_3} \longrightarrow^{j_3}_{sort\_time^y} P_{j'}$ が成り立つ. 従って, $dlist^x$ と $dlist^y$ に共通要素が現れることはない. また, $P_i = P_{i_h}$なので, 補題 2より P_iは $dlist^x$に現れない. ■

アルゴリズム WCSでは, プロセスはすべての居眠りを検知するわけではない. よって, 2つのプロセス P_i,P_jが居眠りするタイミングにより, P_iは「P_jは P_iより

後で時計調整を始めた」と判定するときに, P_jが「P_iは P_jより後で時計調整を始めた」と矛盾した判定をする場合がある. この場合, P_i, P_jは互いに局所時計を参照することなく調整済モードへ移行すると考えられる. しかし, 手続き $sort\_list$ のソートの結果に関して以下の補題が成り立ち, P_i, P_jが十分長く動作しているならば両者が矛盾した判定をすることはない.

補題 6 任意の時刻を t, 任意の異なるプロセスを P_i, P_jとする. プロセス P_i, P_jはそれぞれ時刻 t_i, t_jで $sort_list$を行い, かつ, それぞれ区間 $[t_i, t], [t_j, t]$ で $partial\_reset$ を行わなかったとする. このとき, $work(P_i, t) > 4n$ かつ $work(P_j, t) > 4n$ ならば, $P_j \longrightarrow^i_{t_i} P_i$ または $P_i \longrightarrow^j_{t_j} P_j$ が成り立つ.

(証明) 時刻 t_i, t_jに関して場合分けする.

(1) $\max(t_i, t_j) \geq t - 2n$ のとき

一般性を失うことなく $(t_i, i) \prec (t_j, j)$ と仮定する. このとき, $t-3n \leq t_j-n+1 < t_j \leq t$が成り立つ. P_i, P_jはともに区間 $[t_j-n+1, t_j]$ で正常に動作し続け, この区間のある時刻 t'で P_jは key_iを設定するステップを行う. このとき, 世代 $P_j.Gen(t)$ で時刻 t 以前にプロセス P_jが P_iの居眠りを検知していないならば (第 37 行の条件式が成立), $(t_i, i) \prec (t_j, j)$ より $((P_i.Work\_count(t'-1) - P_j.Work\_count(t'-1)), i) > (0, j)$ なので, $P_i \longrightarrow^j_{t_j} P_j$が成り立つ. 以下では, 世代 $P_j.Gen(t)$ で時刻 t 以前にプロセス P_jが P_iの居眠りを検知していないことを, 背理法により示す.

$work(P_i, t) > 4n$ より, P_iは区間 $[t - 4n + 1, t]$ で居眠りをしていない. また, $2n \leq P_j.Work\_count(t - 3n) \leq 4n$ かつ $steps(P_j, t - 4n + 1, t - 3n + 1) = n$ なので, 事実 3より区間 $[t - 4n + 1, t - 3n]$ のある時刻 t''で P_jは R_i を読み出す. よって, P_jが世代 $P_j.Gen(t)$ を始めてから P_jが t''より後で始めて P_iの居眠りに検知することはない. すなわち, 時刻 t 以前に P_iの居眠りを検知するならば, 遅くとも区間 $[t - 4n + 1, t - 3n]$ で必ず検知する. また, $P_i.Work\_count(t) \geq 5n$ かつ $steps(P_i, t - 2n + 1, t + 1) = 2n$ なので, 事実 3より区間 $[t - 2n + 1, t]$ に P_iが R_j を読み出すステップが必ず存在する. 従って, 区間 $[t'', t]$ で P_iが自身の居眠りを検知する. よって, 区間 $[t'', t]$ のある時刻で P_iは $partial_reset$を行う. 区間 $[t'', t]$ の長さは $4n - 1$ 以下なので, $P_i.Work\_count(t) < 4n$ となり矛盾する.

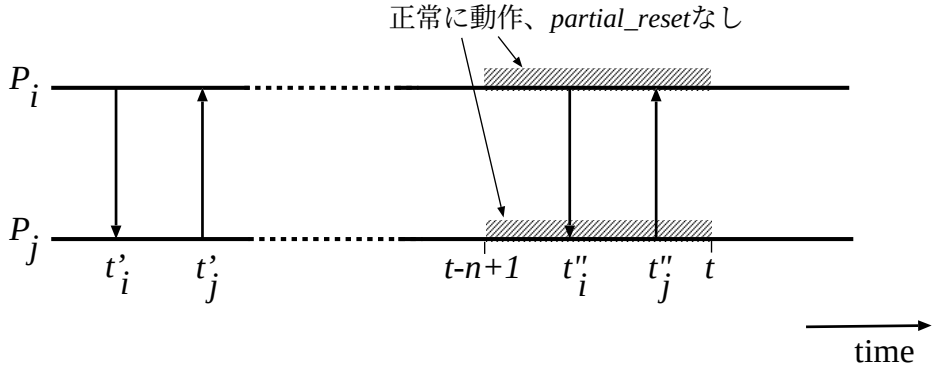


図 2.8 補題６：(2) $\max(t_i, t_j) < t - 2n$ のとき：時刻 t'_i, t'_j, t''_i, t''_j

(2) $\max(t_i, t_j) < t - 2n$ のとき

一般性を失うことなく $i > j$と仮定する. 以下では, $P_j \longrightarrow^i_{t_i} P_i$, $P_i \longrightarrow^j_{t_j} P_j$ がともに成立しないと仮定して矛盾を導く. プロセス P_iが t_i以前に最後に R_j を読み出すステップを行った時刻を t'_iとし, P_jが t_j以前に最後に R_iを読み出すステップを行った時刻を t'_jとする. $P_i \longrightarrow^i_{t_i} P_j$かつ $P_j \longrightarrow^j_{t_j} P_i$ が成り立つので, $P_i.Work\_count(t'_i - 1) \geq P_j.Work\_count(t'_i - 1)$ かつ $P_j.Work\_count(t'_j - 1) > P_i.Work\_count(t'_j - 1)$ であり,

$$steps(P_i, t'_i, t'_j) < steps(P_j, t'_i, t'_j) \tag{2.1}$$

が成り立つ.

ここで, $steps(P_i, t_i, t+1) > 2n$ と $P_i.Work\_count\ (t_i) = 5n$ が成り立つことより, $P_i.Work\_count(t) > 7n$ である. よって, 事実3より区間 $[t-n+1, t]$ のある時刻 t''_iで P_iは R_j を読み出す. 同様にして, 区間 $[t-n+1, t]$ のある時刻 t''_jで P_jは R_iを読み出す. P_i, P_jはそれぞれ区間 $[t_i, t], [t_j, t]$ で $partial_reset$を行わないので,

$$\begin{aligned} steps(P_i, t'_i, t''_i) &\geq steps(P_j, t'_i, t''_i) \\ steps(P_j, t'_j, t''_j) &\geq steps(P_i, t'_j, t''_j) \end{aligned}$$

が成り立つ. ここで, $work(P_i, t) \geq 4n, work(P_j, t) \geq 4n$ なので $steps(P_i, t''_i, t''_j) =$

$steps(P_j, t''_i, t''_j)$ である. よって,

$$\begin{aligned} steps(P_i, t'_i, t'_j) &= steps(P_i, t'_i, t''_i) + steps(P_i, t''_i, t''_j) - steps(P_i, t'_j, t''_j) \\ &\geq steps(P_j, t'_i, t''_i) + steps(P_j, t''_i, t''_j) - steps(P_j, t'_j, t''_j) \\ &= steps(P_j, t'_i, t'_j) \end{aligned}$$

となり, 式 (2.1) に矛盾する. ■

手続き *adjust*の説明で述べたように, プロセス P_iが自分の局所時計をある時刻 $t_{i,j}$でプロセス P_jの局所時計に合わせた, または P_jの局所時計と一致することを確認したあと, P_iが P_jの局所時計を棄却することがある. しかし, $t_{i,j}$以降に P_i, P_j がともに *partial_reset*をすることなく動作し続けるような場合には以下の補題が成り立つ.

補題 7 任意の異なるプロセス P_i, P_jに対し, ある時刻 t で P_i, P_jがともに調整済モードであったとし, $work(P_i, t) > 4n, work(P_j, t) > 4n$ が成り立つとする. また, 時刻 $t_{i,j}$で P_iは自分の局所時計を P_jの局所時計に合わせた, または P_jの局所時計と一致することを確認したとし, 区間 $[t_{i,j}, t]$ で P_i, P_jはともに *partial_reset*を行わないとする. このとき, 区間 $[t_{i,j}, t]$ で P_iは P_jの局所時計を棄却しない.

(証明) 区間 $[t_{i,j}, t]$ で P_iが P_jの時計を棄却すると仮定し, 矛盾を導く. すなわち, 区間 $[t_{i,j}+1, t]$ のある時刻 $t_{i,m}$で, 調整ピリオドの P_iが R_mを読み出したステップで, $P_i.Clock(t_{i,m}-1) \neq P_m.Clock(t_{i,m}-1)$, $\{P_j\} \in P_i.Sync(t_{i,m}-1) \subset P_m.Async(t_{i,m}-1)$ が成り立ち, P_iは P_mと局所時計を合わせた, または局所時計の一致確認をしたと仮定する. P_mが P_jを $P_m.Async$ に挿入した時刻を $t_{m,j}$ とし, $t_{m,j}$より前で最後に P_mが R_jを読み出した時刻を $t'_{m,j}$とする. このとき, 事実3より $steps(P_j, t'_{m,j}, t_{m,j}) < steps(P_m, t'_{m,j}, t_{m,j}) \leq 2n$ が成り立つ. ここで, $t_{m,j} \geq t-2n$ ならば, $steps(P_j, t'_{m,j}, t) = steps(P_j, t'_{m,j}, t_{m,j}) + steps(P_j,\ t_{m,j}, t) < 4n$ かつ P_j は時刻 $t'_{m,j}$以降に居眠りをしている. これは $work(P_j, t) > 4n$ に反する. また, $t_{m,j} < t-2n$ ならば, 事実3より区間 $[t_{m,j}, t-1]$ に P_jが R_mを読み出すステップが存在する. よって, $t'_{m,j}$以降に P_jは必ず *partial_reset*を行うので, 仮定に反する. ■

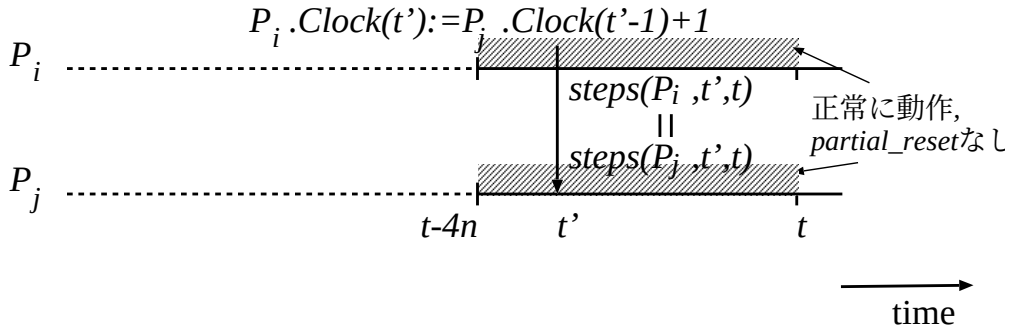


図 2.9 補題８：$t' \geq t - 4n + 1$ の場合

補題 8 任意の時刻を t, 任意の異なるプロセスを P_i, P_jとする. 時刻 t で P_i, P_jがともに調整済モードであったとする. このとき, $work(P_i, t) > 4n, work(P_j, t) > 4n$ が成り立つならば, $P_i.Clock(t) = P_j.Clock(t)$ が成り立つ.

(証明) P_i, P_jは, それぞれ世代 $P_i.Gen(t)$, $P_j.Gen(t)$ において, $sort_list$を行っている. P_i, P_jが $sort_list$を行った時刻をそれぞれ t_i, t_jとすると補題 6より $P_j \longrightarrow^i_{t_i} P_i$ または $P_i \longrightarrow^j_{t_j} P_j$が成り立つ. 一般性を失うことなく, $P_j \longrightarrow^i_{t_i} P_i$ と仮定する. このとき, P_iは世代 $P_i.Gen(t)$ のある時刻 t'で自分の局所時計を P_jの局所時計の値と合わせるか, 一致していることを確認し, 区間 $[t', t]$ で P_jは $partial_reset$を行わない. よって, $P_i.Clock(t') = P_j.Clock(t'-1)+1$ が成り立つ. ここで, 補題 7より, 区間 $[t', t]$ で P_iが P_jの局所時計を棄却することはない. また, 区間 $[t', t]$ で P_i, P_jはともに $partial_reset$をすることはないので, それぞれステップ毎に局所時計の値を１ずつ増やす. 従って,

$$steps(P_i, t', t) = steps(P_j, t', t) \tag{2.2}$$

であることを示せばよい. ここで, $t' \geq t-4n+1$ ならば, 区間 $[t', t]$ では P_i, P_jはともに正常に動作し続けるので明らかに式 (2.2) は成り立つ. 以下では, $t' < t-4n+1$ の場合を考える.

事実 3より, 区間 $[t - 4n + 1, t - n]$ のある時刻 t''で P_iは R_jを読み出す. また, 区間 $[t', t]$ で P_jは調整済モードであり続けるので, $t''(\leq t - n)$ より後で P_jは R_iを

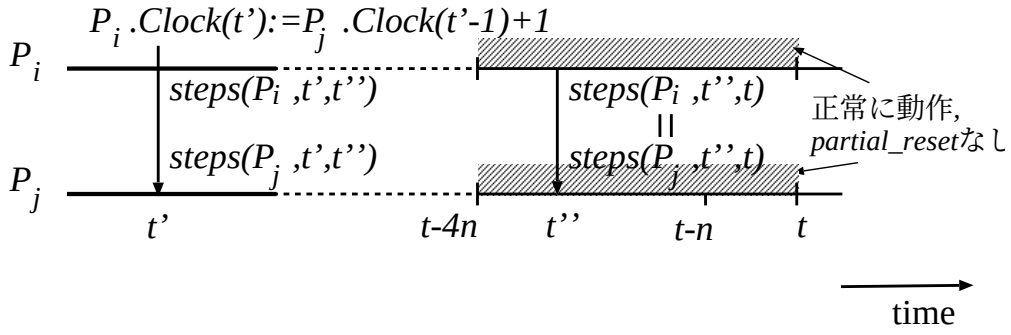


図 2.10 補題８：$t' < t - 4n + 1$ の場合

読み出しをするステップを行っている. 区間 $[t', t]$ で P_i, P_jはともに $partial\_reset$ を行わないので, 区間 $[t', t'']$ の各ステップで P_iが P_iまたは P_jの居眠りを検知することはない. 従って, $steps(P_i, t', t'') = steps(P_j, t', t'')$ である. さらに, 区間 $[t'', t]$ では P_i, P_jはともに正常に動作し続けるので $steps(P_i, t'', t) = steps(P_j, t'', t)$ が成り立つ. 従って, 式 (2.2) は成り立つ. ■

次の補題で, 居眠りをしてから $partial_reset$を行うまでのステップ数を考察する.

補題 9 プロセス P_iが時刻 t で $partial_reset$を実行したならば, $work(P_i, t) < 6n$ が成り立つ.

(証明) P_iが時刻 t で R_mを読み出すとし, t より前で最後に P_iが R_jを読み出すステップを行った時刻を t'とする. 時刻 t の $partial_reset$が時間超過リセットならば, 補題 5より $work(P_i, t) < 6n$ が成り立つ. 以下では, 時刻 t の $partial_reset$が居眠りリセットである場合, すなわち (a)$[\text{naps I}]_i$, (b)$[\text{naps II}]_i$, (c) 第 52–52 行の条件式のうちいずれかが成り立つ場合を考える.

(a) $[\text{naps I}]_i$：$P_i.Work\_count(t-1) \geq n$ かつ $P_i.diff(t) < 0$

プロセス P_iは区間 $[t'+1, t-1]$ で居眠りをしている. $P_i.Work\_count \geq n$, 事実 3 より $steps(P_i, t', t) \leq 2n$ であり, $work(P_i, t) < 2n$ が成り立つ.

(b) $[\text{naps II}]_i$：$P_i.Work\_count(t-1) \geq n$ かつ $P_j.Invalid_i \geq Gen$

P_jが P_iの居眠りを検知し $P_j.Invalid_i = P_i.Gen(t-1)$ にセットした時刻を t_jとする. t_jより前で最後に P_iが R_jを読み出した時刻を t'_jとすると, 区間 $[t'_j, t_j - 1]$ のある時刻で P_iは居眠りをしているので, 事実 3より $steps(P_i, t'_j, t_j) < steps(P_j, t'_j, t_j) \leq 3n - 1$が成り立つ. さらに, 区間 $[t_j + 1, t - 1]$ には P_iが R_jを読み出すステップが存在しないので, $P_i.Work\_count(t-1) \geq n$, 事実 3より $steps(P_i, t_j + 1, t) < 2n$ が成り立つ. 従って, $steps(P_i, t'_j, t) < 5n$ であり, 区間 $[t'_j, t_j - 1]$ のある時刻で P_iは居眠りをしているので $work(P_i, t) < 5n$ が成り立つ.

(c) 第 52–52 行:$P_i.Sync(t-1) \not\subset P_j.Async(t-1)$, $P_j.Sync \neq \emptyset$ かつ $P_i.Clock(t-1) \neq P_j.Clock\ (t-1)$

時刻 t で第 53 行を行うことより $P_i.Work\_count(t-1) < 6n$ であり, $P_i.Sync(t-1) \neq \emptyset$ より P_iは $5n \leq P_i.Work\_count(t_m - 1) < P_i.Work\_count(t-1)$ である時刻 t_mで, 調整済モードのプロセス P_mの局所時計と自分の局所時計を合わせるか, 一致していることを確認している. このとき, $P_i.Clock(t_m) = P_m.Clock(t_m - 1) + 1$ が成り立つ. 以下, $work(P_i, t) \geq 6n$ を仮定して矛盾を導く.

P_i, P_jが t 以前で最後に $sort_list$を行った時刻をそれぞれ t_i, t_jとする. また, t 以前で P_iが最後に $partial_reset$を行った時刻を t^0とする. すなわち, $Work\_count(t^0) = 0$ とする. $work(P_i, t) \geq 6n$ かつ $P_i.Work\_count(t-1) < 6n$ より, P_iは $[t^0, t]$ では居眠りすることなく正常に動作し続ける. 事実 3より, $[t^0 + 1, t^0 + n]$ に P_iが R_j, R_mそれぞれを読み出すステップが存在する. また, P_iは, t_mより前で P_mの, t より前で P_jの居眠りを検知しない. 従って, $work(P_m, t_m - 1) \geq t_m - t^0 - n > 4n$, $work(P_j, t-1) > work(P_j, t_m - 1) > 4n$ が成り立つ. 以下, 時刻 $t_m - 1$ での P_j のモードにより場合分けして考える.

(c1) 時刻 $t_m - 1$ で P_jが調整済モードの場合

補題 8より $P_m.Clock(t_m - 1) = P_j.Clock(t_m - 1)$ が成り立つ. 区間 $[t_m, t-1]$ で P_i, P_jが居眠りすることはないので, $P_i.Clock(t-1) = P_j.Clock(t-1)$ が成り立ち, (c) に矛盾する.

(c2) 時刻 $t_m - 1$ で P_jが調整中モードの場合

P_i, P_jはともに $t_m - 3n(> t - 4n)$ 以降の各プロセスの $Work\_count$ の値を基に $sort_list$を行う. P_i, P_j, P_mは $[t_m - 3n, t_m - 1]$ で正常に動作し続けるので, $P_m \longrightarrow^i_{t_i} P_j$より $P_m \longrightarrow^j_{t_j} P_j$が成り立つ. 従って, P_jは世代 $P_j.Gen(t-1)$ 中のある時刻 t'_mで P_mの局所時計に自分の局所時計を合わせるか, P_mと自分の局所時計が一致することを確認している. すなわち, $P_j.Clock(t'_m) = P_m.Clock(t'_m - 1) + 1$ が成り立つ. P_mは区間 $[t'_m, t_m - 1]$ または $[t_m, t'_m - 1]$ で正常に動作し続け, P_jは区間 $[t'_m, t-1]$ で正常に動作し続ける. 従って, $steps(P_j, t'_m, t) = steps(P_m, t'_m, t_m) + steps(P_i, t_m, t)$ より $P_i.Clock(t-1) = P_j.Clock(t-1)$ が成り立ち, (c) に矛盾する. ■

補題 10 (調整完了性) 任意の $t \geq 1$, 任意のプロセス P_iに対し, $work(P_i, t-1) \geq 12n$ ならば $P_i.Mode\ (t-1) =$ "$adjusted$"が成り立ち, $work(P_i, t) > 12n$ ならば $P_i.Clock(t) = P_i.Clock(t-1) + 1$ が成り立つ.

(証明) プロセス P_iが $t - 12n$ 以前に居眠りしているなら, $t - 12n$ 以前で最後に居眠りした時刻を t_{nap}とする. プロセス P_iが $t - 12n$ 以前に居眠りしていないなら $t_n = 1$ とする. 時刻 t_{nap}以降の時刻 t'で $partial\_reset$ を実行した場合, 補題 9より, $work(P_i, t') < 6n$ が成り立つ. よって, $work(P_i, t) \geq 12n$ ならば, P_iは時刻 $t - 6n$ 以降に $partial_reset$を行うことはない. よって, $P_i.Work\_count(t-1) \geq 6n$ が成り立ち, 補題 5より $P_i.Mode(t-1) =$ "$adjusted$"が成り立つ. また, $work(P_i, t) > 12n$ ならば, P_iは時刻 t で調整済モードのステップを行うので $P_i.Clock(t) = P_i.Clock(t-1) + 1$ が成り立つ. ■

補題 11 (一致性) 任意の $t \geq 1$, 任意のプロセス P_i, P_jに対し, $work(P_i, t) \geq 12n, work(P_j, t) \geq 12n$ ならば $P_i.Clock(t) = P_j.Clock(t)$ が成り立つ.

(証明) 補題 10より, $P_i.Mode(t-1) =$ "$adjusted$" かつ $P_j.Mode(t-1) =$ "$adjusted$"である. よって, 補題 8より $P_i.Clock(t) = P_j.Clock(t)$ が成り立つ. ■

補題 10, 11より, 定理 12 が言える.

定理 12 アルゴリズム WCSは, 同期時間 $12n$ の無待機時計合わせアルゴリズムである. ■

shared variable	**local variable**
$Clock$, $\{0, ..., M-1\}$	cur
$Count$, $\{0, ..., 12n-1\}$	$latest\_gen_j (j = 0..n-1)$, $\{0, 1, ..., n+1\}$
$Work\_count$, $\{-1, 0, ..., 8n\}$	$latest\_invalid_j (j = 0..n-1)$, $\{0, 1, ..., n+1\}$)
Gen, $\{0, 1, ..., n+1\}$	$latest\_L\_mycount_j (j = 0..n-1)$
$Invalid_j$ $(j = 0..n-1)$, $\{0, 1, ..., n+1\}$	$noread_j (j = 0..n-1)$, $\{0, 1, .., 6n-1\}$)
$Last\_count_j$ $(j = 0..n-1)$	$ignore$, "$true$" or "$false$"
$Last\_mycount_j$ $(j = 0..n-1)$	$next$
$Stagnant_j$ $(j = 0..n-1)$	pos
$Mode$, "$adjusting$" or "$adjusted$"	$key_j (j = 0...n-1)$
$Sync$, $\subset \{P_0, P_1, ..., P_{n-1}\}$	$list[0..n-1]$
$Async$, $\subset \{P_0, P_1, ..., P_{n-1}\}$	

図 2.11 アルゴリズム $ssWCS$ (P_iの変数)

2.5. 自己安定無待機時計合わせアルゴリズム

2.5.1 アルゴリズム $ssWCS$

本節で, フェーズ内システムにおける空間複雑度が有界な同期時間 $15n$ の自己安定無待機時計合わせアルゴリズム $ssWCS$を提案する. プロセッサ P_iのプログラムを図 2.11 –2.14に示す. 以下, アルゴリズム WCSからの修正点を説明する.

自己安定性

初期化実行以外の実行を考慮に入れる自己安定性を持たせるため, 以下の 2 点を修正する.

まず, 開始状況における矛盾性を排除するため, プロセス P_iはすべてのステップで 3 つの矛盾性のチェック (inconsistency I)$_i$, (inconsistency II)$_i$, (inconsistency III)$_i$を行う (図 2.13の 102 行目から 107 行目). P_i が矛盾を検出したならば, $partial\_reset$ を行い時計調整をやり直す. この場合の $partial\_reset$ は**矛盾リセット**と呼んで居眠りリセット, 時間超過リセットと区別する.

次に, 居眠りのチェックに関する修正点を述べる. プロセス P_jは, R_iを読み出

```
66  repeat forever do on receipt of a pulse
67      read R_cur;
68      ignore := false;
69      if (check_nap) then partial_reset;      /* 居眠りリセット */
70      elseif (check_consistency) then partial_reset;      /* 矛盾リセット */
71      elseif Mode = "adjusting" then adjust;
72      else /* Mode = "adjusted" */
73          Clock := Clock + 1 (mod M);
74          next := cur + 1 (mod n);
75      Last_count_cur := P_cur.Count;
76      latest_L_mycount_cur := P_cur.Last_mycount_i;
77      Last_mycount_cur := Count;
78      latest_gen_cur := P_cur.Gen;
79      latest_invalid_cur := P_cur.Invalid_i;
80      cur := next;
81      Count := Count + 1;
82      Work_count := min(Work_count, 8n − 1) + 1;
83  end
```

図 2.12 アルゴリズム $ssWCS$ (P_iのメインプログラム)

すステップでは, P_iと P_jの変数 $Count$ の増分の差を比較することにより P_iと P_j の居眠りのチェックを行う. しかし, 開始状況では各変数の値が任意であるため, 開始状況からプロセス P_jが最初に R_iの読み出しをするときは, 正しく居眠りのチェックができない. 従って, P_iが同期時間以上の間, 正常に動作し続けた後でも, P_jが十分に動作していなければ, P_jが最初に R_iを読み出すステップで誤って P_iの居眠りを検出することがある. このような誤りを避けるため, 以下のようにして P_iは P_jに居眠りのチェックをしないよう伝える (図 2.15). P_iは, P_jが R_i の読み出しをしなかった区間に P_iが行ったステップ数を変数 $noread_j$により数える. 各プロセスは R_iを最後に読んだステップにおける $P_i.Count$ の値を表す変数 $P_j.Last_mycount_i$を持っており, この変数が更新されているかどうかにより P_jが R_iの読み出しをしなかった区間を判断できる. アルゴリズム WCSと同様に, アルゴリズム $ssWCS$でも P_jは少なくとも連続する $3n-1$ ステップに 1 回は R_iの

```
84  procedure check_nap;
85      diff := (12n + Count − Last_mycount_cur mod 12n)
86          −(12n + P_cur.Count − Last_count_cur mod 12n);
87      if P_cur.Stagnant_i = Last_mycount_cur and diff > 0    then diff := ⊥;
88      if (P_cur.Last_mycount_i = latest_L_mycount_cur and diff ≠ ⊥)
89          then noread_cur := min(noread_cur + Count − Last_mycount_cur, 3n);
90            if noread_cur ≥ 3n then Stagnant_cur := P_cur.Last_mycount_i;
91                                   else Stagnant_cur := ⊥;
92          else noread_cur := 0;Stagnant_cur := ⊥;
93      if (diff > 0 and P_cur.Gen = latest_gen_cur)
94          then Invalid_cur := P_cur.Gen; ignore := true;
95      return { [naps I]_i: Work_count ≥ n and diff < 0 } or
96              { [naps II]_i: Work_count ≥ n and P_cur.Invalid_i = Gen }
97  end procedure;

98  procedure check_consistency;
99      if (P_cur.Work_count(t − 1) ≥ 5n and P_cur.Mode(t − 1) = "adjusting") or
100         (P_i.Work_count(t − 1) ≥ 5n and P_i.Mode(t − 1) = "adjusting")
101         then ignore := true;
102     return  { [inconsistency I]_i: P_i.Work_count(t − 1) ≥ 6n
103                         and P_i.Mode(t − 1) = "adjusting" } or
104             { [inconsistency II]_i: P_i.Work_count(t − 1) ≤ 5n − 1
105                         and P_i.Mode(t − 1) = "adjusted" } or
106             { [inconsistency III]_i: P_i.Mode = "adjusted" and P_cur.Mode = "adjusted"
107                         and P_i.Invalid_cur ≠ P_cur.Gen and P_i.Clock ≠ P_cur.Clock }
108 end procedure;

109 procedure partial_reset;
110     Gen := min{g|g ∉ {latest_invalid_0, ..., latest_invalid_n−1, Gen}};
111     next := 0; Mode := "adjusting"; Work_count := −1; /* becomes 0 at line 82. */
112 end procedure;

113 procedure sort_list;
114     list[0..n − 1] := sort ids in the descending lexicographic order of (key_id, id);
115     Sync := ∅; Async := ∅; pos := 0; check_adjusted;
116 end procedure;
```

図 2.13 アルゴリズム $ssWCS$ (P_iの手続き1)

```
117 procedure adjust;
118   if 0 ≤ Work_count ≤ 4n − 1 then next := Work_count + 1 (mod n);
119   elseif 4n ≤ Work_count ≤ 5n − 1 then
120     begin
121       if Invalid_cur ≠ P_cur.Gen
122         then key_cur := P_cur.Work_count − Work_count;
123         else key_cur := −1;
124       next := Work_count + 1 (mod n);
125       if Work_count = 5n − 1 then sort_list;
126     end
127   else /* Work_count ≥ 5n */
128     if ignore or P_cur.Work_count < Work_count then
129       begin
130         Async := Async ∪ {P_cur};     /* 無視 */
131         pos := pos + 1; check_adjusted; Clock := Clock + 1 (mod M);
132       end
133     elseif P_cur.Mode = "adjusted" then
134       begin
135         if (Sync ⊈ P_cur.Async and P_cur.Sync ≠ ∅ and Clock ≠ P_cur.Clock)
136           then partial_reset;        /* 居眠りリセット */
137         else
138           begin
139             Clock := P_cur.Clock + 1 (mod M);
140             if Sync ⊆ P_cur.Async
141               then Async := Async ∪ Sync; Sync := ∅;     /* 棄却 */
142             Sync := Sync ∪ {P_cur}; pos := pos + 1; check_adjusted;
143           end
144       end
145     if (Work_count = 5n − 1 and Mode = "adjusting")
146       then partial_reset;          /* 時間超過リセット */
147 end procedure;

148 procedure check_adjusted;
149     if list[pos] = i then Mode := "adjusted"; next := 0; else next := list[pos];
150 end procedure;
```

図 2.14 アルゴリズム $ssWCS$ (P_iの手続き 2)

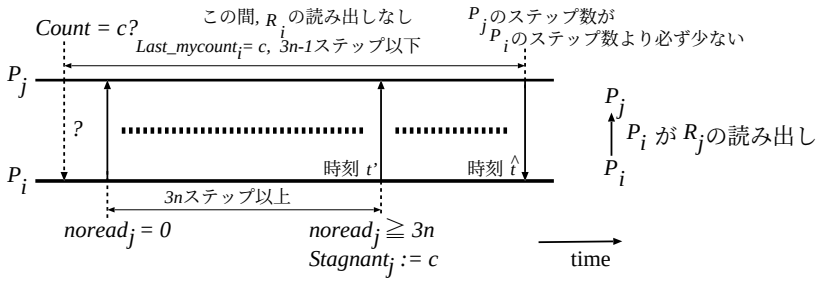


図 2.15 変数 $P_i.noread_j$による居眠り検知

読み出しをする. 従って, $P_i.noread_j > 3n-1$ になれば, P_jが次に R_iの読み出しをするとき (図 2.15の時刻$\hat{t}$) は, 前回 P_jが R_iを読み出したステップまたは開始状況からのステップ数は P_jが $3n-1$ 以下, P_iが $3n$ 以上となり, P_jの方が必ず少ない. そこで, P_iは $P_i.noread_j > 3n-1$ になったとき (図 2.15の時刻 t') に, 変数 $P_i.Stagnant_j$に $P_j.Last_mycount_i$の値を代入する. P_jが次に R_iを読み出す時刻$\hat{t}$では, $P_i.Stagnant_j$より P_jが正確に $count$ の増分の差を比較できないことを知り, P_iの居眠り検出を避ける. この修正により, P_iが十分なステップ数を行った後で P_jが誤って P_i の居眠りを検出することは避けられる. しかし, P_iが十分なステップ数を行うまでは P_jは誤った居眠り検知をおこしうる. その結果, アルゴリズム $ssWCS$の同期時間はアルゴリズム WCSより長くなる.

空間複雑度の有界化

前節のアルゴリズム WCSは空間複雑度の有界性を考慮していないため, 複数の非有界変数を含んでいた. アルゴリズム WCSで用いる 3 つの変数 $Work\_count$, Gen, $Count$ を有界にすれば, 他の非有界変数はこれらの変数のうち 1 つをコピーして用いるため, すべての変数が有界化される. 以下, それぞれの変数の有界化の方法を述べる.

変数 $Work\_count$ はプロセスが現在の世代でのステップ数を表す変数であり, 調整中モードにあるプロセスが他のプロセスが現世代の時計調整をいつ開始したかを知るために使う. 調整中モードの調整ピリオドにいるプロセス P_iは, アルゴリズム WCSの説明のようにそのときの世代を早く開始したプロセスの局所時計

から順に, P_iの局所時計を合わせる, または一致確認を行う. ただし, 読み出し先プロセスがまだ調整中モードにあるときは, そのプロセスが調整済モードへ移行するのを待つ. 調整ピリオドでの読み出し順を世代を早く開始した順にすることにより, P_iが調整ピリオドにいるときに他のプロセスが調整済モードへ移行するのを待つステップ数が n 以下になることが保証される. ここで, 既にプロセス $P_j.Work\_count$ が十分大きいならば, P_i が P_jの調整済モードへの移行を待つことはないので, そのような $Work\_count$ が十分大きいプロセスに対しては時計一致を確認する順序が変わっても, P_iが調整ピリオドにいるときに他のプロセスが調整済モードへ移行するのを待つステップ数が n 以下になることが保証される. そこで, $Work\_count$ の上界を $8n$ に設定することができる.

変数 $P_i.Gen$ は, $partial\_reset$ を行い新しい世代の時計調整を開始するたびに更新する変数であり, 他のプロセス P_jが $P_j.Invalid_i = P_i.Gen$ にセットすることで P_iの居眠り検知を知らせるために使う. アルゴリズム $ssWCS$では, $partial\_reset$ のたびに Gen を 1 増加させている. プロセス P_iは, 変数 Gen を更新することにより $P_j.Invalid_i = P_i.Gen$ を破棄するが, この目的のためには Gen を他のすべてのプロセス P_jの $Invalid_i$ および $partial\_reset$ 前の $P_i.Gen$ と異なる値にセットすれば十分である. アルゴリズム $ssWCS$ではこのように変更することにより, 変数 Gen のサイズを $n+1$ としている. アルゴリズム $ssWCS$では, プロセスは世代番号 gで $partial\_reset$ を行って次の世代の時計調整を開始したあと, 次に $partial\_reset$ を行って開始した世代の世代番号が再び gになることがある. 以下では, 連続しない 2 つの異なる時計調整の世代は, 世代番号が同じでも区別できるものとして説明する.

変数 $Count$ は, ステップするたびに 1 増加する変数であり, 各プロセス P_iが R_jを読み出すとき, P_iが前回 R_jを読み出したときからの P_iと P_jの $Count$ の増分の差を比較することにより, P_i自身または P_jの居眠りをチェックする. ここで, P_iがある時刻 t より前に R_jの読み出しをしているならば, 後で示す補題 17 より $P_i.diff(t) \geq -9n$ が成り立つ. また, 先に少し触れたように, アルゴリズム $ssWCS$でも次の事実が成り立ち, $P_i.diff(t) \leq 3n-1$ が成り立つ.

事実 13 任意のプロセス P_i, P_jと時刻 t に対し, $steps\ (P_i, t', t)$ が以下を満たすような時刻 t'が存在するならば, P_iは区間 $[t', t-1]$ のある時刻で R_j を読み出すス

テップを行っている.

(a) $n \leq P_i.Work\_count(t-1) \leq 5n$ または $7n \leq P_i.Work\_count(t-1)$ ならば, $steps(P_i, t', t) = n$

(b) $5n < P_i.Work\_count(t-1) < 7n$ ならば, $steps(P_i, t', t) = 2n$

(c) $P_i.Work\_count(t-1) < n$ ならば, $steps(P_i, t',\ t) = 3n-1$ ■

アルゴリズム $ssWCS$では, 変数 $Count$ は $12n$ の剰余をとるようにする. $-9n \leq P_i.diff(t) \leq 3n-1$ より, このように変更しても居眠りのチェックが可能である.

2.5.2 アルゴリズム $ssWCS$の正当性

アルゴリズム $ssWCS$が同期時間 $15n$ の自己安定無待機時計合わせプロトコルであることを示す.

アルゴリズム $ssWCS$の正当性は, 同期時間を, プロセスが居眠りをしてから $partial\_reset$ を行うまでのステップ数, 開始状況から $partial\_reset$ を行うまでのステップ数と, 時計調整を始めてから ($partial\_reset$ を行ってから) 調整済モードへ移行するまでのステップ数に分けて考える. まず, 補題 14で矛盾リセットのうち 2 条件は開始状況直後にしか成立しないことを示す. 有界サイズになるように修正した変数 $Work\_count$, Gen, $Count$ に関して, 補題 15, 16, 17でその性質を示す. アルゴリズム $ssWCS$でもアルゴリズム WCSと同様に, 調整ピリオドでの n 回のステップ後も調整中モードから調整済モードへ移行しないならば, $P_i.Work\_count(t'-1) = 6n-1$ なる時刻 t'で時間超過リセットを行う. 補題 20では, 調整中モードのプロセスが時計調整を始めてから居眠りをしていない, かつ居眠りリセットを行わないならば, $P_i.Work\_count(t'-1) = 6n-1$ なる時刻 t'までに調整済モードへ移行することを示す. 補題 24では, 調整済モードの 2 つのプロセスが十分長く居眠りすることなく動作し続けているならば, 両者の局所時計の値が一致することを示す. 補題 25では, プロセスが居眠りをしてから居眠りリセットを行うまでのステップ数が高々$9n$ であることを示す. 補題 26では, プロセスが開始状況から矛盾リセットを行うまでのステップ数が高々$7n$ であることを示す. 補題 27, 28で, 任意の実行に対し調整完了性と一致性が成り立つことを示す.

本節の一部の補題の証明は, 2.4.2節のアルゴリズム WCSの証明と同様の方針によりに示すことができる.

まず, 矛盾リセットの条件のうち,

[inconsistency I]$_i(t)$: $P_i.Work\_count(t-1) \geq 6n$ **and**
$P_i.Mode(t-1) =$ "$adjusting$"

[inconsistency II]$_i(t)$: $P_i.Work\_count(t-1) \leq 5n-1$ **and**
$P_i.Mode(t-1) =$ "$adjusted$"

に関して以下の補題が成り立つ.

補題 14 プロセス P_iが時刻 t で条件 *[inconsistency I]*$_i$, *[inconsistency II]*$_i$の成立により矛盾リセットを実行したならば, $work(P_i, t) = 1$ が成り立つ.

(証明) $work(P_i, t) \geq 2$ を仮定する. このとき, P_iが時刻 $t-1$ で $partial\_reset$ を行うならば, $Work\_count(t-1) = 0 \wedge Mode(t-1) =$ "$adjusting$"が成立し条件 [inconsistency I]$_i(t)$, [inconsistency II]$_i(t)$ に反する. 以下, P_iが時刻 $t-1$ で $partial\_reset$ を行わないときを考える. このとき, 条件 [inconsistency I]$_i(t-1)$, [inconsistency II]$_i(t-1)$ が成り立つことはなく, また $P_i.Work\_count(t-1) = P_i.Work\_count(t-2)+1$ が成り立つ. $P_i.Work\_count(t-2) \geq 6n$ かつ $P_i.Mode(t-2) =$ "$adjusted$" ならば, 時刻 $t-1$ でも調整済モードのままである. $P_i.Work\_count(t-2) = 6n-1$ ならば, 時刻 $t-1$ のステップで手続き $check\_adjusted$ が true を返すならば調整済モードへ移行する. $check\_adjusted$ が false を返すならば 146 行目より時刻 $t-1$ で $partial\_reset$ を実行することになり矛盾する. $P_i.Work\_count(t-2) \leq 5n-2$ かつ $P_i.Mode(t-2) =$ "$adjusting$"ならば, 時刻 $t-1$ でも調整中モードのままである. 従って, 条件 [inconsistency I]$_i(t)$, [inconsistency II]$_i(t)$ が成り立つことはなく, プロセス P_iが時刻 t で条件 [inconsistency I]$_i$, [inconsistency II]$_i$の成立により矛盾リセットを実行することに矛盾する. ■

変数 $Work\_count$ には上界を設定したが, 以下の補題が成り立つ.

補題 15 プロセス P_iがある時刻 t_iで手続き $sort\_list$ を行ったとし, 時刻 t'を P_iが世代 $P_i.Gen(t_i)$ にいる任意の時刻とする. また, P_jを任意のプロセスとする. このとき, *(1)* $t' < t_i$かつ P_i, P_jともに区間 $[t', t_i-1]$ で正常に動作し続けて $partial\_reset$

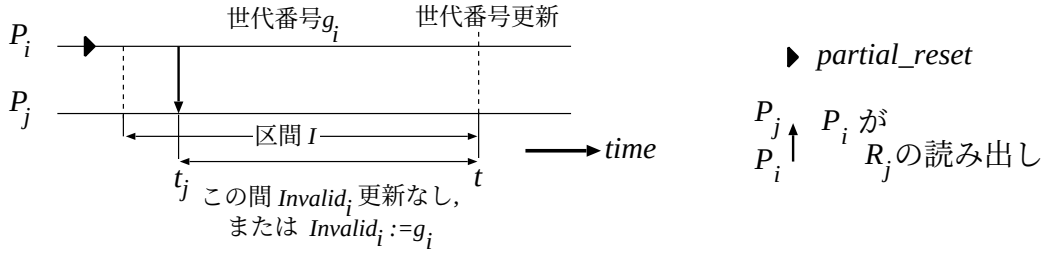


図 2.16 世代番号 Gen の更新

を行わないとき, $((P_j.Work\_count(t_i), j) < (P_i.Work\_count(t_i), i)$ ならば, かつそのときに限り, $((P_j.Work\_count(t'), j) < (P_i.Work\_count(t'), i)$ が成り立つ.
(2) $t_i < t' \leq t_i + 3n$ かつ P_i, P_jともに区間 $[t_i, t' - 1]$ で正常に動作し続けて $partial\_reset$ を行わないとき, $((P_j.Work\_count(t_i), j) < (P_i.Work\_count(t_i), i)$ ならば, $((P_j.Work\_count(t'), j) < (P_i.Work\_count(t'), i)$ が成り立つ.

(証明) $t_i - 5n < t' \leq t_i + 3n$ なる時刻 t'に対して

$$P_i.Work\_count(t_i) = P_i.Work\_count(t') + t_i - t' \tag{2.3}$$

が成り立つ.

(1) $0 < P_j.Work\_count(t_i) < 8n$ ならば $P_j.Work\_count(t_i) = P_j.Work\_count(t') + (t_i - t')$ が成り立つ. このとき, 式2.3より補題が成り立つ. $P_j.Work\_count(t_i) = 8n$ ならば $P_j.Work_count(t_i) - P_i.Work_count(t_i) = 3n > 0$が必ず成り立つ. このとき, $P_j.Work\_count(t_i) = \min\{(P_j.Work\_count(t') + (t_i - t')), 8n\}$ と式2.3より $P_j.Work\_count(t') - P_i.Work\_count(t') \geq P_j.Work\_count(t_i) - P_i.Work\_count(t_i) > 0$ も成り立つ.

(2) $P_j.Work\_count(t'-1) \leq P_i.Work\_count(t'-1) < 8n$ より $P_j.Work\_count(t') = P_j.Work\_count(t_i) + (t' - t_i)$ が成り立つ. このとき, 式 2.3より補題が成り立つ. ■

有界化した世代番号 Gen に関して, 以下の補題が成り立つ.

補題 16 $work(P_i, t) \geq 2n + 1$ かつプロセス P_iは $partial\_reset$ を時刻 t で行うとする. このとき, 任意の $j(j \neq i)$ に対し $P_i.Gen(t) \neq P_j.Invalid_i(t)$ かつ $P_i.Gen(t) \neq P_i.Gen(t-1)$ が成り立つ.

(証明) 補題 14と $work(P_i, t) \geq 2n + 1$ より, 時刻 t の P_iの $partial\_reset$ は条件 [inconsistency I]$_i$, [inconsistency II]$_i$の成立によるものではない. よって, 他の $partial\_reset$ の条件より, $P_i.Work_count(t-1) \geq n$が成り立つ. 従って, $[t-n, t-1]$ では P_iは $partial\_reset$ を行っていない. 事実 13 より P_iは区間 $[t-2n, t-1]$ で各 R_jを少なくとも 1 回ずつ読み出している. ここで, $P_i.last\_invalid_j(t-1)(j \neq i)$ を g_j, $P_i.Gen(t-1)$ を g_iとおく. このとき, 区間 Iを以下のように定義する.

- $[t-2n, t-n-1]$ で P_iは $partial\_reset$ を行っているならば, $I = [t-n, t-1]$ とする. このとき, 事実 13より t_jは区間 $[t-n, t-1]$ に存在する.
- $[t-2n, t-n-1]$ で P_iは $partial\_reset$ を行っていないならば, $I = [t-2n, t-1]$ とする.

図 2.16に示すように, 区間 Iでは P_iは $partial\_reset$ を行っていないので, その間は $P_i.Gen = g_i$が成り立つ. よって, ある P_jが $[t_j, t]$ で $P_j.Invalid_i$を更新するならば, g_iに更新する. 手続き $assign\_gen$ は, 任意の g_jに対して$\bar{g} \neq g_j$となるような番号$\bar{g}$を指定するので, 補題が成り立つ. ■

従って, アルゴリズム $ssWCS$では, プロセス P_i が $partial\_reset$ 前に $2n+1$ パルス以上連続して動作しているときは, 他のプロセス P_jが P_iの居眠りを検知した世代として指定していない世代番号が $partial\_reset$ で割り当てられる.

自己安定アルゴリズム $ssWCS$では, 手続き $check\_nap$ 内で $P_i.Stagnant_j$に $P_j.Last_mycount_i$の値を代入することで, プロセス P_iは P_jに P_iの居眠り判定をしないよう命令することがある. この手続きに関して以下の補題が成り立つ.

補題 17 P_iがある時刻 t で読み出すレジスタを R_jとし, その後の時刻 t'で P_iが再び R_jを読み出したとする. このとき, *(1)* $steps(P_i, t, t') - steps(P_j, t, t') < -9n$ ならば, $P_j.Stagnant_i(t'-1) = P_i.Last\_mycount_j(t'-1)$ が成り立つ. *(2)* $P_j.Stagnant_i(t'-1) = P_i.Last\_mycount_j(t'-1)$ ならば, $steps(P_i, t, t') - steps(P_j, t, t') < 0$ が成り立つ.

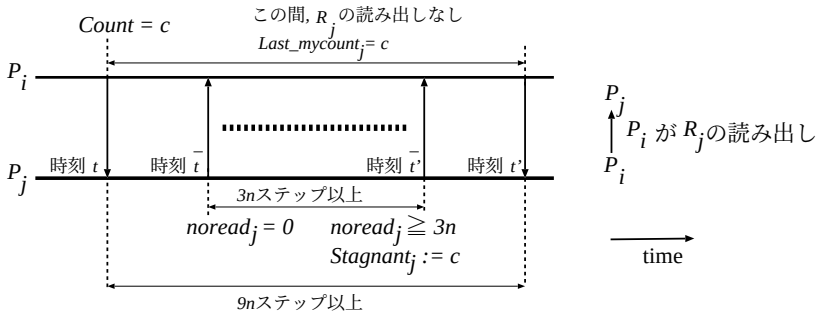


図 2.17 補題 17：$steps(P_i, t, t') - steps(P_j, t, t') < -9n$ の場合

(証明) (1) $steps(P_i, t, t') - steps(P_j, t, t') < -9n$ より $steps(P_j, t, t') > 9n$ が成り立つ. よって, 事実 13より, 時刻 t より後で初めて P_jが R_iを読み出した時刻$\bar{t}$, 時刻 t'より前で最後に P_jが R_iを読み出した時刻$\bar{t}'$が存在し, $steps(P_j, t, \bar{t}) \leq 3n - 1$, $steps(P_j, \bar{t}', t') \leq 3n - 1$ が成り立つ. よって,

$$\begin{aligned} steps(P_j, \bar{t}, \bar{t}') &= steps(P_j, t, t') - steps(P_j, t, \bar{t}) - steps(P_j, \bar{t}', t') \\ &> 3n \end{aligned}$$

が成り立つ (図 2.17). 区間 $[\bar{t}, \bar{t}' - 1]$ では P_iは一回も R_jを読み出していないので, この間は $P_i.Last_count_j$が更新されない. よって, $[\bar{t} + 1, \bar{t}']$ で P_jが R_iを読み出した各時刻で $P_j.noread_i$に $P_j.Count - P_j.Last_mycount_i$が加算される. このとき, $P_j.noread_i(\bar{t}') \geq 3n$ となり, $P_j.Stagnant_i$が $P_i.Last\_mycount_j(\bar{t}' - 1)$ に更新される. 従って, $P_j.Stagnant_i(t' - 1) = P_i.Last\_mycount_j(t' - 1)$ が成り立つ.

(2) 時刻 t より後で初めて P_jが R_iを読み出した時刻を$\bar{t}$, 時刻 t'より前で最後に P_jが R_iを読み出した時刻を$\bar{t}'$とする. 区間 $[\bar{t}, \bar{t}' - 1]$ では P_iは一回も R_jを読み出していないので, この間は $P_i.Last_count_j$が更新されない. よって, $[\bar{t} + 1, \bar{t}']$ で P_jが R_iを読み出した各時刻で $P_j.noread_i$に $P_j.Count - P_j.Last_mycount_i$が加算される. $P_j.Stagnant_i(t' - 1) = P_i.Last\_mycount_j(t' - 1)$ より $P_j.Stagnant_i(\bar{t}') = P_i.Last\_mycount_j(\bar{t}' - 1)$ も成り立つ. すなわち, 時刻$\bar{t}'$では $P_j.noread_i(\bar{t}') \geq 3n$ となる. すなわち, $steps(P_j, \bar{t}, \bar{t}') - steps(P_i, \bar{t}, \bar{t}') > 3n$ が成り立つ. また, 区間 $[t, t']$ では P_iは R_jの読み出しをしないので, 事実 13より $steps(P_i, t, t') - steps(P_i, \bar{t}, \bar{t}') <$

$steps(P_i, t, t') \leq 3n-1$ が成り立つ. 従って,

$$\begin{aligned} steps(P_j, t, t') - steps(P_i, t, t') &\geq steps(P_j, \bar{t}, \bar{t'}) - steps(P_i, t, t') \\ &\geq steps(P_j, \bar{t}, \bar{t'}) - steps(P_i, \bar{t}, \bar{t'}) - (3n-1) \\ &> 0 \end{aligned}$$

が成り立つ. ■

以上より, プロセス P_iが時刻 t で読み出すレジスタ R_jに対し, P_i が t より前にレジスタ R_jを読み出しているならば, $steps(P_i, t, t') - steps(P_j, t, t') < -9n$ のときに P_iは $Count$ の増分差を比較しない (87 行が成立する). 実際, P_iが $Count$ の増分差を比較しないときは P_iの方が区間 $[t, t']$ でのステップ数は少ない.

手続き $sort_list$のソートの結果に関して以下の補題が成り立つ.

補題 18 任意の時刻を t とする. 区間 $[t+2n+1, t+5n]$ において, 3 つのプロセス $P_{j_1}, P_{j_2}, P_{j_3}$は正常に動作し続け, かつ $partial_reset$を行わないとする. また, $(t_1, j_1) \preceq (t_2, j_2) \prec (t_3, j_3)$, $t_1 \geq t+4n+1, t_3 \leq t+5n$ である時刻 t_1, t_2, t_3で, それぞれ $P_{j_1}, P_{j_2}, P_{j_3}$が手続き $sort_list$を行ったとする. このとき, $P_{j_2} \longrightarrow_{t_3}^{j_3} P_{j_3}$ ならば, $P_{j'} \longrightarrow_{t_1}^{j_1} P_{j_1}$ なるすべての j'に対し, $P_{j'} \longrightarrow_{t_3}^{j_3} P_{j_2}$ または $P_{j_3} \longrightarrow_{t_3}^{j_3} P_{j'}$ が成り立つ.

(証明) P_{j_1}は, 時刻 $t_1 - n + j'$のステップで $R_{j'}$を読み出し, $P_{j_1}.key_{j'}$の値を決定し, 時刻 t_1で $sort_list$を行う. ここで, $P_{j_1}.key_{j'} = x$ とすると, $P_{j'} \longrightarrow_{t_1}^{j_1} P_{j_1}$ より $(x, j') > (0, j_1)$ である. 一方 P_{j_3}は, 時刻 $t_3 - n + j'$のステップで $R_{j'}$を読み出して $P_{j_3}.key_{j'}$の値を決定し, 時刻 t_3で $sort_list$を行う. このとき, $t+4n+1 \leq t_1 \leq t_3 \leq t+5n$ より $t_3 - 2n + j' < t_1 - n + j' \leq t_3 - n + j'$が成り立つ. P_{j_3}は区間 $[t_3-2n+j', t_3-n+j']$ では正常に動作し続けるので, $P_{j'}$がこの区間で居眠りをしているならば, 時刻 t_3-n+j'のステップで P_{j_3}はそれを検知する. 時刻 t_3-n+j'に P_{j_3}が $P_{j'}$の居眠りを検知しないなら以下の (a) が成り立ち, $P_{j'}$の居眠りを検知するなら以下の (b) が成り立つ.

(a)

$$(P_{j_3}.key_{j'}, j') = ((P_{j_3}.Work\_count(t_3 - n + j' - 1) -$$

$$
\begin{aligned}
& P_{j'}.Work\_count(t_3 - n + j' - 1)), j') \\
= \ & ((\min\{(4n + j' + x + t_3 - t_1), 8n\} - (4n + j')), j') \\
= \ & (\min\{(x + t_3 - t_1), (4n - j')\}, j') \\
> \ & (t_3 - t_1, j_1) \\
\geq \ & (t_3 - t_2, j_2) \\
= \ & (P_{j_3}.key_{j_2}, j_2) \qquad (\because\ P_{j_2} \longrightarrow_{t_3}^{j_3} P_{j_3} \text{より})
\end{aligned}
$$

(b)

$$
(P_{j_3}.key_{j'}, j') \ = \ (-1, j') \ < \ (0, j_3) \ = \ (P_{j_3}.key_{j_3}, j_3)
$$

(a),(b) はそれぞれ $P_{j'} \longrightarrow_{t_3}^{j_3} P_{j_2}$, $P_{j_3} \longrightarrow_{t_3}^{j_3} P_{j'}$ を意味する. ■

P_iが世代 gの調整ピリオドでのステップ数, すなわち, 時刻 t で $sort_list$を行った後, $partial_reset$を実行するか調整済モードへ移行する時刻 t'までのステップ数 $steps(P_i, t+1, t'+1)$ を$\alpha(P_i, g)$ と記す.

補題 19,20では, 以下に示す条件 B を満たすプロセス P_iを考える.

条件 B： P_iが時刻tで $partial_reset$を実行し, $work\,(P_i, t+6n) \geq 6n$かつ $[t, t+6n]$ で居眠りリセット, 矛盾リセットを行わない.

条件 B を満たす P_iに対し, $\bar{\alpha} = \alpha(P_i, P_i.Gen(t))$ とする. このとき, 世代 $P_i.Gen(t)$ の時計調整で P_iが時間超過リセットを行うならば, 時刻 $t + 6n$ で $partial_reset$を行うので$\bar{\alpha} = n$ が成り立つ. よって, $\bar{\alpha} < n$ が成り立つならば, P_iが時刻 t までに調整済モードへ移行する.

アルゴリズム WCSの条件 A(21ページ) を満たすプロセスと同様にして, 条件 B を満たすプロセス P_iが調整ピリオドにいる区間 $[t+5n+1, t+5n+\bar{\alpha}]$ に属する各時刻 s に対し, 2 種類のプロセス $reader_s, dest_s$を以下のように時刻 $t + 5n + \bar{\alpha}$ から再帰的に定義する. まず, $reader_{t+5n+\bar{\alpha}} = P_i$とする. 時刻 s で $reader_s$ が読み出すレジスタの所有者を $dest_s$と定義する. $reader_s, dest_s (t + 5n + 2 \leq s \leq t + 5n + \bar{\alpha})$ に対し, $reader_s$が時刻 s と $s-1$ で同じレジスタの読み出しをするならば $reader_{s-1} = reader_s$と定義し, 異なるレジスタの読み出しをするならば $reader_{s-1} = dest_s$ と定義する. ここで, $dest_s$ $(t + 5n + 1 \leq s < t + 5n + \bar{\alpha})$ のリス

ト $dlist = dest_{t+5n+1}, dest_{t+5n+2}, \cdots, dest_{t+5n+\bar{\alpha}}$ を, $reader_s$が同一でありかつ連続するように分割した極大部分リスト $dlist^1, dlist^2, \cdots, dlist^h$を考える. 各 $x(1 \leq x \leq h)$ に対し, $dlist^x$で共通の $reader_s$を P_{i_x}, $dlist^x = dest_{s_x}, dest_{s_x+1}, \cdots, dest_{s'_x}$ とし, P_{i_x} が世代 $P_{i_x}.Gen(s'_x - 1)$ において $sort_list$を行うならば, その時刻を $sort_time^x$とする. ここで, 以下の補題が成り立つ.

補題 19 すべての $x(1 \leq x \leq h)$ に対し, 以下が成り立つ.

(a) $dlist^x$の要素はすべて異なる.

(b) P_{i_x}は区間 $[t + 2n + 1, s'_x - 1]$ で居眠りをしない.

(c) $x < h$ ならば, $(t + 4n + 1, 0) \preceq (sort\_time^x, i_x) \prec (sort\_time^{x+1}, i_{x+1})$ ■

補題 19の証明は, 補題 2より示される補題 4の証明と同様にして, 補題 18より示される.

さらに, 補題 19の 3つの性質より, 次の補題が成り立つ.

補題 20 条件 B を満たすプロセス P_iは時刻 $t + 6n$ までに調整済モードへ移行する. ■

補題 20の証明は補題 5と同様にして示される.

次に, 異なる 2プロセスの手続き $sort\_list$ のソートの結果の関係について示すが, 先に以下の補題を示す.

補題 21 任意の時刻を t, 任意の異なるプロセスを P_i, P_jとする. プロセス P_j は時刻 t_jで $sort_list$を行い, かつ, 区間 $[t_j, t]$ で $partial_reset$を行わなかったとする. また, $P_i.Work\_count(t - 1) \geq 5n$, $t_j \geq t - 2n$ と仮定する. このとき, $work(P_i, t) > 4n$, $work(P_j, t) > 4n$ ならば, 世代 $P_j.Gen(t)$ で時刻 t 以前にプロセス P_j が P_iの居眠りを検知しない.

(証明) 世代 $P_j.Gen(t)$ で時刻 t 以前にプロセス P_jが P_iの居眠りを検知すると仮定して矛盾を導く. $work(P_i, t) > 4n$ より, P_iは区間 $[t-4n+1, t]$ で居眠りをしていない. また, $2n \leq P_j.Work_count(t-3n-1) \leq 4n-1$かつ $steps(P_j, t-4n, t-3n) = n$ なので, 事実 13より区間 $[t - 4n, t - 3n - 1]$ のある時刻 t''で P_jは R_i を読み出

す. よって, 世代 $P_j.Gen(t)$ で P_jが t''より後で始めて P_iの居眠りを検知することはない. すなわち, 時刻 t 以前に P_iの居眠りを検知するならば, 遅くとも区間 $[t-4n, t-3n-1]$ で必ず検知する. また, $P_i.Work\_count(t-1) \geq 5n$ かつ $steps(P_i, t-2n+1, t+1) = 2n$ なので, 事実 13より区間 $[t-2n+1, t]$ に P_iが R_j を読み出すステップが必ず存在する. 従って, 区間 $[t'', t]$ で P_iが自身の居眠りを検知する. よって, 区間 $[t'', t]$ のある時刻で P_iは $partial_reset$を行う. 区間 $[t'', t]$ の長さは $4n-1$ 以下なので, $P_i.Work\_count(t) < 4n$ となり矛盾する. ■

補題 22 任意の時刻を t, 任意の異なるプロセスを P_i, P_jとする. 時刻 t に対し, $P_i.Work\_count(t-1) \geq 5n$ かつ $P_j.Work\_count(t-1) \geq 5n$ が成り立つとする. また, $P_i.Work\_count(\bar{t}) = 3n$ なる時刻$\bar{t}$ $(\bar{t} < t)$が存在したとし, P_iが区間 $[\bar{t}, t-1]$ で最後に手続き $sort\_list$ を行った時刻を t_i とする. このとき, $work(P_i, t) > 4n$, $work(P_j, t) > 4n$ かつ $P_i \longrightarrow^i_{t_i} P_j$ ならば, t 以前に P_jが手続き $sort\_list$ を行った時刻が存在し, かつ最後に P_jが手続き $sort\_list$ を行った時刻 t_jに対し $P_i \longrightarrow^j_{t_j} P_j$ が成り立つ.

(証明) まず, $P_j.Work\_count(\tilde{t_j}) = 5n$ なる時刻$\tilde{t_j}$が存在することを示す. 時刻$\tilde{t_j}$ が存在するならば, t 以前に P_jが手続き $sort\_list$ を行った時刻 t_jも存在する.

(1) $t_i \geq t-2n$ の場合

このとき, $t-3n < t_i-n+1 < t_i \leq t$が成り立つ. P_i, P_jはともに区間 $[t_i-n+1, t_i]$ で正常に動作し続け, この区間のある時刻 t'_iで P_iは key_jを設定するステップを行う. 補題 21より世代 $P_i.Gen(t)$ で時刻 t 以前にプロセス P_iが P_jの居眠りを検知することはないので $P_i \longrightarrow^i_{t_i} P_j$より, $P_j.Work\_count(t'_i-1) \leq P_i.Work\_count(t'_i-1)$ が成り立つ. $work(P_i, t) > 4n$, $work(P_j, t) > 4n$ より, $P_j.Work\_count(t_i-n) \leq P_i.Work\_count(t_i-n) = 4n$ も成り立つ. 従って, $P_j.Work\_count(\tilde{t_j}) = 5n$ なる時刻$\tilde{t_j}(\geq t_i-n)$ が区間 $[t-3n, t-n]$ に存在し, 時刻 $t_j(\geq t_i)$ も存在する.

(2) $t_i < t-2n$ の場合

プロセス P_iが t_i以前に最後に R_jを読み出すステップを行った時刻を t'_iとする. $P_i \longrightarrow^i_{t_i} P_j$より,

$$((P_j.Work\_count(t'_i-1) - P_i.Work\_count(t'_i-1)), j) < (0, i) \qquad (2.4)$$

または

$$P_i.Invalid_j(t'_i) = P_j.Gen(t'_i - 1) \tag{2.5}$$

が成り立つ. 特に, $P_i.Work\_count(\bar{t}) = 3n$ なる時刻$\bar{t}$が存在することより P_i が時刻 t'_i以前で最後に R_jを読み出した時刻$\tilde{t'_i}$が存在し, $steps(P_i, \tilde{t'_i}, t'_i) = n > steps(P_j, \tilde{t'_i}, t'_i)$ が成り立つならば, 式 (2.5) が成り立つ. 式 (2.5) が成り立つならば, P_iは時刻 $t'_i (< t-2n)$ 以前に P_jの居眠りを検知している. $P_j.Work\_count(t) > 5n$, $steps(P_i, t-2n, t) = 2n$ と事実 13 より, P_jは区間 $[t'_i, t-1]$ で P_j自身の居眠りを検知して $partial\_reset$ を行い, 次の世代へ移行する. さらに $P_j.Work\_count(t) > 5n$ より, 時刻$\tilde{t_j}, t_j$が必ず存在する. 式 (2.5) が成り立たないならば, 式 (2.4) より $P_j.Work\_count(t'_i - 1) - P_i.Work\_count(t'_i - 1) \leq 0$ が成り立つ. 時刻 t'_iは区間 $[t - 3n, t - 1]$ にあり, かつ $work(P_i, t) > 4n$, $work(P_j, t) > 4n$ より, 時刻 t_jが t_i以降に存在する. また, 式 (2.5) が成り立たないならば $steps(P_j, \tilde{t'_i}, t'_i) \geq steps(P_i, \tilde{t'_i}, t'_i) = n$ が成り立つので, $P_j.Work\_count(\tilde{t_j}) = 5n$ なる時刻$\tilde{t_j}$ も存在する.

次に, P_jが t 以前で最後に手続き $sort\_list$ を行った時刻 t_j に対し $P_i \longrightarrow^j_{t_j} P_j$ が成り立つことを示す.

(a) $t_j \geq t - 2n$ の場合

このとき, $t-3n < t_j-n+1 < t_j \leq t$が成り立つ. P_i, P_jはともに区間$[t_j-n+1, t_j]$で正常に動作し続け, この区間のある時刻 t'_jで P_jは key_iを設定するステップを行う. ここで, $(t_j, j) \prec (t_i, i)$ と仮定する. P_i, P_jはともに区間 $[t_i - n + 1, t_i]$ で正常に動作し続け, この区間のある時刻 t'_iで P_iは key_jを設定するステップを行う. このとき, 補題 21より世代 $P_i.Gen(t)$ で時刻 t 以前にプロセス P_iが P_jの居眠りを検知することはないので, (第 121 行の条件式が不成立) $((P_j.Work\_count(t'_i - 1) - P_i.Work\_count(t'_i - 1)), j) > (0, i)$, すなわち $P_j \longrightarrow^i_{t_i} P_i$が成り立ち, 仮定に反する. 従って, $(t_i, i) \prec (t_j, j)$ が成り立つ. 補題 21より世代 $P_j.Gen(t)$ で時刻 t 以前にプロセス P_jが P_iの居眠りを検知することはないので, 補題 15より $((P_i.Work\_count(t'_j - 1) - P_j.Work\_count(t'_j - 1)), i) > (0, j)$ が成り立つ. 従って, $P_i \longrightarrow^j_{t_j} P_j$が成り立つ.

(b) $t_j < t - 2n$ の場合

アルゴリズムより区間 $[\tilde{t_j}+1, t_j]$ にプロセス P_jが R_iを読み出すステップが存在する. この時刻を t'_jとする. 以下, $P_j \longrightarrow^j_{t_j} P_i$が成り立つと仮定して矛盾を導く. $P_i \longrightarrow^i_{t_i} P_j$かつ $P_j \longrightarrow^j_{t_j} P_i$ が成り立つので, $((P_j.Work\_count\ (t'_i - 1) - P_i.Work\_count(t'_i - 1)), j) > (0, i)$ かつ $((P_i.Work\_count(t'_j - 1) - P_j.Work\_count\ (t'_j - 1)), i) > (0, j)$ であり,

$$steps(P_i, t'_i, t'_j) < steps(P_j, t'_i, t'_j) \tag{2.6}$$

が成り立つ.

ここで, $steps(P_i, t_i, t+1) > 2n$ と $P_i.Work\_count\ (t_i) = 5n$ が成り立つことより, $P_i.Work\_count(t) > 7n$ である. よって, 事実 13より区間 $[t-n+1, t]$ のある時刻 t''_iで P_iは R_j を読み出す. 同様にして, 区間 $[t-n+1, t]$ のある時刻 t''_jで P_jは R_iを読み出す. P_i, P_jはそれぞれ区間 $[t_i, t], [t_j, t]$ で $partial_reset$を行わないので,

$$\begin{aligned} steps(P_i, t'_i, t''_i) &\geq steps(P_j, t'_i, t''_i) \\ steps(P_j, t'_j, t''_j) &\geq steps(P_i, t'_j, t''_j) \end{aligned}$$

が成り立つ. ここで, $work(P_i, t) \geq 4n, work(P_j, t) \geq 4n$ なので $steps(P_i, t''_i, t''_j) = steps(P_j, t''_i, t''_j)$ である. よって,

$$\begin{aligned} steps(P_i, t'_i, t'_j) &= steps(P_i, t'_i, t''_i) + steps(P_i, t''_i, t''_j) - steps(P_i, t'_j, t''_j) \\ &\geq steps(P_j, t'_i, t''_i) + steps(P_j, t''_i, t''_j) - steps(P_j, t'_j, t''_j) \\ &= steps(P_j, t'_i, t'_j) \end{aligned}$$

となり, 式 (2.6) に矛盾する. ■

プロセス P_iが自分の局所時計をある時刻 $t_{i,j}$でプロセス P_jの局所時計に合わせた, または P_jの局所時計と一致することを確認したあと, $t_{i,j}$以降に P_i, P_jがともに $partial_reset$をすることなく動作し続けるような場合には, P_iが P_jの局所時計を棄却することはなく, 次の補題が成り立つ.

補題 23 任意の異なるプロセス P_i, P_jに対し, ある時刻 t で P_i, P_jがともに調整済モードであったとし, $work(P_i, t) > 4n, work(P_j, t) > 4n$ が成り立つとする. P_iに対しては, $P_i.Work\_count(t_i) = 5n$ なる時刻 $t_i (t_i < t)$ が存在したとする. また,

時刻 $t_{i,j}$で P_iは自分の局所時計を P_jの局所時計に合わせた, または P_jの局所時計と一致することを確認したとし, 区間 $[t_{i,j}, t]$ で P_i, P_jはともに $partial_reset$を行わないとする. このとき, 区間 $[t_{i,j}, t]$ で P_iは P_jの局所時計を棄却しない. ■

補題 24 任意の時刻を t, 任意の異なるプロセスを P_i, P_jとする. 時刻 t で P_i, P_jがともに調整済モードであったとする. このとき, $work(P_i, t) > 4n, work(P_j, t) > 4n$ が成り立つならば, $P_i.Clock(t) = P_j.Clock(t)$ が成り立つ.

(証明) 少なくとも一方のプロセス Pに対し, $P.Work\_count(\bar{t}) = 3n$ なる時刻$\bar{t}$ $(\bar{t} < t)$ が存在するかによって場合分けする.

(1) 少なくとも一方のプロセス Pに対し, $P.Work\_count(\bar{t}) = 3n$ なる時刻$\bar{t}$ $(\bar{t} < t)$ が存在する場合

一般性を失うことなく $P_i.Work\_count(\bar{t}) = 3n$ なる時刻$\bar{t}$ $(\bar{t} < t)$ が存在すると仮定する. このとき補題 22より, $P_j.Work\_count(t_j) = 5n$ なる時刻 t_j $(t_j < t)$ が存在して $P_i \longrightarrow^j_{t_j} P_j$が成り立つ, または, $P_j \longrightarrow^i_{t_i} P_i$が成り立つ. $P_i \longrightarrow^j_{t_j} P_j$ならば $P_i = P_y, P_j = P_x$, $P_j \longrightarrow^i_{t_i} P_i$ならば $P_i = P_x, P_j = P_y$とおく. このとき, 以下が成り立つ.

- $P_y \longrightarrow^x_{t_x} P_x$かつ $P_x.Work\_count(t_x) = 5n$ なる時刻 t_x $(t_x < t)$ が存在する. P_xは世代 $P_x.Gen(t)$ のある時刻 t'で自分の局所時計を P_yの局所時計の値と合わせるか, 一致していることを確認し, 区間 $[t', t]$ で P_yは $partial_reset$を行わない.

このとき, $P_x.Clock(t') = P_y.Clock(t' - 1) + 1$ が成り立つ. ここで, 補題 23より, 区間 $[t', t]$ で P_xが P_yの局所時計を棄却することはない. また, 区間 $[t', t]$ で P_x, P_yはともに $partial_reset$をすることはないので, それぞれステップ毎に局所時計の値を 1 ずつ増やす. 従って,

$$steps(P_x, t', t) = steps(P_y, t', t) \tag{2.7}$$

であることを示せばよい. ここで, $t' \geq t - 4n + 1$ ならば, 区間 $[t', t]$ では P_x, P_yはともに正常に動作し続けるので明らかに式 (2.7) は成り立つ. 以下では, $t' < t - 4n + 1$ の場合を考える.

事実 13より, 区間 $[t-4n+1, t-n]$ のある時刻 t''で P_xは R_yを読み出す. また, 区間 $[t', t]$ で P_yは調整済モードであり続けるので, $t''(\leq t-n)$ より後で P_yは R_xを読み出す. 区間 $[t', t]$ で P_x, P_yはともに $partial_reset$を行わないので, 区間 $[t', t'']$ の各ステップで P_xが P_xまたは P_yの居眠りを検知することはない. 従って, $steps(P_x, t', t'') = steps(P_y, t', t'')$ である. さらに, 区間 $[t'', t]$ では P_x, P_yはともに正常に動作し続けるので $steps(P_x, t'', t) = steps(P_y, t'', t)$ が成り立つ. 従って, 式 (2.7) は成り立つ.

(2) 任意の時刻$\bar{t}(\leq t)$ に対し, $P_i.Work\_count(\bar{t}) > 3n$ かつ $P_j.Work\_count(\bar{t}) > 3n$ が成り立つ場合

このとき, P_i, P_jはともに区間$[1, t]$で$partial_reset$を行わない. また, $work(P_i, t) \geq 4n$, $work\ (P_j, t) \geq 4n$ より $P_i.Work\_count(t-n) > 6n$, $P_j.Work\_count(t-n) > 6n$, すなわち P_i, P_jはともに時刻$t-n$で調整済モードにある. $P_i.Work\_count(t-1) \geq 7n$, $P_j.Work\_count(t-1) \geq 7n$, 事実 13より, P_iは区間 $[t-n+1, t]$ のある時刻 t'''で R_jの読み出しをし, P_jは区間 $[t-n+1, t]$ のある時刻で R_i の読み出しをしている. ここで, $[\text{inconsistency III}]_i(t)$ が成立しないことから, $P_i.Invalid_j(t''') = P_j.Gen(t'''-1)$ または $P_i.Clock(t'''-1) = P_j.Clock(t'''-1)$ が成り立つ. ここで, $P_i.Invalid_j\ (t''') = P_j.Gen(t'''-1)$ が成り立つ場合を考える. $steps(P_i, t-4n, t-2n) = 2n$, 事実 13より P_iは区間 $[t-4n, t-2n]$ で少なくとも１回は R_j の読み出しをする. また区間 $[t-4n, t]$ で P_jは居眠りをしない. よって, P_iが区間 $[t-4n, t-2n]$ では P_jの居眠りを検知せずに区間 $[t-2n, t]$ で新たに P_j の居眠りを検知することはなく, $P_i.Invalid_j = P_j.Gen(t'''-1)$ にセットしたのは時刻 $t-2n$ より前である. P_jは区間 $[t-n+1, t]$ のある時刻で R_iの読み出しをするので, これは区間 $[1, t]$ で $partial\_reset$ を行わないことに矛盾する. 従って, $P_i.Clock(t'''-1) = P_j.Clock(t'''-1)$が成り立つ. $steps(P_i, t''', t) = steps(P_j, t''', t)$ より, $P_i.Clock(t-1) = P_j.Clock(t-1)$ も成り立つ. ■

居眠りリセットに関して, 次の補題が成り立つ. 開始状況後に他のプロセスが十分なステップ数の動作をしていないときに不正確に居眠りの検知をする場合がある点でアルゴリズム WCSの補題 9と異なる.

補題 25 プロセス P_iが時刻 t で居眠りリセットを実行したならば, $work(P_i, t) < 9n$ が成り立つ.

(証明) P_iが時刻tでR_jを読み出すとする. ここで, 事実13より少なくとも$3n-1$ステップに1回はP_iはR_jの読み出しをする. よって, 時刻tより前にR_jを読み出すステップが存在しないならば, 明らかに$work(P_i,t) < 9n$が成り立つ. 以下ではtより前にR_jを読み出す場合を考える. tより前で最後にP_iがR_jを読み出すステップを行った時刻をt'とする. P_iは時刻tで居眠りリセットを実行するので, (a)[naps I]$_i$, (b)[naps II]$_i$, (c) 第135行の条件式のうちいずれかが成り立つ.

(a) [naps I]$_i$: $P_i.Work\_count(t-1) \geq n$ かつ $P_i.diff(t) < 0$

プロセスP_iは区間$[t'+1, t-1]$で居眠りをしている. $P_i.Work\_count \geq n$, 事実13より$steps(P_i,t',t) \leq 2n$であり, $work(P_i,t) < 2n$が成り立つ.

(b) [naps II]$_i$: $P_i.Work\_count(t-1) \geq n$ かつ $P_j.Invalid_i(t-1) = P_i.Gen(t-1)$

$work(P_i,t) \geq 8n$を仮定し, 矛盾を導く. 前提条件より$P_i.Work_count(t-1) \geq n$なので, 事実13より$steps(P_i,t',t) \leq 2n$である.

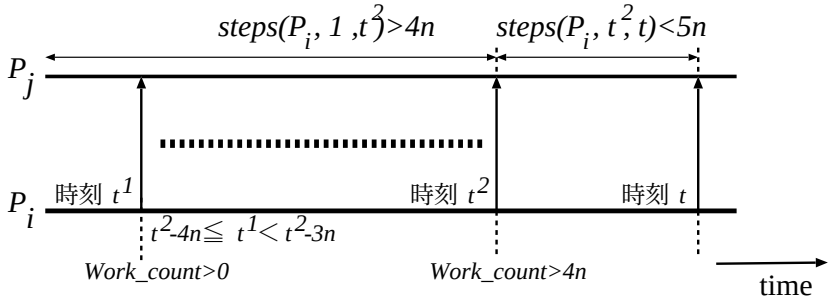
(b1) 時刻tより前でP_jが$P_j.Invalid_i$を更新していない場合

$P_i.Gen(t'-1) = P_j.Invalid_i(t'-1)$かつ$P_i.Work_count(t'-1) < n$が成り立つならば, 時刻$t'$以前のある時刻で$P_i$は$partial_reset$を行っている. 時刻$t'$で[naps II]$_i$が成り立つならば, 時刻$t'$で$P_i$は$partial_reset$を行っている. $P_i.Gen(t'-1) = P_j.Invalid_i(t'-1) = P_j.Invalid_i(t-1)$ならば, $P_i.Gen(t-1) = P_j.Invalid_i(t-1)$より区間$[t'+1, t-1]$のある時刻で$partial_reset$を行い$P_i.Gen$を更新する. しかし, いずれの場合も$partial_reset$を行った時刻$t_p$に対して$work(P_i,t_p) > 2n$が成り立つので, 補題16より$P_i$は世代番号を$P_j.Invalid_i(t-1)$と異なる値にする. 従って, $P_j.Invalid_i(t-1) = P_i.Gen(t-1)$に矛盾する.

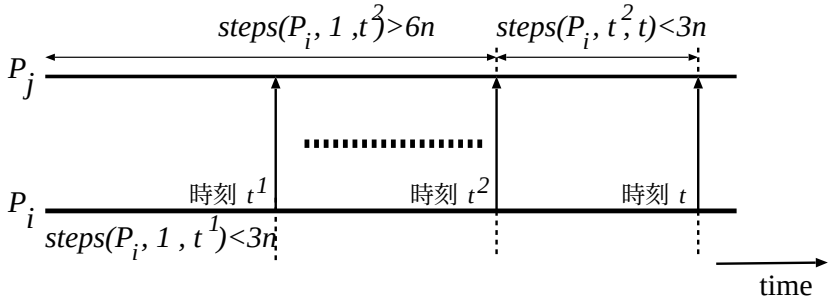
(b2) 時刻tより前でP_jが$P_j.Invalid_i$を更新している場合

時刻tより前で最後にP_jが$P_j.Invalid_i$を更新した時刻をt_jとする. 以下, さらにP_jが時刻t_jより前にR_iの読み出しをしているかにより場合分けする.

(b2-1) 時刻t_jより前にP_jがR_iの読み出しをしている場合



(i) $n < steps(P_i, t^2, t') < 3n$ が成り立つ場合



(ii) $steps(P_i, t^2, t') \leq n$ が成り立つ場合

図 2.18 補題 25：(b2-2) 時刻 t_j より前に P_jが R_iの読み出しをしていない場合 (1)

時刻 t_j より前で最後に P_jが R_iの読み出しをした時刻を t'_jとする. 時刻 t_jでプロセス P_jは P_iの居眠りを検知し, 世代番号を更新する. すなわち, P_iは区間 $[t'_j, t_j]$ で居眠りをしている. よって, 事実 13より, $steps(P_i, t'_j, t_j) < steps(P_j, t'_j, t_j) \leq 3n - 1$ が成り立つ. さらに, 区間 $[t_j + 1, t - 1]$ には P_iが R_jを読み出すステップが存在しないので, $P_i.Work\_count(t - 1) \geq n$, 事実 13より $steps(P_i, t_j + 1, t) < 2n$ が成り立つ. 従って, $steps(P_i, t'_j, t) < 5n$ であり, 区間 $[t'_j, t_j - 1]$ のある時刻で P_iは居眠りをしているので $work(P_i, t) \geq 9n$ に反する.

(b2-2) 時刻 t_j より前に P_jが R_iの読み出しをしていない場合

$steps(P_i, 1, t_j) < 7n$ ならば, $steps(P_i, t_j, t) \leq 2n$ より $steps(P_i, 1, t_j) < 9n$ が成り立つ. これは $work(P_i, t) \geq 9n$ に矛盾する. 以下では, $steps(P_i, 1, t_j) \geq 7n$ の場

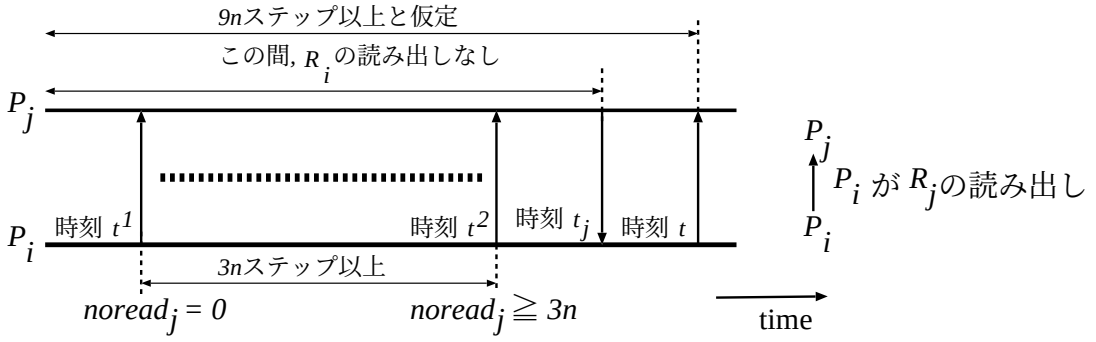


図 2.19 補題 25：(b2-2) 時刻 t_j より前に P_jが R_iの読み出しをしていない場合 (2)

合を考える.

$steps(P_i, t_j + 1, t) < 2n$ と事実 13より, 時刻 t_j以前に P_iが R_jの読み出しをする時刻が存在する. 時刻 t_j以前で最後に P_iが R_jを読み出した時刻を t'とし, 時刻 t'より前で最後に P_iが R_jを読み出した時刻を t^2とする. ここで, ある時刻 t^1が存在し, $steps(P_i, t^1, t^2) \geq 3n$ が成り立ち, かつ時刻 t^1で P_iが R_jを読み出すことを示す (図 2.18).

(i) $n < steps(P_i, t^2, t') < 3n$ が成り立つ場合

事実 13より $4n < P_i.Work\_count(t^2) < 6n$ が成り立つ. また, $steps(P_i, t^2, t) < 5n$ より $steps(P_i, 1, t^2) > 4n$ が成り立つ. よって, 事実 13より区間 $[t^2 - 4n, t^2 - 3n - 1]$ に P_iが R_jを読み出した時刻が存在する. この時刻を t^1とすると, $steps(P_i, t_1, t_2) \geq 3n$ が成り立つ.

(ii) $steps(P_i, t^2, t') \leq n$ が成り立つ場合

$steps(P_i, t^2, t) < 3n$ より $steps(P_i, 1, t^2) > 6n$ が成り立つ. P_iが開始状況から最初に R_jを読み出した時刻を t^1とすると, 事実 13より $steps(P_i, 1, t^1) < 3n$ が成り立つ. 従って, $steps(P_i, t_1, t_2) \geq 3n$ が成り立つ.

以上より, $steps(P_i, t^1, t^2) \geq 3n$ となるある時刻 t^1が存在し, t^1で P_iは R_jの読み出しをする. このとき, $[t^1, t^2]$ では P_jが R_iの読み出しを 1 回もしないことから, 図 2.19に示すように $P_i.noread_j(t^2) \geq 3n$ となり, $P_i.Stagnant_j$が $P_j.Last\_mycount_i$

(t_j-1) の値に更新される. すなわち, $P_j.Last\_mycount_i(t_j-1) = P_i.Stagnant_j(t_j-1)$ が成り立つ. 時刻 t_jで P_jは 87 行目で $diff = \perp$にセットして $P_i.Count$ と $P_j.Count$ それぞれの増分比較を行わないので, $P_j.Invalid_i$の更新をすることに矛盾する.

(c) 第 135 行:$P_i.Sync(t-1) \not\subset P_j.Async(t-1)$, $P_j.Sync \neq \emptyset$ かつ $P_i.Clock(t-1) \neq P_j.Clock\ (t-1)$

時刻 t で第 136 行を行うことより $P_i.Work\_count(t-1) < 6n$ であり, $P_i.Sync(t-1) \neq \emptyset$ より P_iは $5n \leq P_i.Work\_count(t_m-1) < P_i.Work\_count(t-1)$ である時刻 t_mで, 調整済モードのプロセス P_mの局所時計と自分の局所時計を合わせるか, 一致していることを確認している. このとき, $P_i.Clock(t_m) = P_m.Clock(t_m-1)+1$ が成り立つ. 以下, $work(P_i,t) \geq 9n$ を仮定して矛盾を導く.

時刻 t 以前で P_iが最後に $partial_reset$を行った時刻を t^0とする. すなわち, $Work\_count(t^0) = 0$ とする. $work(P_i,t) \geq 6n$ かつ $P_i.Work\_count(t-1) < 6n$ より, P_iは $[t^0,t]$ では居眠りすることなく正常に動作し続ける. 事実 13より, $[t^0+1, t^0+n]$ に P_iが R_j, R_mそれぞれを読み出すステップが存在する. また, P_iは, t_mより前で P_mの, t より前で P_jの居眠りを検知しない. 従って, $work(P_m,t_m-1) \geq t_m - t^0 - n > 4n$, $work(P_j,t-1) > work(P_j,t_m-1) > 4n$ が成り立つ. 以下, 時刻 t_m-1 での P_jのモードにより場合分けして考える.

(c1) 時刻 t_m-1 で P_jが調整済モードの場合

補題 24より $P_m.Clock(t_m-1) = P_j.Clock(t_m-1)$ が成り立つ. 区間 $[t_m, t-1]$ で P_i, P_jが居眠りすることはないので, $P_i.Clock(t-1) = P_j.Clock(t-1)$ が成り立ち, (c) に矛盾する.

(c2) 時刻 t_m-1 で P_jが調整中モードの場合

P_i, P_jはともに $t_m-3n(>t-4n)$ 以降の各プロセスの $Work\_count$ の値を基に $sort_list$を行う. プロセス P_i, P_jが t 以前で最後に $sort_list$を行った時刻をそれぞれ t_i, t_jとする. P_i, P_j, P_mは $[t_m-3n, t_m-1]$ で正常に動作し続けるので, $P_m \longrightarrow^i_{t_i} P_j$ と補題 15より $P_m \longrightarrow^j_{t_j} P_j$が成り立つ. 従って, P_jは世代 $P_j.Gen(t-1)$ 中のある

時刻 t'_m で P_m の局所時計に自分の局所時計を合わせるか, P_m と自分の局所時計が一致することを確認している. すなわち, $P_j.Clock(t'_m) = P_m.Clock(t'_m - 1) + 1$ が成り立つ. P_m は区間 $[t'_m, t_m - 1]$ または $[t_m, t'_m - 1]$ で正常に動作し続け, P_j は区間 $[t'_m, t - 1]$ で正常に動作し続ける. 従って, $steps(P_j, t'_m, t) = steps(P_m, t'_m, t_m) + steps(P_i, t_m, t)$ より $P_i.Clock(t-1) = P_j.Clock(t-1)$ が成り立ち, (c) に矛盾する. ■

また, 以下の補題より連続して $7n$ ステップした後で矛盾リセットが実行されることはないことが言える.

補題 26 プロセス P_i が時刻 t で矛盾リセットを実行したならば, $work(P_i, t) < 7n$ が成り立つ.

(証明) 補題 14より, 条件 [inconsistency I]$_i$, [inconsistency II]$_i$ の成立により矛盾リセットを実行したならば, $work(P_i, t) = 1$ が成り立つ. 以下, [inconsistency III]$_i$ の成立により矛盾リセットを実行する場合を考える.

[inconsistency III]$_i(t)$: $P_i.Mode(t-1) =$ "$adjusted$" **and** $P_{cur}.Mode(t-1) =$ "$adjusted$" **and** $P_i.Invalid_{cur}(t) \neq P_{cur}.Gen(t-1)$ **and** $P_i.Clock(t-1) \neq P_{cur}.Clock(t-1)$

プロセス P_i が時刻 t で読み出すレジスタを R_j とする. $work(P_i, t) \geq 7n$ を仮定したとき, [inconsistency III]$_i(t)$ が成立しないことを示す. 事実 13より, 区間 $[t - 7n + 1, t - 4n]$ で少なくとも 1 回は P_i は R_j の読み出しをしている. ここで, P_j が区間 $[t - 4n, t - 1]$ で居眠りをするならば, P_i は同区間で P_j の居眠りを必ず検知できるので, $P_i.Invalid_{cur}(t) = P_j.Gen(t-1)$ または $P_{cur}.Mode(t-1) =$ "$adjusting$"が成り立つ. また, P_j が区間 $[t - 4n, t - 1]$ で居眠りをしないならば, $work(P_j, t) > 4n$ が成り立つ. 補題 24より, $P_i.Mode(t-1) =$ "$adjusted$"かつ $P_j.Mode(t-1) =$ "$adjusted$"ならば, $P_i.Clock(t-1) \neq P_j.Clock(t-1)$ が成り立つ. 以上より, $work(P_i, t) \geq 7n$ ならば [inconsistency III]$_i(t)$ が成立しない. 従って, $work(P_i, t) < 7n$ が成り立つ. ■

補題 27 (調整完了性) 任意の $t \geq 1$, 任意のプロセス P_i に対し, $work(P_i, t-1) \geq 15n$ ならば $P_i.Mode(t-1) =$ "$adjusted$"が成り立ち, $work(P_i, t) > 15n$ ならば $P_i.Clock(t) = P_i.Clock(t-1) + 1$ が成り立つ.

(証明) 時刻$t-15n$以前にP_iが居眠りしているなら, 時刻$t-15n$以前でP_iが最後に居眠りした時刻をt_{nap}とする. 時刻$t-15n$以前にP_iが居眠りしていないなら, $t_{nap}=1$とする. 時刻t_{nap}以降の時刻t'でP_iが$partial_reset$を行った場合, 補題20,25, 26より$work(P_i,t')<9n$が成り立つ. よって, $work(P_i,t)\geq 15n$ならば, P_iは時刻$t-6n$以降に$partial_reset$を行うことはない. よって, $P_i.Work_count(t-1)\geq 6n$なので$P_i.Mode(t-1)=$"$adjusted$"が成り立つ. また, $work(P_i,t)>15n$ならば, P_iは時刻tで調整済のステップを行うので, $P_i.Clock(t)=P_i.Clock(t-1)+1$が成り立つ. ■

補題 28 (一致性) 任意の$t\geq 1$, 任意のプロセスP_i,P_jに対し, $work(P_i,t)\geq 15n, work(P_j,t)\geq 15n$ならば$P_i.Clock(t)=P_j.Clock(t)$が成り立つ.

(証明) 補題27より, $P_i.Mode(t)=$"$adjusted$", $P_j.Mode=$"$adjusted$"である. よって, 補題24より$P_i.Clock(t)=P_j.Clock(t)$が成り立つ. ■

補題27, 28より, 定理29が言える.

定理 29 プロトコル$ssWCS$は, 空間複雑度が有界な同期時間$15n$の自己安定無待機時計合わせプロトコルである. ■

2.6. むすび

本章では, フェーズ内システムとよばれる同期式共有メモリシステムにおける時計合わせアルゴリズムを考察した. まず, フェーズ内システム上の無待機時計合わせアルゴリズムの同期時間の下界が$n-2$であることを示した. さらに, 同期時間$12n$の無待機時計合わせアルゴリズム, 空間複雑度が有界な同期時間$15n$の自己安定無待機時計合わせアルゴリズムを提案した. 過去に提案された最も効率的な無待機時計合わせアルゴリズムはPapatriantafilouらが提案した同期時間$4n^2-3n-1$の自己安定無待機時計合わせアルゴリズムがあり, 本章で提案した２つのアルゴリズムは同期時間に関して大きく改善している. また, 提案した２つのアルゴリズムは同期時間の関してオーダー的に最適である.

同期時間$O(n^2)$のDolevら, Papatriantafilouらのプロトコルでは１ステップの計算量が定数である. これに対し, プロトコルWCS, $ssWCS$は手続き$sort_list$

を行うステップでプロセスが要素数 n のソートを行うため, 1 ステップで計算量 $O(n \log n)$ の内部計算が可能なシステムにしか適用できない. しかし, プロトコル WCS, $ssWCS$をヒープを使用するように修正すれば, 同じ同期時間で 1 ステップを計算量 $O(\log n)$ で行うプロトコルを構成できる.

本章に関連する今後の課題として, 次の問題が挙げられる. 本章で提案した 2 つの無待機時計合わせアルゴリズムは同期時間に関してオーダー的に最適であるが, 本章で示した下界の結果とは係数のギャップがある.

第 3 章

線形化可能性を保証する共有オブジェクトの無待機な実現

3.1. はじめに

本章では, 同期式メッセージパッシングシステム上に線形化可能性を保証する共有オブジェクトを無待機に実現するアルゴリズムについて述べる. 共有オブジェクト実現アルゴリズムの評価尺度として, その共有オブジェクトが提供する操作または操作集合に対する最悪応答時間を用いる.

本論文では, システム内の各プロセスが実時間スピードの局所時計を保持し, すべてのメッセージ遅延が各プロセスに既知である定数 $d, u(o < u \leq d)$ に対し $[d-u, d]$ であるようなメッセージパッシングシステムを考える. このようなシステムにおいて, 無待機性を考慮しない線形化可能性を保証する共有オブジェクトの実現アルゴリズムに関するさまざまな結果が示されている. 過去の線形化可能性を保証する実現アルゴリズムを表 3.1に示す. Attiya ら [3], Mavronicolas ら [16] は非同期時計モデルにおける read/write レジスタの実現アルゴリズムの最悪応答時間の下界を示した. Attiya らは write 操作の下界が $u/2$ であること, Mavronicolas らは read 操作の下界が $u/2$, read 操作と write 操作の最悪応答時間の和の下界が $d + u/2$ であることを示した. Mavronicolas らは u-同期時計モデルでの read/write レジスタの実現アルゴリズムも提案した. また, 井上らは非同期

時計モデルにおける一般オブジェクトの実現アルゴリズムを提案した [11]. 井上らの実現アルゴリズムでは, 一般オブジェクトの操作を応答値の種類が単一な ack 型操作と応答値の種類が複数の val 型操作に分類し, それぞれの最悪応答時間が u, $2d$ の実現アルゴリズムを提案した. また, 井上らは FIFO キューの実現アルゴリズムに関して, 最悪応答時間の下界が $u \leq (2/3)d$ であるときの dequeue 操作に対して $d + u/2$, enqueue 操作に対して $u(n-1)/n$ であることを示した [11].

本論文では, システム内のプロセスは停止故障をおこしうると仮定し, 任意個のプロセスの停止故障に対する耐性を意味する無待機性を持つ実現アルゴリズムを提案する. まず, さまざまな放送モデル, 時計モデルに対する 4 種類の read/write レジスタの無待機な実現アルゴリズムを提案する. 信頼放送モデルに対し, 非同期時計モデルにおける write 操作の応答時間 d, read 操作の応答時間 u の実現アルゴリズム, u-同期時計モデルにおける write 操作の応答時間 $u + \alpha \cdot \max\{d - 2u, 0\}$, read 操作の応答時間 $u + (1-\alpha)\max\{d - 2u, 0\}$(ただし, $0 \leq \alpha \leq 1$) の実現アルゴリズム, 無信頼放送モデルに対し, 非同期時計モデルにおける write 操作 d, read 操作 u の実現アルゴリズム, u-同期時計モデルにおける write 操作 u, read 操作 d の実現アルゴリズムを提案する. また, 信頼放送モデルにおいて, 2 種類の時計モデルに対する一般オブジェクトの実現アルゴリズムを提案する. 非同期時計モデルにおける write 操作 u, read 操作 $2d$ の実現アルゴリズム, u-同期時計モデルにおける write 操作 u, read 操作 $d + u$ の実現アルゴリズムを提案する.

以下, 3.2節で本章で提案するアルゴリズムに関する定義を行う. 3.3節では, 信頼放送／無信頼放送, 非同期時計／ u-同期時計モデル上に read/write レジスタを無待機に実現するアルゴリズムを提案する. 3.5節では, 信頼放送, 非同期時計／ u-同期時計モデル上に一般オブジェクトを無待機に実現するアルゴリズムを提案する.

[a] 文献 [16] では, Mavronicolas らは任意の小さい定数 b に対し write 操作の応答時間 $4u + \alpha(d-u)$ read 操作の応答時間 $4u + (1-\alpha)(d-u) + b$ の実現アルゴリズムを示している. ここで, 定数 b は放送を行うための時間を表す. 本章では放送時間を無視し, $b = 0$ として考える.

表 3.1 過去の線形化可能性を保証する実現アルゴリズム

read/write レジスタ

	時計モデル		最悪応答時間 write 操作	最悪応答時間 read 操作
上界	u-同期時計	[16]	$4u + \alpha \cdot m$ $(0 < \alpha \leq 1, m = d - u)$	$4u + (1 - \alpha)m$ [a]
下界	非同期時計	[3]	$u/2$	
		[16]	和が $d + u/2$	
				$u/2$

FIFO キュー

			dequeue 操作	enqueue 操作
上界	非同期時計	[11]	$d + u/2$ if $u \leq (2/3)d$	$u(n-1)/n$

一般オブジェクト

			ack 型操作	val 型操作
上界	非同期時計	[11]	u	$2d$

メッセージ遅延 : $[d - u, d]$, プロセス数 : n

3.2. 諸定義

本節でシステムのモデル, 共有オブジェクトの無待機な実現アルゴリズムを定義する.

3.2.1 システム

複数のプロセスと相互通信網から成るメッセージパッシングシステムを考える. 各プロセスは 0 から $n-1$ までの相異なる識別子を持つとし, 識別子 i のプロセスを P_iと表す. プロセスは相互通信網を介してメッセージを送信, 受信することによってのみ通信を行うことができる. 相互通信網内では, メッセージの欠落, 複

表 3.2 本論文で提案する線形化可能性を保証する無待機な実現アルゴリズム

read/write レジスタ

放送モデル	時計モデル	最悪応答時間 write 操作	最悪応答時間 read 操作
信頼放送	非同期時計	d	u
	u-同期時計	$u+\alpha\cdot m'$ $(0\le\alpha\le 1, m'=\max\{d-2u,0\})$	$u+(1-\alpha)m'$ $(0\le\alpha\le 1, m'=\max\{d-2u,0\})$
無信頼放送	非同期時計	d	d
	u-同期時計	u	d

一般オブジェクト

		ack 型操作	val 型操作
信頼放送	非同期時計	u	$2d$
	u-同期時計	u	$d+u$

メッセージ遅延 : $[d-u, d]$

製はおこらないとする. 任意のプロセス間でのメッセージ遅延は, 各プロセスに既知である定数 $d, u(0 < u \le d)$ に対し $[d-u, d]$ の範囲であるとする. 各プロセスは, 大域時間と同じ速度の局所時計を保持する[a]. プロセスは, 局所時計により局所的な時刻を参照し, また, タイマーとして使う. ある定数ϵに対し, 任意の 2 プロセスの局所時計の値の差はϵとする. このようなモデルをϵ-同期時計モデルとよぶ. また, システム内のプロセスは停止故障をおこすとする. 停止故障をおこしたプロセスは, 以降の動作を一切行わない.

すべてのプロセスに対し同一のメッセージを送信することを放送とよぶ. 本論文では, 放送に関して以下の 2 種類のモデルを考える.

信頼放送モデル 放送するメッセージは, すべてのプロセスに受信される, または, すべてのプロセスが受信しないことを保証する. このような放送を**信頼性**

[a] 本章ではシステムの振舞を大域時刻を使って説明する. ただし, 各プロセスは大域時計は参照できず, 局所時計のみを参照できる.

があるという. 本モデルにおける放送は, すべてのプロセスへメッセージを送信する原子イベントによりモデル化する.

無信頼放送モデル 放送中にプロセスが故障したならば, 放送するメッセージが各プロセスに受信されるか否かについて, 何も保証しない. すなわち, 放送メッセージを一部のプロセスのみが受信する場合がある. 本モデルにおける放送では, 1プロセスへメッセージを送信するイベントをすべてのプロセスに対して連続して生起することによりモデル化する.

プロセス P_iは状態機械によりモデル化される. プロセス P_iの状態は, イベントが P_iでおこったときに変化する. システムの大域状況を, 全プロセスの状態, 通信中のメッセージの集合 $\mathcal{N}$, プロセス P_iがセット中のアラームの集合 $\mathcal{A}_i$により定義する. 通信中のメッセージは3項組 (M, s, r) により定義する. ここで, Mはメッセージ, s は送信者, rは送信先を表す. アラームは2項組 (K, t) により定義する. ここで, K はアラームの型, t はアラームの生起する局所時刻を表す. 以降, このシステムの大域状況のことを, 単に状況という. 各プロセス P_iは以下のイベントを持つ.

通信イベント: プロセスがメッセージを送受信するイベント. 信頼放送モデルでは, 放送イベント $BroadCast(i, M)$ と受信イベント $Receive(i, j, M)$ のみ生起する. 無信頼放送モデルでは, 送信イベント $Send(i, j, M)$ と受信イベント $Receive(i, j, M)$ のみ生起する.

送信イベント $Send(i, j, M)$**:** プロセス P_iがプロセス P_jへメッセージ M を送信する. 3項組 (M, i, j) が $\mathcal{N}$へ追加される.

放送イベント $BroadCast(i, M)$**:** プロセス P_iがメッセージ M を放送, すなわち, 全プロセスへ Mを送信する[b]. すべての P_jに対し, 3項組 (M, i, j) が $\mathcal{N}$へ追加される.

受信イベント $Receive(i, j, M)$**:** プロセス P_iがプロセス P_j からメッセージ Mを受信する. 3項組 (M, j, i) が $\mathcal{N}$から削除される.

[b]本章では便宜上, 放送により自分を含めたすべてのプロセスに同一のメッセージを送信するものとする

時間イベント: 局所時計に関するイベント.

タイマセット イベント $TimerSet(i, \bar{t}, K)$**:** プロセス P_i が型 K のアラームが $\bar{t}$ 後に生起するようにセットする. イベント $TimerSet(i, \bar{t}, K)$ が局所時刻 t で生起するとき, 2 項組 $(K, t + \bar{t})$ が $\mathcal{A}_i$ へ追加される.

アラームイベント $Alarm(i, K)$**:** 型 K のアラームがプロセス P_i でおこる. イベント $Alarm(i, K)$ が局所時刻 t で生起するとき, 2 項組 (K, t) が $\mathcal{A}_i$ から削除される.

時計読み出しイベント $ReadClock(i, s)$**:** プロセス P_i が局所時計から値 s を得る.

故障イベント $Stop(i)$**:** プロセス P_i が故障する. このイベントにおいてプロセス P_i は**故障状態**に遷移し, 以降の動作は一切行わない.

その他のイベント: プロセス P_i はシステムの外部 (以降, 環境とよぶ) とも通信を行う. プロセスと環境の間の通信に関するイベントは後で述べる.

受信イベント $Receive(i, j, M)$, アラームイベント $Alarm(i, K)$, 故障イベント $Stop(i)$ はプロセスの制御外で生起する**入力イベント**である.

システムの実行は状況とイベント生起の無限または有限交替列 $E = c_0, (e_1, t_1), c_1, \cdots, c_k, (e_{k+1}, t_{k+1}), c_{k+1}, \cdots,$ で表す. ここで, 各 $c_k (k \geq 0)$ は状況, $(e_k, t_k) (k \geq 1)$ はイベントの生起を表す. また, 各 (e_k, t_k) に対し, e_k はイベント, t_k は e_k がおこった大域時刻を表す. 各 t_k を $time(e_k)$ により表す. 状況 c_k におけるプロセス P_i の状態は, c_k の P_i に対する射影により定義され, $c_k|i$ で表す. c_0 は各プロセスが特定の初期状態にあり, $\mathcal{N}$ とすべての i に対する $\mathcal{A}_i$ が空な初期システム状況を表す. 各 k に対し, $t_k \leq t_{k+1}$ が成り立つ. 実行 E は, 各 $k (k \geq 0)$ に対し, 時刻 t_{k+1} にあるプロセス P_i でイベント e_{k+1} が生起し, P_i の状態が $c_k|i$ から $c_{k+1}|i$ へ遷移した ($\mathcal{N}$, $\mathcal{A}_i$ が遷移することもある) ことを意味する. 簡単化のため, 実行内のすべてのイベントは区別できるとする. 任意の実行に対し, 以下の条件を仮定する.

- $\mathcal{A}_i$ が 2 項組 (K, t) を含むなら, 時刻 t で $Alarm(P_i, K)$ が生起する, または, 時刻 t で P_i は故障状態にある. 逆に, (K, t) が $\mathcal{A}_i$ に含まれる場合のみ, $Alarm(P_i, K)$ は生起する.

$$OP = \{write(v) | v \in D\} \cup \{read\}$$
$$RES = \{\text{ack}\} \cup D$$
$$Q = D$$
$$\forall v, v' \quad \delta(v, write(v')) = (v', \text{ack})$$
$$\forall v \quad \delta(v, read) = (v, v)$$

図 3.1 定義域 Dの read/write レジスタの型

- 大域時刻Tに 3 項組 (M, i, j) が $\mathcal{N}$へ追加されたならば, 区間 $[T+d-u, T+d]$ で $Receive(j, i, M)$ が生起する, または, 大域時刻 $T+d$ で P_jは故障状態にある. 逆に, (M, i, j) が $\mathcal{N}$に含まれる場合のみ, $Receive(j, i, M)$ は生起する.

3.2.2 共有オブジェクトの無待機な実現

決定性共有オブジェクト (以降, **オブジェクト**とよぶ) を定義する. オブジェクトは複数のアプリケーションプロセスが並行にアクセスできるデータ構造であり, 相異なる**名前**と**型**により定義される. 型は $(OP, RES, Q, q_0, \delta)$ で表記される. ここで, OPは操作の集合，RESは応答値の集合，Q は状態の集合である．また，q_0 は初期状態である．$\delta : Q \times OP \to Q \times RES$は**逐次仕様**と呼ばれる関数である．逐次仕様は操作が逐次的に適用されたときの共有オブジェクトの振舞を定義している. 操作が一つずつ適用されている場合，もし操作 op が状態 qに適用され，状態 q'に変化し，応答 res を返すならば，$\delta(q, op) = (q', res)$ と定義される．このようなオブジェクトを, 逐次仕様が関数なので, 決定性オブジェクトとよぶ. ある定義域 D上の値を扱い，初期値を q_0とする read/write レジスタの型を図 3.1に示す. また, もしオブジェクトのある操作 op が常に同じ値を返すなら, すなわち, $|\{res | \exists s, s'[\delta(s, op) = (s', res)]\}| = 1$ ならば, 操作 op を ack 型操作とよぶ. ack 型以外の操作を val 型操作とよぶ. 以下, op_aは任意の ack 型操作, op_vは任意の val 型操作を表す.

次に, 型 $(OP, RES, Q, q_0, \delta)$ のオブジェクト Oの**実現**を定義する. 本論文では, 環境に並行してアクセスされる仮想的なオブジェクトを実現する. オブジェクトを実現するシステムを図 3.2に示す. オブジェクトはプロセスの集合 $\{P_0, ..., P_{n-1}\}$

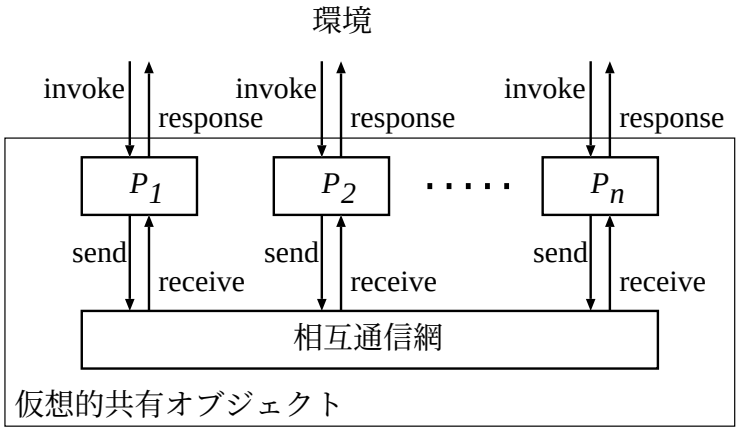


図 3.2 共有オブジェクトの実現

により実現される. 環境は, あるプロセス P_iと通信することにより, オブジェクトにアクセスする. 環境とプロセス P_i間の通信は以下のイベントにより定義される.

呼出イベント $Invoke(P_i, op)$**:** 環境がオブジェクト Oに操作 $op(\in O)$ を適用するためにプロセス P_iへ呼び出す.

応答イベント $Response(P_i, res)$**:** 環境の呼び出しに対し, プロセス P_iが応答 res $(\in RES)$ を返す.

呼出イベントは入力イベントである. 環境とプロセス P_i間の通信に関して, 任意の実行に対し, 以下の条件を仮定する.

- 環境がプロセス P_iへ操作を呼び出したならば, その呼び出しに対する応答が返るまで環境は次の操作を P_iへ呼び出さない.

無矛盾な応答を返すため, プロセスは互いにメッセージ交換を行う.

各プロセス P_iに対し, 実行 Eからあるプロセス P_iの呼出イベントと応答イベントを取り出した系列を考える. 実現されたオブジェクトでは, ある操作がプロセス P_iに呼び出されたならば, かつそのときに限り, P_iは応答を返す. よって, 得られた系列は交替列 $Inv_1, Res_1, Inv_2, Res_2, \cdots$となる. ただし, 各 $k(k \geq 1)$ に対し Inv_kは呼出イベント, Res_kは応答イベントを表す. 各呼出イベント Inv_kに対

し, 次のイベント Res_k を**対応する**イベントとよぶ. また, 対応するイベントの組 (Inv_k, Res_k) を**操作実行**とよぶ. ある操作の呼出イベントが対応するイベントを持たないとき, その操作または呼出イベントが**ペンディング**しているという. 呼出イベントに対応するイベントがあるなら, その操作は**完了**したという.

実現されたオブジェクトは, 並行するアクセスに対してコンシステンシを保証しなければならない. 本論文では, オブジェクトの実現のコンシステンシとして, **線形化可能性** (*linearizability*)[10] を採用する. Herlihy らは文献 [10] において，線形化可能性の局所性 *locality* を示した．局所性とはシステム内の個々の共有オブジェクトの実現が線形化可能であるとき，かつそのときのみ，そのシステム全体も線形化可能であるという性質である．よって，本論文では一つの線形化可能な共有オブジェクトの実現を考え，１つのオブジェクトの実現の定義をする．局所性より, 今回提案する１つのオブジェクトの実現を組み合わせることにより, 複数のオブジェクトからなるシステムを実現できる.

次に, 線形化可能性, 無待機性を定義する. ある操作実行列$\tau = (Inv_1, Res_1), (Inv_2, Res_2), \cdots$を考える. 各操作実行 (Inv_k, Res_k) に対し, その操作を op_kとする. 型 $(OP, RES, Q, q_0, \delta)$ のオブジェクト Oに対し, $\delta(q_{k-1}, op_k) = (q_k, res_k)$ なる Oの状態列$\theta = q_0, q_1, \cdots$が存在するならば, τは**正当**であるという. 実行 Eにおいて, ２つの操作実行 $op_k = (Inv_k, Res_k)$, $op_l = (Inv_l, Res_l)$ に対して $time(Res_k) \leq time(Inv_l)$ が成り立つとき, または, 同一プロセス上の操作実行が op_k, op_lの順で生起するとき, op_k は op_lに先行するといい, $op_k \xrightarrow{E} op_l$で表す. 実行 Eを完了した操作の呼出イベント, 応答イベントに制限した系列を $complete(E)$ で表す.

定義 3 実行 Eから以下を満たす実行 E'が得られるなら, その実行 Eを**線形化可能**であるという.

- 実行 E'は, 実行 Eに等しい, または, 実行 Eにペンディングしているいくつかの呼出イベントに対応する応答イベントを追加することにより得られる.
- $complete(E')$ 内のすべての操作実行から成るようなある正当な系列τが存在し, $op_1 \xrightarrow{complete(E')} op_2$を満たす任意の２つの操作実行に対し, 系列$\tau$で op_1は op_2より前に現れる. ■

定義 4 ある実現 Iに対し, 任意の実行が線形化可能ならば, その実現 Iは線形化可能であるという. ■

定義 5 実行 Eに現れる任意の呼出イベント Invが以下の条件の一方を満たすとき, 実行 Eは**無待機**であるという.

- 対応する応答イベントが存在する.
- Invが生起したプロセス P_iに対し, Invの後で停止イベント $Stop(i)$ が生起する.

また, 実現 Iに対し, 任意の実行が無待機ならば, その実現 Iは無待機であるという. ■

実現 Iの評価尺度として, 操作または操作集合の最悪応答時間を用いる. 操作実行 $ope = (Inv, Res)$ に対し, $time(Res) - time(Inv)$ を**応答時間**と定義し, $ope.op$ をイベント Invで呼び出された操作とする. $OPE(E)$ を実行 Eに現れる操作実行の集合とする. 型 $(OP, RES, Q, q_0, \delta)$ のオブジェクト Oの実現 Iに対し, 操作または操作集合 op の最悪応答時間を

$$\max\{res\_time(ope) | ope \in OPE(E), ope.op \in op, E \text{ is an execution of } I\}$$

により定義し, $res\_time(op)$ により表す.

3.3. 信頼放送による read/write レジスタの無待機実現

本節では, 信頼放送モデルにおける 2 種類の read/write レジスタの実現アルゴリズムとして, 非同期時計モデルのアルゴリズム $register_{RB-AC}$ と u-同期時計モデルのアルゴリズム $register_{RB-uC}$を提案する. read/write レジスタの実現アルゴリズムの効率は, read 操作の最悪応答時間 $res\_time(read)$ とすべての write 操作から成る集合の最悪応答時間 $res\_time(write) = \max\{res\_time(write(v))\}$ により評価する.

まず, 本論分で提案する無待機実現アルゴリズムの基本方針を示す. すべての read/write レジスタの無待機実現アルゴリズムにおいて, 各プロセスは read/write レジスタの局所コピーを持つ. 操作 $write(v)$ が呼び出されたとき, プロセスはその write 操作にタイムスタンプ ts を割り当て, v, ts を含む**更新メッセージ**を放送する. 更新メッセージを受信したプロセスは, その内容に従って局所コピーを更新する. read 操作が呼び出されたとき, 操作中のある決められた時間後の局所コピーの値を応答値として返す. 以降, read 操作実行 Rに対し, Rが応答値として返す値に局所コピーを更新するときに基になった更新メッセージを放送した write 操作実行を $Write(R)$ と記す. ある read 操作実行 Rに対する $Write(R)$ であるような write 操作を**有効**な write 操作とよぶ.

実現アルゴリズムは, 入力イベントに対するイベント駆動型で示す. 各イベントとそれに続く内部計算の連続が原子的に行われる. すなわち, プロセスはその連続中に故障することはない. 複数の入力イベントが同時刻に生起する場合, 故障イベントを除いてそれらのイベントはプログラムに現れる順序で処理する.

3.3.1 アルゴリズム $register_{RB-AC}$

まず, 非同期時計モデルの実現アルゴリズム $register_{RB-AC}$を説明する. アルゴリズム $register_{RB-AC}$ は $res\_time(write) = d$, $res\_time(read) = u$ で read/write レジスタを実現している. 実現アルゴリズム $register_{RB-AC}$のプログラムを図 3.3 に示す.

先にプロセスの故障が発生しないことを仮定して説明する. 故障に関しては後述する. write 操作では, プロセスはまず更新メッセージを放送する. 更新メッセージを受信したプロセスは, メッセージの内容に従って局所コピーを更新する. write 操作は, 呼び出しから d 時間後に ack を返して完了する. read 操作では, 呼び出し時の局所コピーの値を応答値として返す. 呼び出しから u 時間後に, その値を返して完了する.

アルゴリズム $register_{RB-AC}$では, タイムスタンプとして整数 $count$ を用いる. 各プロセスは局所変数として $count$ を持つ. 整数 $count$ は write 操作呼び出し時に 1 ずつ増やす. また, 受信した更新メッセージに含まれる $count$ が自分の $count$ より大きいならば, $count$ を更新メッセージに含まれる値に合わせる. 図 3.4に示

data type
timestamp=(integer, process identifier);

variables
$count$, **type** integer, **init** 0;
res_val, **type** value of the register ;
$last\_up\_ts$, **type** timestamp, **init** $(0, 0)$;
$local\_copy$, **type** value of the register, **init** initial value of the register;

transition functions of process P_i
$Invoke(i, write(v))$:
$count := count + 1$;
$BroadCast(i, \text{update}(v, (count, i)))$; /* update message */
$TimerSet(i, d, \text{WRITE})$;

$Invoke(i, read)$:
$res\_val := local\_copy$;
$TimerSet(i, u, \text{READ})$;

$Receive(i, j, \text{update}(v, (recvd\_count, recvd\_uid)))$:
$count := \max(count, recvd\_count)$;
if $last\_up\_ts <^a (recvd\_count, recvd\_uid)$
then $local\_copy := v$; $last\_up\_ts := (recvd\_count, recvd\_uid)$;

$Alarm(i, \text{WRITE})$:
$Response(i, \text{ack})$;

$Alarm(i, \text{READ})$:
$Response(i, res\_val)$;

$Stop(i)$:
No events can happen after this event.

---

[a] $(a_1, b_1) < (a_2, b_2) \Leftrightarrow a_1 < a_2 \vee (a_1 = a_2 \wedge b_1 < b_2)$.

図 3.3 実現アルゴリズム $register_{RB-AC}$ (P_iのプログラム)

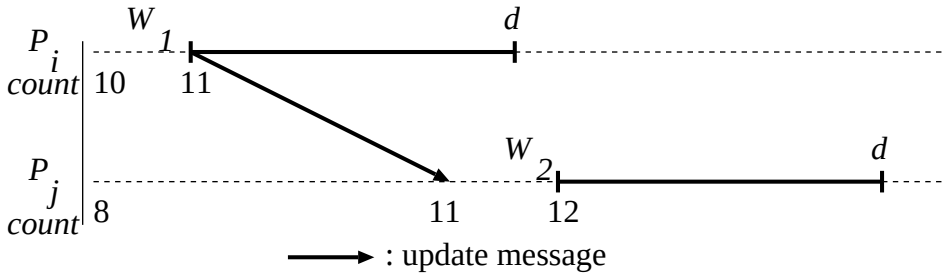


図 3.4 整数 $count$

すように, 任意のメッセージ遅延は高々d なので, $W_1 \xrightarrow{E} W_2$なる実行 Eの２つの write 操作実行 W_1, W_2に対し, W_2のタイムスタンプは W_1のタイムスタンプより大きくなる.

プロセスは, 自分の局所コピーを更新メッセージにより知らされた値に, 各更新メッセージのタイムスタンプの順に更新していく. ただし, タイムスタンプの値が等しい２つの更新メッセージ間では, それぞれの更新メッセージを送信したプロセスの識別子の大小により順序付けする. しかし, 更新メッセージはそのタイムスタンプの順に受信するとは限らない. ある更新メッセージに従って局所コピーを更新した後で, それより小さいタイムスタンプを持つ更新メッセージを受信することがある. このような場合, 小さいタイムスタンプを持つ更新メッセージは既に処理し, さらに上書きされたとみなす. このようにして, プロセスは小さい更新メッセージを無視する. 図 3.5のようにある時刻 t に放送された更新メッセージは必ず区間 $[d+t-u, d+t]$ で各プロセスに受信され, その更新メッセージに従った更新を行う, または無視する. 従って, $R_1 \xrightarrow{E} R_2$なる実行 Eの２つの read 操作実行 R_1, R_2に対し, 操作実行 $Write(R_2)$ は $Write(R_1)$ より大きいまたは等しいタイムスタンプを持つことが保証される.

次に，プロセスが故障する場合を考える．ある操作の呼び出し後，応答を返す前にプロセスが故障すると，その操作の呼び出しはペンディングしたままとなる．呼び出しがペンディングとなる操作のうち，他のプロセスに影響を及ぼす可能性があるのは，write 操作において放送イベントの生起後に故障が発生した場合のみである．しかし，ここでは放送に信頼性のあるシステムを仮定しているため，

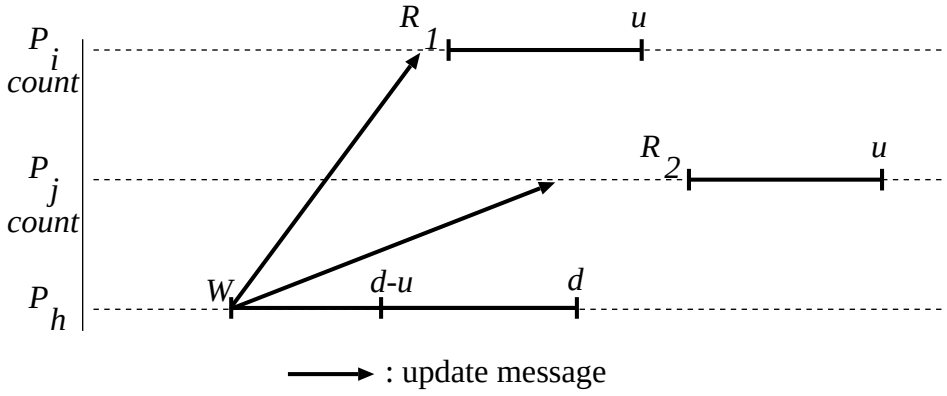


図 3.5 更新メッセージの受信

放送イベントが発生し，あるプロセスが更新メッセージを受信したならば, 他のすべてのプロセスもその更新メッセージを受信することが保証される．よって，すべてのプロセス間で矛盾なく処理を行うことができる．また，どのプロセスも更新メッセージを受信しないならば, どのプロセスもこの write 操作に関する処理は行われないため, この場合もすべてのプロセス間で矛盾は生じない．よって，プロセスが故障する場合でもアルゴリズム $register_{RB-AC}$は正しく動作する．

アルゴリズム $register_{RB-AC}$が線形化可能性を保証する無待機な read/write レジスタの実現であることを示す．実行 Eに対し, E内のペンディングしている有効な write 操作の呼出イベント e に対応する応答イベントを時刻 $time(e) + d$ に加えた実行を E'とする.

補題 30 実行 E'内の任意の完了した $write$ 操作実行 Wの呼出イベントを Invとし, Invで放送された更新メッセージを Mとする. プロセス P_iが Mに従って局所コピーの更新をするならば, 区間 $[time(Inv) + d - u, time(Inv) + d]$ で行う.

(証明) メッセージ遅延より, プロセス P_iは区間 $[time(Inv)+d-u, time(Inv)+d]$ でメッセージ Mを受信する. アルゴリズムより, プロセス P_iがメッセージ Mに従って局所コピーの更新をするならば, それをメッセージ M受信時に行う. 従って, 補題は成り立つ. ■

各 write 操作 Wに対し, $ts(W)$ を Wに割り当てられたタイムスタンプと Wが生起しているプロセスの識別子の組とする.

補題 31 実行 E'内の完了した *write* 操作実行 W_1, W_2に対し, $W_1 \stackrel{E'}{\rightarrow} W_2$ならば, $ts(W_1) < ts(W_2)$ が成り立つ.

(証明) write 操作実行 W_1, W_2が実行されたプロセスをそれぞれ P_1, P_2とする. 各プロセスは write 操作を呼び出された d 経過後に応答を返すので, write の応答時間は d である. W_1の更新メッセージの送信時刻を t_1とすると, W_1の更新メッセージの受信時刻は遅くとも $t_1 + d$ である. ここで, W_1は W_2に先行しているので, W_2の呼び出し時刻 t_2に対し $t_1 + d < t_2$が成り立ち, W_1の更新メッセージは W_2にタイムスタンプを割り当てる前に P_2が受信している. つまり, W_1の更新メッセージが受信されたとき, P_2の *count* は W_1に割り当てられたタイムスタンプの *count* より大きくなる. *count* は減少することはないため, その後に呼び出される W_2には W_1より大きいタイムスタンプが割り当てられる. よって補題が成り立つ. ■

補題 32 実行 E'内の完了した *read* 操作実行 R_1, R_2に対し, $R_1 \stackrel{E'}{\rightarrow} R_2$ならば, $ts(Write(R_1)) < ts(Write(R_2))$ または $Write(R_1) = Write(R_2)$ が成り立つ.

(証明) read 操作実行 R_1, R_2が実行されたプロセスをそれぞれ P_1, P_2とし, その呼び出し時刻をそれぞれ t_1, t_2とする. read 操作の応答時間は u であり, $R_1 \stackrel{E'}{\rightarrow} R_2$であるため, $t_2 > t_1 + u$が成り立つ. プロセス P_1は t_1以前に $Write(R_1)$ の更新メッセージを受信して局所コピーの更新を行っているので, $Write(R_1)$ は $t_1 - (d - u)$ 以前に呼び出されている. 以下, P_2が $Write(R_1)$ の更新メッセージを受信したときの処理に関して場合分けする.

(a) P_2が $Write(R_1)$ の更新メッセージに従って局所コピーの更新をする場合

補題 30より, P_2はこの更新を $t_1 + u(< t_2)$ 以前に行う. ここで, $Write(R_1) \neq Write(R_2)$ とすると, $Write(R_1)$ の更新メッセージ受信時に局所コピーの更新をしたあとで $Write(R_2)$ の更新メッセージに従って更新を行うので, $ts(Write(R_1)) < ts(Write(R_2))$ を意味する.

(b) P_2が $Write(R_1)$ の更新メッセージを無視する場合

プロセス P_2における $Write(R_1)$ の更新メッセージ受信直前の $last\_up\_ts$ の値を lts_1，$Write(R_2)$ の更新メッセージ受信直前の $last\_up\_ts$ の値を lts_2とする．P_2 は $Write(R_1)$ の更新メッセージを無視することより, $lts_1 > ts(Write(R_1))$ が成り立つ．また, P_2における時刻 t_2での局所コピーの値を書き込んだ write 操作実行は $Write(R_2)$ なので, $lts_2 \leq ts(Write(R_2))$ が成り立つ. $last\_up\_ts$ は小さくなることはないので, $ts(Write(R_1)) < lts_1 \leq lts_2 \leq ts(Write(R_2))$ が成り立つ． ■

定理 33 アルゴリズム $register_{RB-AC}$は, 信頼放送, 非同期時計モデルにおける $read/write$ レジスタの線形化可能かつ無待機な実現アルゴリズムであり, $res\_time(write) = d$, $res\_time(read) = u$ である.

(証明) アルゴリズム $register_{RB-AC}$が無待機であること, 最悪応答時間が $res\_time(write) = d$, $res\_time(read) = u$ であるのは明らかである．以下, アルゴリズム $register_{RB-AC}$が線形化可能であることを示す.

$complete(E')$ に含まれるすべての操作実行から成る正当な系列τ を構成し, 任意の操作実行 op_1, op_2に対して $op_1 \overset{complete(E')}{\rightarrow} op_2$ならば$\tau$で op_1が op_2より前に現れることを示す. 系列τは初期値を書き込む仮想的な write 操作 W_0から始まると仮定してτの構成法を説明する. まず, $complete(E')$ に現れるすべての write 操作実行をそのタイムスタンプ順 (タイムスタンプが同じときは操作が実行されるプロセスの識別子により順序付けする) に W_0の後ろに並べる. 次に, 各 read 操作実行 Rを, その操作が呼び出された大域時刻の順に, $Write(R)$ の次の write 操作の直前に挿入していく.

以下, 操作実行 op_1, op_2が write 操作か read 操作により場合わけする.

(a) op_1, op_2が write 操作実行 W_1, W_2のとき

補題 31より, $W_1 \overset{complete(E')}{\rightarrow} W_2$ならば $ts(W_1) < ts(W_2)$ が成り立つ. 構成法よりτで W_1は W_2より前に現れる.

(b) op_1, op_2が read 操作実行 R_1, R_2のとき

補題 32より, $R_1 \overset{complete(E')}{\rightarrow} R_2$ならば, $ts(Write(R_1)) < ts(Write(R_2))$ または $Write(R_1) = Write(R_2)$ が成り立つ. どちらの場合も, 構成法よりτで R_1は R_2より前に現れる.

(c) op_1が write 操作実行 W, op_2が read 操作実行 Rのとき

read 操作実行 Rが実行されたプロセスをそれぞれ P_Rとし, W, Rの呼び出し時刻をそれぞれ t_W, t_Rとする. $W \overset{complete(E')}{\rightarrow} R$より, $t_W + d < t_R$が成り立つ.

(c1) P_Rが Wの更新メッセージを無視する場合: P_Rは Wの更新メッセージを受信する前に, Wより大きいタイムスタンプを持つ更新メッセージに従って局所コピーを更新している.

(c2) P_Rが Wの更新メッセージに従って局所コピーを更新する場合: 補題 30より, その更新は時刻 $t_W + d$ 以前に行われる. W_1に対する更新のあと, P_Rがさらに局所コピーの更新を行うならば, W_1より大きいタイムスタンプを持つ更新メッセージに従っている.

read 操作実行 Rは時刻 $t_R (> t_W + d)$ における P_Rの局所コピーの値を返すので, $W = Write(R)$ または $ts(W) < ts(Write(R))$ が成り立つ. 従って, 構成法よりτで Wは Rより前に現れる.

(d) op_1が read 操作実行 R, op_2が write 操作実行 Wのとき

$R, W, Write(R)$ の呼び出し時刻をそれぞれ $t_R, t_W, t_{W(R)}$とする. 補題 30より, $t_{W(R)} < t_R - (d-u)$が成り立つ. また, $t_R < t_W - u$ より, $t_{W(R)} < t_W - d$, すなわち, $Write(R) \overset{complete(E')}{\rightarrow} W$が成り立つ. 従って, 補題 31より $ts(Write(R)) < ts(W)$ が成り立ち, 構成法よりτで Rは $Write(R)$ と Wの間に現れる.

最後に, τの構成法より, すべての read 操作実行が前にある最も近い write 操作実行に書き込まれた値を返している. 従って, 系列τは正当である. ■

3.3.2 アルゴリズム $register_{RB-uC}$

次に, u-同期時計モデルの実現アルゴリズム $register_{RB-uC}$を説明する. アルゴリズム $register_{RB-uC}$は, $res\_time(write) = u + \alpha \cdot \max\{d - 2u, 0\}$, $res\_time(read) = u + (1-\alpha)\max\{d-2u, 0\}$ で read/write レジスタを実現している. ここで, 任意のαは $0 \leq \alpha \leq 1$ なる任意の定数であり, 更新を頻繁に行う場合はαを小さく, 読み出しを頻繁に行う場合はαを大きく設定すればすべての操作の応答時間の平均

が短縮される. 以下, write 操作の応答時間 $u + \alpha \cdot \max\{d - 2u, 0\}$ を $|W|$, read 操作の応答時間 $u + (1 - \alpha)\max\{d - 2u, 0\}$ を $|R|$ と記す. 実現アルゴリズム $register_{RB-uC}$のプログラムを図 3.6に示す.

非同期時計モデルのアルゴリズム $register_{RB-AC}$ではタイムスタンプとして整数 $count$ を使っていたが, u-同期時計モデルのアルゴリズム $register_{RB-uC}$では局所時計の値をタイムスタンプとして使う. u-同期時計モデルでは局所時計の値の差が高々u なので, すべての write 操作の応答時間が u 以上ならば, $W_1 \stackrel{E}{\rightarrow} W_2$なる実行 Eの 2 つの write 操作実行 W_1, W_2に対し, W_2のタイムスタンプは W_1のタイムスタンプより大きくなることが保証される. また, アルゴリズム $register_{RB-AC}$の場合と同様にして, すべての read 操作実行の応答時間が u 以上ならば, $R_1 \stackrel{E}{\rightarrow} R_2$なる実行 Eの 2 つの read 操作実行 R_1, R_2に対し, 操作実行 $Write(R_2)$ は $Write(R_1)$ より大きいまたは等しいタイムスタンプを持つことが保証される.

アルゴリズム $register_{RB-uC}$は, read 操作に対する応答値の決定が呼び出しから $\max\{|R|, d - u\}$ 時間後に行われる点がアルゴリズム $register_{RB-AC}$と異なる. この変更により, read 操作実行 Rに先行する $Write(R)$ 以外の任意の write 操作実行のタイムスタンプが $Write(R)$ より小さいことが保証される.

アルゴリズム $register_{RB-uC}$の正当性は次のように示される. 実行 Eに対し, E 内のペンディングしている有効な write 操作の呼出イベント e に対応する応答イベントを時刻 $time(e) + d$ に加えた実行を E'とする. このとき, 3.3.1節の補題 30 と同様にして, 次の補題が成り立つ.

補題 34 実行 E'内の任意の完了した $write$ 操作実行 Wの呼出イベントを Invとし, Invで放送された更新メッセージを Mとする. プロセス P_iが Mに従って局所コピーの更新をするならば, 区間 $[time(Inv) + d - u, time(Inv) + d]$ で行う. ■

また, u-同期時計モデルなので, write 操作の応答時間が u 以上であることより次の補題が成り立つ.

補題 35 実行 E'内の完了した $write$ 操作実行 W_1, W_2に対し, $W_1 \stackrel{E'}{\rightarrow} W_2$ならば, $ts(W_1) < ts(W_2)$ が成り立つ. ■

補題 36 実行 E'内の完了した $read$ 操作実行 R_1, R_2に対し, $R_1 \stackrel{E'}{\rightarrow} R_2$ならば, $ts(Write(R_1)) < ts(Write(R_2))$ または $Write(R_1) = Write(R_2)$ が成り立つ.

constant

$|W| = u + \alpha \cdot \max\{d - 2u, 0\}$, $|R| = u + (1 - \alpha)\max\{d - 2u, 0\}$

data type

timestamp=(integer, process identifier);

variables

$count$, **type** integer, **init** 0;

res_val, **type** value of the register ;

$last\_up\_ts$, **type** timestamp, **init** $(0, 0)$;

$local\_copy$, **type** value of the register, **init** initial value of
the register;

transition functions of process P_i

$Invoke(i, write(v))$:
$ReadClock(i, local\_cl)$;
$BroadCast(i, \text{update}(v, (local\_cl, i)))$; /* update message */
$TimerSet(i, |W|, \text{WRITE})$;

$Invoke(i, read)$:
$TimerSet(i, \min\{|R|, d - u\}, \text{SET\_VAL})$;
$TimerSet(i, |R|, \text{READ})$;

$Receive(i, j, \text{update}(v, (recvd\_cl, recvd\_uid)))$:
if $last\_up\_ts < (recvd\_cl, recvd\_uid)$
then $local\_copy := v$; $last\_up\_ts := (recvd\_cl, recvd\_uid)$;

$Alarm(i, \text{WRITE})$:
$last\_up\_ts := write\_ts$;
$Response(i, \text{ack})$;

$Alarm(i, \text{SET\_VAL})$:
$res\_val := local\_copy$;

$Alarm(i, \text{READ})$:
$Response(i, res\_val)$;

$Stop(i)$:
No events can happen after this event.

図 3.6 実現アルゴリズム $register_{RB-uC}$ (P_iのプログラム)

(証明) read 操作実行 R_1, R_2が実行されたプロセスをそれぞれ P_1, P_2とし, その呼び出し時刻をそれぞれt_1, t_2とする. read 操作の応答時間は u 以上であり, $R_1 \stackrel{E'}{\rightarrow} R_2$ であるため, $t_2 > t_1 + u$ が成り立つ. プロセス P_1は $t_1 + \min\{|R|, d-u\}$ 以前に $Write(R_1)$ の更新メッセージを受信して局所コピーの更新を行っているので, $Write(R_1)$ は $t_1 + \min\{|R|, d-u\} - (d-u)$ 以前に呼び出されている.

(a) P_2は $Write(R_1)$ の更新メッセージに従って局所コピーの更新をする場合

補題 34より, P_2はこの更新を $t_1 + \min\{|R|, d-u\} + u (< t_2 + \min\{|R|, d-u\})$ 以前に行う. ここで, $Write(R_1) \neq Write(R_2)$ とすると, $Write(R_1)$ の更新メッセージ受信時に局所コピーの更新をしたあとで $Write(R_2)$ の更新メッセージに従って更新を行うので, $ts(Write(R_1)) < ts(Write(R_2))$ を意味する.

(b) P_2は $Write(R_1)$ の更新メッセージを無視する場合

プロセス P_2における $Write(R_1)$ の更新メッセージ受信直前の $last\_up\_ts$ の値を lts_1, $Write(R_2)$ の更新メッセージ受信直前の $last\_up\_ts$ の値を lts_2とする. P_2は $Write(R_1)$ の更新メッセージを無視することより, $lts_1 > ts(Write(R_1))$ が成り立つ. また, P_2における時刻 $t_2 + \min\{|R|, d-u\}$ での局所コピーの値を書き込んだ write 操作実行は $Write(R_2)$ なので, $lts_2 \leq ts(Write(R_2))$ が成り立つ. $last\_up\_ts$ は小さくなることはないので, $ts(Write(R_1)) < lts_1 \leq lts_2 \leq ts(Write(R_2))$ が成り立つ. ■

定理 37 アルゴリズム $register_{RB-uC}$は, 信頼放送, u-同期時計モデルにおける $read/write$ レジスタの線形化可能かつ無待機な実現アルゴリズムであり, 任意の $\alpha (0 \leq \alpha \leq 1)$ に対し $res\_time(write) = u + \alpha \cdot \max\{d-2u, 0\}$, $res\_time(read) = u + (1-\alpha)\max\{d-2u, 0\}$ である.

(証明) アルゴリズム $register_{RB-uC}$が無待機であること, 最悪応答時間が $res\_time(write) = u + \alpha \cdot \max\{d-2u, 0\}$, $res\_time(read) = u + (1-\alpha)\max\{d-2u, 0\}$ であるのは明らかである. 以下, アルゴリズム $register_{RB-AC}$が線形化可能であることを示す.

3.3.1節の定理 33と同様に, $complete(E')$ におけるすべての操作実行から成る正当な系列τを構成し, 任意の操作実行 op_1, op_2に対して $op_1 \stackrel{complete(E')}{\rightarrow} op_2$ならば

τで op_1が op_2より前に現れることを示す. 以下, 操作実行 op_1, op_2が write 操作か read 操作により場合わけする.

(a) op_1, op_2が write 操作実行 W_1, W_2のとき

補題 35より, $W_1 \overset{complete(E')}{\rightarrow} W_2$ならば $ts(W_1) < ts(W_2)$ が成り立つ. 構成法よりτで W_1は W_2より前に現れる.

(b) op_1, op_2が read 操作実行 R_1, R_2のとき

補題 36より, $R_1 \overset{complete(E')}{\rightarrow} R_2$ならば, $ts(Write(R_1)) < ts(Write(R_2))$ または $Write(R_1) = Write(R_2)$ が成り立つ. どちらの場合も, 構成法よりτで R_1は R_2より前に現れる.

(c) op_1が write 操作実行 W, op_2が read 操作実行 Rのとき

read 操作実行 Rが実行されたプロセスを P_Rとし, W, Rの呼び出し時刻をそれぞれ t_W, t_Rとする. $W \overset{complete(E')}{\rightarrow} R$より, $t_W + |W| < t_R$が成り立つ.

(c1) P_Rが Wの更新メッセージを無視する場合: P_Rは Wの更新メッセージを受信する前に, Wより大きいタイムスタンプを持つ更新メッセージに従って局所コピーを更新している.

(c2) P_Rが Wの更新メッセージに従って局所コピーを更新する場合: 補題 34より, その更新は時刻 $t_W + d$ 以前に行われる. W_1に対する更新のあと, P_Rがさらに局所コピーの更新を行うならば, W_1より大きいタイムスタンプを持つ更新メッセージに従っている.

read 操作実行 Rは時刻 $t_R + \min\{|R|, d-u\}$ における P_Rの局所コピーの値を返す. $|W| = u + \alpha \cdot \max\{d-2u, 0\}$, $|R| = u + (1-\alpha)\max\{d-2u, 0\}$ より $\min\{|R|, d-u\} + |W| \geq d$が成り立ち, $t_W + |W| < t_R$より $t_R + \min\{|R|, d-u\} > t_W + d$が成り立つので, $W = Write(R)$ または $ts(W) < ts(Write(R))$が成り立つ. 従って, 構成法よりτで Wは Rより前に現れる.

(d) op_1が read 操作実行 R, op_2が write 操作実行 Wのとき

$R, W, Write(R)$ の呼び出し時刻をそれぞれ $t_R, t_W, t_{W(R)}$とする． 補題 34より, $t_{W(R)} < t_R + \min\{|R|, d-u\} - (d-u)$が成り立つ. また, $t_R + |R| < t_W$より,

$$\begin{aligned} t_{W(R)} &< t_W - |R| + \min\{|R|, d-u\} - (d-u) \\ &= t_W - \max\{|R|, d-u\} \\ &\leq t_W - |R| \\ &\leq t_W - u \end{aligned}$$

が成り立つ. アルゴリズムより, $ts(Write(R)) < ts(W)$が成立し, 構成法よりτでRは$Write(R)$とWの間に現れる.

最後に, τの構成法より, すべての read 操作実行が前にある最も近い write 操作実行に書き込まれた値を返している. 従って, 系列τは正当である. ■

3.4. 無信頼放送による read/write レジスタの無待機実現

本節では, 無信頼放送モデルにおける 2 種類の read/write レジスタの実現アルゴリズムとして, 非同期時計モデルのアルゴリズム $register_{UB-AC}$ と u-同期時計モデルのアルゴリズム $register_{UB-uC}$を提案する. 無信頼放送モデルでは, プロセスが放送中に故障するならば, すべての正常なプロセスに同じ送信メッセージが受信されることは保証されない. 一部の正常プロセスのみが受信するようなメッセージを, **不完全放送メッセージ**とよぶ.

3.4.1 アルゴリズム $register_{UB-AC}$

まず, write 操作, read 操作の最悪応答時間がともに d 時間のアルゴリズム $register_{UB-AC}$を説明する. 実現アルゴリズム $register_{UB-AC}$のプログラムを図 3.7に示す.

まず, アルゴリズム $register_{RB-AC}$を無信頼放送モデルに適用することを考える. 不完全放送の更新メッセージ Mは, 一部の正常プロセスの局所コピー更新には反映されない. このとき, 次のように線形化可能性が破棄される. 図 3.8に示す

transition functions of process P_i

$Invoke(i, \text{Write}(v))$:

$count := count + 1$;

for $j = 1$ **to** n /* broadcast an original update message */

do $Send(i, j, \text{update}(v, (count, i)))$;

$TimerSet(i, d, \text{WRITE})$;

$Invoke(i, \text{Read})$:

$res\_val := local\_copy$;

for $j = 1$ **to** n /* broadcast an additional update message */

do $Send(i, j, \text{update}(res\_val, last\_up\_ts))$;

$TimerSet(i, d, \text{READ})$;

$Receive(i, \text{update}(v, (recvd\_ct, recvd\_uid)))$:

$count := \max(count, recvd\_ct)$;

if $last\_up\_ts < (recvd\_ct, recvd\_uid)$

then $local\_copy := v$; $last\_up\_ts := (recvd\_ct, recvd\_uid)$;

$Alarm(i, \text{WRITE})$:

$Return(i, \text{ack})$;

$Alarm(i, \text{READ})$:

$Return(i, res\_val)$;

$Stop(i)$:

No events can happen after this event.

図 3.7 実現アルゴリズム $register_{UB-AC}$ (P_iのプログラム)

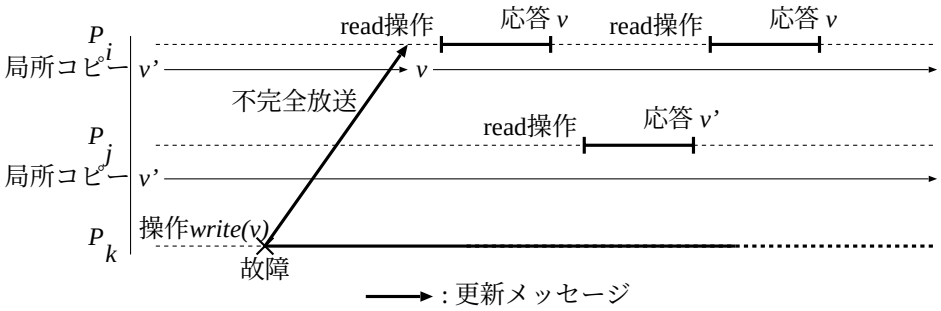


図 3.8 無信頼放送モデルで線形化可能性を破棄する例

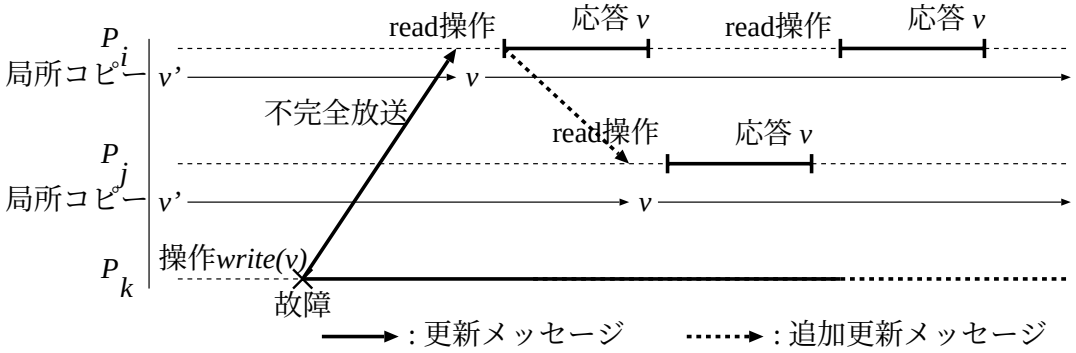


図 3.9 アルゴリズム $register_{UB-AC}$を無信頼放送モデルに適用

プロセス P_i のように, $write(v)$ 操作の更新メッセージ Mを受信して Mに従って局所コピーの更新をするならば, その受信後に呼び出される read 操作 Rに対し, Mに含まれる値 v を応答値として返す. しかし, プロセス P_jのようにメッセージ Mを受信しない正常プロセスは, Rに先行される read 操作実行に対し, 応答値として vを返すことができない. P_jで呼び出された read 操作実行完了後に P_iで呼び出された read 操作実行は，再び vを応答値として返す．すなわち，値 v'の read/write レジスタに対して $write(v)$ 操作は 1 回しか適用されてないが，図 3.8 に示す実行では read/write レジスタは 2 回 vに更新されなければならない.

そこで，メッセージ Mを受信しない正常プロセスが，Mを受信したプロセス

から情報を受け取る方法を考える．図 3.9に示すように, あるプロセス P_iで read 操作 Rが呼び出されたとき，その read 操作で応答値として返す値を**追加更新メッセージ**を放送する. この変更により，write 操作 $Write(R)$ 中の更新メッセージ M が不完全放送であっても, その更新メッセージ Mを受信できなかったプロセスは追加更新メッセージにより局所コピーを更新できる. つまり，プロセス P_iが write 操作 $Write(R)$ の代わりに，すべてのプロセスに更新メッセージ送信をする．プロセス P_iが故障することなく正常に read 操作を完了した場合，すべてのプロセスが write 操作 $Write(R)$ の情報を知ることができる. ここで, 線形化可能性を維持するには, read 操作実行 Rの完了直後に呼び出される他の read 操作も Rと同じ応答値を返さなければならない. そのため，read の応答時間を d に設定することにより, Rで放送される追加更新メッセージを受信できない read 操作実行 R'との先行関係を解消する．これで，read 操作が完了したとき，すべてのプロセスに $Write(R)$ の更新メッセージの情報が受信され，その情報に従って局所コピーが更新されることを保証する．

また，局所コピーの更新は write 操作に割り当てられたタイムスタンプの順序で行うとする．read 操作 Rで放送される追加更新メッセージに, write 操作 $Write(R)$ に割り当てられたタイムスタンプを含む. プロセスは, 従来の更新メッセージと追加更新メッセージを区別することなく受信し, アルゴリズム $register_{AB-AC}$と同様に, 大きいタイムスタンプを含む更新メッセージを受信したときのみ局所コピーを更新する.

次に, アルゴリズム $register_{UB-AC}$が線形化可能性を保証する無待機な read/write レジスタの実現であることを示す. 実行 Eに対し, 以下のように実行 E'を構成する．実行 E内のペンディングしている有効な write 操作 Wに対し, Wの書き込み値を応答値として最も早く返す read 操作実行を R_1とし, R_1の応答時刻と同時刻に Wの応答イベントを加えたものを E'とする．以下, 各 read 操作 Rに対し, $ts(R) = ts(Write(R))$ とする. また, write 操作を含む各操作 op に対し, 操作呼び出し時の $count$ の値と操作が実行されるプロセスの識別子との組を $ts\_op(op)$ とする. 操作 opが write 操作ならば $ts(op) = ts\_op(op)$ が成り立ち, 操作 opが read 操作ならば $ts(op) < ts\_op(op)$ が成り立つ.

補題 38 実行 E'において, 任意の完了した操作実行 op_1, op_2に対し, $op_1 \xrightarrow{E'} op_2$な

らば $(ts(op_1), op\_ts(op_1)) < (ts(op_2), op\_ts(op_2))$ が成り立つ.

(証明) 操作実行 op_1, op_2が実行されたプロセスをそれぞれ P_1, P_2とする. 実行 E' の定義より, プロセス P_1が操作 op_1中に故障をするならば, op_1は有効な write 操作である. $op_1 \stackrel{E'}{\rightarrow} op_2$より, プロセス P_1が write 操作 op_1中に故障をしても, 実行 E' 内では op_1の書き込む値を応答値として最も早く返す read 操作 op_3の応答時刻と同時刻に op_1の応答イベントがある. ここで, 操作 op_3はプロセス P_3で実行されたとする. 操作 op_3は応答値を返して完了しているので, P_3は op_3が完了するまで故障しない. よって, op_3の応答時間は d である. 故障することなく応答を返す場合の応答時間は d であり，また操作実行 op_1中に P_1が故障する場合でも op_1の応答時間は op_3の応答時間より長いので, プロセス P_1の故障の有無に関わらず, op_1の応答時間は d 以上である. ここで, op_1の呼び出し時刻を t_1, op_3の呼び出し時刻を t_3 とする. 各正常プロセスは, op_1の更新メッセージを受信するならば, 遅くとも時刻 $t_1 + d$ までに受信する. このとき, op_1は op_2に先行しているので，op_2の呼び出し時刻を t_2とすると $t_2 > t_1 + d$ が成り立ち, プロセス P_2は op_2にタイムスタンプを割り当てる前に op_1の更新メッセージを受信する. 各正常プロセスは, op_1の更新メッセージを受信しない場合でも, 遅くとも時刻 $t_3 + d$ までに op_1のタイムスタンプを含む op_3の追加更新メッセージを受信する. このとき, op_3は op_2に先行しているので $t_2 > t_3 + d$ が成り立ち, プロセス P_2は op_2にタイムスタンプを割り当てる前に op_3の追加更新メッセージを受信する. op_1の更新メッセージまたは op_3 の追加更新メッセージを受信したとき, P_2の $count$ は少なくとも op_1に割り当てられたタイムスタンプの $count$ 以上である．以下, op_2が write 操作か read 操作かによって場合わけする. $count$ は減少することはないため，その後に呼び出される op_2呼び出し時には $op\_ts(op_1) < op\_ts(op_2)$ が成り立つ. また, op_2が write 操作ならば $ts(op_1) < ts(op_2)$ が成り立ち, op_2が read 操作ならば op_1のタイムスタンプ以上のタイムスタンプを持つ write 操作の書き込み値を返すので, $ts(op_1) \leq ts(op_2)$ が成り立つ. 以上より, $(ts(op_1), op\_ts(op_1)) < (ts(op_2), op\_ts(op_2))$ が成り立つ. ■

定理 39 アルゴリズム $register_{UB-AC}$は, 無信頼放送, 非同期時計モデルにおけるの *read/write* レジスタの線形化可能かつ無待機な実現アルゴリズムであり, $res\_time(write) = d$, $res\_time(read) = d$ である.

(証明) アルゴリズム $register_{UB-AC}$が無待機であること, 最悪応答時間が $res\_time(write) = d$, $res\_time(read) = d$ であるのは明らかである. 以下, アルゴリズム $register_{UB-AC}$が線形化可能であることを示す.

$complete(E')$ におけるすべての操作実行から成る正当な系列τ を構成し, 任意の操作実行 op_1, op_2に対して $op_1 \overset{complete(E')}{\rightarrow} op_2$ならば$\tau$で op_1が op_2より前に現れることを示す. 系列τは初期値を書き込む仮想的な write 操作 W_0から始まると仮定する. ただし, $ts(W_0) = op\_ts(W_0) = (0, 0)$ とする. $complete(E')$ に現れるすべての操作 op を $(ts(op), op\_ts(op))$ の辞書式順序で並べてτを構成する.

補題 38より, 任意の操作実行 op_1, op_2に対し, $op_1 \overset{complete(E')}{\rightarrow} op_2$ならば$\tau$で op_1は op_2より前にある. また, 各 read 操作実行 Rに対し, $W = Write(R)$ かつ $ts(W) = ts(R)$ なる write 操作実行 Wが存在する. また, このような操作実行 R, Wに対し, $op\_ts(W) < op\_ts(R)$ が成り立つ. さらに, W より大きいタイムスタンプが割り当てられるすべての write 操作実行 W_gに対し $ts(W') > ts(W) = ts(R)$ が成り立ち, W より小さいタイムスタンプが割り当てられるすべての write 操作実行 W_sに対し $ts(W_2) < ts(W)$ が成り立つ. 従って, τにおいて Wと Rの間には他の write 操作実行は現れないので, τは正当である. ■

3.4.2 アルゴリズム $register_{UB-uC}$

次に, 無信頼放送, u-同期時計モデルで $res\_time(write) = u$, $res\_time(read) = d$ のアルゴリズム $register_{UB-uC}$を説明する. 実現アルゴリズム $register_{UB-uC}$のプログラムを図 3.10に示す.

アルゴリズム $register_{UB-uC}$では, タイムスタンプとして局所時計の値を使う. また, read 操作では, 呼び出しから $d - u$ 時間後の局所コピーの値を, 呼び出しから d 時間後に応答値として返す, u-同期時計モデルでは局所時計の値の差が高々 u なので, すべての write 操作の応答時間が u 以上ならば, $W_1 \overset{E}{\rightarrow} W_2$なる実行 E の 2 つの write 操作実行 W_1, W_2に対し, W_2のタイムスタンプは W_1のタイムスタンプより大きくなることが保証される. 従って, アルゴリズム $register_{UB-uC}$では write 操作の応答時間が u に短縮できる.

アルゴリズム $register_{UB-uC}$が線形化可能性を保証する無待機な read/write レジスタの実現であることを示す. 実行 Eに対し, アルゴリズム $register_{UB-AC}$の

transition functions of process P_i

$Invoke(i, \text{Write}(v))$:

$ReadClock(i, local_cl)$;

for $j = 1$ **to** n /* broadcast an original update message */

do $Send(i, j, \text{update}(v, (local_cl, i)))$;

$TimerSet(i, u, \text{WRITE})$;

$Invoke(i, \text{Read})$:

$TimerSet(i, d - u, \text{SET\_VAL})$;

for $j = 1$ **to** n /* broadcast an additional update message */

do $Send(i, j, \text{update}(res\_val, last\_up\_ts))$;

$TimerSet(i, d, \text{READ})$;

$Receive(i, \text{update}(v, (recvd\_ct, recvd\_uid)))$:

if $last\_up\_ts < (recvd\_ct, recvd\_uid)$

then $local\_copy := v$; $last\_up\_ts := (recvd\_ct, recvd\_uid)$;

$Alarm(i, \text{WRITE})$:

$Return(i, \text{ack})$;

$Alarm(i, \text{SET\_VAL})$;

$res\_val := local\_copy$;

$Alarm(i, \text{READ})$:

$Return(i, res\_val)$;

$Stop(i)$:

No events can happen after this event.

図 3.10 実現アルゴリズム $register_{UB-uC}$ (P_iのプログラム)

場合 (82ページ) と同様に, 以下のように実行 E'を構成する. 実行 E内のペンディングしている有効な write 操作 Wに対し, Wの書き込み値を応答値として最も早く返す read 操作実行を R_1とし, R_1の応答時刻と同時刻に Wの応答イベントを加えたものを E'とする. 実行 E'に対して以下の補題が成り立つ.

補題 40 実行 E'において, 任意の操作実行 op_1, op_2に対し, $op_1 \stackrel{E'}{\rightarrow} op_2$ならば $(ts(op_1), op\_ts(op_1)) < (ts(op_2), op\_ts(op_2))$ が成り立つ.

(証明) 操作実行 op_1, op_2が実行されたプロセスをそれぞれ P_1, P_2とする. op_1が read 操作か write 操作かにより場合わけする.

(a) op_1が read 操作のとき

実行 E'の定義より, P_1が操作 op_1中に故障することはない. op_1の呼び出し時刻を t_1とすると, P_1は時刻 t_1+d-u に応答値を決定する. よって. $Write(op_1)$ は t_1以前に呼び出される. また, 各正常プロセスは op_1の追加更新メッセージを遅くとも t_1+d までに受信する. op_1は op_2に先行しているので, op_2の呼び出し時刻を t_2 とすると $t_2 > t_1+d$ が成り立ち, プロセス P_2は op_2にタイムスタンプを割り当てる前に op_1の追加更新メッセージを受信する. よって, op_2が write 操作ならば u-同期時計より $ts(op_1) = ts(Write(op_1)) < ts(op_2)$ が成り立つ. op_2が read 操作ならば, P_2は応答値を決定する時刻 t_2+d-u 以前に op_1の追加更新メッセージを受信するので, op_2 に対して $ts(op_1)$ 以上のタイムスタンプを持つ write 操作の書き込み値を返す. すなわち, $ts(op_1) \leq ts(op_2)$ が成り立つ. また, u-同期時計より $op\_ts(op_1) < op\_ts(op_2)$ が成り立つ. 従って, $(ts(op_1), op\_ts(op_1)) < (ts(op_2), op\_ts(op_2))$ が成り立つ.

(b) op_1が write 操作のとき

実行 E'の定義より, プロセス P_1が操作 op_1中に故障するならば, op_1は有効な write 操作である. $op_1 \stackrel{E'}{\rightarrow} op_2$より, プロセス P_1が write 操作 op_1中に故障をしても, 実行 E'内では op_1の書き込む値を応答値として最も早く返す read 操作 op_3の応答時刻と同時刻に op_1の応答イベントがある. ここで, 操作 op_3はプロセス P_3で実行されたとする. 操作 op_3は応答値を返して完了しているので, P_3は op_3が完了するまで故障しない. よって, op_3の応答時間は dである. 故障することなく応答を返す

場合の write 操作の応答時間は u であり，また操作実行 op_1中に P_1が故障する場合でも op_1の応答時間は op_3の応答時間より長いので, プロセス P_1の故障の有無に関わらず, op_1の応答時間は u 以上である．ここで, op_2が write 操作ならば, u-同期時計より $ts(op_1) < ts(op_2)$ は明らかである．以下では, op_2が read 操作の場合を考える．op_1の呼び出し時刻を t_1, op_3の呼び出し時刻を t_3 とする．各正常プロセスは, op_1の更新メッセージを受信するならば, 遅くとも時刻 t_1+d までに受信する．このとき, op_1は op_2に先行しているので，op_2の呼び出し時刻を t_2とすると $t_2 > t_1+u$ が成り立つ．P_2は op_2の応答値を $t_2+d-u(> t_1+d)$ に決定するので, プロセス P_2 は op_2の応答値を決定する前に op_1の更新メッセージを受信する．各正常プロセスは, op_1の更新メッセージを受信しない場合でも, 遅くとも時刻 t_3+d までに op_1のタイムスタンプを含む op_3の追加更新メッセージを受信する．このとき, op_3は op_2 に先行しているので $t_2 > t_3+d$ が成り立つ．すなわち, プロセス P_2は op_2の応答値を決定する前に op_3 の追加更新メッセージを受信する．よって, op_2に割り当てられるタイムスタンプは op_1に割り当てられたタイムスタンプ以上である．よって, $ts(op_1) \leq ts(op_2)$ が成り立つ．また, u-同期時計より $op\_ts(op_1) < op\_ts(op_2)$ が成り立つ．従って, $(ts(op_1), op\_ts(op_1)) < (ts(op_2), op\_ts(op_2))$ が成り立つ． ■

補題 40より, 定理 39と同様にして次の定理を示せる．

定理 41 アルゴリズム $register_{UB-uC}$は，無信頼放送，u-同期時計モデルにおけるの $read/write$ レジスタの線形化可能かつ無待機な実現アルゴリズムであり，$res\_time(write) = u$, $res\_time(read) = d$ である． ■

3.5. 信頼放送による一般オブジェクトの無待機実現

本節では, 信頼放送モデルにおける 2 種類の一般オブジェクトの実現アルゴリズムとして, 非同期時計モデルのアルゴリズム $general_{RB-AC}$ と u-同期時計モデルのアルゴリズム $general_{RB-uC}$を提案する．一般オブジェクトの実現アルゴリズムの効率は, 任意の ack 型操作の最悪応答時間 $res\_time(op_a)$ と任意の val 型操作の最悪応答時間 $res\_time(op_v)$ により評価する．

一般オブジェクトの実現アルゴリズムでは, 各プロセスは実現するオブジェクトの局所コピーを持ち, 全プロセス間共通の順序で操作を局所コピーに適用して

いく. 一般オブジェクトの実現では, すべての操作を局所コピーに反映させる点で read/write レジスタの実現と異なる.

3.5.1 アルゴリズム $general_{RB-AC}$

アルゴリズム $general_{RB-AC}$は, 文献[11]の井上らの $res\_time(op_a) = u$, $res\_time(op_v) = 2d$ のプロセス故障を仮定していない実現アルゴリズムを基にしている. 以降, 井上らのアルゴリズムを $general_{IMT}$と記す. アルゴリズム $general_{IMT}$はプロセス故障がおきる場合に線形化可能性を保証できず, 無待機ではない. 3.5.1 節では, 信頼放送モデル上で無待機性を保証できるようアルゴリズム $general_{IMT}$ を修正する.

まず, 文献 [11] のアルゴリズム $general_{IMT}$を説明し, 次に本論文で提案するアルゴリズム $general_{RB-AC}$への修正点を述べる. アルゴリズム $general_{IMT}$では全プロセス間共通の操作の全順序を次のように決定する. すべてのプロセスは更新メッセージ以外に**報告メッセージ**を使う. ある操作 op がプロセス P_iで呼び出されたとき, プロセス P_iは操作の内容を更新メッセージとして放送する. 呼び出しから $d-u$ 時間後, P_iは全順序で op より前にある操作を決定する. P_iはそのときまでに更新メッセージを受信した操作を op より前の操作とみなし, その順序関係を報告メッセージを用いて放送して他のプロセスに知らせる. このように, あるプロセスが操作 op_1と op_2の順序関係を付けたならば, 操作 op_1は op_2 の呼び出しより前に呼び出されている. また, 任意の操作の応答時間が u 以上で, かつある実行 E上で $op_1 \xrightarrow{E} op_2$が成り立つならば, 実行 Eでは共通の全順序で op_1は op_2より前にあるとみなされる. ここで, 実行 Eにおいて各プロセスに報告メッセージにより知らされる 2 つの操作実行間の半順序関係を $\mathcal{PO}^E$, 実行 Eにおける共通の全順序関係を $\mathcal{TO}^E$で表す. アルゴリズム $general_{IMT}$の正当性は以下のように示される. アルゴリズム $general_{IMT}$の任意の実行を Eとする. 先に述べたような 2 つの操作間の先行関係の集合からなる半順序関係 $\mathcal{PO}^E$から, すべてのプロセスが共通の規則に従って全順序関係 $\mathcal{TO}^E$に拡張できる. また, 操作実行 op の呼び出しから $2d$ 時間後に, op までの全順序関係を決定できる. 従って, val 型操作は応答時間 $2d$ で実現できる. ack 型操作の場合, プロセスは応答値を決定する必要はないが, 線形化可能性を保証するために u 時間を必要とする. 以降, 操作実行 op ま

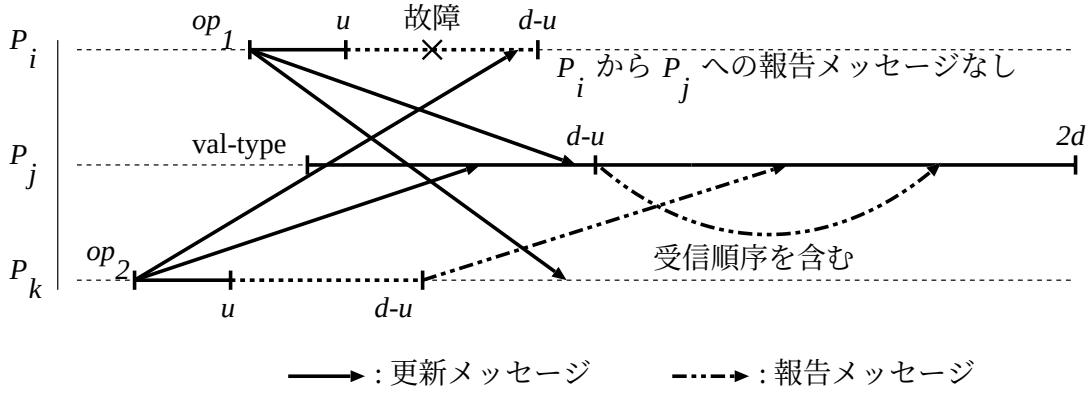


図 3.11 アルゴリズム $general_{RB-AC}$の部分実行

での全順序関係を $\mathcal{TO}^E(op)$ と記す.

次に, 無待機性を持たせるための修正点を述べる. アルゴリズム $general_{IMT}$が線形化可能性を破棄するのは, プロセスが ack 型操作の完了直後に故障する場合である. 実行 Eで, ある ack 型操作実行 op_1がプロセス P_iで呼び出されたとき, op_1は呼び出しから u 時間後に完了し, 呼び出しから $d-u$ 時間後に op_1に対する報告メッセージが放送される. ここで, $d-u > u$ かつ P_iが op_1完了後に報告メッセージを放送する前に故障するならば, 他のプロセスが op_1に関する順序関係を知ることができない. プロセス P_jが応答値を決定するとき, P_jは全順序関係 $\mathcal{TO}$への拡張法に従って操作 op_1を別の操作 op_2より前に処理するかもしれないが, 実際は $op_2 \xrightarrow{E} op_1$かもしれない.

そこで, アルゴリズム $general_{RB-AC}$では以下のように修正する. 図 3.11のように実行 Eにおいて $op_2 \xrightarrow{E} op_1$ならば, P_iを含むすべてのプロセスが op_1の更新メッセージより前に op_2の更新メッセージを受信する. アルゴリズム $general_{RB-AC}$では, このような受信順序を報告メッセージの中に添付する. プロセスが局所コピーに操作の適用をするとき, ある報告メッセージが op_1より op_2の更新メッセージを先に受信したと報告され, かつ逆の順序で受信したという報告メッセージがないならば, $(op_2, op_1) \in \mathcal{PO}^E$としてプロセスは op_1より op_2を先に適用する.

アルゴリズム $general_{RB-AC}$の正当性を示す. 一般オブジェクトの実現アルゴ

リズムの実行に対し, ペンディングしている操作実行のうちその操作実行内で放送された更新メッセージをすべての正常プロセスが受信するような操作実行を**有効**とよぶ. また, 故障により報告メッセージが放送されないような操作実行を**無報告**操作実行とよぶ. 実行 Eに対し, E内のペンディングしている有効な ack 型操作の呼び出しイベント e_aに対応する応答イベントを時刻 $e_a + u$ に, ペンディングしている有効な val 型操作の呼び出しイベント e_vに対応する応答イベントを時刻 $e_v + 2d$ に加えた実行を E'とする.

まず, 文献 [11] より以下の補題が成り立つ.

補題 42 実行 $complete(E')$ において, 任意の操作実行を op_1とし, op_1の呼び出し時刻を t_1とする. このとき, $\mathcal{TO}^{complete(E')}(op_1)$ は時刻 $t + 2d$ までに決定する. ■

また, 実行 $complete(E')$ 内の任意の操作実行に対して以下が成り立つ.

補題 43 実行 E'において, 任意の操作実行を op_1, op_2とし, op_1, op_2の呼び出し時刻をそれぞれ t_1, t_2とする. このとき, $op_1 \overset{complete(E')}{\rightarrow} op_2$ならば, $(op_1, op_2) \in \mathcal{TO}^{complete(E')}$が成り立つ.

(証明) 操作実行 op_1, op_2が実行されたプロセスをそれぞれ P_1, P_2とする. 実行 $complete(E')$ には, 報告メッセージを放送する操作実行と無報告操作実行とが存在しうる. 以下, 操作実行 op_2が報告メッセージを放送しているか無報告かにより場合わけする.

(a) op_2が報告メッセージを放送するとき

実現アルゴリズムより op_1の応答時間は u 以上なので, $t_1 + u < t_2$が成り立つ. このとき, プロセス P_2は op_1の更新メッセージを時刻 $t_2 + d - u$ までに受信するので, P_2は op_2が op_1より後になることを報告メッセージとして知らせる. 従って, $(op_1, op_2) \in \mathcal{TO}^{complete(E')}$ が成り立つ.

(b) op_2は無報告操作実行のとき

実現アルゴリズムより op_1の応答時間は u 以上なので, $t_1 + u < t_2$が成り立つ. このとき, 正常なプロセスは時刻 $t_1 + d$ までに op_1の更新メッセージを受信し, 時

刻 $t_2 + d - u(> t_1 + d)$ よりあとで op_2の更新メッセージを受信する．ここで, $(op_1, op) \in \mathcal{PO}^{complete(E')}$かつ $(op_2, op) \in \mathcal{PO}^{complete(E')}$なる操作実行 op が存在したとする．操作実行 op では報告メッセージを放送する前に op_1, op_2の順序で更新メッセージを受信しているので, 報告メッセージにこの受信順序が添付される．すなわち, どの報告メッセージにも op_2, op_1の順の受信順序は添付されない．従って, $(op_1, op_2) \in \mathcal{TO}^{complete(E')}$ が成り立つ． ■

定理 44 アルゴリズム $general_{RB-AC}$は, 信頼放送, 非同期時計モデルにおけるの一般オブジェクトの線形化可能かつ無待機な実現アルゴリズムであり, $res\_time(op_a) = u$, $res\_time(op_v) = 2d$ である．

(証明) アルゴリズム $general_{RB-AC}$が無待機であること, 最悪応答時間が $res\_time(op_a) = u$, $res\_time(op_v) = 2d$ であるのは明らかである．以下, アルゴリズム $general_{RB-AC}$が線形化可能であることを示す．

$complete(E')$ におけるすべての操作実行から成る正当な系列τ を構成し, 任意の操作実行 op_1, op_2に対して $op_1 \overset{complete(E')}{\rightarrow} op_2$ならば$\tau$で op_1が op_2より前に現れることを示す．$complete(E')$ に現れるすべての操作 op を $\mathcal{TO}^{complete(E')}$の順序で並べて$\tau$を構成する．補題 43より, 任意の操作実行 op_1, op_2に対し, $op_1 \overset{complete(E')}{\rightarrow} op_2$ ならばτで op_1は op_2より前にある．また, 各プロセス P_iは val 型操作実行 op_vの応答値を以下のように決定する．操作実行 op_vが呼び出された時刻を t_vとする．補題 42より, プロセス P_iは全順序関係 $\mathcal{TO}^{complete(E')}$において op_vより前になる操作実行の更新メッセージは時刻 $t_v + 2d$ までに受信する．プロセス P_iは, 局所コピーの初期状態に $(op, op_v) \in \mathcal{TO}^{complete(E')}$なるすべての操作実行を局所コピーに適用した結果を op_vの応答値に決定する．従って, τは正当である． ■

3.5.2 アルゴリズム $general_{RB-uC}$

次に, 非同期時計モデルにおける $res\_time(op_a) = u$, $res\_time(op_v) = d + u$ の一般オブジェクトの無待機な実現アルゴリズムを提案する．実現アルゴリズム $general_{RB-uC}$のプログラムを図 3.12に示す．

アルゴリズム $general_{RB-uC}$では, 各操作実行に割り当てられたタイムスタンプによって, 操作実行に対してすべてのプロセスに共通の順序を決める．ある操作

variables
op_ts, **type** timestamp ;
$local\_copy$, **type** value of the object, **init** initial value of the object;
$update\_buffer$, **init** empty;

transition functions of process P_i
$Invoke(i, op)$:
    $ReadClock(i, local\_cl)$;
    $BroadCast(i, \text{update}(op, (local\_cl, i)))$; /* update message */
    if op is ack-type **then** $TimerSet(i, u, \text{ack})$;
    else /* op is val-type */
        $op\_ts := (local\_cl, i)$; $TimerSet(i, d + u, \text{val})$;

$Receive(i, j, \text{update}(v, (recvd\_cl, recvd\_uid)))$:
    $update\_buffer := update\_buffer \cup (op, (recvd\_cl, recvd\_uid))$

$Alarm(i, \text{ack})$:
    $Response(i, res)$ where res is a unique response value for
        current op;

$Alarm(i, \text{val})$:
    while $op\_ts \geq \min\{ts | (op, ts) \in update\_buffer\}$ **do**
        $smallest := (op, ts)$ where ts is the smallest in $update\_buffer$;
        apply $smallest$ to $local\_copy$;
        $update\_buffer := update\_buffer - \{smallest\}$;
    $Response(i, res)$;

$Stop(i)$:
    No events can happen after this event.

図 3.12 実現アルゴリズム $general_{RB-uC}$ (P_iのプログラム)

op がプロセス P_i に呼び出されたとき, P_i は操作 op に局所時計の値を割り当て, 操作の内容とタイムスタンプを更新メッセージとして送信する. 更新メッセージを受信したプロセスは, 局所変数 $update\_buffer$ にその情報を格納する. u-同期時計モデルなので, メッセージ遅延が高々 d であることより, ある操作実行 op_1 の呼び出しから $d+u$ 時間以降に各正常プロセスで更新メッセージが受信される操作実行 op_2 に対し, op_2 は op_1 より大きいタイムスタンプが割り当てられる. すなわち, op_1 より小さいタイムスタンプが割り当てられる操作実行は, op_1 の呼び出しから $d+u$ 時間後までにすべて知ることができる. 従って, val 型操作 op に対し, 呼び出しから $d+u$ 後に操作 op までの全順序を決定し, 応答値を返すことができる. ack 型操作の場合, プロセスは応答値を決定する必要はないが, 線形化可能性を保証するために u 時間を必要とする. また, プロセスが故障しない限り, 任意の操作の呼び出しに対し一定時間後に応答が返されるので, アルゴリズム $general_{RB-uC}$ の無待機性が保証される.

アルゴリズム $general_{RB-uC}$ の正当性は次のように示される. 実行 E に対し, E 内のペンディングしている有効な ack 型操作の呼び出しイベント e_a に対応する応答イベントを時刻 e_a+u に, ペンディングしている有効な val 型操作の呼び出しイベント e_v に対応する応答イベントを時刻 e_v+d+u に加えた実行を E' とする.

このとき, 実行 E' 内の任意のイベントに対して以下が成り立つ.

補題 45 実行 E' 内の任意の完了した操作実行の呼び出しイベントを Inv とし, Inv で放送された更新メッセージを M とする. プロセス P_i が M に従って局所コピーを更新するならば, M を区間 $[time(Inv)+d-u, time(Inv)+d]$ で受信する. ■

u-同期時計モデルなので, 任意の操作の応答時間が u 以上であることより次の補題が成り立つ.

補題 46 実行 E' 内の完了した操作実行 op_1, op_2 に対し, $op_1 \stackrel{E'}{\rightarrow} op_2$ ならば $ts(op_1) < ts(op_2)$ が成り立つ. ■

また, 任意の val 型操作に対して以下の補題が成り立つ.

補題 47 実行 E' 内の任意の完了した val 型操作実行を op_v とし, op_v が実行されたプロセスを P_v, op_v の呼び出し時刻を t_v とする. また, $ts(op_1) < ts(op_v)$ なる任意

の操作実行を op_1とし, op_1で放送された更新メッセージを Mとする. このとき, プロセス P_iは Mを時刻 $t_v + d + u$ より前に受信する.

(証明) 操作実行 op_1が実行されたプロセスを P_1とし, op_1の呼び出し時刻を t_1とする. u-同期時計モデルなので, $ts(op_1) < ts(op_v)$ ならば $t_1 < t_v + u$ が成り立つ. 従って, 補題 45よりプロセス P_iは Mを時刻 $t_v + d + u$ より前に受信する. ■

定理 48 アルゴリズム $general_{RB-uC}$は, 信頼放送, 非同期時計モデルにおけるの一般オブジェクトの線形化可能かつ無待機な実現アルゴリズムであり, $res\_time(op_a) = u$, $res\_time(op_v) = d + u$ である.

(証明) アルゴリズム $general_{RB-uC}$が無待機であること, 最悪応答時間が $res\_time(op_a) = u$, $res\_time(op_v) = d + u$ であるのは明らかである. 以下, アルゴリズム $general_{RB-uC}$が線形化可能であることを示す.

$complete(E')$ におけるすべての操作実行から成る正当な系列τ を構成し, 任意の操作実行 op_1, op_2に対して $op_1 \overset{complete(E')}{\rightarrow} op_2$ならば$\tau$で op_1が op_2より前に現れることを示す. $complete(E')$ に現れるすべての操作 op を $ts(op)$ の順序で並べてτ を構成する. 補題 46より, 任意の操作実行 op_1, op_2に対し, $op_1 \overset{complete(E')}{\rightarrow} op_2$ならば$\tau$で op_1は op_2より前にある. また, 各プロセス P_iは val 型操作実行 op_v内において受信した更新メッセージを基に局所コピーの更新を行うが, 補題 47より P_iは局所コピーの初期状態に op_vよりタイムスタンプの小さいすべての操作実行を局所コピーに適用した結果を op_vの応答値に決定する. 従って, τは正当である. ■

3.6. むすび

本章では, 同期式メッセージパッシングシステムにおける線形化可能性を保証する共有オブジェクトの無待機な実現について考察した. 文献 [12] より, 完全非同期メッセージパッシングシステムでは線形化可能性を保証する read/write レジスタの無待機な実現アルゴリズムは存在しない. そこで, 同期式メッセージパッシングシステム上に線形化可能性を保証する read/write レジスタの実現アルゴリズムを提案した. 放送モデルとして信頼放送モデルまたは無信頼放送モデル, 時計モデルとして非同期時計モデルまたは u-同期時計モデルを仮定し, それぞれの

モデルを組み合わせた表 3.2に示す 4 種類の実現アルゴリズムを提案した. また, 同期式メッセージパッシングシステムにおける線形化可能性を保証する一般オブジェクトの実現アルゴリズムとして, 信頼放送モデルの下で時計モデルとして非同期時計モデルまたは u- 同期時計モデルを仮定した表 3.2に示す 2 種類の実現アルゴリズムを提案した.

一般に, 非同期時計モデルの実現アルゴリズムは, 他の条件が等しい u-同期時計モデルの実現アルゴリズムよりも応答時間が長くなる. しかし, 非同期時計モデル上でシステム内のプロセスが同期手続きを実行することにより, 局所時計の値の差を高々u にして u-同期時計モデルのための実現アルゴリズムを採用することができる. 同期手続きとして, Mavronicolas らの提案した手続き Synch[15] などがある. 同期手続きのコストを考慮すると, 非同期時計モデルでも, 頻繁に操作が呼び出されるような場合は同期手続きを実行し, u 同期時計モデルのための実現アルゴリズムを採用した方が効率的である.

本章に関連する今後の課題として, 次の問題が挙げられる. まず, 線形化可能性を保証する実現アルゴリズムに対し, いくつかの最悪応答時間に関する下界の結果が示されている [3, 16, 11]. これらの結果と本章の結果にギャップがある. また, 本章では提案していない無信頼放送モデルにおける一般オブジェクトの実現アルゴリズムに関しては, 最悪応答時間がプロセス数に比例するような実現アルゴリズムは存在する. しかし, 最悪応答時間がプロセス数に独立な実現アルゴリズムが存在するかどうかはまだわかっていない.

第 4 章

結論

本論文では, 高度な故障耐性を有する分散アルゴリズムである無待機分散アルゴリズムの基本的な問題について考察した.

まず, システム内のプロセスが保持する局所時計の値を一致させる無待機時計合わせアルゴリズムを取り上げた. 本論文では特にフェーズ内システムとよばれる同期式共有メモリシステム上の無待機時計合わせアルゴリズムについて考察した. フェーズ内システム上の無待機時計合わせアルゴリズムの同期時間の下界が $n-2$ であることを示した. さらに, 同期時間 $12n$ の無待機時計合わせアルゴリズム, 空間複雑度が有界な同期時間 $15n$ の自己安定無待機時計合わせアルゴリズムを提案した. 過去に提案された最も効率的な無待機時計合わせアルゴリズムとして同期時間 $4n^2-3n-1$ の自己安定無待機時計合わせアルゴリズムがあったが, 本論文で提案された 2 つのアルゴリズムは同期時間に関して大きく改善している. また, 提案した 2 つのアルゴリズムは同期時間の関してオーダー的に最適である.

次に, メッセージパッシングシステムにおける線形化可能性を保証する共有オブジェクトの無待機な実現を取り上げた. 文献 [12] により, 完全非同期メッセージパッシングシステム上では線形化可能性を保証する read/write レジスタの無待機な実現は不可能であることが示されていた. そこで本論文では, 同期式メッセージパッシングシステム上に線形化可能性を保証する read/write レジスタの実現アルゴリズムを提案した. 放送モデルとして信頼放送モデルまたは無信頼放送モデル, 時計モデルとして非同期時計モデルまたは u-同期時計モデルを仮定し, それぞれのモデルを組み合わせた 4 種類の実現アルゴリズムを提案した. また, 同期

式メッセージパッシングシステムにおける線形化可能性を保証する一般オブジェクトの実現アルゴリズムとして, 信頼放送モデルの下で時計モデルとして非同期時計モデルまたは u-同期時計モデルを仮定した 2 種類の実現アルゴリズムを提案した.

本論文では基本的な無待機分散アルゴリズムとして 2 つのアルゴリズムを取り上げたが, それらのアルゴリズム以外に, システム内のプロセスをある値で合意に導く合意アルゴリズム, プロセスに新たな固有の識別子を与える名前付け替えアルゴリズムなどの無待機性に関する研究がされている. 無待機性は任意個のプロセスの停止故障, 居眠り故障に対する耐性を意味するが, このような故障は実際のシステムでしばしばおこりうる. 従って, 無待機分散アルゴリズムを考察することは, 分散システムを実際に実現する場合に構築したシステムの動作の正しさに理論的な根拠を与える意味でも意義深いと考えられる.

謝辞

本研究の全過程を通じて, 適切な御指導と御助言を賜わりました藤原秀雄教授に深く感謝の意を表します.

本論文の執筆にあたり貴重な御助言をいただきました福田晃教授, 伊藤実教授に深謝致します.

本研究の全過程を通じて, 方針, 問題点などのあらゆる面において, 懇切丁寧に直接的な御指導をして頂きました増澤利光助教授に深く感謝致します.

本研究をまとめるに際し, 様々な面でお世話になり, 御討論, 御協力頂きました井上美智子助手に深く感謝致します.

本研究を進めるにあたって, 日頃から様々な面での熱心な御助言, 御協力を頂きました広島市立大学井上智生助教授 (前本学助手), 本学大竹哲史助手に深く感謝致します.

また, 日頃より有益な御討論, 御援助を頂きました情報論理学講座の皆様に深く感謝致します. 特に, 本研究をすすめるにあたり熱心な御協力を頂きました須田克朗氏, 有益な御討論を頂きました上田英一郎氏, 石水隆氏, 浮穴学慈氏に感謝致します.

参考文献

[1] A. Arora, S. Dolev, and M. Gouda. Maintaining digital clocks in step. *Parallel Processing Letters*, 1(1):11–18, 1991.

[2] H. Attiya. Implementing fifo queues and stacks. *Proceedings of the 5th International Workshop on Distributed Algorithms(LNCS579)*, pages 80–94, 1991.

[3] H. Attiya and J. L. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, May 1994.

[4] H. Attiya and J. L. Welch. *Distributed Computing*. McGraw Hill, 1998.

[5] D. Dolev, J.Y. Halpern, and H.R. Strong. On the possibility and impossibility of achieving clock synchronization. *Journal of Computer System Science*, 32(2):230–250, 1986.

[6] S. Dolev. Possible and impossible self-stabilizing digital clock synchronization in general graphs. *Technical Report TR 96-06, Department of Mathematics and Computer Science. Ben-Gurion University*, 1996.

[7] S. Dolev and J.L. Welch. Wait-free clock synchronization. *Proceedings of the 12th ACM Symposium on Principles of Distributed Computing*, pages 97–108, 1993.

[8] J. Halpern, B. Simons, R. Strong, and D. Dolev. Fault-tolerant clock synchronization. *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, pages 89–102, 1984.

[9] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

[10] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transaction on Programming Languages and Systems*, 12(3):463–492, 1990.

[11] M. Inoue, T. Masuzawa, and N. Tokura. Efficient linearizable implementation of shared fifo queues and general objects on a distributed system. *IEICE Transactions on Fundamentals on Electronics, Communications and Computer Sciences*, E81-A(5):768–775, May 1998.

[12] J. James and A. K. Singh. Fault tolerance bounds for memory consistency. *Proceedings of the 11th International Workshop on Distributed Algorithms (LNCS1320)*, pages 200–214, 1997.

[13] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess program. *IEEE Transaction on Computers*, 28(9):690–691, 1979.

[14] L. Lamport and P.M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):1–36, 1985.

[15] M. Mavronicolas and D. Roth. Sequential consistency and lineariability:read/write objects. *Proceedings of the 29th Annual Allerton Conference on Communication, Control and Computing*, Oct. 1991.

[16] M. Mavronicolas and D. Roth. Efficient, strongly consistent implementations of shared memory. *Proceedings of the 6th International Workshop on Distributed Algorithms(LNCS647)*, pages 346–361, 1992.

[17] M. Papatriantafilou and P. Tsigas. On self-stabilizing wait-free clock synchronization. *Proceedings of the 4th Scandinavian Workshop on Algorithm Theory(LNCS 824)*, pages 267–277, 1994.

[18] T.K. Srikanth and S. Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, 1987.

[19] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 1994.

[20] J.W. Welch and N. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77(1):1–36, 1988.

[21] 亀田恒彦, 山下雅史. 分散アルゴリズム. 近代科学社, 1994.