非同期共有メモリシステムにおける
ポイント競合度適応型繰り返し改名アルゴリズム

梅谷 真也 [†]　井上 美智子 [†]　増澤 利光 [‡]　藤原 秀雄 [†]

[†] 奈良先端科学技術大学院大学 情報科学研究科
[‡] 大阪大学大学院 基礎工学研究科

あらまし　本レポートでは，非同期共有メモリシステム上で，効率良く名前の獲得と解放を繰り返し行う分散アルゴリズム (繰り返し M-改名アルゴリズム) を提案する．システムに属する N 個のプロセスは，$0, 1, \cdots, N-1$ の範囲の相異なる名前を最初に持っている．繰り返し M-改名アルゴリズムを実行することで，$0, 1, \cdots, M-1$ の範囲の名前を新しく獲得し，その名前を使用した後に解放する．複数のプロセスが同時に同一の名前を保持することはない．

従来の最良の $k(2k-1)$-改名アルゴリズムとして，Afek らがステップ計算量 $O(k^3)$，空間計算量 $O(n^3N)$ のアルゴリズムを提案した [14]．ここで，k はポイント競合度，n は k の上界を示す．ポイント競合度とは，名前を獲得する際に，同時にステップを実行したり名前を保持しているプロセスの最大数である．さらに，変数の値が非有界となるが，ステップ計算量 $O(k^2 \log k)$，空間計算量 $O(n^3N)$ の $k(2k-1)$-アルゴリズムも提案している [14]．本レポートでは，これら従来の $k(2k-1)$-改名アルゴリズムより効率の良い，ステップ計算量 $O(k^2)$，空間計算量 $O(n^2N)$ かつ変数の値が有界な $k(2k-1)$-改名アルゴリズムを提案する．

Point-contention Adaptive Long-lived Renaming Algorithm for Asynchronous Shared Memory Systems

Shinya Umetani[†]　Michiko Inoue[†]　Toshimitsu Masuzawa[‡]　Hideo Fujiwara[†]

[†]Graduate School of Information Science, Nara Institute of Science and Technology
[‡]Graduate School of Engineering Science, Osaka University

Abstract. This report presents an adaptive long-lived M-renaming algorithm in an asynchronous shared memory system. The system consists of N asynchronous process, and each process initially has a distinct name in the range $\{0, 1, \cdots, N-1\}$. Every process acquires a new name in the range $\{0, 1, \cdots, M-1\}$, and releases it after the use. No two processes holds the same name concurrently.

The previous best $k(2k-1)$-renaming algorithm is the algorithm with $O(k^3)$ step complexity and $O(n^3N)$ space complexity presented by Afek et. al [14], where k is the point contention and n is upper bound of k. The point contention is the maximum number of processes that actually take steps or hold a name while the new name is being acquired. They also presented $k(2k-1)$-renaming algorithm with $O(k^2 \log k)$ step complexity and $O(n^3N)$ space complexity under the condition where unbounded values are allowed. This report shows $k(2k-1)$-renaming algorithm which is more efficient than two previous algorithms. The step complexity of this algorithm is $O(k^2)$, and space complexity is $O(n^2N)$.

1 はじめに

分散システムは，自律的に動作する N 個のプロセスと通信メディアから構成される．分散システムにはシステム全体を集中管理する計算機は存在しないので，各プロセスが通信メディアを介して情報を交換しながら，協調して計算を行うアルゴリズムを設計する必要がある．このような分散システムのためのアルゴリズムを分散アルゴリズムと呼ぶ．分散システムは拡張性，可用性，資源共有性等に優れた特性を持っており，分散アルゴリズムに関する研究の重要性はますます高まっている．

分散システムの代表的なモデルに，プロセス間で直接通信を行うメッセージパッシングシステムと，各プロセスが共通に接続されているメモリを介して通信を行う共有メモリシステムがある．本レポートでは，プロセスの実行速度，プロセスの局所時計の速度に仮定を置かず，プロセスは共有メモリへ非同期にアクセスする非同期共有メモリシステムにおける分散アルゴリズムについて議論する．

分散システムを構成する N 個のプロセスは，$0, 1, ..., N-1$ の範囲の相異なる名前を持っている．分散アルゴリズムの中には，計算量がプロセスの名前空間の大きさに依存するものが数多く存在する．このようなアルゴリズムを実行する場合，プロセスの名前空間が大きいほど，多くの計算量が必要となる．同時に共有メモリへアクセスすることを競合すると言い，競合するプロセスの個数を競合度と呼ぶ．現実のシステムにおいて，システムを構成する全てのプロセスが競合することは少なく，競合度はプロセスの総数に比べて非常に小さい場合が多い．

そこで，そのようなアルゴリズムを実行するプロセスに，競合度に応じた小さな名前を新しく与えられれば，結果としてアルゴリズムの計算量を小さくできる．新しく与える名前の範囲を $0, 1, ..., M-1$ とすると，このようなプロセスの名前空間の大きさを変更する問題を M-改名問題と言う．最近では，システム内に膨大な数のプロセスが含まれることが多いため，改名を行う意義は大きい．これまでに，コレクトアルゴリズム [12] や，スナップショットアルゴリズム [13] 等，様々なアルゴリズムに M-改名アルゴリズムが応用されている．

表 1: 一回 M-改名アルゴリズム

文献	名前空間	ステップ計算量	分類
[1]	$2n_o-1$	$O(4^{n_o}N)$	無待機
[3]	$2n_o-1$	$O({n_o}^2N)$	無待機
[11]	$2n_o-1$	$O({n_o}^4)$	高速
[4]	$2k_o-1$	$O(n_o\log n_o)$	高速
	$6k_o-1$	$O(k_o\log k_o)$	適応型
[16]	$2k_o-1$	$O({k_o}^2)$	適応型

M-改名問題は，プロセスが高々1 回だけ新しい名前を獲得する一回 M-改名問題と，名前の獲得とその解放を繰り返し行う繰り返し M-改名問題に分類することができる．

表 1 に，これまでに提案されている一回 M-改名アルゴリズムを示す．ここで，N はシステムに属するプロセスの総数，k_o はアクティブなプロセスの個数である競合度，n_o は k_o の上界を示す．アクティブなプロセスとは，すでに一回 M-改名アルゴリズムの実行を開始しているプロセスのことを言う．上でも述べたように，M-改名問題はアルゴリズムの計算量を削減することを目的としているので，M-改名アルゴリズム自身の計算量も小さい方が望ましい．M-改名問題の研究初期段階において，一回 $(2n_o-1)$-改名アルゴリズムが提案された [1, 3]．これらのアルゴリズムは，他のプロセスの動作にかかわらず，有限個のステップで改名を行えることを保証する．このような M-改名アルゴリズムを，無待機一回 M-改名アルゴリズムと言う．しかし，これら一回 $(2n_o-1)$-改名アルゴリズムのステップ計算量は，$O(4^{n_o}N)$ [1] や $O({n_o}^2N)$ [3] であり，システムに含まれるプロセスの総数 N に依存するものであった．

その後，ステップ計算量が N に依存せず，競合度 k_o の上界 n_o にのみ依存する高速一回 M-改名アルゴリズムが提案された．文献 [11]，[4] ではステップ計算量がそれぞれ $O({n_o}^4)$，$O(n_o\log n_o)$ の高速一回 $(2k_o-1)$-改名アルゴリズムが提案された．

さらに，高速一回 M-改名アルゴリズムの中でも，ステップ計算量が競合度 k_o のみに依存するアルゴリズムを，適応型一回 M-改名アルゴリズムと言う．文献 [4] では，ステップ計算量が $O(k_o\log k_o)$ の適応型一回 $(6k_o-1)$-改名アルゴリズム，文献

表 2: 繰り返し M-改名アルゴリズム

文献	名前空間	ステップ計算量	空間計算量	変数の値	分類
[15]	$4{k_i}^2$	$O({k_i}^2)$	$O({n_i}^2N)$	有界	適応型
	$k(2k-1)$	$O(k^2\log k)$	非有界	有界	適応型
[14]	$k(2k-1)$	$O(k^2\log k)$	$O(n^3N)$	非有界	適応型
	$2k-1$	$Exp(k)$	$O(n^3N)$	非有界	適応型
	$k(2k-1)$	$O(k^3)$	$O(n^3N)$	有界	適応型
[6]	$2k-1$	$O(k^4)$	非有界	非有界	適応型
本レポート	$k(2k-1)$	$O(k^2)$	$O(n^2N)$	有界	適応型

[16] ではステップ計算量が $O({k_o}^2)$ の適応型一回 $(2k_o-1)$-改名アルゴリズムが提案された．

表 2 に，これまでに提案されている繰り返し M-改名アルゴリズムと，本レポートで提案する繰り返し M-改名アルゴリズムを示す．ここで，N はシステムに属するプロセスの総数，k_i は繰り返し M-改名アルゴリズムの実行区間においてアクティブになるプロセスの総数である区間競合度，k は繰り返し M-改名アルゴリズムの実行区間において同時にアクティブになるプロセスの最大数であるポイント競合度を示す．n_i，n はそれぞれ k_i，k の上界とする．アクティブなプロセスとは，繰り返し M-改名アルゴリズムを実行している，もしくは名前を保持しているプロセスのことを言う．

繰り返し M-改名アルゴリズムの中で，ステップ計算量が競合度のみに依存するアルゴリズムを，適応型繰り返し M-改名アルゴリズムと言う．適応型繰り返し M-改名アルゴリズムは競合度の定義に従って，区間競合度適応型 M-改名アルゴリズムとポイント競合度適応型 M-改名アルゴリズムに分類できる．

文献 [15] では，空間計算量が有界な区間競合度適応型繰り返し $4{k_i}^2$-改名アルゴリズムと，空間計算量が非有界なポイント競合度適応型繰り返し $k(2k-1)$-改名アルゴリズムが提案された．文献 [14] では，文献 [15] のアルゴリズムが改良され，変数の値が非有界ではあるが，空間計算量を $O(n^3N)$ に削減したポイント競合度適応型繰り返し $k(2k-1)$-改名アルゴリズムが提案された．文献 [14] では他に，ステップ計算量が $O(Exp(k))$ と大きいが，改名後の名前空間の大きさが最適である [10] ポイント競合度適応型繰り返し $(2k-1)$-改名アルゴリズムと，ステップ計算量は $O(k^3)$ と大きくなるが変数の値を有界に改良した，ポイント競合度適応型繰り返し $k(2k-1)$-改名アルゴリズムも提案されている．文献 [6] では，ポイント競合度適応型繰り返し $(2k-1)$-改名アルゴリズムを，$O(k^4)$ のステップ計算量で実現している．

本レポートでは，M-改名アルゴリズムにおける最近の研究の中心である，ポイント競合度適応型繰り返し M-改名アルゴリズムを対象とし，従来のアルゴリズムより効率の良いポイント競合度適応型繰り返し $k(2k-1)$-改名アルゴリズムを提案する．本改名アルゴリズムは，Afek らが文献 [14] で提案したステップ計算量が $O(k^3)$ の $k(2k-1)$-改名アルゴリズム (Afek らのアルゴリズム) を改良したものである．Afek らのアルゴリズムでは，名前を獲得するために，シーブと呼ばれる基本要素の配列を用いている．各シーブでは，束合意 [4] と呼ばれるオブジェクトを利用することで，そのシーブにアクセスしているプロセスのスナップショットを得る．本アルゴリズムでは，束合意をより単純なオブジェクトであるコレクトリストに置き換えることにより，変数の大きさは有界のままに，ステップ計算量を $O(k^2)$，空間計算量を $O(n^2N)$ に削減することができた．コレクトリストは，適応型コレクトアルゴリズム [5] に基づいている．

本レポートの構成は次の通りである．2 節では非同期共有メモリのモデルと，このモデルにおける改名問題，さらに改名アルゴリズムの評価尺度ついて定義する．3 節と 4 節では，それぞれ本レポートで提案するコレクトリストと改名アルゴリズムについて詳しく述べる．最後に 5 節では，まとめと今後の課題について述べる．

2　諸定義

2.1　非同期共有メモリシステム

本レポートで提案する M-改名アルゴリズムは，非同期共有メモリシステムを対象とする．このシステムは，N 個のプロセス $p_0, p_1, ..., p_{N-1}$ と，共有メモリによって構成される (図 1)[7, 17]．

プロセスは状態機械であるとし，共有メモリへのアクセス，またはプロセスの内部計算等のイベントによって状態遷移を行う．全てのプロセスは相異なる名前を持ち，他のプロセスと区別することができる．また各プロセスは非同期に動作するとし，各プロセスによるイベントの生起間隔に関しては何も仮定しない．

共有メモリは，複数のプロセスに共有される共有変数の集合である．共有メモリに対する命令について，読み込み (read) 操作と書き込み (write) 操作のみが利用可能な read/write モデルや，read-modify-write 操作が利用可能な read-modify-write モデル等がある．read-modify-write 操作のような強力な命令を使うと，read/write モデルよりも小さな計算量で問題が解決できたり，read/write モデルでは解決できない問題が解決できる場合がある．しかし，現実には一般的な計算機が強力な操作を備えているとは必ずしも言えない．そこで本レポートでは，より一般的な read/write モデルを仮定する．従って，プロセス内部では以下のイベントのみが発生するものとする．

1. 共有変数への read 操作
2. 共有変数への write 操作
3. 内部計算

全てのプロセスの状態と，全ての共有変数の状態からなるベクトルを，システムの状況と言う．特に全プロセスが初期状態にあり，かつ全共有変数が初期値を保持している状況を初期状況と呼ぶ．

あるイベントの有限 (または無限) 系列 $E = e_0, e_1, e_2, ...$ に対して，初期状況 c_0 から始まるある状況の有限 (または無限) 系列 $C = c_0, c_1, c_2, ...$ が存在し，各状況 c_{i-1} $(i \geq 1)$ においてイベント e_{i-1} が生起することで状況 c_i に遷移するとき，E を実行と言う．なお，read 操作や write 操作は原子操作であるので，これらの操作によるイベントが複数のプロセスにおいて同時に生起することはない．この性質により，同一の共有変数に対して複数のプロセスが同時にアクセスすることはないので，read 操作や write 操作の競合について考慮する必要はない．原子操作を保証する共有変数の実現に関しては，文献 [2, 7, 8, 9] を参照されたい．

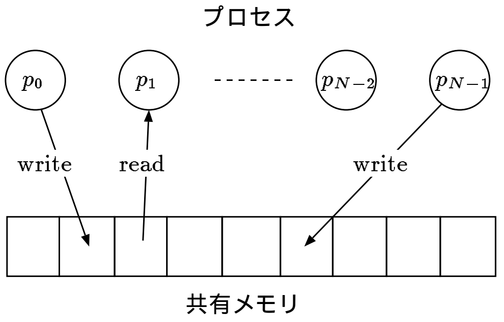

図 1: 非同期共有メモリシステム

2.2　M-改名問題

M-改名問題とは，プロセスの名前空間の大きさを縮小する問題であり，一回 M-改名問題と繰り返し M-改名問題に分類できる．M-改名アルゴリズムの任意の実行が次の条件を満たすとき，アルゴリズムは M-改名問題を解決すると定義する [11]．

一回 M-改名アルゴリズム　あるプロセス p_i は，名前を獲得するための手続き getName を高々1 回実行する．手続き getName の返り値 $name_i$ に対して，以下が成り立つ．

1. プロセス p_i が名前 $name_i$ を獲得したなら $name_i \in \{0, 1, ..., M-1\}$
2. 異なるプロセス p_i，p_j がそれぞれ名前 $name_i$，$name_j$ を獲得したなら，$name_i \neq name_j$

繰り返し M-改名アルゴリズム　あるプロセス p_i は，名前を獲得するための手続き getName と名前の使用，名前を解放するための手続き releaseName を繰り返し実行する．図 2 に，繰り返し M-改名アルゴリズムの構成を示す．ここで，$RemainderSection$ は名前の獲得，使用及び解放に関係の無い部分，$WorkingSection$ は獲得した名前を使用する部分である．

図 2: 繰り返し M-改名アルゴリズムの構成

```
Process $p_i$        // $0 \le i \le N-1$
Non-shared variables $name_i$ : $0, ..., M-1$
    while (true) do
        $RemainderSection$;
        $name_i$ := getName($i$);
        $WorkingSection$;
        releaseName($name_i$);
    od;
```

プロセス p_i は，手続き getName の返り値 $name_i$ を p_i の局所変数に代入してから，手続き releaseName による名前の解放を開始するまでの間，名前 $name_i$ を保持しているという．任意の状況において，以下が成り立つ．

1. プロセス p_i が名前 $name_i$ を保持しているなら $name_i \in \{0, 1, ..., M-1\}$
2. 異なるプロセス p_i，p_j がそれぞれ名前 $name_i$，$name_j$ を同時に保持しているなら，$name_i \neq name_j$

2.3 競合度

2.3.1 一回 M-改名アルゴリズムにおける競合度

一回 M-改名アルゴリズムのある実行を α，α の接頭部を α' とする．もし α' の最後のイベント e において，プロセス p_i がすでに手続き getName を呼び出しているなら，プロセス p_i はイベント e においてアクティブであるという．すなわち，アクティブなプロセスは，名前の獲得を試みているか，もしくは既に新しい名前を獲得している．イベント e においてアクティブなプロセスの個数を競合度と呼び，k_o と表記する．また，k_o の上界を n_o とする．

一般に非同期共有メモリシステムでは，アルゴリズムの実行前に，任意の実行における競合度を知ることはできない．しかし，任意の実行に対する競合度の上界は，あらかじめ与えられているとする．

2.3.2 繰り返し M-改名アルゴリズムにおける競合度

繰り返し M-改名アルゴリズムのある実行を α，α の接頭部を α' とする．もし α' の最後のイベント e において，プロセス p_i がすでに手続き getName を呼び出しており，かつそれに対応する手続き releaseName から返っていないならば，p_i はイベント e においてアクティブであると言う．すなわち，アクティブなプロセスは，名前の獲得を試みている，もしくは新しい名前を保持している．

実行 α に対して，$\alpha = \alpha_1 \beta \alpha_2$ なる実行区間 β を考える．ここで，α_1，α_2 は長さ 0 以上の部分実行，β は長さ 1 以上の部分実行である．このとき，実行区間 β における競合度として，区間競合度とポイント競合度がこれまでに提案されている．

区間競合度　プロセス p_i が，実行区間 $\alpha_1\beta$ のある接頭部 $\alpha_1\beta'$ の最後のイベントにおいてアクティブであるとき，プロセス p_i は区間 β でアクティブであると言う．実行区間 β においてアクティブなプロセスの個数を β の 区間競合度 と呼ぶ．区間競合度を k_i，k_i の上界を n_i と表記する．

ポイント競合度　ある実行区間 $\alpha_1\beta$ の最後においてアクティブなプロセスの個数を，$\alpha_1\beta$ の最後における競合度と言う．実行区間 $\alpha_1\beta$ の全ての接頭部 $\alpha_1\beta'$ の最後における最大の競合度を，区間 β における ポイント競合度 と呼ぶ．ポイント競合度を k，k の上界を n と表記する．

図 3 に，繰り返し M-改名アルゴリズムの実行例を示す．この例では，実行区間 β における区間競合度は 3，ポイント競合度は 2 である．

一般に非同期共有メモリシステムでは，アルゴリズムの実行前に，任意の実行区間における区間競合度やポイント競合度をあらかじめ知ることはできない．しかし，任意の実行に対する各競合度の上界は，あらかじめ与えられているとする．

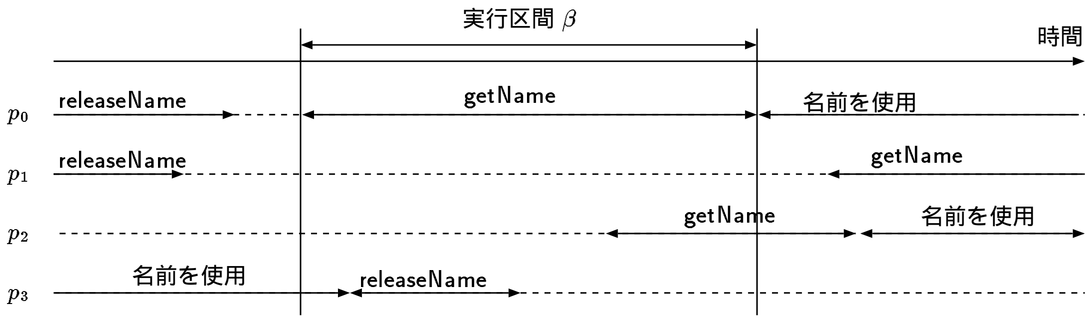

図 3: 区間競合度とポイント競合度

2.4 アルゴリズムの評価尺度

2.4.1 改名後の名前空間の大きさ

改名の動機から明らかなように，改名後の名前空間の大きさ M はできるだけ小さい方が望ましい．M をどこまで小さくできるかは，システムの計算モデルに依存する．例えば，read 操作と write 操作だけを使用できる read/write モデルでは，$M < 2k - 1$ ならば M-改名を無待機で実現するようなアルゴリズムは存在しないことが証明されている [10]．また，複数の read 操作と write 操作を組み合わせた強力な操作である read-modify-write 操作を使用する read-modify-write モデルでは，k-改名アルゴリズムが提案されている [11]．

2.4.2 ステップ計算量

改名を行うプロセスは，共有メモリに対してアクセスする必要がある．一回 M-改名アルゴリズムにおけるステップ計算量は，1 つのプロセスが新しい名前を獲得するために必要な共有メモリへのアクセス回数とする．また，繰り返し M-改名アルゴリズムでは，1 つのプロセスが新しい名前を 1 回獲得し，獲得した名前を 1 回解放するために必要な共有メモリへのアクセス回数とする．

M-改名アルゴリズムのステップ計算量は，システムに属するプロセスの個数 N と競合度 (一回 M-改名アルゴリズムにおける競合度 k_o や，繰り返し M-改名アルゴリズムにおける区間競合度 k_i，またはポイント競合度 k)，そして競合度の上界 (n_o や n_i，または n) の関数で与えられる．ステップ計算量を示す関数の性質に従って，M-改名アルゴリズムは無待機，高速及び適応型の 3 つに分類できる．

ステップ計算量が有界である M-改名アルゴリズムを，無待機 M-改名アルゴリズムと言う．無待機 M-改名アルゴリズムを実行するプロセスは，自らがアルゴリズムの計算ステップを有限回実行することで，改名を完了することが保証される．従って，自分よりも実行速度の速いプロセスに先を越されて完了できないことはなく，自分よりも遅いプロセスの結果を待ち続けることもない．さらに，他のプロセスが故障により停止した場合であっても改名が可能なことから，無待機 M-改名アルゴリズムは，プロセスの停止故障に対して耐性を持つと言える．

無待機 M-改名アルゴリズムの中で，ステップ計算量がプロセス数 N に依存せず，手続き getName の実行区間における競合度の上界に依存するアルゴリズムを，高速 M-改名アルゴリズムと言う．

高速 M-改名アルゴリズムの中で，ステップ計算量が手続き getName の実行区間における競合度のみに依存するアルゴリズムを，適応型 M-改名アルゴリズムと言う．さらに，ステップ計算量が手続き getName の実行区間におけるポイント競合度 k のみに依存する繰り返し M-改名アルゴリズムを，ポイント競合度適応型繰り返し M-改名アルゴリズムと言う．

2.4.3 空間計算量

改名アルゴリズムの実行に必要な共有変数の個数を空間計算量と呼ぶ．当然，空間計算量はできるだけ小さい方が望ましい．各変数の値について，有界であるか非有界であるかに分かれる．非有界であれば，あらかじめどの程度の記憶領域を確保しておけば十分であるか見積もることが困難である．このため，有界である方が望ましい．

3　コレクトリスト

3.1　アルゴリズム

コレクトリストは，n_t 個のスプリッタと呼ばれる基本要素の配列と，3 つの手続き register，collect，clear_list で構成されている．各スプリッタには順に 0 から $n_t - 1$ のレベルを与える (図 4)．n_t については，3.2. において説明する．手続き register ではコレクトリストに属するいずれかのスプリッタへ登録し，collect ではスプリッタへ登録したプロセスの集合 (登録者集合) を収集する，手続き clear_list では，register や collect で使用したスプリッタを初期化する．コレクトリストのコードを図 6 に示す．

各スプリッタに対して，手続き splitter を実行することができる．splitter は，返り値として **stop**，**abort**，もしくは **next** のいずれかを返す．d 個 $(d > 1)$ のプロセスが同一のスプリッタに対して splitter を実行すると，高々1 個のプロセスが **stop** を返され，それぞれ高々 $d - 1$ 個のプロセスが **abort** や **next** を返される (図 5)．d 個すべてのプロセスに同一の値を返されることはなく，あるスプリッタに対して 1 つのプロセスのみが splitter を実行すると **stop** を返される．すなわち，スプリッタを利用することで，同一のスプリッタにアクセスするプロセスの集合を，より小さなプロセスの集合に分割することができる．

手続き register では，レベル 0 のスプリッタから順に手続き splitter を実行する．各レベルのスプリッタに対する手続き splitter から，**stop** が返されるとそのスプリッタへの登録に成功し，そのスプリッタのレベルを返して register を終了する．**abort** が返されるとスプリッタへの登録に失敗し，⊥ を返して register を終了する．**next** が返されると，次のレベルのスプリッタに進み，再び手続き splitter を実行する．

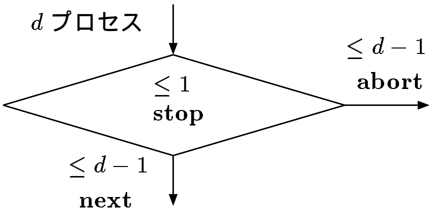

図 5: スプリッタ

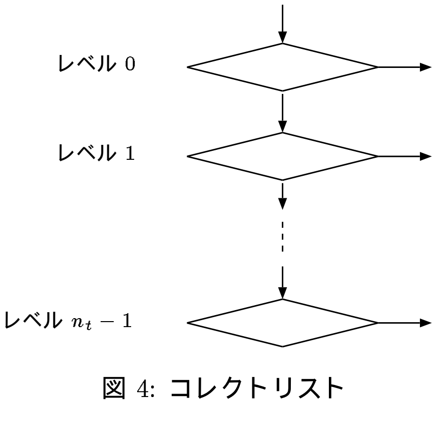

図 4: コレクトリスト

手続き register において，各スプリッタに対して手続き splitter を実行する直前に，そのスプリッタのフラグ $mark$ を立てる．フラグ $mark$ を立てられたスプリッタは，マークされているという．手続き collect では，マークされているスプリッタをレベル 0 から順に調べる．マークされているスプリッタの中で，手続き register を実行したプロセスによってすでに登録されているスプリッタについて，そのスプリッタのレベルと，登録しているプロセスの識別子の二項組を集める．このようにして集められる二項組の集合を登録者集合といい，手続き collect は登録者集合を返す．

コレクトリストには，各スプリッタに任意のデータを持たせることができ，ここでは繰り返し M-改名アルゴリズムで利用する共有変数 $done$ と $view$ を各スプリッタに持たせる．このデータを読み込むためには手続き read_list，書き込むためには write_list を用いる．コレクトリストの使用が終了すると，手続き clear_list を用いることで，コレクトリストを初期化する．初期化を実行することで，再び同一のコレクトリストを初期状態から使用できる．

3.2　コレクトリストの正当性

以下の補題では，コレクトリストに属するすべての共有変数の値が初期値である状況から，初期化の手続き clear_list が最初に開始される直前までの実行 R^{col} における，手続き register や collect，splitter の性質や計算量を示す．ここで，任意の実行 R^{col} において，手続き register を実行するプロセスの個数を k_t，k_t の上界を n_t とする．なお，共有変数 $done$ と $view$ を読み書きするための手続き read_list，write_list や，共有変数 $mark$ を読

図 6: コレクトリスト

```
Shared variables :
    $list$ { // collect list
        $mark[0, ..., n_t - 1]$
            : Boolean, initially **false**;
        $id[0, ..., n_t - 1]$
            : id, initially $\perp$;
        $X[0, ..., n_t - 1]$
            : integer, initially $\perp$;
        $Y[0, ..., n_t - 1]$
            : Boolean, initially **false**;
        $done[0, ..., n_t - 1]$
            : Boolean, initially **false**;
        $view[0, ..., n_t - 1]$
            : set of $\langle$id,integer$\rangle$, initially $\perp$;
        // $done$ and $view$ are the variables
        // used in the renaming algorithm.
    }

Non-shared variables :
    $sp$, $move$ : integer, initially 0;
    $V$ : set of $\langle$id,integer$\rangle$, initially $\emptyset$;

procedure splitter($list, sp$)
    $list.X[sp] := p_i$;
    if ($list.Y[sp]$ = **true**) then return **abort**;
    $list.Y[sp]$ := **true**;
    if ($list.X[sp] = p_i$) then return **stop**;
    else return **next**;

procedure clear_list($list, sp$)
    while ($list.mark[sp]$) do
        write initial value to $sp$ in $list$;
        $sp$++;
    od;
```

```
procedure register($list$)
    $sp := 0$;
    while (**true**) do
        $list.mark[sp]$ := **true**;
        $move$ := splitter($list, sp$);
        if ($move$ = **next**) then $sp$++;
        if ($move$ = **abort**) then return $\perp$;
        if ($move$ = **stop**) then
            $list.id[sp] := p_i$;
            return $sp$;
    od;

procedure collect($list$)
    $sp := 0$; $V := \emptyset$;
    while ($list.mark[sp]$ = **true**) do
        if ($list.id[sp] \neq \perp$) then
            $V := V \cup \{\langle list.id[sp], sp \rangle\}$
        $sp$++;
    od;
    return $V$;

procedure read_mark($list, sp$)
    return $list.mark[sp]$;

procedure read_list($list, sp, var$)
    switch ($var$) {
        case 'done':return $list.done[sp]$;
        case 'view':return $list.view[sp]$;
    }

procedure write_list($list, sp, var, value$)
    switch ($var$) {
        case 'done':$list.done[sp] := value$;
        case 'view':$list.view[sp] := value$;
    }
```

み込むための手続き read_mark は，以下の補題には関係しない．

スプリッタは，2 つの共有変数 X と Y を用いる．初期値はそれぞれ $X = \bot$，$Y = \mathbf{false}$ である．手続き splitter を実行するプロセス p_i は，まず X に自らの識別子 p_i を書き込み，次に Y の値を読み込む．もし $Y = \mathbf{true}$ であれば，p_i は $\mathbf{abort}$ を返す．$Y = \mathbf{false}$ であれば，Y に $\mathbf{true}$ を書き込み，X の値を確認する．X が自らの識別子 p_i を格納していれば $\mathbf{next}$ を返し，そうでなければ $\mathbf{next}$ を返す．アルゴリズムより，次に示すスプリッタの性質が導かれる．

補題 1 任意の実行 R^{col} において，d 個のプロセスが同一のスプリッタにアクセスしたとき，以下の性質が成り立つ．

1. 高々 1 個のプロセスが $\mathbf{stop}$ を返される．
2. 高々 $d-1$ 個のプロセスが $\mathbf{abort}$ を返される．
3. 高々 $d-1$ 個のプロセスが $\mathbf{next}$ を返される．

補題 2 任意の実行 R^{col} において，あるコレクトリストに属するスプリッタ v のレベルが $l(0 \leq l \leq k_t)$ であるとき，高々 $k_t - l$ 個のプロセスが v に対して手続き splitter を実行する．

(証明) スプリッタ v のレベル l についての帰納法で証明する．手続き register を実行するプロセスは，レベル 0 のスプリッタから順に手続き splitter を実行するので，$l = 0$ のときは明らかに補題が成り立つ．次に，レベル $l-1(0 < l < k_t)$ のスプリッタ u について補題が成り立つと仮定する．仮定より，u に対して高々 $k_t - l + 1$ 個のプロセスが splitter を実行する．よって補題 $1(3)$ より，レベル l のスプリッタ v に対して，高々 $k_t - l$ 個のプロセスが splitter を実行する．従って，レベル l のスプリッタ v についても仮定が満たされる．■

補題 2 より，レベル k_t 以上のスプリッタに対して手続き splitter を実行するプロセスは存在しない．従って，手続き register を実行するプロセスは，レベル 0 からレベル $k_t - 1$ のいずれかのスプリッタにおいて，スプリッタへの登録に成功する，もしくは登録に失敗することにより register を終了する．また，補題 $1(1)$ より，各スプリッタへ高々 1 個のプロセスが登録する．

補題 3 任意の実行 R^{col} において，手続き register を実行するプロセスは，レベル $k_t - 1$ 以下のスプリッタへ登録する．また，他のプロセスが同一のスプリッタへ登録することはない．

補題 3 より，手続き register および collect では高々 k_t 個のスプリッタに対して手続き splitter を実行する．また，各スプリッタで必要なステップ計算量，および空間計算量は定数であるので，次の補題が導かれる．

補題 4 任意の実行 R^{col} において，手続き register および collect のステップ計算量は $O(k_t)$，空間計算量は $O(k_t)$ である．

次に，手続き collect が返す登録者集合の性質について述べる．補題 3 より，コレクトリストで使用するスプリッタの数は高々 k_t 個である．補題 $1(1)$ より，各スプリッタへ高々 1 個のプロセスだけが登録する．

補題 5 任意の実行 R^{col} において，手続き collect が返す登録者集合の要素数は高々 k_t である．

補題 6 任意の実行 R^{col} において，プロセス p_i，p_j が同一のコレクトリストに対して手続き collect を実行し，それぞれ登録者集合 V_1，V_2 を返されたとする．p_i による collect が終了した後に，p_j による collect が開始されたなら，$V_1 \subseteq V_2$ が成り立つ．

(証明) 補題 3 より，あるプロセスがスプリッタ sp へ登録するなら，そのプロセス以外が $list.id[sp]$ を更新することはない．また collect では，マークされているスプリッタをレベル 0 から順に調べる．

任意のスプリッタ sp_x をプロセス p_x が登録し，$list.id[sp_x]$ に p_x の識別子を書き込むとする．実行 R^{col} において，$list.id[sp_x]$ に $\bot$ が書き込まれることはない．p_i が sp_x を調べる前に p_x が識別子を書き込む場合，$\langle p_x, sp_x \rangle \in V_1, V_2$ となる．p_i が sp_x を調べた後に p_x が識別子を書き込む場合，$\langle p_x, sp_x \rangle \notin V_1$ かつ $\langle p_x, sp_x \rangle \in V_2$ となる．よって補題が成り立つ．■

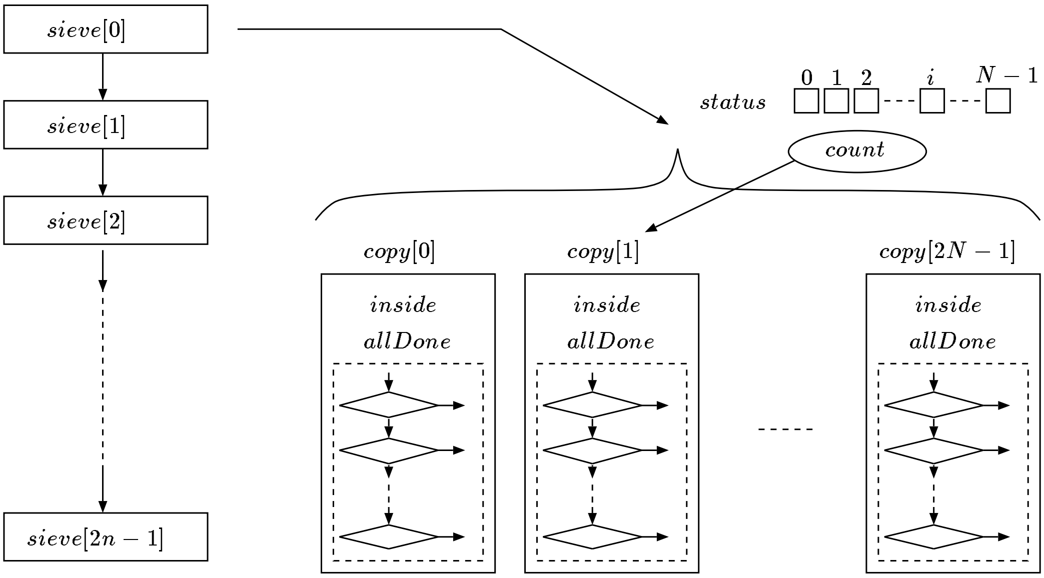

図 7: シーブの概念図

4　改名アルゴリズム

4.1　アルゴリズム

本改名アルゴリズムでは，$0, 1, 2, ..., 2n-2$ と番号付けされたシーブと呼ばれる基本要素の配列を使用して，ポイント競合度適応型繰り返し $k(2k-1)$-改名アルゴリズムを実現する．

本改名アルゴリズムは，名前を獲得するための手続き getName と，獲得した名前を解放するための手続き releaseName からなる．手続き getName を実行するプロセスは，名前の獲得に成功するまで，0 番目のシーブから順に手続き interleaved_sc_sieve を実行する．手続き interleaved_sc_sieve からは手続き sc_sieve が呼ばれる．手続き sc_sieve は，同一のシーブから同時に名前を獲得する可能性のあるプロセスの集合 (候補者集合) を返す．候補者集合に sc_sieve を呼んだプロセスの識別子が含まれているならば，そのプロセスは新しい名前を獲得できる．新しい名前は，この sc_sieve を呼んだシーブの番号と，候補者集合におけるこのプロセスのランクから構成される．

本改名アルゴリズムのコードを，図 8 から 図 10 に示す．

1 つのシーブは $0, 1, ..., 2N-1$ と番号付けされた $2N$ 個のコピーと，共有変数 $count$，$status$ から構成される (図 7)．コピーは手続き sc_sieve を実行するための作業領域であり，$count$ は現在 sc_sieve を受け付けているコピーを指すポインタである．$status$ はこのシーブで獲得した名前を保持している，もしくは名前の獲得を試みているアクティブなプロセスを示す配列である．

各コピーは，共有変数 $inside$，$allDone$ と前節で述べたコレクトリスト $list$ からなる．なお，コレクトリストに属する各スプリッタには，共有変数 $view$ と $done$ を持たせている．

あるシーブ s における手続き sc_sieve では，共有変数 $sieve[s].count$ が指すコピー c に対して手続き open を実行する．open では，このときコピー c を使用しているプロセスがまだ存在せず，かつ直前のコピー $c-1 \bmod 2N$ における候補者集合に含まれているすべてのプロセスが終了してるかをチェックする．条件を満たさなければ sc_sieve は空集合を返す．この条件を満たせば，共有変数 $inside$ に **true** を書き込んだ後に，コピー c のコレクトリストに対して手続き register によるスプリッタへの登録を試みる．登録できなければ sc_sieve は空集合を返す．登録できれば手続き partial_scan を呼び出し，コピー c のコレクトリストに属するスプリッタへ登録したプロセスの集合 (スナップショット) をとる．このときスナップショットをとることができたならば，計算したスプリッタが持つ共有変数 $view$ にスナップショットを格納する．

図 8: 改名アルゴリズム (1/3)

```
Shared variables :
    $sieve[0, ..., 2n-1]$ {
        $count$ : ⟨integer,{0, 1}⟩, initially ⟨0, 1⟩;
        $status[0, ..., N-1]$ : Boolean, initially **false**;
        $copy[0, ..., 2N-1]$ {
            $allDone$ : {0, 1}, initially 1;
            $inside$ : Boolean, initially **false**;
            $list$ : collect list;
    }}

Non-shared Global variables :
    $s$, $c$ : integer, initially 0;
    $dirtyB$ : {0, 1}, initially 0;
    $mysplitter$ : integer, initially $\perp$;
    $W$ : set of ⟨ id,integer ⟩, initially $\emptyset$;

procedure getName()
    $s := 0$;
    while (**true**) do
        $s{+}{+}$;
        $sieve[s].status[i]$ := **active**;
        $\langle c, dirtyB \rangle := sieve[s].count$;
        if (($c \bmod N = i$) or ($sieve[s].status[c \bmod N]$ = **idle**)) then
            $W :=$ interleaved_sc_sieve($s, c, dirtyB$);
            if ($\langle p_i, * \rangle \in W$) then
                $sieve[s].count := \langle c+1 \bmod 2N$,next_dirtyB($dirtyB, c$)⟩;
                return ⟨$s$,rank of $p_i$ in $W$⟩;
            else-if ($sieve[s].copy[c].allDone$ = next_dirtyB($dirtyB, c$)) then
                clear($s, c, 0$);
        $sieve[s].status[i]$ := **idle**;
    od;

procedure releaseName()
    leave($s, c, dirtyB$);
    if ($sieve[s].copy[c].allDone$ = next_dirtyB($dirtyB, c$)) then clear($s, c, 0$);
    $sieve[s].status[i]$ := **idle**;
```

図 9: 改名アルゴリズム (2/3)

```
Shared variables :
    $last\_modified$ : points to last shared variable modified by $p_i$;
            // $last\_modified$ is assumed to be updated immediately before the write.

procedure interleaved_sc_sieve($s, c, dirtyB$)
            // interleave is a two part construct. Part I of the interleave is executed after every
            // read or write to a shared variable in Part II,the sc_sieve() and any procedure
            // recursively called from sc_sieve().
    $last\_modified := \bot$;
    interleave { // Part I
        if ($sieve[s].copy[c].allDone =$ next_dirtyB($dirtyB, c$)) then
            if ($last\_modified$ is a variable $mark$ in some splitter of $sieve[s].copy[c].list$) then
                clear($s, c$, level of $last\_modified$);
            else-if ($last\_modified \neq \bot$) then
                write initial value to $last\_modified$;
            return $\emptyset$;                // abort current sc_sieve(), and continue to next sieve.
    }{ // Part II
        return sc_sieve($s, c, dirtyB$);
    }

procedure sc_sieve($s, c, dirtyB$)
    if (open($s, c, dirtyB$) = **true**) then
        $sieve[s].copy[c].inside$ := **true**;
        $mysplitter$ := register($sieve[s].copy[c].list$);
        if ($mysplitter \neq \bot$) then
            $V$ := partial_scan($sieve[s].copy[c].list$);
            write_list($sieve[s].copy[c].list, mysplitter$, 'view', $V$);
            $W$ := candidates($s, c$);
            if ($\langle p_i, mysplitter \rangle \in W$) then return $W$;
            write_list($sieve[s].copy[c].list, mysplitter$, 'done', **true**);
            $W$ := candidates($s, c$);
            leave($s, c, dirtyB$);
    return $\emptyset$;

procedure next_dirtyB($dirtyB, c$)
    if ($c = 0$) then return $\neg dirtyB$;
    else return $dirtyB$;
```

図 10: 改名アルゴリズム (3/3)

```
procedure leave($s, c, dirtyB$)
write_list($sieve[s].copy[c].list, mysplitter, \text{'done'}, \mathbf{true}$);
if ($W \neq \emptyset$ and for every $\langle p_j, sp \rangle \in W$,
        read_list($sieve[s].copy[c].list, sp, \text{'done'}$) $= \mathbf{true}$) then
    $sieve[s].copy[c].allDone :=$ next_dirtyB($dirtyB, c$);

procedure partial_scan($list$)
$V_1 :=$ collect($list$);
$V_2 :=$ collect($list$);
if ($V_1 = V_2$) then return $V_1$;
else return $\emptyset$;

procedure candidates($s, c$)
$sp := 0$; $V := \emptyset$;
while (read_mark($sieve[s].copy[c].list, sp$)) do
    $v :=$ read_list($sieve[s].copy[c].list, sp, \text{'view'}$);
    if ( $v \neq \perp$ and $v \neq \emptyset$) then $V := V \cup \{v\}$;
    $sp{+}{+}$;
od;
if ($V = \emptyset$) then return $\emptyset$;
$U := \min\{view | view \in V\}$;
if (for every $\langle p_j, sp \rangle \in U$, read_list($sieve[s].copy[c].list, sp, \text{'view'}$) $\supseteq U$
        or read_list($sieve[s].copy[c].list, sp, \text{'view'}$) $= \emptyset$) then
    return $U$;
else
    return $\emptyset$;

procedure open($s, c, dirtyB$)
if (($sieve[s].copy[c-1 \bmod 2N].allDone = dirtyB$) and ($sieve[s].copy[c].inside = \mathbf{false}$)) then
    return $\mathbf{true}$;
else
    return $\mathbf{false}$;

procedure clear($s, c, sp$)
$sieve[s].copy[c].inside := \mathbf{false}$;
clear_list($sieve[s].copy[c].list, sp$);
```

スナップショットをとった後に，各プロセスのスナップショットを比較して，最小のスナップショットを計算する．最小のスナップショットは，このコピーにおける候補者集合となる．候補者集合を獲得できたプロセスが，候補者集合に含まれているならば，このプロセスは名前の獲得に成功する．候補者集合に含まれていなければ，手続き leave を呼び，登録したスプリッタに持たせた共有変数 $done$ に **true** を書き込み，シーブ s での手続きを終了する．手続き leave では，候補者集合に含まれるすべてのプロセスが，すでに共有変数 $done$ へ **true** を書き込んでいたなら，共有変数 $allDone$ を手続き next_dirtyB が返す値に更新する．

手続き releaseName では，登録したスプリッタに持たせた共有変数 $done$ に **true** を書き込み，名前を解放したことを他のプロセスに知らせる．

4.2 コピーの再使用

本改名アルゴリズムでは，空間計算量を有界にするために，コピーを再使用する．つまり，コピー $2N-1$ を使用した次は，コピー 0 を使用する．コピーの再使用を実現するためには，コピーの使用が終了した後に，そのコピーに属するすべての共有変数にそれぞれの初期値を書き込むことで，コピーを初期化しなければならない．このための手続きが clear であり，コピーの初期化が終了すると，同一のコピーを再び使用することができる．

コピーの使用が終了したことは，共有変数 $allDone$ の値の変化によって検知する．$allDone$ からコピーの使用が終了したことを示す値を読み込んだプロセス，もしくは $allDone$ をその値に更新したプロセスは，コピーの初期化を開始する．しかし，コピーの初期化をする際には，次の 2 点に注意しなければならない．

1 つ目は，あるプロセスがコピーを初期化しているときに，他のプロセスがそのコピーに属する変数の値を読み書きする可能性があることである．このようなプロセスは矛盾した値を読み込み，予期しない動作をする可能性がある．そこで，手続き interleaved_sc_sieve，および手続き interleaved_sc_sieve から呼ばれる手続きにおいて，共有変数の値を読み書きする毎に $allDone$ の値を確認する．もし，$allDone$ の値が更新されていたなら，手続き sc_sieve を中断し，直前に書き込んだ共有変数には初期値を書き込む，もしくは手続き clear を実行する．これを インタリーブ機構と呼ぶ．この機構により，$allDone$ が新しい値に更新された後には，各プロセスはそのコピーにおいて高々1 ステップしか実行しないので，コピーの初期化を保証できる．

2 つ目は，コピーの初期化において単に $allDone$ に初期値を書き込むのでは，コピーの使用が終了したことを正しく検知することができない可能性がある．そこで，$allDone$ は 2 値変数とし，コピーの使用が終了するとこの値を入れ換える．コピーの使用が終了しているのかどうかは，$allDone$ の値と $count$ の第二項から読み込んだ値を比較することで判別できる．

4.3 改名アルゴリズムの正当性

改名アルゴリズムの実行 R はイベントの系列 $e_0, e_1, e_2, ...$ である．R に属するイベントの中で，あるプロセス p_i が実行する手続き getName の開始から，それに対応する releaseName の最後までのイベント列を β とする．β において，p_i が局所変数 s に x を書き込むイベントから $x+1$ を書き込むイベントの直前までを，シーブアクセス A^x_β とし，このとき p_i がシーブ x にアクセスするという．$x+1$ を書き込むイベントが存在しない場合は，A^x_β は実行 β の最後までとする．つまり，p_i の実行 β はアクセスするシーブごとに分割でき，$\beta = A^0_\beta, A^1_\beta, ...$ と書ける．以下の議論では，A^x_β を A^x と省略することもある．

改名アルゴリズムの実行 R に属するイベントの中で，シーブ s にアクセスしているプロセスが実行するイベントの部分系列を，シーブ s の実行 $R^s = e^s_0, e^s_1, e^s_2, ...$ とする．このとき，R^s に属するイベント $t^s_0, t^s_1, ..., t^s_{l-1}, t^s_l, ...$ を，次のように帰納的に定義する．

- $t^s_0 = e^s_0$
- $l \geq 1$ のとき t^s_{l-1} が存在するなら，t^s_{l-1} より後に局所変数 c に $l \bmod 2N$ を読み込む最初のイベントを t^s_l とする．

このとき，イベント t^s_l から t^s_{l+1} の直前までを，

シーブ s の実行 R^s の区間 l とする．t^s_{l+1} が存在しなければ，R^s は l 番目の区間で終了している．

プロセス p_i が実行するシーブアクセス A^s において，$sieve[s].count$ を局所変数 $\langle c, dirtyB\rangle$ に読み込むイベントを $e \in A^s$ とする．このとき，R^s においてイベント e が属する区間を $int(A^s)$ とし，$int(A^s) = l$ を満たすシーブアクセスは R^s の区間 l に属すると言う．また，イベント e において，局所変数 c に読み込む値を $rv(A^s)$，局所変数 $dirtyB$ に読み込む値を $rd(A^s)$ とする．

以下の用語を，本改名アルゴリズムの正当性の証明で用いる．

- プロセス p_i が実行するシーブアクセス A^s に属するイベント e の直前において，共有変数 $sieve[s].status[i] = \textbf{active}$ ならば，A^s は e においてアクティブであると言う．
- プロセス p_i が実行するシーブアクセス A^s に属するイベント e の直前において，局所変数 $c = x$ ならば，A^s は e においてシーブ s のコピー x を使用していると言う．
- プロセス p_i が実行するシーブアクセス A^s が $int(A^s) \bmod 2N = x$ を満たすとする．イベント $e \in A^s$ の直前までに，すでに p_i がシーブ s のコピー x の共有変数 $inside$ に **true** を書き込んでいるなら，A^s は e においてシーブ s のコピー x に入っていると言う．
- プロセス p_i が実行するシーブアクセス A^s が，コピー c のコレクトリストに属するあるスプリッタ sp_i へ登録するとする．このとき，イベント $e \in A^s$ においてスプリッタ sp_i が持つ変数 $done$ に **true** を書き込んでいるなら，A^s は e において完了していると言う．
- 手続き partial_scan が返す空でない集合をスナップショット，手続き candidates が返す空でない集合を候補者集合と言う．
- プロセス p_i がシーブアクセス A^s においてスプリッタ sp_i へ登録し，手続き candidates から候補者集合 W を返されたとする．このとき $\langle p_i, sp_i\rangle \in W$ ならば，p_i を区間 $int(A^s)$ における勝利プロセス，A^s を区間 $int(A^s)$ における勝利シーブアクセスと言う．

補題 7 任意のシーブ s の実行 R^s において，以下の性質が成り立つ．

1. 任意のシーブアクセス A^s が，イベント $e \in A^s$ においてアクティブであり，e が R^s の区間 l に属しているなら，$int(A^s) \geq l - N$．
2. 任意のシーブアクセス A^s が，イベント $e \in A^s$ の直前においてすでに $sieve[s].count$ の値を読み込んでいるなら，$rv(A^s) = int(A^s) \bmod 2N$．
3. 任意のシーブアクセス A^s が，イベント $e \in A^s$ の直前においてすでに $sieve[s].count$ の値を読み込んでいるなら，$rd(A^s) = db(int(A^s) - 1)$．

   ここで，$x \geq 0$ のとき $db(x) = (x \operatorname{div} 2N) \bmod 2$，$x < 0$ のとき $db(x) = 1$ であり，div は商を表す．
4. 任意のシーブアクセス A^s が，イベント $e \in A^s$ においてあるコピーに入っているとする．このとき，$int(A^s) = l \geq 1$ ならば，区間 $l-1$ に属し，e の直前までに $sieve[s].copy[l - 1 \bmod 2N].allDone$ に $db(l-1)$ を書き込んだシーブアクセスが少なくとも *1* つ存在する．
5. 任意のシーブアクセス A^s が，イベント $e \in A^s$ においてあるコピーに入っているとする．このとき，e において $int(A^s_w) < int(A^s)$ を満たす任意の勝利シーブアクセス A^s_w は完了している．
6. イベント t^s_l が存在するなら，t^s_l の直前にシーブ s のコピー $l \bmod 2N$ を使用しているアクティブなシーブアクセスは存在しない．
7. イベント t^s_l が存在するなら，t^s_l の直前にシーブ s のコピー $l \bmod 2N$ に属する任意の共有変数は初期値である．
8. シーブアクセス A^s_1，A^s_2 が手続き partial_scan を実行し，スナップショット V_1，V_2 をそれぞれ返されたとする．このとき，$int(A^s_1) = int(A^s_2)$ ならば，$V_1 \subseteq V_2$ もしくは $V_2 \subseteq V_1$ を満たす．
9. シーブアクセス A^s_1，A^s_2 が手続き candidates を実行し，候補者集合 W_1，W_2 をそれぞれ返されたとする．このとき，$int(A^s_1) = int(A^s_2)$ ならば，$W_1 = W_2$ を満たす．

(証明) 任意のシーブ s の実行 R^s の長さに関する帰納法で補題を証明する．まず，R^s の最初のイベント e_0^s において，補題 7 が成り立つことを示す．

1. イベント e_0^s がシーブアクセス A^s に属しているとする．e_0^s において A^s は明らかにアクティブではないので，e_s^s において *1.* が成り立つ．
2. イベント e_0^s がシーブアクセス A^s に属しているとする．e_0^s では A^s はまだ $sieve[s].count$ を読み込んでいないので，e_0^s において *2.* が成り立つ．
3. イベント e_0^s がシーブアクセス A^s に属しているとする．e_0^s では A^s はまだ $sieve[s].count$ を読み込んでいないので，e_0^s において *3.* が成り立つ．
4. イベント e_0^s は R^s の最初のイベントなので，イベント e_0^s において，コピーに入っているシーブアクセスは存在しない．よって，e_0^s において *4.* が成り立つ．
5. イベント e_0^s は R^s の最初のイベントなので，イベント e_0^s において，勝利シーブアクセスは存在しない．よって，e_0^s において *5.* が成り立つ．
6. イベント $t_0^s = e_0^s$ は R^s の最初のイベントなので，t_0^s の直前に，シーブ s のコピー 0 を使用しているアクティブなシーブアクセスは存在しない．よって，t_0^s において *6.* が成り立つ．
7. イベント $t_0^s = e_0^s$ は R^s の最初のイベントなので，t_0^s の直前に，シーブ s のコピー 0 に属する任意の共有変数は初期値である．よって，t_0^s において *7.* が成り立つ．
8. イベント e_0^s は R^s の最初のイベントなので，e_0^s において，すでにスナップショットを返しているシーブアクセスは存在しない．よって，e_0^s において *8.* が成り立つ．
9. イベント e_0^s は R^s の最初のイベントなので，e_0^s において，すでに候補者集合を返しているシーブアクセスは存在しない．よって，e_0^s において *9.* が成り立つ．

次に帰納段として，任意のイベント $e \in R^s$ の直前において補題 7 が成り立つと仮定し，e においても成り立つことを示す．

1. あるシーブアクセス A^s が，イベント $e \in A^s$ おいてアクティブであり，e は R^s の区間 l に属しているとし，背理法で証明する．A^s が $int(A^s) < l - N$ を満たすと仮定する．

   N は正の定整数なので，$int(A^s) < l' < l$ かつ $l' \bmod N = i$ を満たす l' が存在する．アルゴリズムより，A^s は区間 $int(A^s)$ で $sieve[s].count$ を読み込み，その後区間 l のイベント e までは少なくともアクティブである．よって A^s は，区間 l' に含まれるすべてのイベントにおいてアクティブである．

   区間 l' で共有変数 $sieve[s].count$ を読み込む，任意のシーブアクセス A_1^s を考える．1 つのシーブアクセスにおいて，$sieve[s].count$ を読み込むのは 1 度だけなので，A^s と A_1^s は異なるシーブアクセスである．定義より $int(A_1^s) = l'$ なので，*2.* より $rv(A_1^s) = int(A_1^s) \bmod 2N = l' \bmod 2N$．このため，$A_1^s$ は $sieve[s].count$ の第一項から $l' \bmod 2N$ を読み込むが，区間 l' で A^s はアクティブなので，A_1^s は共有変数 $sieve[s].count$ の第一項を $l' + 1 \bmod 2N$ に更新することはない．また *6.* より，イベント $t_{l'}^s$ の直前にコピー $l' \bmod 2N$ を使用しているアクティブなシーブアクセスは存在しない．よって，イベント $t_{l'}^s$ 以降に $sieve[s].count$ の第一項へ $l' + 1 \bmod 2N$ を書き込むシーブアクセスは存在しないので，区間は $l' + 1$ 以上にならない．これは e が区間 l に属するという仮定に矛盾するので，*1.* が成り立つ．
2. 背理法で証明する．$int(A^s) = l$ を満たすシーブアクセス A^s が，イベント $e \in A^s$ において $rv(A^s) \neq int(A^s) \bmod 2N$ を満たすと仮定する．この仮定は，イベント t_l^s が発生してから A^s が $sieve[s].count$ の値を読み込むまでに，$sieve[s].count$ の値を書き換えた勝利シーブアクセス A_1^s が存在し，その値を A^s が読み込んだことを示している．

   背理法の仮定より $int(A_1^s) \leq l$．$int(A_1^s) = l$ とすると，帰納法の仮定より $int(A_1^s) \bmod 2N =$

$rv(A_1^s) = l \bmod 2N$ なので，A_1^s は $sieve[s].count$ の第一項に $l+1 \bmod 2N$ を書き込む．このため $rv(A^s) = l+1 \bmod 2N$ となるので，$int(A^s) = l+1$ となる．また $int(A_1^s) = l-1$ とすると，同様に $rv(A^s) = l \bmod 2N$ となるので，$int(A^s) = l$ となる．これでは背理法の仮定にそれぞれ矛盾するので，$int(A_1^s) = l' < l-1$ である．

区間 l が存在するためには，区間 $l'+1$ における勝利シーブアクセス A_2^s が存在し，共有変数 $sieve[s].count$ の第一項に $l'+2 \bmod 2N$ を書き込む必要がある．A_2^s が区間 $l'+1$ においてコピーに入るとき，*5.* より区間 l' における勝利シーブアクセスは，A_1^s も含みすべて完了しているはずである．これは，区間 l において A_1^s が共有変数 $sieve[s].count$ を書き換えるという仮定に矛盾するので，*2.* が成り立つ．

3. $int(A^s) = l$ を満たすシーブアクセス A^s が，イベント e を含むとする．このとき，$e = t_l^s$ である，もしくは e の直前までにイベント t_l^s が存在する．アルゴリズムより，共有変数 $sieve[s].count$ の値を更新するのは，各区間における勝利シーブアクセスのみである．よって，$l \bmod 2N$ を共有変数 $sieve[s].count$ の第一項に書き込んだ，区間 $l-1$ における勝利シーブアクセス A_w^s が少なくとも 1 つ存在する．

   帰納法の仮定より $rd(A_w^s) = db(int(A_w^s)-1) = db(l-2)$ であり，A_w^s は手続き next_dirtyB が返す値を $sieve[s].count$ の第二項に書き込む．もし，$l-1 \bmod 2N = 0$ であれば next_dirtyB は $\neg db(l-2)$ を返すので，$\neg db(l-2) = \neg((l-2) \operatorname{div} 2N) \bmod 2 = ((l-1) \operatorname{div} 2N) \bmod 2 = db(l-1)$．また，$l-1 \bmod 2N \neq 0$ であれば next_dirtyB は $db(l-2)$ を返すので，$db(l-2) = ((l-2) \operatorname{div} 2N) \bmod 2 = ((l-1) \operatorname{div} 2N) \bmod 2 = db(l-1)$．

   6. より，イベント t_{l-1}^s の直前にコピー $l-1 \bmod 2N$ を使用しているアクティブなシーブアクセスは存在しない．よって，$int(A_1^s) \equiv l-1 \pmod{2N}$ かつ $int(A_1^s) < l-1$ を満たすシーブアクセス A_1^s が，イベント t_{l-1}^s 以降に共有変数 $sieve[s].count$ の第二項に $\neg db(l-1)$ を書き込むことはない．

   以上の議論により，イベント e において $rd(A^s) = db(l-1)$ が言えるので，*3.* が成り立つ．

4. イベント $e \in A^s$ において，シーブアクセス A^s がコピーに入っていないならば *4.* が成り立つ．以下では，イベント $e \in A^s$ において，A^s がコピーに入っているときについて考える．

   アルゴリズムより，$int(A^s) \bmod 2N = c-1$ を満たすシーブアクセスのみが，$sieve[s].copy[c-1].allDone$ を更新することができる．$l-2N-1 \geq 0$ の場合，帰納法の仮定より，区間 $l-2N-1$ に属し，$sieve[s].copy[c-1].allDone$ に $db(l-2N-1)$ を書き込むシーブアクセス $A_{w'}^s$ が少なくとも 1 つ存在する．このシーブアクセスは，*6.* よりイベント t_{l-1}^s の直前までにアクティブではなくなる．ここで，$db(l-2N-1) = ((l-2N-1) \operatorname{div} 2N) \bmod 2 = \neg((l-1) \operatorname{div} 2N) \bmod 2 = \neg db(l-1)$ なので，t_{l-1}^s の直前に $sieve[s].copy[c-1].allDone$ の値は $\neg db(l-1)$ である．$l-2N-1 < 0$ の場合，アルゴリズムより，t_{l-1}^s の直前に $sieve[s].copy[c-1].allDone$ の値はは初期値 $1 = \neg db(l-1)$ である．

   2. より $rv(A^s) = int(A^s) \bmod 2N = l \bmod 2N$ なので，A^s はコピー $c = l \bmod 2N$ に入っている．*3.* より $rd(A^s) = db(l-1)$．従って，A^s は手続き enter の 97 行目において，共有変数 $sieve[s].copy[c-1].allDone$ から $db(l-1)$ を読み込んだと言える．従って，区間 $l-1$ に属し，$sieve[s].copy[c-1].allDone$ に $db(l-1)$ を書き込むシーブアクセス A_w^s が少なくとも 1 つ存在し，その値を A^s が読み込んだと言える．

5. イベント $e \in A^s$ において，シーブアクセス A^s がコピーに入っていないならば *5.* が成り立つ．以下では，イベント $e \in A^s$ において，A^s がコピーに入っているときについて考える．

   $int(A^s) = l$ とする．*4.* より，区間 $l-1$ に属し，イベント e の直前までに $sieve[s].copy[c-1].allDone$ に $db(l-1)$ を書き込んだシーブアクセスが，少なくとも 1 つ存在する．このシーブアクセスを A_1^s とする．

   A_1^s が実行した手続き candidates が返す候補者集合を W とすると，*9.* より区間 $l-1$ にお

ける任意の勝利プロセスは W に属する．A^s_1 は，任意の要素 $\langle p_j, sp_j\rangle \in W$ について，スプリッタ sp_j が持つ変数 $done$ から $\mathbf{true}$ を読み込む．よって A^s がコピー $l \bmod 2N$ に入るとき，区間 $l-1$ におけるすべての勝利プロセスは完了している．帰納法の仮定より，イベント e において $int(A^s) < l-1$ を満たす任意の勝利シーブアクセスは完了しているので，イベント e において *5.* が成り立つ．

6. 任意の区間 l $(l > 0)$ に対して，イベント $e \neq t^s_l$ ならば *6.* が成り立つ．以下では，$e = t^s_l (l \geq 1)$ のときについて背理法で証明する．

t^s_l の直前において，すなわち区間 $l-1$ においてコピー $l \bmod 2N$ を使用しているシーブアクセス A^s が存在すると仮定する．アルゴリズムより $rv(A^s) = l \bmod 2N$，*2.* より $rv(A^s) = int(A^s) \bmod 2N$，さらに背理法の仮定より $int(A^s) < l$ なので，$int(A^s) \leq l-2N$ である．ところが，*1.* より $int(A^s) \geq (l-1)-N$．よって矛盾するので，*6.* が成り立つ．

7. 任意の区間 l $(l > 0)$ に対して，イベント $e \neq t^s_l$ ならば *7.* が成り立つ．以下では，$e = t^s_l (l \geq 1)$ のときについて議論する．

$l < 2N$ ならば，イベント t^s_l の直前までにコピー $c = l \bmod 2N$ を使用したシーブアクセスは存在しないので，*7.* が成り立つ．

$l \geq 2N$ ならば，*4.* より区間 $l-2N$ に属し，$sieve[s].copy[c].allDone$ に $db(l-2N)$ を書き込むシーブアクセスが少なくとも 1 つ存在する．ここで，$c = l \bmod 2N$．このシーブアクセスを A^s とすると，A^s は続いて手続き clear を実行することで，コピー c に属する共有変数を初期化する．つまり $sieve[s].copy[c].inside$ や，レベル 0 以上のマークされているスプリッタを初期化する．さらに，区間 $l-2N$ に属するシーブクアクセスの中で，手続き clear を実行する任意のシーブアクセスを A^s_c とする．*2.* より $int(A^s_c) = l-2N$ なので，任意のイベント $e_c \in A^s_c$ において A^s_c がアクティブであるなら，*1.* より e_c が属する区間は高々 $l-2N+N = l-N$ である．従って A^s_c による手続き clear は，イベント t^s_l の直前までに終了する．

次に，区間 $l-2N$ で使用されたすべてのスプリッタが，手続き clear_list によって初期化されることを背理法で示す．区間 $l-2N$ に属するシーブアクセスがマークしたスプリッタの中で，イベント t^s_l の直前に初期化されていないスプリッタが存在すると仮定し，このようなスプリッタの中で最も低いレベルのスプリッタを sp_d (レベル l_d) とする．$l_d = 0$ の場合，明らかに A^s が sp_d を初期化する．$l_d > 0$ の場合，sp_d の変数 $mark$ が初期化されているなら，$mark$ に初期値を書き込んだシーブアクセスが sp_d を初期化する．sp_d の変数 $mark$ が初期化されていないなら，仮定よりレベル $l_d - 1$ のスプリッタ sp_{d-1} は初期化されているので，sp_{d-1} を初期化したシーブアクセスが続いて sp_d も初期化する，もしくは，sp_d の $mark$ に $\mathbf{true}$ を書き込むシーブアクセスが，インタリーブ機構により sp_d を初期化する．よって背理法の仮定に矛盾するので，区間 $l-2N$ で使用されたすべてのスプリッタは初期化される．

以上の議論により，イベント e において *7.* が成り立つ．

8. 一般性を失うことなく，A^s_1 が V_1 を返された後に，A^s_2 が V_2 を返されると仮定する．イベント e の直前までに V_1 と V_2 の両方がすでに返されている場合には，*8.* が成り立つ．イベント e の直前までに V_1 が返されており，e において V_2 が返された場合について考える．

$int(A^s_1) = int(A^s_2) = l$ なので，*2.* より A^s_1 と A^s_2 は同一のコピー $l \bmod 2N$ を用いる．*7.* より，イベント t^s_l の直前にコピー $l \bmod 2N$ に属する任意の共有変数は初期値である．よって，手続き partial_scan の実行では，初期化された状態から始まるコレクトリストの実行 R^{col} において，手続き collect が 2 回呼ばれる．A^s_1 の実行において呼ばれる collect の実行を順に cop^1_i，cop^2_i とする．A^s_2 の実行においても同様に cop^1_j，cop^2_j とする．さらに，collect の実行で返される登録者集合をそれぞれ v^1_i，v^2_i，v^1_j，v^2_j とする．

一般性を失うことなく，cop^1_i が cop^1_j よりも先

に終了すると仮定する．このとき，cop_i^1 が終了した後に cop_j^2 が開始されると言えるので，補題 6 より $v_i^1 \subseteq v_j^2$ が成り立つ．アルゴリズムより $V_1 = v_i^1 = v_i^2$，$V_2 = v_j^1 = v_j^2$ なので，$V_1 \subseteq V_2$ が成り立つ．よって，イベント e において *8.* が成り立つ．

9. 一般性を失うことなく，A_1^s において W_1 が返された後に，A_2^s において W_2 が返されると仮定する．イベント e の直前までに W_1 と W_2 の両方がすでに返されている場合には，*9.* が成り立つ．

イベント e の直前までに W_1 が返されており，e において W_2 が返された場合について，背理法で証明する．$W_1 \neq W_2$ であると仮定する．$int(A_1^s) = int(A_2^s) = l$ なので，*7.* より，イベント t_l^s の直前にコピー $l \bmod 2N$ に属する任意の共有変数は初期値である．よってアルゴリズムより，W_1 と W_2 は区間 l に属するシーブアクセスが実行した手続き partial_scan によって返されたスナップショットでもあるので，*8.* より $W_1 \subset W_2$ もしくは $W_1 \supset W_2$ が成り立つ．ここでは，一般性を失うことなく $W_1 \subset W_2$ と仮定する．

手続き partial_scan を実行して W_1 を獲得するプロセスを p_i，手続き candidates を実行して W_2 を獲得するプロセスを p_j とする．*2.* より A_1^s と A_2^s は同一のコピー $l \bmod 2N$ を用いる．p_i はスプリッタ sp_i へ登録した後に partial_scan を実行するので，少なくとも p_i は W_1 に属している．$c = l \bmod 2N$ とすると，区間 l において変数 $sieve[s].copy[c].view[sp_i]$ を更新するプロセスは p_i のみなので，その値は $\perp$ もしくは W_1 のみである．しかし，p_j が candidates において W_2 を返すためには，$sieve[s].copy[c].view[sp_i] \supseteq W_2$ もしくは $sieve[s].copy[c].view[sp_i] = \emptyset$ を満たす値が，$sieve[s].copy[c].view[sp_i]$ に格納されていなければならない．よって，背理法の仮定に矛盾するので，イベント e において *9.* が成り立つ．

■

定理 1 では補題 7 を用いて，複数のプロセスが同時に同一の名前を保持することはないことを示す．

定理 1 プロセス p_i，p_j が同時にそれぞれ名前 y_i，y_j を保持しているとき $(i \neq j)$，y_i と y_j は異なる．

(証明) 手続き getName が返す新しい名前は，名前の獲得に成功したシーブの番号と，そのときの候補者集合におけるプロセスのランクの二項組からなる．よって，プロセス p_i と p_j が異なるシーブから名前を獲得したときは，明らかに名前 y_i と y_j は異なる．

次に，プロセス p_i と p_j が，同一のシーブ s から名前を獲得したときを考える．p_i の勝利シーブアクセス A_1^s が名前 y_i を返し，p_j の勝利シーブアクセス A_2^s が名前 y_j を返すとする．もし $int(A_1^s) = int(A_2^s)$ ならば，補題 7-*9.* より p_i と p_j は同一の候補者集合を得る．よって，同一の候補者集合における各プロセスのランクは相異なるので，y_i と y_j は異なる．$int(A_1^s) \neq int(A_2^s)$ ならば (一般性を失うことなく $int(A_1^s) < int(A_2^s)$ と仮定する)，補題 7-*5.* よりシーブ s の区間 $int(A_1^s)$ におけるすべての勝利プロセスが完了しなければ，A_2^s がシーブ s のコピー $int(A_2^s) \bmod 2N$ に入ることができない．よって，プロセス p_i，p_j が同時にそれぞれ名前 y_i，y_j を持つことはない． ■

本改名アルゴリズムで用いるシーブの数の上界を示すために，以下の補題 8 から補題 12 を示す．これらの補題により，本改名アルゴリズムが与える新しい名前の範囲 (定理 2) や，本改名アルゴリズムの計算量 (定理 3) を示すことができる．

補題 8 任意のシーブ s の区間 l に属し，かつ手続き partial_scan を実行したシーブアクセスの中の少なくとも *1* つはスナップショットを獲得する．

(証明) 背理法で証明する．シーブ s に区間 l に属し，かつ手続き partial_scan を実行したすべてのシーブアクセスが，スナップショットを獲得できないと仮定する．補題 7-*2.* より，これらの中の任意のシーブアクセス A^s は，$rv(A^s) = int(A^s) \bmod 2N = l \bmod 2N$ を満たすので，コピー $c = l \bmod 2N$ を使用する．背理法の仮定より，コピー c のコレクトリストに属する任意のスプリッタ sp について，sp が持つ変数 $view$ へスナップショットを書き込

むシーブアクセスは存在しない．さらに，区間lに属するシーブアクセスの中で，手続きcandidatesを実行して候補者集合を返されるものは存在しない．このため，$sieve[s].copy[c].allDone$の値を更新するシーブアクセスが存在しないので，手続きclearを実行するシーブアクセスは存在しない．

区間lに属するシーブアセクスの中で，手続きregisterにおいてスプリッタへ登録し，スプリッタの変数idに識別子を書き込む最後のシーブアクセスをA_1^sとする．A_1^sは続いて手続きpartial_scanを実行するが，このとき任意のスプリッタの変数idが更新される，もしくは初期化されることはない．よって，partial_scanから呼ばれる手続きcollectは同一の登録者集合を返すので，A_1^sはスナップショットを得ることができる．これは背理法の仮定に矛盾している． ■

補題 9 任意のシーブsの区間lに属し，かつコピーに入ったシーブアクセスの中の少なくとも*1*つは勝利シーブアクセスとなる．

(証明) 補題8より，シーブsの区間lに属するシーブアクセスの中の少なくとも1つは，手続きpartial_scanを実行することでスナップショットを獲得する．これらのスナップショットの中で，プロセスp_iのシーブアクセスA^sが獲得するスナップショットWが最小であるとする．A^sはスプリッタsp_iへ登録したとする．

Wに属するシーブアクセスの中で，スナップショットをスプリッタsp_jの変数$view$へ書き込む最後のシーブアクセスを，プロセスp_jのシーブアクセスA_1^sと仮定する．仮定より，A^sがp_iの変数$view$にWを書き込んだ後に，A_1^sはp_jの変数$view$にスナップショットを書き込み，続いて手続きcandidatesを実行する．このとき，A_1^sはスプリッタsp_iの変数$view$からWを読み込む．Wに属するすべてのシーブアクセスは，補題*7-8.*より$W' \supseteq W$もしくは$W' = \emptyset$を満たす値W'をすでに変数$view$に書き込んでいる．よって，A_1^sの手続きcandidatesは候補者集合としてWを返す．アルゴリズムより明らかに$\langle p_j, sp_j \rangle \in W$なので，$A_1^s$は勝利シーブアクセスとなる． ■

補題 10 任意のプロセスp_iの区間lに属するシーブアクセスA^sが，イベント$e \in A^s$においてシーブsのコピー$c = l \bmod 2N$に入っているとする．このとき，高々k個のシーブアクセスが同時にシーブsのコピーcに入る．ここでkは，p_iが実行する手続きgetNameの実行区間におけるポイント競合度とする．

(証明) 区間lに属するあるシーブアクセスA_1^sが，イベント$e' \in A_1^s$においてシーブsのコピーcに入っているとする．$int(A^s) = int(A_1^s) = l$なので，補題*7-2.*より$rv(A^s) = rv(A_1^s) = l \bmod 2N$を満たす．よって，$A^s$と$A_1^s$は共にコピー$c = l \bmod 2N$に入っている．

区間lにおいて，A^sとA_1^sが共にシーブsのコピーcに入るには，A^sとA_1^sが63行目で$sieve[s].copy[c].inside$から**false**を読み込んだ後に，64行目で$sieve[s].copy[c].inside$に**true**を書き込む必要がある．補題*7-6.*より，区間lにおいてA^sが$sieve[s].copy[c].inside$に**true**を書き込んだ後に，区間l' $(l' < l)$に属するシーブアクセスが，手続きclearにおいて$sieve[s].copy[c].inside$に**false**を書き込み，A_1^sがコピーcに入ることはない．従ってA^sとA_1^sは，$sieve[s].copy[c].inside$に対する読み込みと書き込みの間で，並行にgetNameを実行している．

ポイント競合度の定義より，p_iが実行する手続きgetNameの中の任意のイベントにおいて，アクティブなプロセスは高々k個である．よって，p_iがシーブsのコピーcに入っているとき，高々k個のシーブアクセスが同時にコピーcに入る． ■

あるシーブsの実行R^sにおいて，sにアクセスしているプロセスがいない実行区間を休止期間という．また，2つの連続する休止期間にはさまれた実行区間を使用期間という．ある使用期間の中で，最初にreleaseNameが終了してから使用期間の終りまで，シーブsは使用可能状態であるという．

補題 11 あるシーブsの実行R^sにおいて，任意の使用期間R_bに存在する最初の区間をlとする．区間lに属し，かつR_bに属するイベントを実行するシーブアクセスの中で，少なくとも*1*つはコピー$l \bmod 2N$に入ることができる．

(証明) あるシーブ s の実行 R^s は，使用期間・休止期間と区間の関係に注目することで，図 11 に示す (a) ~ (c)，(a') の四種類の実行に分類できる．以下ではこれらの分類に従って，任意の使用期間 R_b について補題が成り立つことを証明する．

(a) 使用区間 R_b は，シーブ s の実行 R^s の最初のイベント e_0 から始まり，区間 0 の最後のイベント以降まで続くとする．イベント e_0 を実行し，シーブ 0 にアクセスするシーブアクセスを A_0^s とする．補題 *7-2.* より，A_0^s はコピー 0 を使用する．イベント e_0 の直前において，コピー 0 の共有変数 $sieve[s].copy[0].inside$ の値は **false**，$sieve[s].copy[0].allDone$ の値は 1 である．よって，区間 0 に属するシーブアクセスの中の少なくとも 1 つは 40 行目と 97 行目を満たし，コピー 0 に入ることができる．

なお (a') に示すように，イベント e_0 から始まり，区間 0 の最後のイベントの直前までに終る使用区間 R_b は存在しない．なぜなら (a) と同様に，e_0 から R_b の最後のイベントまでに，少なくとも 1 つのシーブアクセスがコピー 0 に入ることができる．補題 9 より，それらのシーブアクセスの中の少なくとも 1 つは勝利シーブアクセス A_w となる．A_w は，対応する手続き releaseName の実行を終えるまでシーブ s にアクセスしているので，releaseName の最後のイベントは R_b に属する．R_b に続く R_i の後に，$R_{b'}$ の最初のイベント e_1 は区間 0 に属するはずである．しかし，e_1 を実行するシーブアクセスは，共有変数 $sieve[s].count$ の第一項から，A_w が更新した値 1 を読み込むはずである．従って，$R_{b'}$ の最初のイベントは区間 1 に属するので，(a') に示すような使用区間 R_b は存在しない．

(b) 使用期間 R_b は，区間 l の最初のイベント t_l^s から始まり，区間 l の最後のイベント以降まで続くとする．(a') と同様の議論により，区間 l の最後のイベントの直前までに，R_b が終ることはない．

補題 *7-7.* より，R_b の最初のイベント t_l^s において，コピー l の共有変数 $sieve[s].copy[l].inside$ の値は **false** である．また，R_b の直前の使用区間 $R_{b'}$ と R_b は，休止区間 R_i を挟んでいるので，e の直前までに区間 $l-1$ に属するシーブアクセスはすべて終了してる．よって補題 9 より，区間 $l-1$ に勝利シーブアクセスが存在し，$sieve[s].copy[l-1].allDone$ の値は $db(l-1)$ となっている．よってアルゴリズムより，区間 l に属するシーブアクセスの中の少なくとも 1 つはコピー $l \bmod 2N$ に入ることができる．

(c) 使用期間 R_b は，区間 l の途中のイベント e から始まり，区間 l の最後のイベント以降まで続くとする．(a') と同様の議論により，区間 l の最後のイベントの直前までに R_b が終ることはない．また，イベント t_l^s から，R_b の直前の使用期間 $R_{b'}$ の最後のイベントまでに，コピー $l \bmod 2N$ に入るシーブアクセスは存在しない．

R_b の最初のイベント e の直前までにイベント t_l^s が存在するので，補題 *7-7.* より，e においてコピー l の共有変数 $sieve[s].copy[l].inside$ の値は **false** である．また，R_b の直前の使用区間 $R_{b'}$ と R_b は，休止区間 R_i を挟んでいるので，e の直前までに区間 $l-1$ に属するすべてのシーブアクセスは終了している．よって補題 9 より，区間 $l-1$ に勝利シーブアクセスが存在し，$sieve[s].copy[l-1].allDone$ の値は $db(l-1)$ に更新されている．このため，区間 l に属するシーブアクセスの中で，少なくとも 1 つはコピー $l \bmod 2N$ に入ることができる．

■

補題 12 プロセス p_i がシーブ s にアクセスしており，p_i が実行する getName におけるポイント競合度を k とする．このとき，任意の $s'(\leq s)$ について，シーブ $s', \ldots, \infty$ にアクセスしているプロセスの個数 $S_p(s')$ と，シーブ $s', \ldots, \infty$ の中で使用可能状態にあるシーブの個数 $S_u(s')$ の和 $S(s')$ は，高々 $2k - s' - 1$ である．

(証明) プロセス p_i が実行する手続き getName の実行区間の長さについての帰納法で証明する．

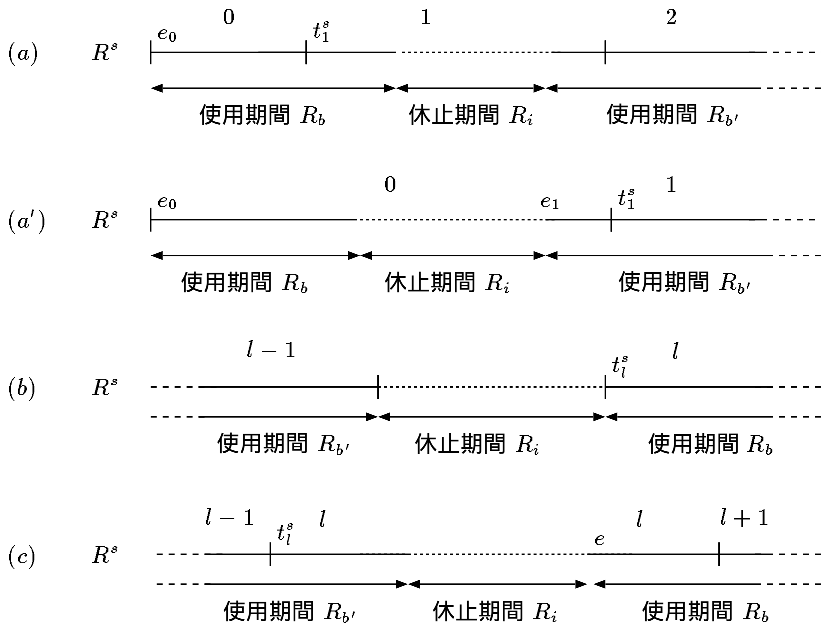

図 11: 実行 R_b の分類

まず初期段として，プロセス p_i が手続き getName を開始するとき，つまりシーブ 0 へのアクセスを開始するときについて示す．p_i がアクセスを開始する直前には，高々 $k-1$ 個のプロセスがシーブ $0,\ldots,\infty$ にアクセスしている．すべての使用可能状態のシーブには，それぞれ少なくとも 1 つのプロセスがまだアクセスしているので，シーブ $0,\ldots,\infty$ の中で高々 $k-1$ 個が使用可能状態である．よって，プロセス p_i 自身を合わせて $S(0)=(k-1)+(k-1)+1=2k-1$．

次に帰納段について示す．プロセス p_i の手続き getName の実行において，$S(s')$ の値の増加は以下の三つの場合が考えられる．

1. プロセス p_i がシーブ $s(\geq 0)$ にアクセスしているときに，あるプロセス $p_j(\neq p_i)$ が getName を開始する場合，$S(0)$ の値が増加する．このとき，初期段と同様にして補題が成り立つ．

2. あるシーブ s_0 が使用可能状態になる場合，使用可能状態の定義より，あるプロセスが releaseName を終了する．このとき，$S_u(s_0)$ の値は 1 つ増加すると同時に，$S_p(s_0)$ の値は 1 つ減少する．よって任意の $s'(\leq s)$ に対する $S(s')$ の値に変化はないので，補題が成り立つ．

3. あるプロセス p_j がシーブ s_1-1 $(s_1 \geq 1)$ へのアクセスを終了した後に，シーブ s_1 へのアクセスを開始する場合を考える．$s_1 > s$ ならば，p_j がシーブ s_1 にアクセスする前にシーブ s_1-1 にアクセスしているので，$S_p(s')$ $(s' \leq s)$ の値は増加しない．また p_j はシーブ s_1-1 で手続き releaseName を実行しないので，$S_u(s')$ の値は増加しない．よって補題が成り立つ．

   $s_1 \leq s$ ならば，帰納法の仮定より，p_j がシーブ s_1 へのアクセスを開始する直前に $S(s_1-1)$ の値は高々 $2k-(s_1-1)-1=2k-s_1$ である．ここで，シーブ s_1-1 のある使用期間 R_b において，p_j がシーブ s_1-1 へのアクセスを終了する最後のプロセスである場合を考える．補題 11 より，R_b の最初のイベントが属する区間 l において，コピーに入ることができるシーブアクセスが少なくとも 1 つは存在する．補題 9 より，区間 l において少なくとも 1 つの勝利シーブアクセスが存在し，p_j 以外のプロセス p_w が実行するシーブクアクセスである．p_w のシーブアクセスは，手続き releaseName を実行した後に，シーブ s_1-1 へのアクセスを終了する．このため，R_b の最後のイベントにおいてシーブ s_1-1 は使用可能状態である．従って，p_j が s_1 へのアクセスを開始するとき，シーブ s_1-1 にアクセスしているプロセスの個数

が 1 つ減少し，シーブ s_1 にアクセスしているプロセスの個数が 1 つ増加する．また，シーブ $s_1 - 1$ は使用可能状態ではなくなる．以上より，p_j が s_1 へのアクセスを開始するとき，$S(s_1)$ の値は高々 $2k - s_1 - 2 + 1 = 2k - s_1 - 1$ となる．よって補題が成り立つ．

次に，p_j がシーブ $s_1 - 1$ へのアクセスを終了する最後のプロセスではない場合を考える．p_j がシーブ s_1 へのアクセスを開始するとき，シーブ $s_1 - 1$ にアクセスしているプロセスの個数が 1 つ減少し，シーブ s_1 にアクセスしているプロセスの個数が 1 つ増加する．また，$S_u(s_1 - 1)$ の値に変化は無い．このため，p_j が s_1 へのアクセスを開始するとき，$S(s_1 - 1)$ の値は高々 $2k - s_1 - 1$ となるので，$S(s_1)$ の値も高々 $2k - s_1 - 1$ である．よって補題が成り立つ．

■

補題 13 あるプロセス p_i が手続き getName を実行したときのポイント競合度を k とする．このとき，p_i は getName において高々 $2k - 1$ 個のシーブにアクセスする．

(証明) 補題 12 より，シーブ $2k - 1, ..., \infty$ にアクセスしているプロセスや，シーブ $2k - 1, ..., \infty$ の中で使用可能状態にあるシーブは存在しない．よって，プロセス p_i は手続き getName において，0 から $2k - 2$ の高々 $2k - 1$ 個のシーブにアクセスする． ■

定理 2 手続き getName は $0, 1, ..., k(2k - 1) - 1$ の範囲の名前を返す．ここで，k は手続き getName の実行区間におけるポイント競合度とする．

(証明) 補題 13 より，高々 $2k - 1$ 個のシーブにアクセスすればランクを獲得できる．手続き candidates が返す候補者集合に属する要素は高々 k 個なので，候補者集合におけるランクは高々 k である．手続き getName が返す名前はランクとシーブ番号の二項組なので，名前の範囲は $0, 1, .., k(2k - 1) - 1$ である． ■

定理 3 本改名アルゴリズムは，ポイント競合度適応型繰り返し $k(2k - 1)$-改名アルゴリズムであり，ステップ計算量は $O(k^2)$，空間計算量は $O(n^2 N)$ である．ここで，k は手続き getName の実行区間におけるポイント競合度，n は k の上界，N はシステムに属するプロセスの総数とする．

(証明) 手続き getName では，補題 13 より高々 $2k - 1$ 個のシーブにアクセスする．各シーブにおいて，シーブに入ることができたなら手続き register を呼ぶ．register によりスプリッタへ登録できると，手続き partial_scan，さらに手続き collect を呼ぶ．補題 10 より register と partial_scan を呼ぶプロセスは高々 k 個なので，補題 4 よりこれらの手続きのステップ計算量は $O(k)$ である．補題 5 よりスナップショットに属するプロセスは高々 k 個なので，candidates のステップ計算量は $O(k)$ である．シーブに入ることができない場合のステップ計算量は $O(1)$ なので，getName のステップ計算量は $O((2k - 1)(k + [k + k] + 1)) = O(k^2)$ である．

手続き releaseName では，ステップ計算量 $O(k)$ の手続き leave を実行する．続いて，もし共有変数 *allDone* がすでに更新されていれば，手続き clear を実行する．clear では，コピーに属する共有変数とコレクトリストを初期化する．補題 3 よりマークされているスプリッタは高々 k 個であり．1 つのスプリッタを初期化するためのステップ計算量は $O(1)$ である．よって，clear のステップ計算量は $O(k)$ なので，releaseName のステップ計算量は $O(k + k) = O(k)$ である．

補題 13 より，アクセスされるシーブの数は高々 $2k - 1$ 個 なので，$2n - 1$ 個のシーブを用意する．また補題 6 より，各シーブには $2N$ 個のコピーを用意すれば，コピーを正しく再利用できる．補題 3 より，1 つのコピーにつき n 個のスプリッタを用意し，1 つのスプリッタの空間計算量は $O(1)$ である．よって，本改名アルゴリズムの空間計算量は，$O((2n - 1) \cdot 2N \cdot n) = O(n^2 N)$ である． ■

5　まとめ

本レポートでは，非同期共有メモリシステムにおける，ポイント競合度適応型繰り返し $k(2k-1)$-改名アルゴリズムを提案した．本改名アルゴリズムのステップ計算量は $O(k^2)$，空間計算量は $O(n^2N)$，共有レジスタの値は有界である．従って，これらの点において従来のアルゴリズム [14] よりも優れている．なお，k はポイント競合度，n は k の上界，N はシステムに属するプロセスの総数を示す．

これまでに，改名アルゴリズムを応用したアルゴリズム (応用アルゴリズム) がいくつも提案されている [12, 13]．本改名アルゴリズムは，これらの応用アルゴリズムで用いられている改名アルゴリズムよりも，ステップ計算量や空間計算量が優れている．そこで，これらの応用アルゴリズムに本改名アルゴリズムを応用することで，その計算量を削減できると考えられる．

今後の課題は，本改名アルゴリズムのアイデアを，read/write メモリモデルにおいて名前空間の大きさが最適な [10] 繰り返し $(2k-1)$-改名アルゴリズムに改良することである．一般に，繰り返し $(2k-1)$-改名アルゴリズムでは，繰り返し $k(2k-1)$-改名アルゴリズムよりも複雑な制御を必要とし，多くのレジスタが必要であると予想できる [6]．このため，多項式ステップ計算量かつ有界な空間計算量の繰り返し $(2k-1)$-改名アルゴリズムは，これまでに提案されていない．

また，本改名アルゴリズムのステップ計算量や空間計算量を，さらに改善することも今後の課題として挙げられる．

参考文献

[1] A.Bar-Noy and D.Dolev. Shared memory versus message-passing in an asynchronous distributed environment. In *Proc. 8th ACM Symp. on Principles of Dist. Comup.*, pages 307–318, 1989.

[2] A.Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, Inc., 1996.

[3] E.Borowsky and E.Gafni. Immediate atomic snapshots and fast renaming. In *Proc. 12th ACM Symp. on Principles of Dist. Comp.*, pages 41–52, 1993.

[4] H.Attiya and A.Fouren. Adaptive wait-free algorithms for lattice agreement and renaming. In *Proc. 17th ACM Symp. on Principles of Dist. Comp.*, pages 277–286, 1998.

[5] H.Attiya and A.Fouren. An adaptive collect algorithm with applications. 1999. Available at www.cs.tehnion.ac.il/∼hagit/pubs/AF99ful.ps.gz.

[6] H.Attiya and A.Fouren. Polynomial and adaptive long-lived (2k-1)-renaming. In *Proc. 14th Int. Symp. on Dist. Comp.*, pages 149–163, 2000.

[7] H.Attiya and J.Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics.* McGraw-Hill Publishing Company, 1998.

[8] L.Lamport. On interprocess communication, part i : Basic formalism. *Distributed Computing*, 1(2):77–85, 1986.

[9] L.Lamport. On interprocess communication, part ii : Algorithms. *Distributed Computing*, 1(2):86–101, 1986.

[10] M.Herlihy and N.Shavit. The asynchronous computability theorem for t-resilient tasks. In *Proc. 25th ACM Symp. on Theory of Comp.*, pages 111–120, 1993.

[11] M.Moir and J.H.Anderson. Fast, long-lived renaming. In *Proc. 8th Int. Workshop on Dist. Algorithms*, pages 141–155, 1994.

[12] Y.Afek, G.Stupp, and D.Touitou. Long-lived and adaptive collect with applications. In *Proc. 40th Symp. on Foundations of Computer Science*, pages 262–272, 1999.

[13] Y.Afek, G.Stupp, and D.Touitou. Long-lived and adaptive atomic snapshot and immediate snapshot. In *Proc. 19th ACM Symp.*

on Principles of Dist. Comp., pages 71–80, 2000.

[14] Y.Afek, H.Attiya, A.Fouren, G.Stupp, and D.Touitou. Adaptive long-lived renaming using bounded memory. 1999. Available at www.cs.technion.ac.il/~hagit/pubs/AAFST99disc.ps.gz.

[15] Y.Afek, H.Attiya, A.Fouren, G.Stupp, and D.Touitou. Long-lived renaming made adaptive. In *Proc. 18th ACM Symp. on Principles of Dist. Comp.*, pages 91–103, 1999.

[16] Y.Afek and M.Merritt. Fast, wait-free (2k-1)-renaming. In *Proc. 18th ACM Symp. on Principles of Dist. Comp.*, pages 105–112, 1999.

[17] Y.Zomaya, editor. *Parallel and Distributed Computing Handbook*, chapter 5. McGraw-Hill Publishing Company, 1996.