並列・分散アルゴリズムの基礎

並列・分散アルゴリズムの基礎
Agenda

Part I　排他制御 1 ダイクストラ

- ダイクストラのMutual Exclusion アルゴリズム 1
- ダイクストラのMutual Exclusion アルゴリズム 詳細

Part II　排他制御 2 ランポート

- ランポートのbakeryアルゴリズム 1
- ランポートのbakeryアルゴリズムの正しさ
- 新しいbakeryアルゴリズムと並列性の表現

並列・分散アルゴリズムの基礎
Agenda

Part III　生産者消費者同期

- ダイクストラのsemaphore
- 生産者消費者同期アルゴリズムを記述する
- Producer-Consumerアルゴリズムの振る舞いを図解する

Part IV　アルゴリズムの正しさへのアプローチ

- アルゴリズムとプログラム
- 状態モデルとアルゴリズムの「不変量」
- イベントの順序関係とランポートの「論理的時計」
- 状態とアクションと時制論理式

Part I
排他制御 ダイクストラ

並列・分散アルゴリズムの基礎

Part I　排他制御　ダイクストラ

- ダイクストラのMutual Exclusion アルゴリズム 1
- ダイクストラのMutual Exclusion アルゴリズム 詳細

ダイクストラの
Mutual Exclusion アルゴリズム

ダイクストラのMutual Exclusion アルゴリズム

並列アルゴリズムの代表例として、まず、ダイクストラの Mutual Exclusion のアルゴリズムを紹介する。

今回は、ダイクストラがどのような問題を解こうとしたのかについて述べる。この部分は、図を除けば、基本的には、ダイクストラの論文をそのまま引用している。

アルゴリズム自体の説明は、次回になる。

Solution of a Problem in Concurrent Programming Control

制限された相互通信手段しか持たない、複数の基本的には独立したシーケンシャルあるいはサイクリックなプロセスが、どの時点でも、一つのそしてーつだけのプロセスしか、サイクルの"critical section" を実行しないようにできるだろうか。

Dijkstra, E.W. Solution of a problem in concurrent programming control. *Commun. ACM 8*, 9 (September 1965), 569.
http://rust-class.org/static/classes/class19/dijkstra.pdf

Introduction

この論文で与えられているのは、ある問題への解法である。著者の知る限りでは、この問題は少なくとも1962年から、解決可能性の議論とは無関係に、未解決問題であった。

この論文は、次の三つの部分からなる。「問題」「解法」「証明」。

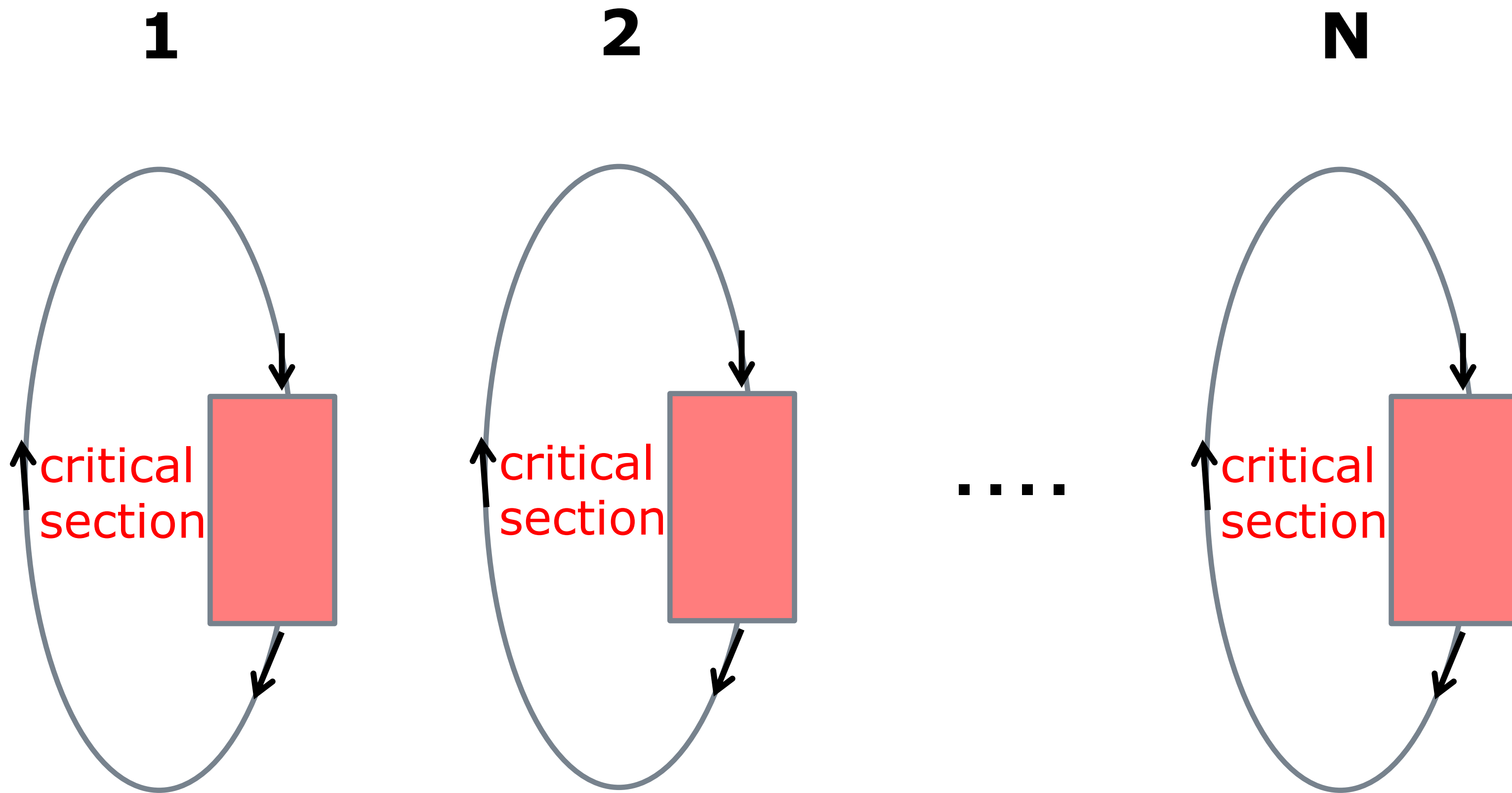
問題の立て方は、最初はアカデミックなものに見えるかもしれないが、コンピュータの結合から生まれる論理的な問題をよく知っている人は、誰でも、この問題が実際に解けるという事実の重要性は評価してくれるだろうと著者は信じている。

以下、「問題」の部分を紹介する。

The Problem
critical sectionとMutual Exclusion

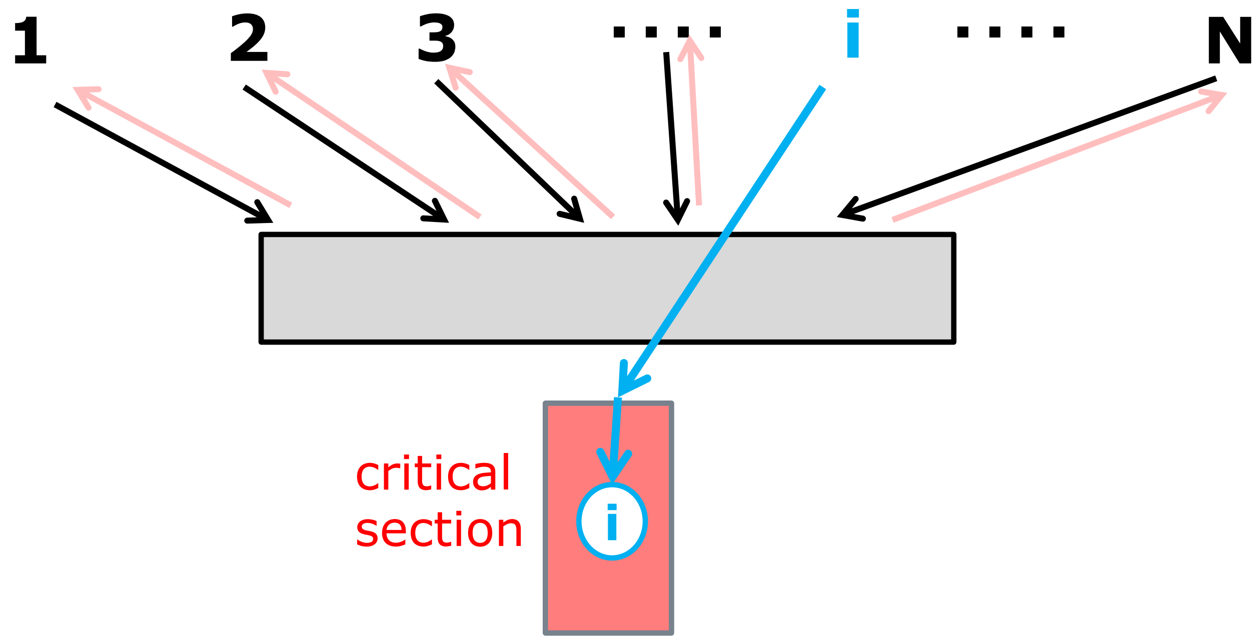
最初に、N台のコンピュータを考えよう。それぞれは、我々の想定では、サイクリックなプロセスを実行しているものと考えることができる。

それぞれのサイクルには、いわゆる "critical section" といわれるものがあり、どの時点でもN個のプロセスのサイクルのうち一つのプロセスだけがそのcritical sectionにいるように、コンピュータはプログラムされてなければならない。



N台のコンピュータを考えよう。それぞれは、我々の想定では、
サイクリックなプロセスを実行しているものと考えることができる。

それぞれのサイクルには、いわゆる “critical section” といわれるものがある。



どの時点でもN個のプロセスのサイクルのうち一つのプロセスだけがそのcritical sectionにいるように、コンピュータはプログラムされてなければならない。

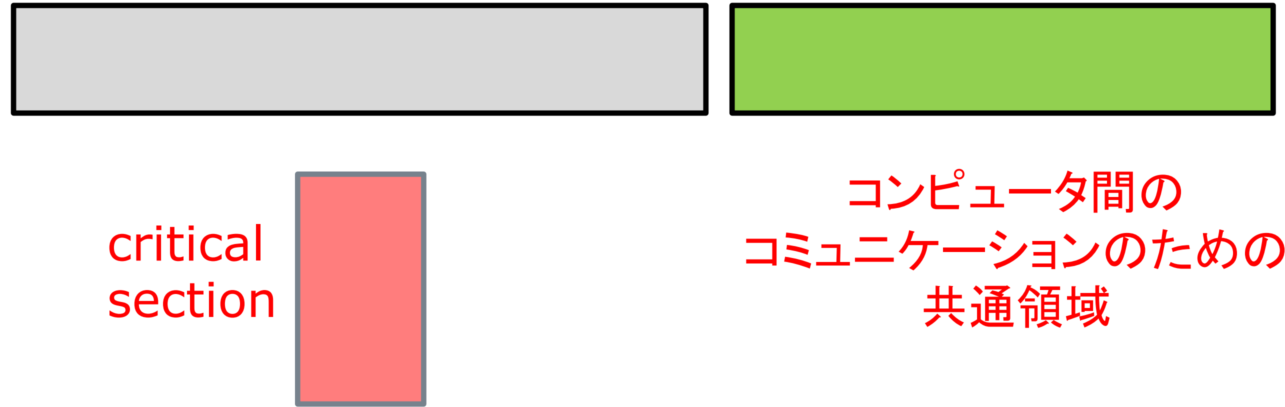
共通領域を通じた
コンピュータ同士のコミュニケーション

このようにお互いに排除し合うようなcritical sectionの実行を実現するためには、コンピュータは、共通な領域を通じてお互いにコミュニケーションができなければならない。

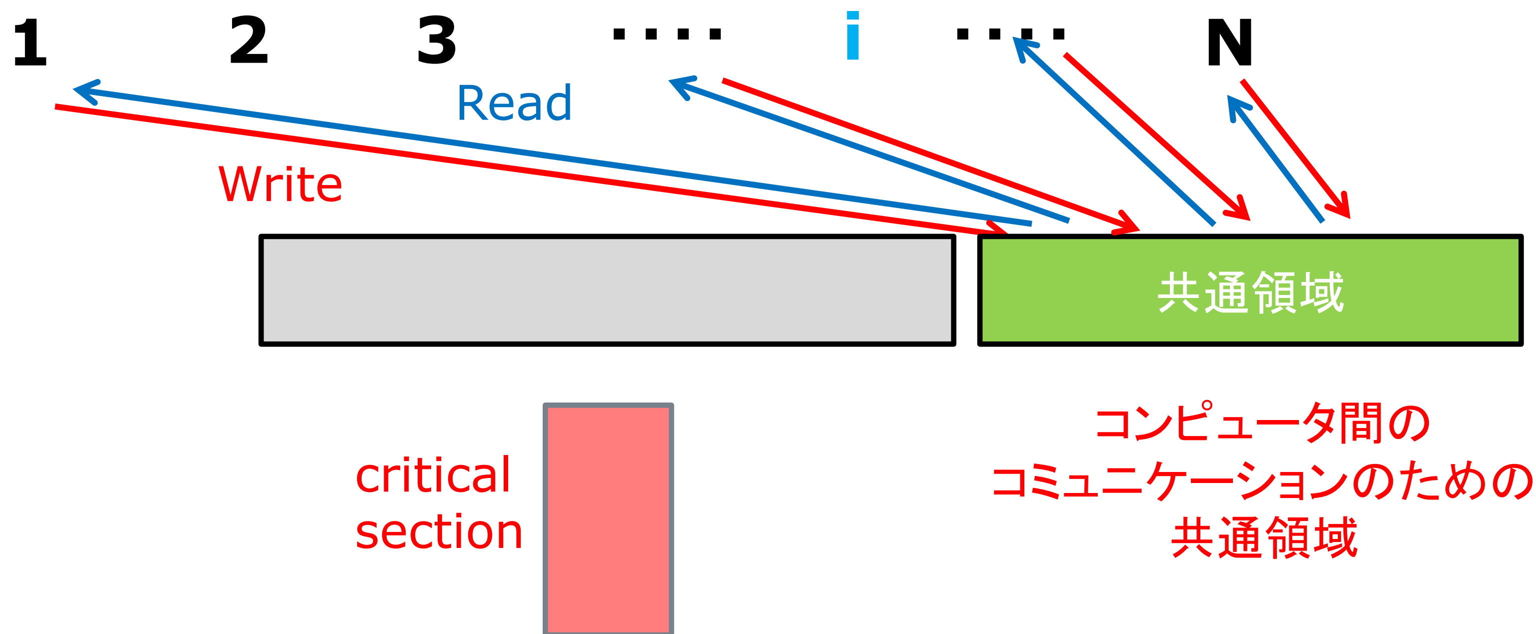
この領域への情報の書き込み、あるいはこの領域からの非破壊的な情報の読み出しは、分割できない操作である。

すなわち、2台以上のコンピュータが共通の同一の場所を同時に（読み出しあるいは書き込みで）コミュニケートしようとするなら、こうしたコミュニケーションは、順序は不明だが、ひとつづつ順番に実行されるだろうということである。

1　　2　　3　　‥‥　　i　　‥‥　　N



このようにお互いに排除し合うようなcritical sectionの実行を
実現するためには、コンピュータは、共通な領域を通じてお互いに
コミュニケーションができなければならない。



この領域への情報の書き込み、あるいはこの領域からの非破壊的な
情報の読み出しは、分割できない操作である。

こうしたコミュニケーションは、順序は不明だが、ひとつづつ順番に
実行されるだろうということである。

解が満たすべき条件

この問題の解は、次のような要請を満たさなければならない。

- この解は、N台のコンピュータの間で対称的でなければならない。その結果、コンピュータの間に、静的な優先権を導入することは許されていない。
- N台のコンピュータの相対的なスピードについては、いかなる想定もなされていない。我々は、それらのスピードが一定であるという想定さえ行わないだろう。
- もし、critical sectionの外側で、どれかのコンピュータが停止したとしても、そのことが、他のコンピュータのブロックを潜在的に導くようなことは許されない。

解が満たすべき条件

この問題の解は、次のような要請を満たさなければならない。

- もし、一台以上のコンピュータがcritical sectionに入ろうとした時、有限のスピードでは、彼らが何かを考えるのは、不可能に違いない。こうした場合、どの一台が先にcritical sectionに入るかを決定する判断は、永遠に延期される。別の言葉で言えば、「お先にどうぞ」「お先にどうぞ」というブロックのある解の構成は、可能である。あまり、ありそうもないのだが、こうした解は、妥当な解とは見なされないだろう。

読者への挑戦

ここで読者に挑戦したい。しばしたちどまって、自分自身で考えてほしい。

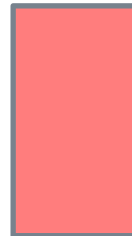
というのも、そうすることが、それぞれのコンピュータは、同時には一方向のメッセージを要求することしかできないという事実のトリッキーな帰結を感じてもらう唯一の方法のように思えるからだ。

そして、それのみが、どの程度までこの問題が、トリビアルな問題からは遠い問題であることを、読者に理解させるだろう。

ダイクストラのアルゴリズム

1　　2　　3　　・・・・　　i　　・・・・　　N

critical section



integer k
boolean b[1:N]
boolean c[1:N]

ダイクストラのアルゴリズム

```
integer j;
Li0: b[i] := false;
Li1: if k ≠ i then
Li2: begin c[i] := true;
Li3: if b[k] then k := i;
      go to Li1
      end
        else
Li4: begin c[i] := false;
        for j := 1 step 1 until N do
            if j ≠ i and not c[j] then go to Li1
       end ;
       critical section;
       c[i] := true; b[i] := true;
       remainder of the cycle in which stopping is allowed;
       go to Li0"
```

ダイクストラの
Mutual Exclusion アルゴリズム詳細

このセッションの目的

このセッションは、ダイクストラが提示したアルゴリズムを理解することを目的とする。

わかりやすい説明を心がけたつもりだが、アルゴリズムが全体として大きなループで循環する構造をしているので、アルゴリズムを説明する叙述の順序も循環しているように思われるかもしれない。

本当は、アルゴリズムのパーツのどの部分から理解を始めても構わないのだが、全体像を再構成するには、やや、時間がかかるかもしれない。 根気よく取り組んでもらえればと思う。

基本的には、ダイクストラの論文の引用は、ミドリ色の背景を持つ見出しの下にまとめ、丸山による補足説明は、ライトブルーの背景を持つ見出しの下にまとめた。

留意すべきこと

ただ、ダイクストラのアルゴリズムについては、留意すべきことがある。

それは、今回説明したアルゴリズム自身について、いくつか留保すべきことがあるということである。

それについては、別の機会に「クヌースらによるダイクストラ・アルゴリズムの改善」のセッションで述べようと思う。
（このセッションは、割愛された。）

ダイクストラによる問題の「解」

共通領域は、次のものからなる。

Boolean array b[1:N];

Boolean array c[1:N];

integer k ;　整数 k は、$1 \leq k \leq N$ を満たす。

b[i] と c[i] は、i 番目のコンピュータのみによってセットされる。
これらの変数は、他のコンピュータによってチェックされる。
全てのコンピュータは、critical sectionの外側からスタートし、また、前述の共通領域のBoolean arrayは全てtrueにセットされていると想定されている。kの最初の値は、重要ではない。

i番目($1 \leq i \leq N$) のコンピュータのプログラムは、次のようになる。

N個のコンピュータに対する ダイクストラのアルゴリズム

1　2　3　・・・・　k　・・・・　N

critical
section

integer k
boolean b[1:N]
boolean c[1:N]

ダイクストラによる
問題の「解」

i 番目のコンピュータが
実行するプログラム

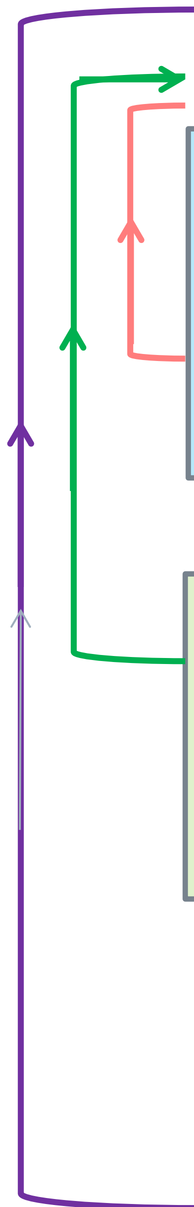
```
integer j;
Li0: b[i] := false;
Li1: if k ≠ i then
Li2: begin c[i] := true;
Li3: if b[k] then k := i;
      go to Li1
      end
        else
Li4: begin c[i] := false;
        for j := 1 step 1 until N do
            if j ≠ i and not c[j] then go to Li1
       end ;
       critical section;
       c[i] := true; b[i] := true;
       remainder of the cycle in which stopping is allowed;
       go to Li0"
```

最初に、critical section への接近を果たすのは、k = i を満たすk番目のコンピュータである。

```
integer j;
Li0: b[i] := false;
Li1: if k ≠ i then
Li2: begin c[i] := true;
Li3: if b[k] then k := i;
      go to Li1
      end
        else
Li4: begin c[i] := false;
        for j := 1 step 1 until N do
          if j ≠ i and not c[j] then go to Li1
      end ;
      critical section;
      c[i] := true; b[i] := true;
      remainder of the cycle in which stopping is allowed;
      go to Li0"
```

k ≠ i の時、このブロックが実行される

k = i の時、このブロックが実行される

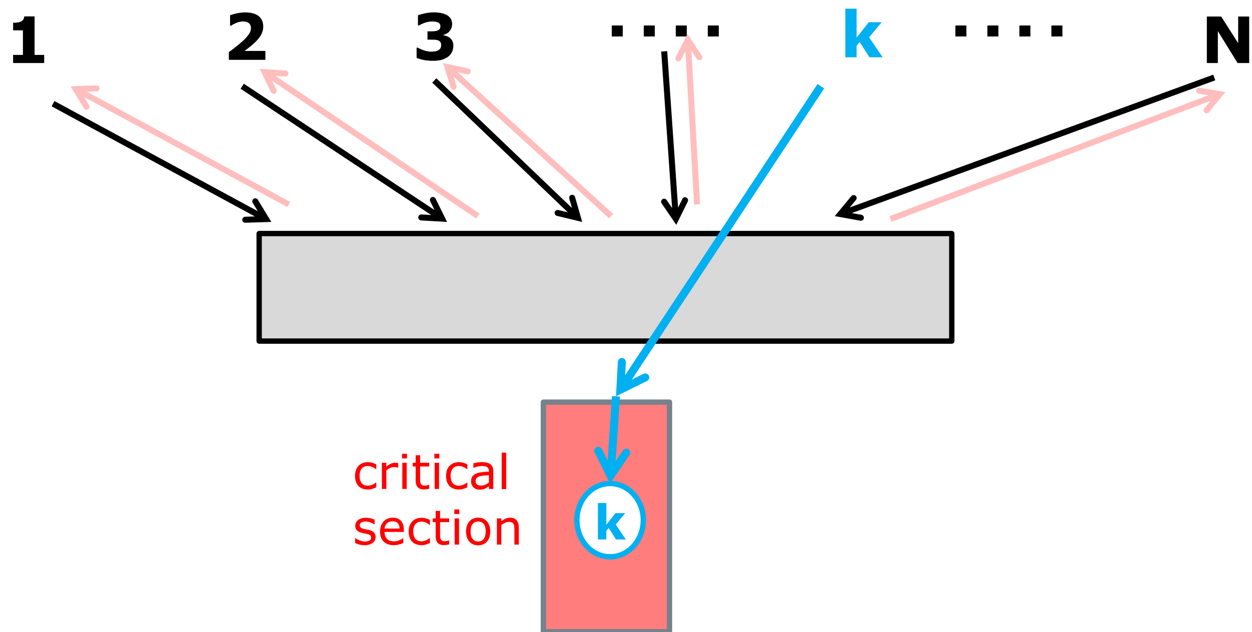


```
integer j;
Li0: b[i] := false;
Li1: if k ≠ i then
Li2: begin c[i] := true;
Li3: if b[k] then k := i;
     go to Li1
     end
        else
Li4: begin c[i] := false;
        for j := 1 step 1 until N do
            if j ≠ i and not c[j] then go to Li1
     end ;
     critical section;
     c[i] := true; b[i] := true;
     remainder of the cycle in which stopping is allowed;
     go to Li0"
```

ただし、すぐに critical section へ入れるわけではない。条件を満たさないとLi1へ loop back する。

このelse は必要か？

critical sectionに入る条件



どの時点でもN個のプロセスのサイクルのうち一つのプロセスだけがそのcritical sectionにいるように、コンピュータはプログラムされてなければならない。

The Proof
ダイクストラによる「証明」

まず最初に、二つのコンピュータが同時にcritical sectionに入ることはできないという意味で、この解はsafe であることを見ておこう。

なぜなら、critical sectionに入る唯一の方法は、Li4複合文の実行がLi1にジャンプして戻ることがない場合だけである。すなわち、自分自身のcの値をfalseにした後で、その他のすべてのcの値がtrueであることが分かった場合だけである。

```
integer j;
Li0: b[i] := false;
Li1: if k ≠ i then
Li2: begin c[i] := true;
Li3: if b[k] then k := i;
      go to Li1
      end
        else
Li4: begin c[i] := false;
        for j := 1 step 1 until N do
            if j ≠ i and not c[j] then go to Li1
      end ;
      critical section;
      c[i] := true; b[i] := true;
      remainder of the cycle in which stopping is allowed;
      go to Li0"
```

critical sectionに入る為には、直前の
このLi4ブロックで、Li1に戻されてはいけない。

```
integer j;
Li0: b[i] := false;
Li1: if k ≠ i then
Li2: begin c[i] := true;
Li3: if b[k] then k := i;
      go to Li1
      end
        else
Li4: begin c[i] := false;
        for j := 1 step 1 until N do
            if j ≠ i and not c[j] then go to Li1
       end ;
       critical section;
       c[i] := true; b[i] := true;
       remainder of the cycle in which stopping is allowed;
       go to Li0"
```

critical sectionに入る為には、直前のLi4ブロックで、Li1に戻されてはいけない。
その条件は、自分以外($j \neq i$) のすべての $c[j]$ が真であることである。

```
integer j;
Li0: b[i] := false;
Li1: if k ≠ i then
Li2: begin c[i] := true;
Li3: if b[k] then k := i;
      go to Li1
      end
        else
Li4: begin c[i] := false;
        for j := 1 step 1 until N do
            if j ≠ i and not c[j] then go to Li1
       end ;
       critical section;
       c[i] := true; b[i] := true;
       remainder of the cycle in which stopping is allowed;
       go to Li0"
```

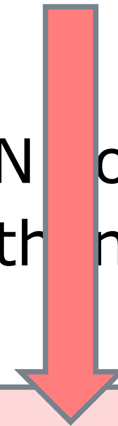
critical sectionに入る為には、直前のLi4ブロックで、Li1に戻されてはいけない。

その条件は、自分以外($j \neq i$) のすべてのc[j] が真であることである。

このことは、二つのプロセスが同時にcritical seectionに入ることはないことを保証する。

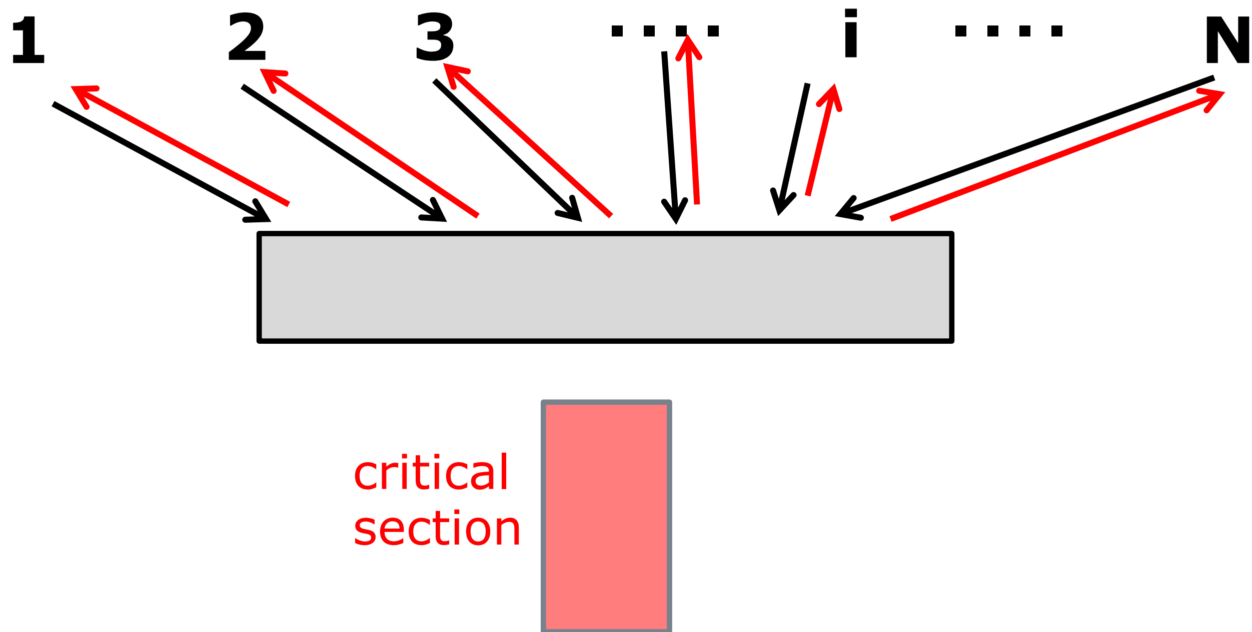
```
integer j;
Li0: b[i] := false;
Li1: if k ≠ i then
Li2: begin c[i] := true;
Li3: if b[k] then k := i;
      go to Li1
      end
        else
Li4: begin c[i] := false;
        for j := 1 step 1 until N do
            if j ≠ i and not c[j] then go to Li1
       end ;
```

critical sectionの中で、c[i], b[i]は共にtrueにセットされる。



```
critical section;
c[i] := true; b[i] := true;
remainder of the cycle in which stopping is allowed;
go to Li0"
```

どのコンピュータも critical sectionにいない場合



どのコンピュータもcritical sectionにいない場合、ループしている（すなわち、Li1にジャンプバックしている）コンピュータの少なくとも一つは、それゆえ正確に一つは、適当な時期に、critical sectionに、いつか入ることが許されるだろう

The Proof
ダイクストラによる「証明」

証明の第二の部分で、無限の「お先にどうぞ」「お先にどうぞ」というブロックが起こりえないことを示す必要がある。

すなわち、どのコンピュータもcritical sectionにいない場合、ループしている（すなわち、Li1にジャンプバックしている）コンピュータの少なくとも一つは、それゆえ正確に一つは、適当な時期に、critical sectionに、いつか入ることが許されるだろうということである。

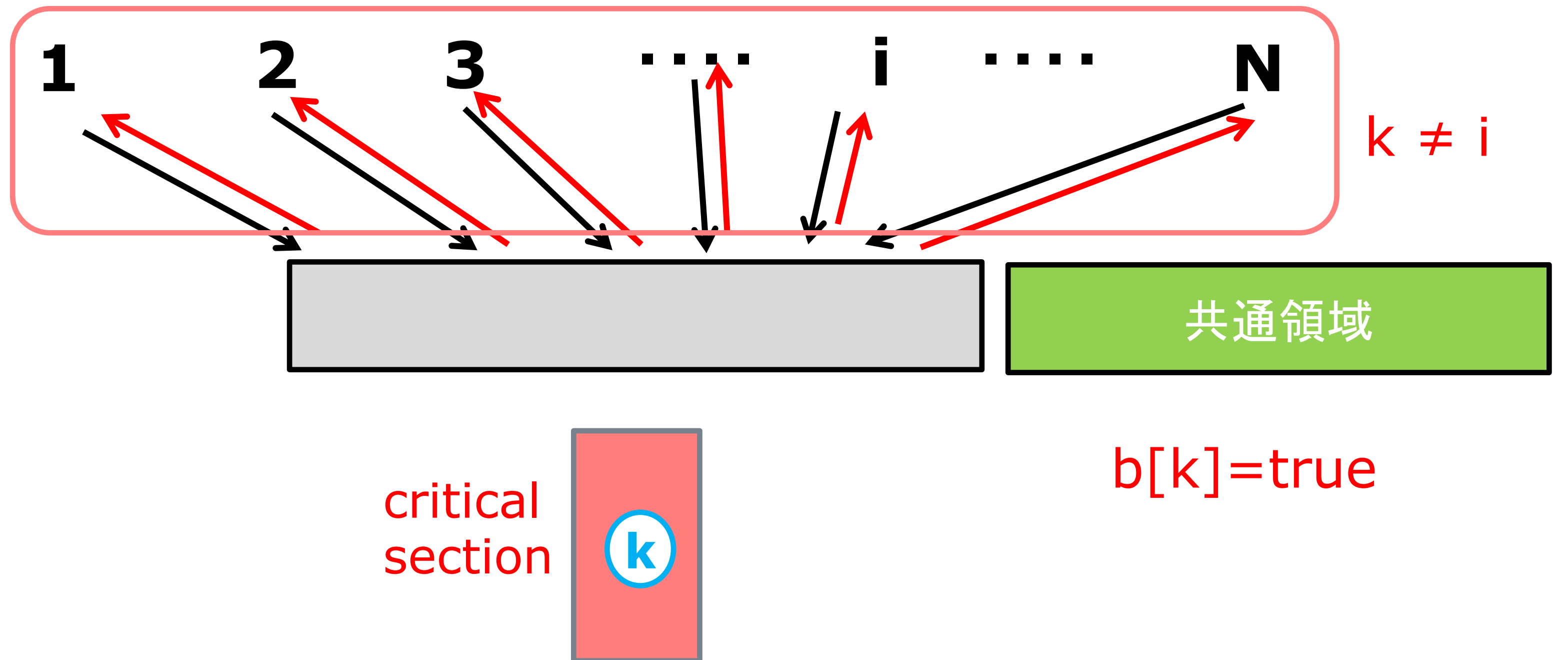
```
integer j;
Li0: b[i] := false;
Li1: if k ≠ i then
Li2: begin c[i] := true;
Li3: if b[k] then k := i;
     go to Li1
     end
        else
Li4: begin c[i] := false;
        for j := 1 step 1 until N do
            if j ≠ i and not c[j] then go to Li1
     end ;
     critical section;
     c[i] := true; b[i] := true;
     remainder of the cycle in which stopping is allowed;
     go to Li0"
```

Li1へのループバック

k ≠ i の場合

k = i の場合

k番目のコンピュータが critical sectionにいる場合



もし、k番目のコンピュータがループして待機しているコンピュータのグループに属さないとすれば、b[k]はtrueで、ループしているコンピュータはすべて、k ≠ i であることがわかる。

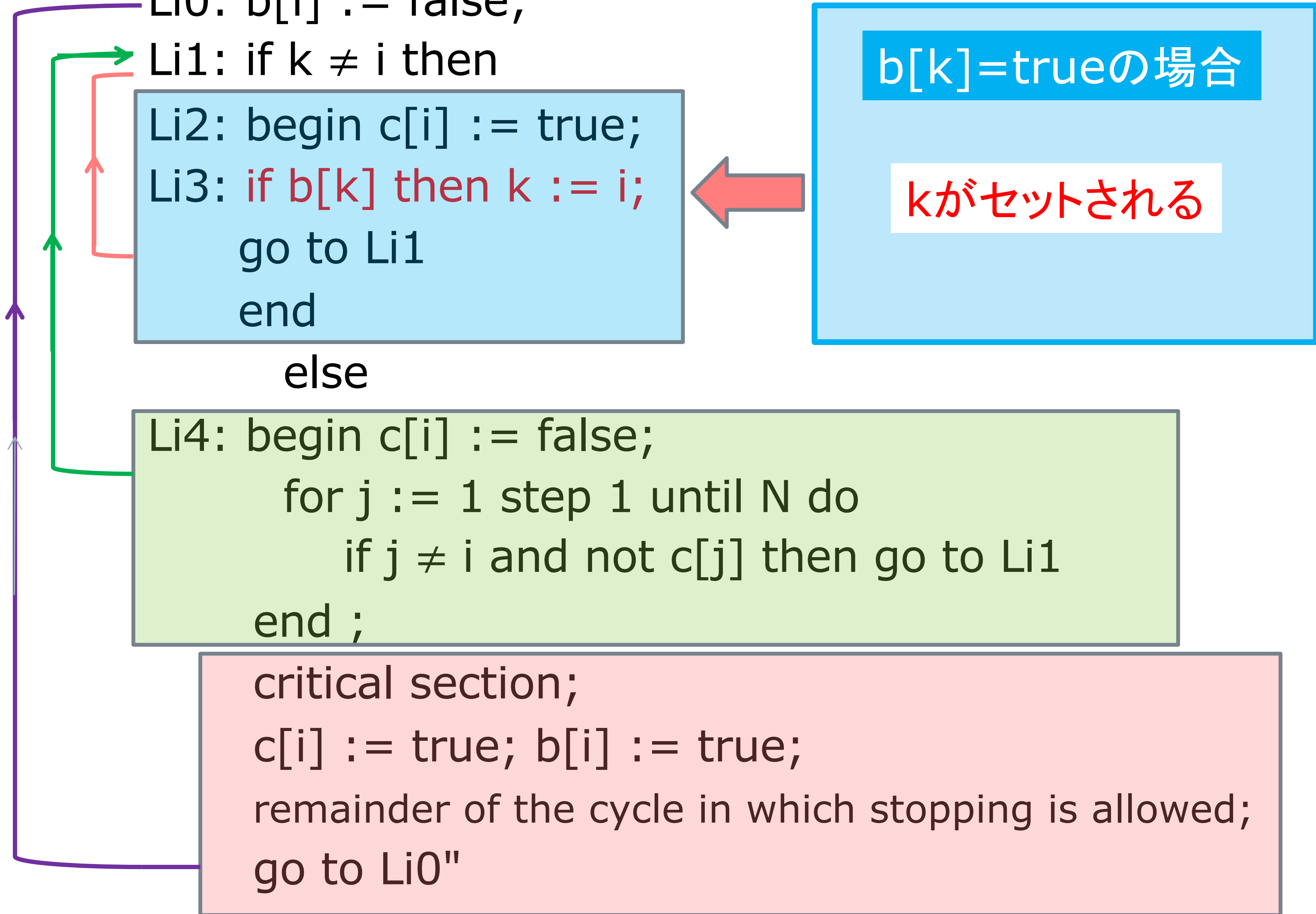
The Proof
ダイクストラによる「証明」

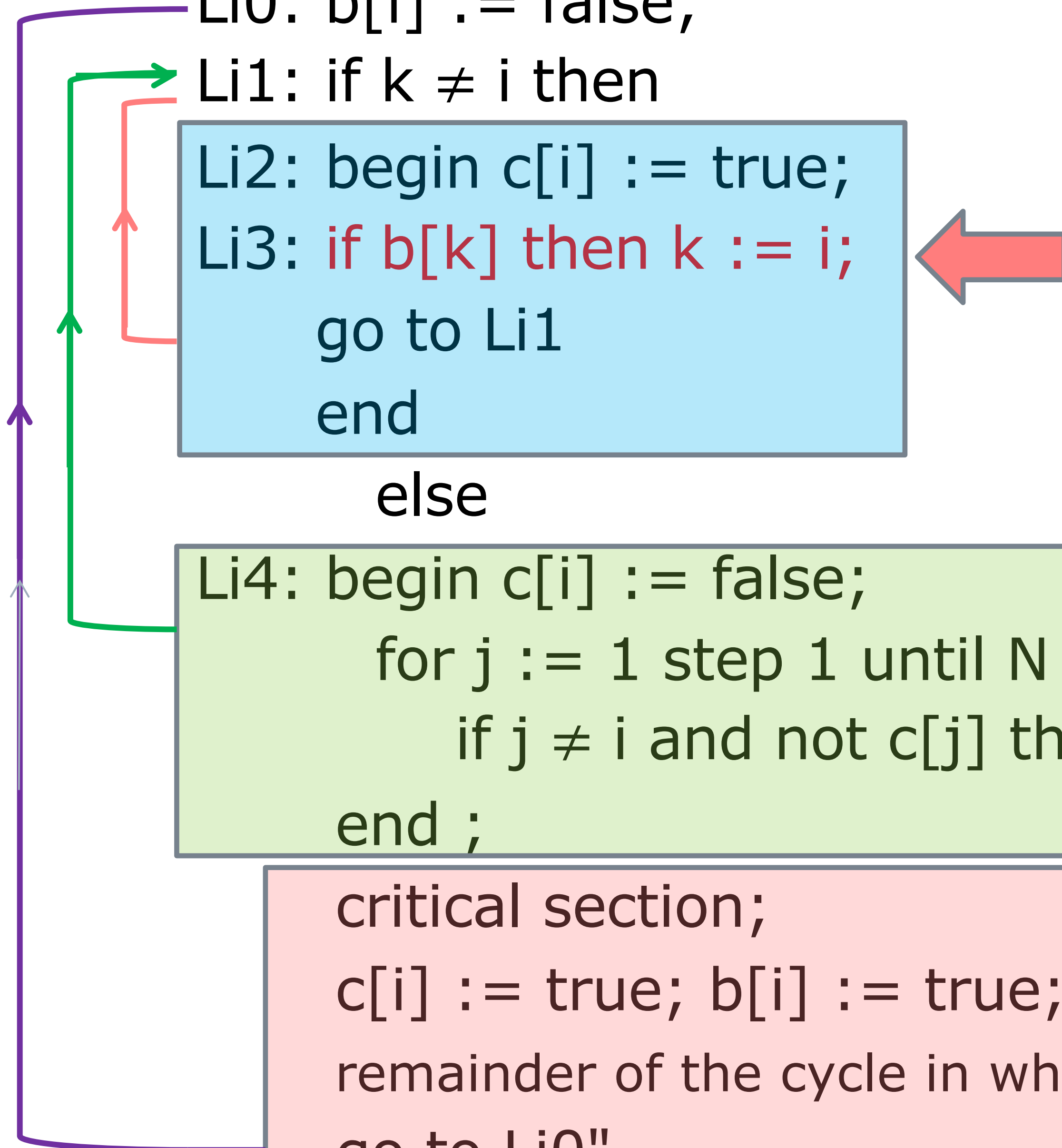
もし、k番目のコンピュータがループして待機しているコンピュータのグループに属さないとすれば、b[k]はtrueで、ループしているコンピュータはすべて、k ≠ i であることがわかる。

The Proof
ダイクストラによる「証明」

その結果、それらの中の一つ以上のコンピュータは、Li3でb[k]がtrueの時、“k := i” という値の割り当てを決定するだろう。

最初の“k := i” の割り当ての後、b[k]はfalseとなり、新しいコンピュータは、kに再度新しい値を割り当てることを決めることができなくなる。

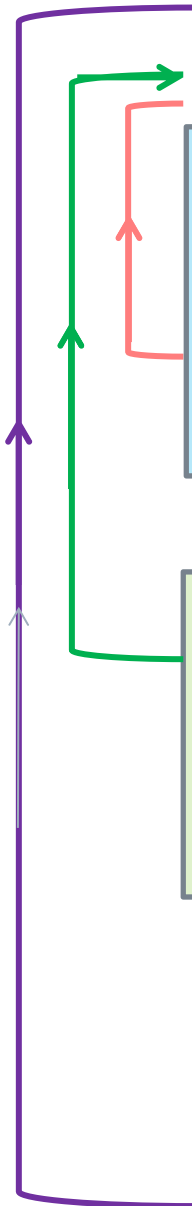

integer j;
Li0: b[i] := false;
Li1: if k ≠ i then
Li2: begin c[i] := true;
Li3: if b[k] then k := i;
go to Li1
end
else
Li4: begin c[i] := false;
for j := 1 step 1 until N do
if j ≠ i and not c[j] then go to Li1
end ;
critical section;
c[i] := true; b[i] := true;
remainder of the cycle in which stopping is allowed;
go to Li0"
b[k]=trueの場合
kがセットされる



```
integer j;
Li0: b[i] := false;
Li1: if k ≠ i then
Li2: begin c[i] := true;
Li3: if b[k] then k := i;
        go to Li1
        end
          else
Li4: begin c[i] := false;
          for j := 1 step 1 until N do
              if j ≠ i and not c[j] then go to Li1
       end ;
       critical section;
       c[i] := true; b[i] := true;
       remainder of the cycle in which stopping is allowed;
       go to Li0"
```

b[k]=true ?

これは、b[k]=trueに最初に気づいたマシンの番号iにkがセットされるということである。



```
integer j;
Li0: b[i] := false;
Li1: if k ≠ i then
Li2: begin c[i] := true;
Li3: if b[k] then k := i;
      go to Li1
      end
         else
Li4: begin c[i] := false;
         for j := 1 step 1 until N do
             if j ≠ i and not c[j] then go to Li1
      end ;
      critical section;
      c[i] := true; b[i] := true;
      remainder of the cycle in which stopping is allowed;
      go to Li0"
```

b[k]がtrueにセットされるのは

critical section の内部でのみ

```
integer j;
Li0: b[i] := false;
Li1: if k ≠ i then
Li2: begin c[i] := true;
Li3: if b[k] then k := i;
      go to Li1
      end
        else
Li4: begin c[i] := false;
        for j := 1 step 1 until N do
            if j ≠ i and not c[j] then go to Li1
      end ;
      critical section;
      c[i] := true; b[i] := true;
      remainder of the cycle in which stopping is allowed;
      go to Li0"
```

ただし、critical sectionを抜けると

b[k]はfalseにリセットされる

The Proof
ダイクストラによる「証明」

kへの決定された値の割り当てがすべて実行された時、kはループ上のコンピュータの一つを指している。そして、当分の間、その値を変えないだろう。

すなわち、b[k]がtrueになるまで、すなわち、k番目のコンピュータが、critical sectionを終了するまで。

The Proof
ダイクストラによる「証明」

kの値が、それ以上変わらないことになれば、すぐに、k番目のコンピュータは、(Li4の複合文によって)、その他のすべてのcがtrueになるまで待機する。

しかし、すでにそうなっていなくても、この状況は確実に起こることである。なぜなら、その他のループ上のコンピュータは、 k ≠ i であるので cをtrueにセットすることを強制されるからである。

そして、このことが、証明を完成させると著者は信じている。

Part II

排他制御 ランポート

並列･分散アルゴリズムの基礎

Part II　排他制御　ランポート

- ランポートのbakeryアルゴリズム 1
- ランポートのbakeryアルゴリズムの正しさ
- 新しいbakeryアルゴリズムと並列性の表現

ランポートのbakeryアルゴリズム 1

ランポートのbakeryアルゴリズム

「bakeryアルゴリズム」は、「パン屋のアルゴリズム」ということ。
混雑している人気のパン屋をイメージしている。こんな感じ。

- パン屋の店先には、番号付きの整理券を発行する機械がある。整理券の番号は一つづつ増えていく。
- お客は、パン屋に入る前に、機械から整理券を受け取る。同時に一人しか整理券を受け取れない。
- パン屋には、一つのレジしかない。一番小さな番号の整理券を持ったお客から、パンを買うことができる。

パン屋じゃなくても病院でも銀行でも、身の回りの日常で混雑整理の方法として、よくみる風景である。

ダイクストラらの排他制御アルゴリズムの性質

最初に、先行したダイクストラらのアルゴリズムの特徴をおさらいしておこう。それは、次のような性質を持っていた。

- どの時点でも、多くても一つのコンピュータしかcritical sectionに入ることができない。
- どのコンピュータも、（途中でマシンが停止しなければ）最終的にはいずれcritical sectionに入ることができる。
- どのコンピュータも、非critical sectionで停止しても構わない。また、それぞれのコンピュータの処理スピードは問わない。

ランポートのアルゴリズムは、
排他制御アルゴリズムである

ランポートのアルゴリズムでは、最後の実際にレジでパンを買うところが、critical sectionに入ることに相当する。ダイクストラらの排他制御アルゴリズムの特徴を、ランポートのbakery アルゴリズムが持つことは、直感的には明らかに思える。

- レジでパンを買えるのは、一人だけ。
- 整理券を持っていれば、いずれ、パンを買える。
- 途中で店を抜けても、時間をかけてゆっくりパンを選んでも構わない。

その上、アルゴリズムはとてもわかりやすい。

ダイクストラらのアルゴリズムの問題
ランポートのアルゴリズムの背景

ただ、アルゴリズムのわかりやすさだけが、ランポートのアルゴリズムのメリットではない。

実は、ダイクストラらのアルゴリズムには（クヌース、ド・ブルイジンらの「改良版」でも共通に）、実践的には問題があった。

それは、N台のコンピュータがコミュニケーションに利用する「共有領域」の読み書きに障害が発生すると、このアルゴリズムは機能しなくなるということである。

ランポートのアルゴリズムでは、排他制御に重要な役割をはたすそれぞれのマシンが受け取ったチケットの番号を、N台のマシン全体が共有する領域に置くのではなく、それぞれのマシンが持つ。

その値を、外部のマシンから読み出すことは可能であることにする。

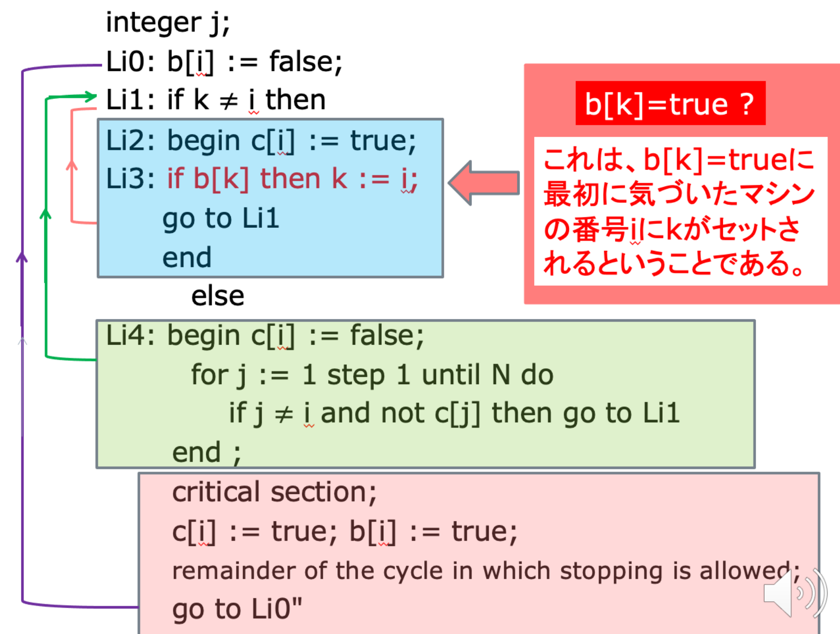
微妙な問題

ダイクストラらのアルゴリズムは、共有領域に置かれた k, b[k]の読み書きが、アトミックに行われることに依存している。

別の言葉で言えば、システム全体の排他性は、システムの一部の排他性を保証することによって成り立っている。

ランポートのアルゴリズムで、他のマシンのチケット番号を全て読み出す操作は、アトミックなものにはなり得ないはずだ。

それに対するランポートの答えは驚くべきものだ。



まず、そのアルゴリズムを見ておこう。

A new solution of Dijkstra's concurrent programming problem

Leslie Lamport
Communications of the ACM Volume 17 Issue 8
Aug. 1974 pp 453–455
https://dl.acm.org/doi/pdf/10.1145/361082.361093

ランポートのbakeryアルゴリズム

```
begin integer j;
  L1: choosing [i] := 1;
      number[i] := 1 + maximum (number[1], . . . , number[N]);
      choosing[i] := 0;
      for j = 1 step 1 until N do
        begin
          L2: if choosing[j] ≠ 0 then goto L2;
          L3: if number[j] ≠ 0 and (number [j], j) < (number[i],
                    i) then goto L3;
        end;
      critical section;
      number[i] := 0;
      noncritical section;
      goto L1;
end
```

ランポートのbakeryアルゴリズムへのコメント

```
begin integer j;
  L1: choosing [i] := 1;
      number[i] := 1 + maximum (number[1], . . . , number[N]);
      choosing[i] := 0;
      for j = 1 step 1 until N do
        begin
          L2: if choosing[j] ≠ 0 then goto L2;
          L3: if number[j] ≠ 0 and (number [j], j) < (number[i],
                  i) then goto L3;
        end;
      critical section;
      number[i] := 0;
      noncritical section;
      goto L1;
end
```

自分のチケット番号選択中

ランポートのbakeryアルゴリズムへのコメント

```
begin integer j;
  L1: choosing [i] := 1;
      number[i] := 1 + maximum (number[1], . . . , number[N]);
      choosing[i] := 0;
      for j = 1 step 1 until N do
        begin
          L2: if choosing[j] ≠ 0 then goto L2;
          L3: if number[j] ≠ 0 and (number [j], j) < (number[i],
                  i) then goto L3;
        end;
      critical section;
      number[i] := 0;
      noncritical section;
      goto L1;
end
```

それは、選択された全てのチケット番号で最大なものに、1を加えたものである。

ランポートのbakeryアルゴリズムへのコメント

```
begin integer j;
  L1: choosing [i] := 1;
      number[i] := 1 + maximum (number[1], . . . , number[N]);
      choosing[i] := 0;        チケット番号選択終了
      for j = 1 step 1 until N do
        begin
          L2: if choosing[j] ≠ 0 then goto L2;
          L3: if number[j] ≠ 0 and (number [j], j) < (number[i],
                 i) then goto L3;
        end;
      critical section;
      number[i] := 0;
      noncritical section;
      goto L1;
end
```

ランポートのbakeryアルゴリズムへのコメント

```
begin integer j;
   L1: choosing [i] := 1;
       number[i] := 1 + maximum (number[1], . . . , number[N]);
       choosing[i] := 0;
       for j = 1 step 1 until N do
          begin
             L2: if choosing[j] ≠ 0 then goto L2;
             L3: if number[j] ≠ 0 and (number [j], j) < (number[i],
                        i) then goto L3;
          end;
       critical section;
       number[i] := 0;
       noncritical section;
       goto L1;
end
```

N台のマシンのチケット番号をチェックする

ランポートのbakeryアルゴリズムへのコメント

```
begin integer j;
  L1: choosing [i] := 1;
      number[i] := 1 + maximum (number[1], . . . , number[N]);
      choosing[i] := 0;
      for j = 1 step 1 until N do
        begin
          L2: if choosing[j] ≠ 0 then goto L2;
          L3: if number[j] ≠ 0 and (number [j], j) < (number[i],
                    i) then goto L3;
        end;
      critical section;
      number[i] := 0;
      noncritical section;
      goto L1;
end
```

チケット番号を選択中なら、
選択が終わるまで待機する

ランポートのbakeryアルゴリズムへのコメント

```
begin integer j;
  L1: choosing [i] := 1;
      number[i] := 1 + maximum (number[1], . . . , number[N]);
      choosing[i] := 0;
      for j = 1 step 1 until N do
        begin
          L2: if choosing[j] ≠ 0 then goto L2;
          L3: if number[j] ≠ 0 and (number [j], j) < (number[i],
                  i) then goto L3;
        end;
      critical section;
      number[i] := 0;
      noncritical section;
      goto L1;
end
```

チケット番号がセットされていて、その番号が自分の番号より小さければ，そちらに優先権があるので待機。

ランポートのbakeryアルゴリズムへのコメント

```
begin integer j;
  L1: choosing [i] := 1;
      number[i] := 1 + maximum (number[1], . . . , number[N]);
      choosing[i] := 0;
      for j = 1 step 1 until N do
        begin
          L2: if choosing[j] ≠ 0 then goto L2;
          L3: if number[j] ≠ 0 and (number [j], j) < (number[i],
                    i) then goto L3;
        end;
      critical section;
      number[i] := 0;
      noncritical section;
      goto L1;
end
```

ここにいたるのは、先行するforループの中で、
全てのマシンが、L3の待機状態を脱した時である。
すなわち、全てのマシンのチケット番号より、自分の
チケット番号が小さい時、for ループは終了し、
自分が、critical sectionに入る。

ランポートのbakeryアルゴリズムへのコメント

```
begin integer j;
  L1: choosing [i] := 1;
      number[i] := 1 + maximum (number[1], . . . , number[N]);
      choosing[i] := 0;
      for j = 1 step 1 until N do
        begin
          L2: if choosing[j] ≠ 0 then goto L2;
          L3: if number[j] ≠ 0 and (number [j], j) < (number[i],
                  i) then goto L3;
        end;
      critical section;
      number[i] := 0;
      noncritical section;
      goto L1;
end
```

critical sectionを通過したら、自分のチケット番号をリセットする。その後、L1に戻って、再度、チケット番号を選択する。

ランポートのbakeryアルゴリズムの正しさ

ランポートによる
bakeryアルゴリズムの正しさの証明

ランポートは、アルゴリズムについての「三つの主張」が正しいことを示すことで、排他制御アルゴリズムの正しさを証明しようとする。

このセッションでは、その「三つの主張」を見ていこうと思う。

- 主張 1:　パン屋に先に入った者の番号が小さい
- 主張 2:　critical sectionに入った者の番号は、パン屋の中にいる他の者の番号より小さい
- 主張 3:　誰も critical section にいなければ、いずれ誰かがそこに入る

ランポートによる
bakeryアルゴリズムの正しさの証明

三つの主張で重要なのは、critical section に入れるのは一つのプロセッサーだけだという主張（主張 2）と、レジ待ちのプロセッサーは、いずれ、critical section に入るという主張（主張 3）の二つである。

前者（主張 2）は、排他制御の safety という性質であり、
後者（主張 3）は、排他制御の liveness という性質である。

ただし、この論文では、ランポートは、こうした言葉を使っていない。

ランポートのbakeryアルゴリズム

改めて、彼のアルゴリズムを確認しよう。

式の中で、

(number [i], i) < (number [k], k) が成り立つのは、

number [i] < number [k] の場合、あるいは、

もし、number [i] = number [k] なら、i < k の場合である。

ランポートのbakeryアルゴリズム

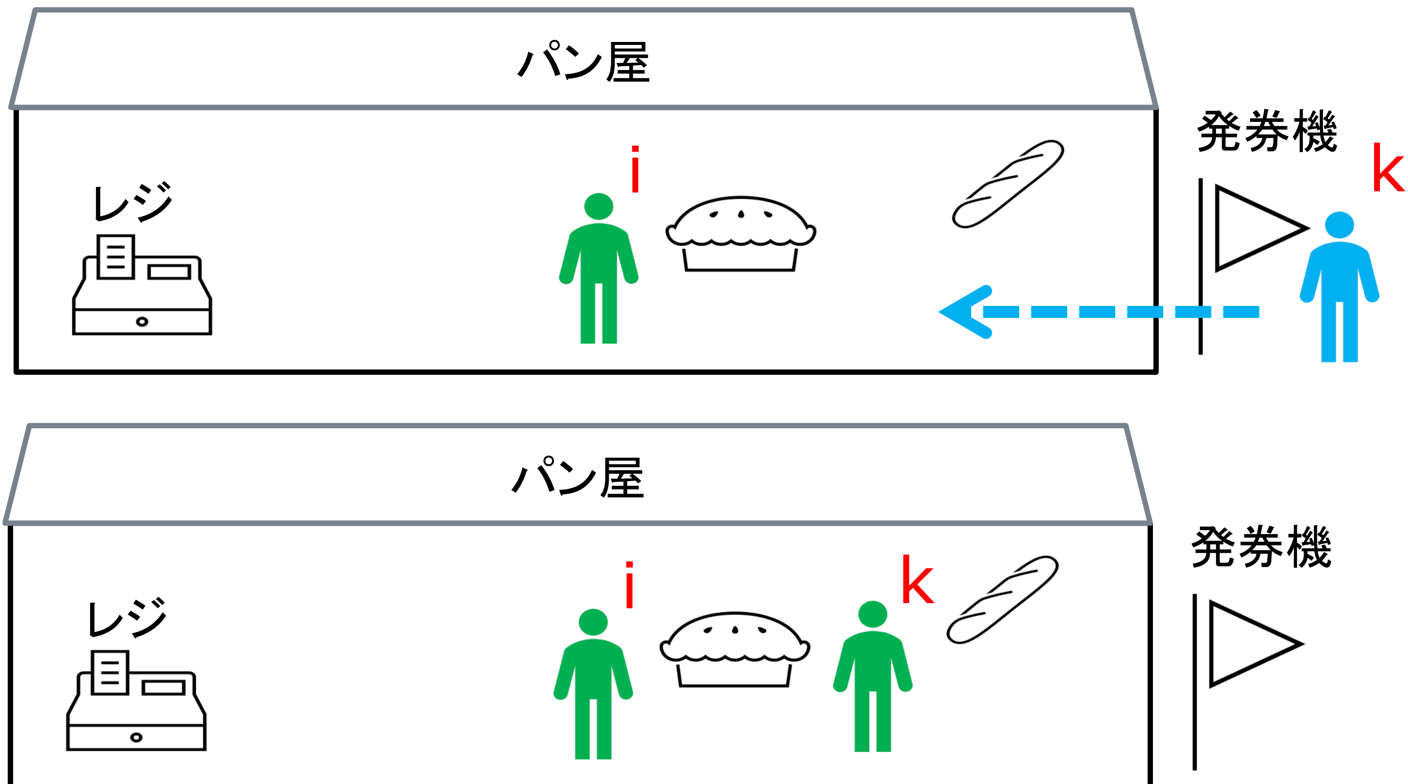
```
begin integer j;
  L1: choosing [i] := 1;
      number[i] := 1 + maximum (number[1], . . . , number[N]);
      choosing[i] := 0;
      for j = 1 step 1 until N do
        begin
          L2: if choosing[j] ≠ 0 then goto L2;
          L3: if number[j] ≠ 0 and (number [j], j) < (number[i],
                    i) then goto L3;
        end;
      critical section;
      number[i] := 0;
      noncritical section;
      goto L1;
end
```

```
L1: choosing [i] := 1;
    number[i] := 1 + maximum (number[1], . . . , number[N]);
    choosing[i] := 0;
```

```
for j = 1 step 1 until N do
  begin
    L2: if choosing[j] ≠ 0 then goto L2;
    L3: if number[j] ≠ 0 and (number [j], j) < (number[i],
            i) then goto L3;
  end;
```

主張 1

先に入った者の番号が小さい



number [i] < number [k]

Proof of Correctness
Assertion 1

主張 1:

もしプロセッサー i と k がパン屋の中にいて、kがパン屋の戸口に立つ前に、i がパン屋に入ったとすると、
number [i] < number [k] である。

証明:

仮定から、k が現在の値 number[k] を選んでいる間に、number[i] は現在の値を持っていた。だから、k は、number[k] ≥ 1 + number[i] である数字を選んでいたはずである。

主張 2

critical sectionに入った者の番号は パン屋の中にいる他の者の番号より少ない

$k \neq i$ なら、

(number [i], i) < (number [k], k)

critical section に入れるのは一人だけということ。
これは、排他制御の safety という性質である。

Proof of Correctness
Assertion 2

主張 2: もし、プロセッサー i が critical section にいて、プロセッサー k がパン屋の中にいて、k ≠ i なら、

(number [i], i) < (number [k], k) である。

証明:

choosing[k] は、本質的には ゼロか非ゼロかの二つの値を持つだけである。processor i からみれば、その値の読み・書きは瞬時に終わる。そして、読みと書きが同時に起きることはない。

たとえば、choosing[k]の値が、プロセッサー i の読みの途中で0から1に変わったとしても、その読みはもし値 0を得たのなら、読みが最初に起きたと考えられる。もし、そうでないなら、書きが最初に起きたと言われる。この証明での全ての時間は、プロセッサー i からの視点のものだ。

Proof of Correctness
Assertion 2

L2 で j = k の最後の実行の間に、プロセッサー i が、choosing [k] を読む時間を t_{L2} とし、L3 で j = k の最後の実行をプロセッサー i が始める時間を t_{L3} としよう。この時、
$t_{L2} < t_{L3}$ である。

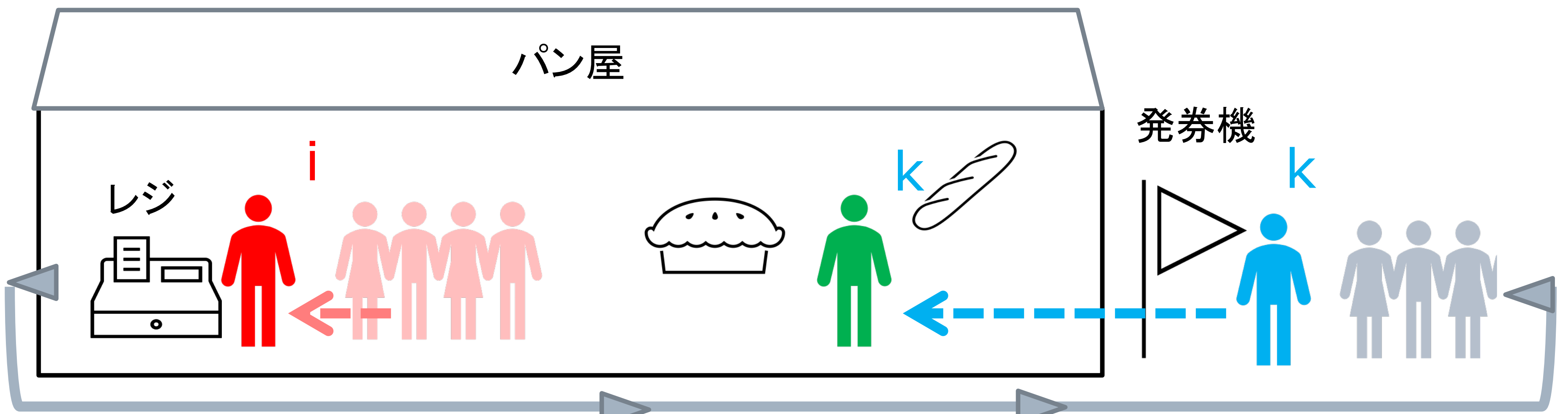
プロセッサー k が、number[k] の現在の値を選んでいる時、k が戸口に立った時間を t_e 、number[k] の値を書き終えた時間を t_w、戸口を離れた時間を t_c としよう。この時、
$t_e < t_w < t_c$ である。

なぜなら、choosing[k] は、時間 t_{L2} で 0 に等しく、

(a) $t_{L2} < t_e$ か、

(b) $t_c < t_{L2}$ の、

どちらかになる。



パン受取　レジ待ち　入店して買物　整理券受取　発券待ち

kの振る舞い

t_e

```
L1: choosing [i] := 1;
    number[i] := 1 + maximum (number[1], . . . , number[N]);
    choosing[i] := 0;
```

t_c　t_w

$t_e < t_w < t_c$

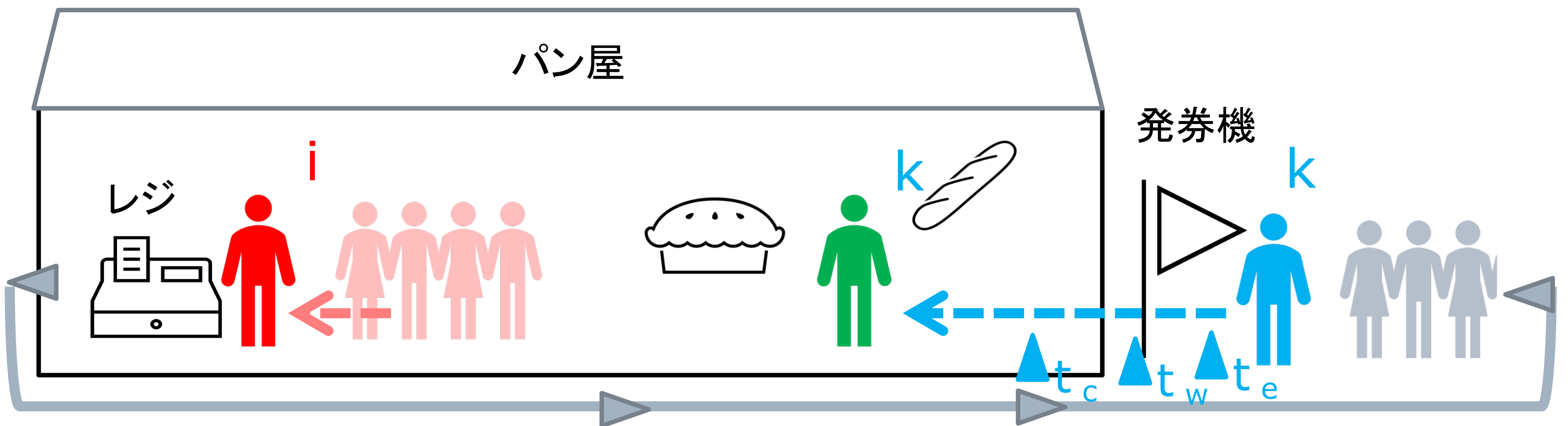
i の振る舞い

```
for j = 1 step 1 until N do
  begin
    L2: if choosing[j] ≠ 0 then goto L2;
    L3: if number[j] ≠ 0 and (number [j], j) < (number[i],
          i) then goto L3;
  end;
```

t_{L2}　t_{L3}

$t_{L2} < t_{L3}$

critical section へ



パン受取　レジ待ち　入店して買物　整理券受取　発券待ち

kの振る舞い

t_e

```
L1: choosing [i] := 1;
    number[i] := 1 + maximum (number[1], . . . , number[N]);
    choosing[i] := 0;
```

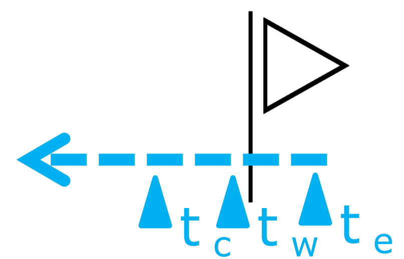
t_c t_w

i の振る舞い

```
for j = 1 step 1 until N do
  begin
    L2: if choosing[j] ≠ 0 then goto L2;
    L3: if number[j] ≠ 0 and (number [j], j) < (number[i],
          i) then goto L3;
  end;
```

t_{L2} t_{L3}

critical section へ



choosing[k]=0
at t_{L2}

パン屋

発券機

レジ

i

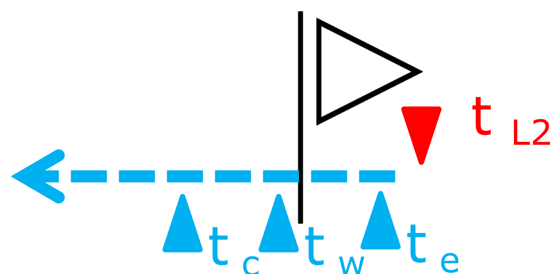
k

k

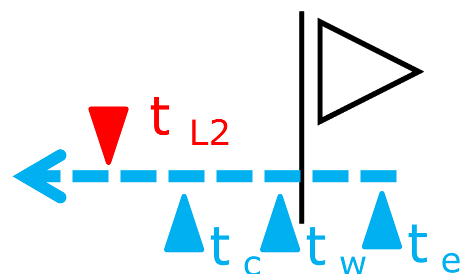
t_c t_w t_e

パン受取　レジ待ち　入店して買物　整理券受取　発券待ち

choosing[k]=0 at t_{L2} となるのは、次の二つの場合



あるいは



$t_{L2} < t_e$

ケース (a)

あるいは

$t_c < t_{L2}$

ケース (b)

Proof of Correctness
Assertion 2

ケース (a) $t_{L2} < t_e$ の場合、iの方がkより先に発券を受けて先に店に入ったということ。主張 1. は number [i] < number [k] を導く。それゆえ、主張 2. も成り立つ。

ケース (b) $t_c < t_{L2}$ の場合、次の式が成り立つ。

$t_w < t_c < t_{L2} < t_{L3}$ だから、$t_w < t_{L3}$ 。

よって、t_{L3} で始まる命令 L3 の実行中は、プロセッサー i は、t_w で書き込まれたnumber[k]の現在の値を読む。

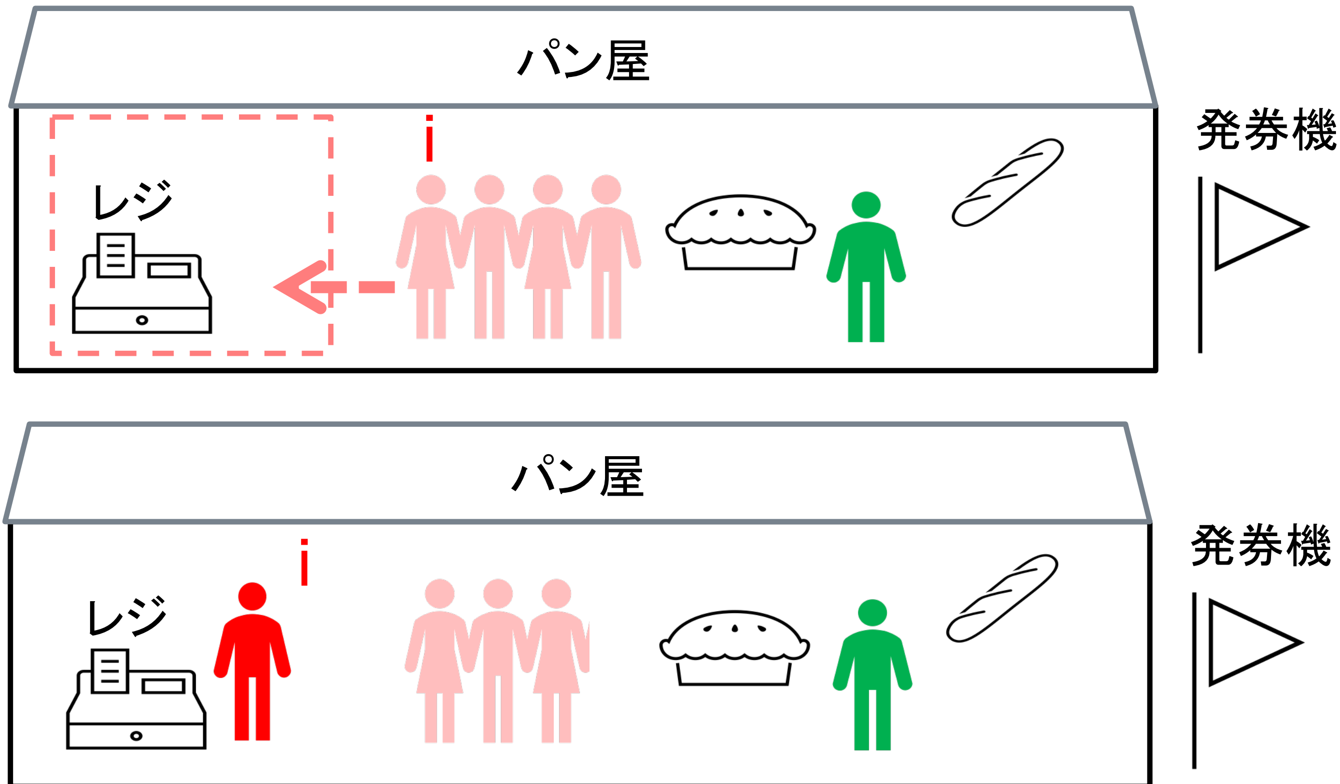
i は、j = k を再び実行することはないので、

(number [i], i) < (number [k], k) でなければならない。

よって、このケースでも、主張 2. は成り立つ。

主張 3

誰も critical section にいなければ、いずれ誰かがそこに入る



レジ待ちのプロセッサー i は、いずれ、レジ待ちの for ループを抜け、critical section に入る。
これは、排他制御の liveness という性質である。

Proof of Correctness
Assertion 3

主張 3:

制限された数だけしかプロセッサーのエラーは起きないとしよう。critical section にプロセッサーがおらず、パン屋の中にはプロセッサーがいる場合、あるプロセッサーは、いずれcritical sectionに入らなければならない。

証明:

どのプロセッサーも、ずっとcritical sectionに入らないとしよう。そうだとすれば、どのプロセッサーもそれ以降はパン屋に出入りしない時間があるだろう。この時点で、プロセッサー i がパン屋内で最小の(number [i], i)を持っていたとしよう。そうすれば、プロセッサー i は、いつか for ループを抜け、critical sectionに入らなければならない。これは矛盾する。

三つの主張の帰結

主張 2. は、どの時点でも、critical section　には多くても一つのプロセッサーしかいないことを意味する。

主張 1. と主張 2. は、プロセッサーは「最初に入ったものが最初にサービスを受ける」というルール・ベースで、critical section に入ることを証明する。だから、システム全体がデッドロックにならなければ、個別のプロセッサーはブロックされえない。

主張 3. は、システムは、次の場合にのみデッドロックに陥ることを意味する。すなわち、あるプロセッサーがcritical section 内で停止する場合か、あるいは、無制限にプロセッサーのエラーが続き再試行を繰り返す場合である。

後者の場合は、次のようにしてシステムは停止する。もしプロセッサー j が連続的にエラーを起こし再起動を繰り返すなら、不運なプロセッサー i は、つねに choosing[j] = 1 を見つけてしまい、L2で無限ループする。

留意すべきこと
number[i]は際限なく増大する

このアルゴリズムの問題は、パン屋の中に常に少なくても一つのプロセッサーが入っているのなら、number[i] の値は、際限なくどんどん大きなものになりうるということである。

この問題は、有限個の整数の集合をサイクリックに利用するというような単純な方法では、解決できない。 ...

幸運なことに、実践的に考えれば、たとえば 1ミリ秒に一回の割合で、number[i] を1だけ増やしたとしても、この操作を一年間続けても、たかだか number[i] $< 2^{35}$ である。...

読み出した値が正しければ、このアルゴリズムは正しい

先に見たこのアルゴリズムの正しさの証明は、チケット番号全体に対する読み書きが、アトミックに行われる必要はないことを示している。

このbakeryアルゴリズムは、その値がコンカレントに書き込まれたものでないとしても、その読み出しが正しい値を返すならば、正しいのである。書き込みと重なった読み出しがどんな値を返すかは問題ではないのだ。

コンカレントな書き込みによって9から10に変わろうとしているある数の呼び出しが、2496という値を返したとしても、このアルゴリズムは正しい。

下位のレベルの相互排除を前提としない、相互排除アルゴリズム

bakeryアルゴリズムの、この驚くべき性質は、それがプロセスがチケット番号に相互排除的にアクセスすることを想定していない相互排除のアルゴリズムの実装であることを意味している。**それは、下位のレベルの相互排除を前提としない、相互排除アルゴリズムの最初の実装だった。**

1973年当時は、そうしたことは不可能だと考えられていた。1990年になっても、専門家はそれは不可能だと考えていた。

このアルゴリズムの厳密な証明は、次の論文で与えられる。

Lamport, L. A new approach to proving the correctness of multi-process programs. *ACM Trans. Program.Lang. Syst. 1, 1 (July 1979), 8497.*
https://lamport.azurewebsites.net/pubs/new-approach.pdf

新しいbakeryアルゴリズムと
並列性の表現

新しいbakeryアルゴリズム

Lamport, L. A new approach to proving the correctness of multi-process programs. *ACM Trans. Program.Lang. Syst. 1, 1 (July 1979), 8497.*
https://lamport.azurewebsites.net/pubs/new-approach.pdf

ランポートのbakeryアルゴリズム 1973年

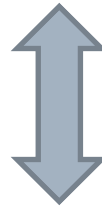
```
begin integer j;
  L1: choosing [i] := 1;
      number[i] := 1 + maximum (number[1], . . . , number[N]);
      choosing[i] := 0;
      for j = 1 step 1 until N do
        begin
          L2: if choosing[j] ≠ 0 then goto L2;
          L3: if number[j] ≠ 0 and (number [j], j) < (number[i],
                  i) then goto L3;
        end;
      critical section;
      number[i] := 0;
      noncritical section;
      goto L1;
end
```

新しいbakeryアルゴリズム
1979年

integer j;
repeat *noncritical section*;
L1: $n[i] :> 0$;
L2: $n[i] :> maximum\ (n[1], \dots, n[N])$;
L3: **for all** $j \in \{1, \dots, N\}$ **do**
wait until $n[j] = 0$ **or** $(n[j], j) \geq (n[i], i)$ **od**;
critical section;
L4: $n[i] := 0$
end repeat

新旧のアルゴリズムの比較

$L1:$ *choosing* $[i] := 1;$
number $[i] := 1 + \mathit{maximum}\ (\mathit{number}[1], \ldots, \mathit{number}[N]);$
choosing $[i] := 0;$



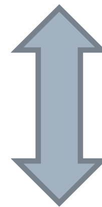
L1: $n[i] :> 0;$
L2: $n[i] :> \mathit{maximum}\ (n[1], \ldots, n[N]);$

代入操作 $''a :> b''$

“a ：> b” は、”aにbより大きい任意の値を代入する”こととしよう。
それは、”a ：= b” が”aにbの値を代入する“ のと同様である。

新旧のアルゴリズムの比較

```
for j = 1 step 1 until N do
  begin
    L2: if choosing[j] ≠ 0 then goto L2;
    L3: if number[j] ≠ 0 and (number [j], j) < (number[i],
            i) then goto L3;
  end;
```

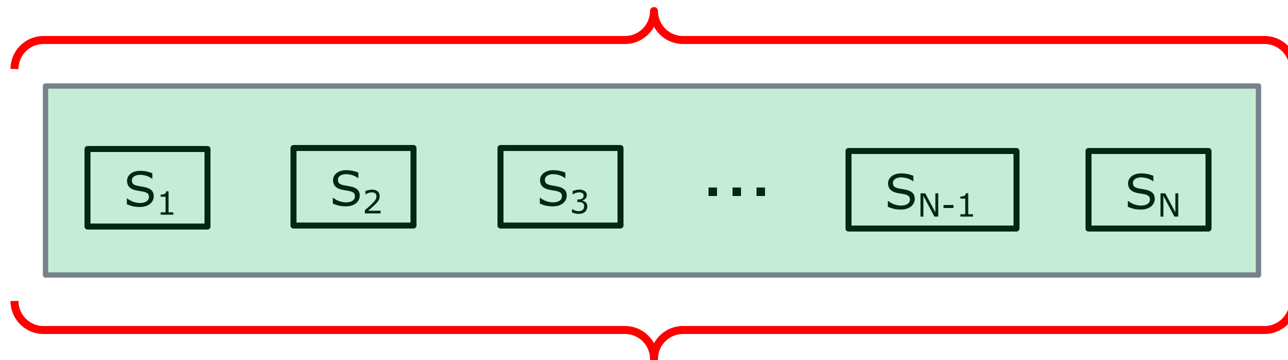


```
L3: for all j ∈ {1, ... , N} do
        wait until n[j] = 0 or (n[j], j) ≥ (n[i], i) od;
```

並列実行の命令
$\boldsymbol{for\ all}\ j \in \{1, \ldots, N\}\ \boldsymbol{do}\ S_j\ \boldsymbol{od}$

$$\boldsymbol{for\ all}\ j \in \{1, \ldots, N\}\ \boldsymbol{do}\ S_j\ \boldsymbol{od}$$

この命令は、命令 S_1 , …, S_N がコンカレントに（すなわち、順序を問わず並列に）実行されるべきことを意味する。ここで、S_1 は、命令中のS_j の j を1に置き換えた命令である。



S_1 , …, S_N はコンカレントに 実行される

待機命令 wait until

次の命令を追加した。

wait until *condition*

次の命令の短縮形である。Condition が成り立つまで待機する。

L: if not *condition* **then goto L fi.**

$\boldsymbol{wait\ until}\ n[j] = 0\ or\ (n[j], j) \geq n[i].i)$

$n[j] = 0\ or\ (n[j], j) \geq n[i].i)$ が成り立つまで待機

成り立ったら待機を抜けて次のステップに

$\boldsymbol{for\ all\ j} \in \{1, \ldots, N) \ \boldsymbol{do}$
$\boldsymbol{wait\ until}\ n[j] = 0\ or\ (n[j], j) \geq n[i].i)\ \boldsymbol{od}$
の働き

次のN個のプロセスを並列実行する

TEST n[1] $\equiv n[1] = 0\ or\ (n[1], 1) \geq n[i].i)$ が真なら次のステップに
TEST n[2] $\equiv n[2] = 0\ or\ (n[2], 2) \geq n[i].i)$が真なら次のステップに
TEST n[3] $\equiv n[3] = 0\ or\ (n[3], 3) \geq n[i].i)$が真なら次のステップに

⋮

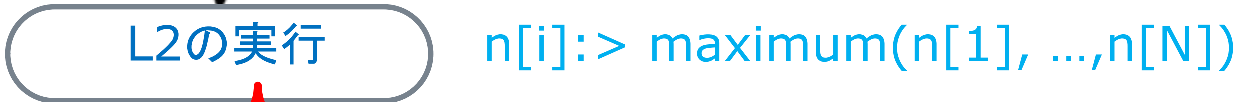
⋮

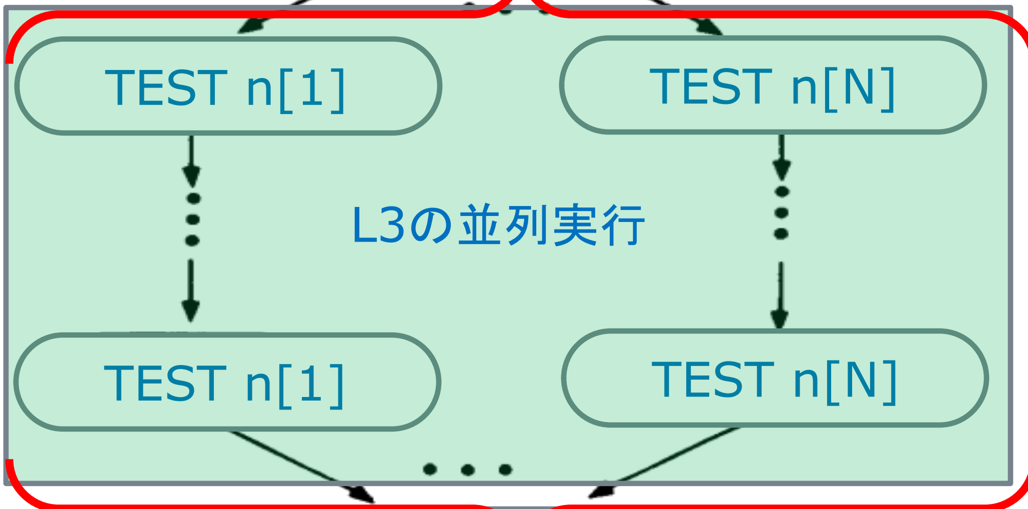
TEST n[N] $\equiv n[N] = 0\ or\ (n[N], N) \geq n[i].i)$が真なら次のステップに

```
integer j;
repeat noncritical section;
    L1: n[i] :> 0;
    L2: n[i] :> maximum (n[1], ... , n[N]);
    L3: for all j ∈ {1, ... , N} do
            wait until n[j] = 0 or (n[j], j) ≥ (n[i], i) od;
        critical section;
    L4: n[i] := 0
end repeat
```

非critical sectionの実行

n[i]:>0





$\boldsymbol{for\ all}\ j \in \{1, \ldots, N)\ \boldsymbol{do}$
$\boldsymbol{wait\ until}\ n[j] = 0\ or$
$(n[j], j) \geq n[i]. i)\ \boldsymbol{od}$

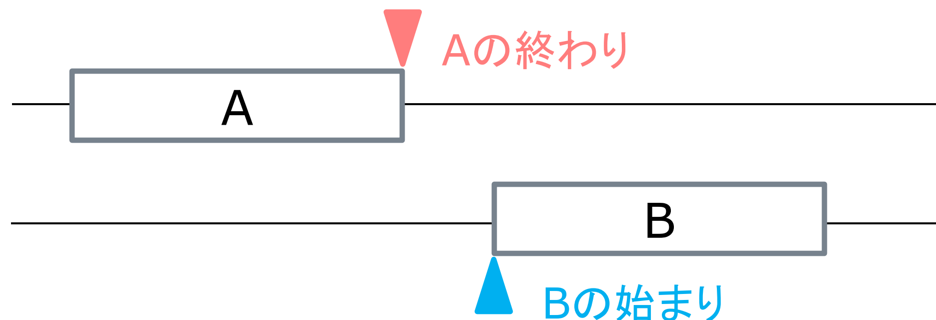
n[i]:=0

非critical sectionの実行

プロセス実行の時間的順序

基本的な順序 1
Bが始まる前にAが終わる

プロセスBが始まる前にプロセスAが終わる場合、



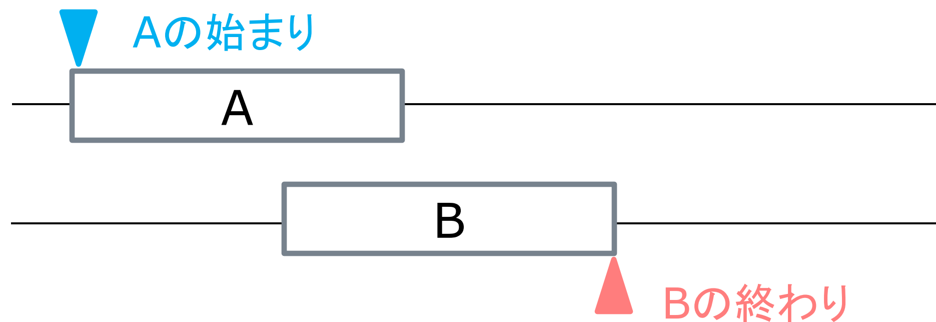
この時、

A ⟶ B

と表す。

基本的な順序 2
Bが終わる前にAが始まる

プロセスBが終わる前にプロセスAが始まる場合、



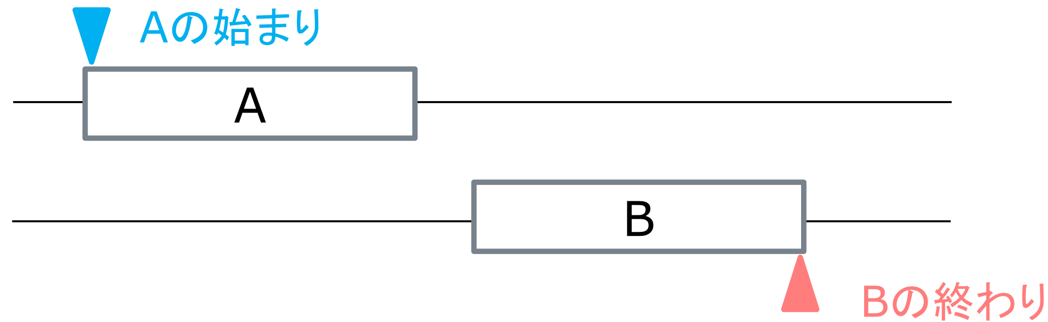
この時、

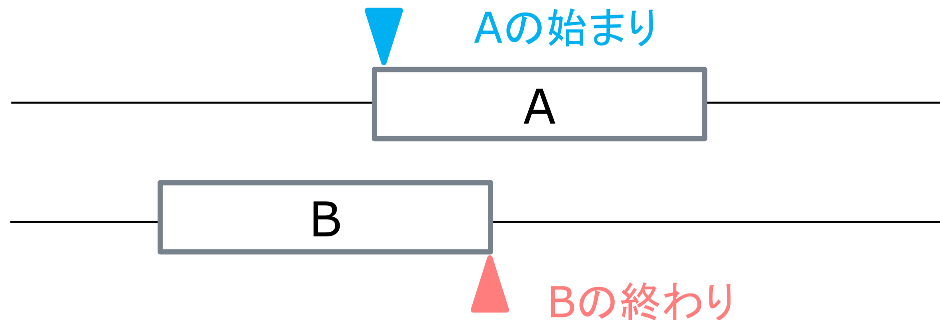
A - - → B

と表す。

A - - -> Bのバリエーション

プロセスBが終わる前にプロセスAが始まる、





二つの順序関係の満たす性質

A1. (a) $A \longrightarrow B \longrightarrow C$ implies $A \longrightarrow C$

($\longrightarrow$ transitively closed)

(b) $A \not\longrightarrow A$. ($\longrightarrow$ irreflexive)

A2. $A \longrightarrow B$ implies $A \dashrightarrow B$ and

$B \not\dashrightarrow A$.

A3. $A \longrightarrow B \dashrightarrow C$ or $A \dashrightarrow B \longrightarrow C$

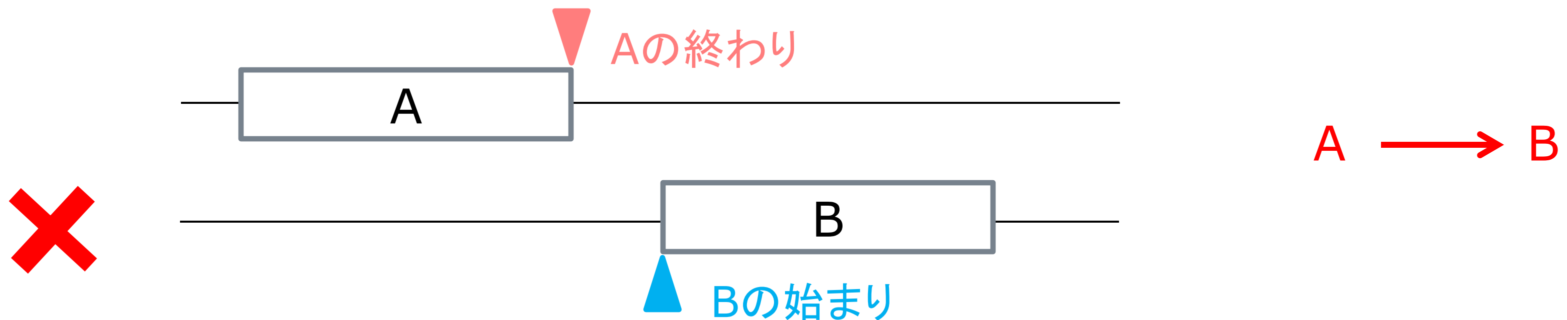
implies $A \dashrightarrow C$.

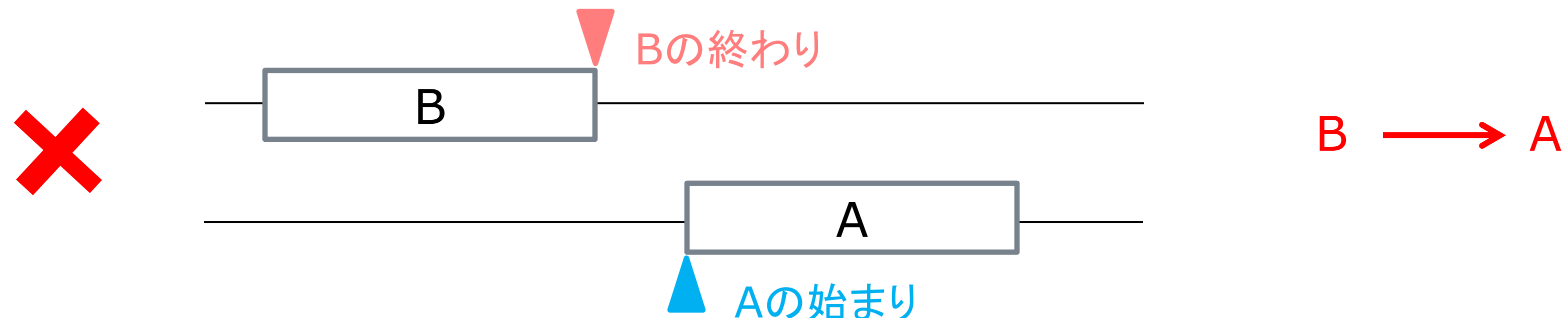
A4. $A \longrightarrow B \dashrightarrow C \longrightarrow D$ implies $A \longrightarrow D$.

AとBが並列であるということ

AとBが並列であるということは、Aが終わってからBが始まるわけでも、Bが終わってからAが始まるわけでもないということ。

次の図式は、いずれも並列ではない。





AとBが並列であるということ

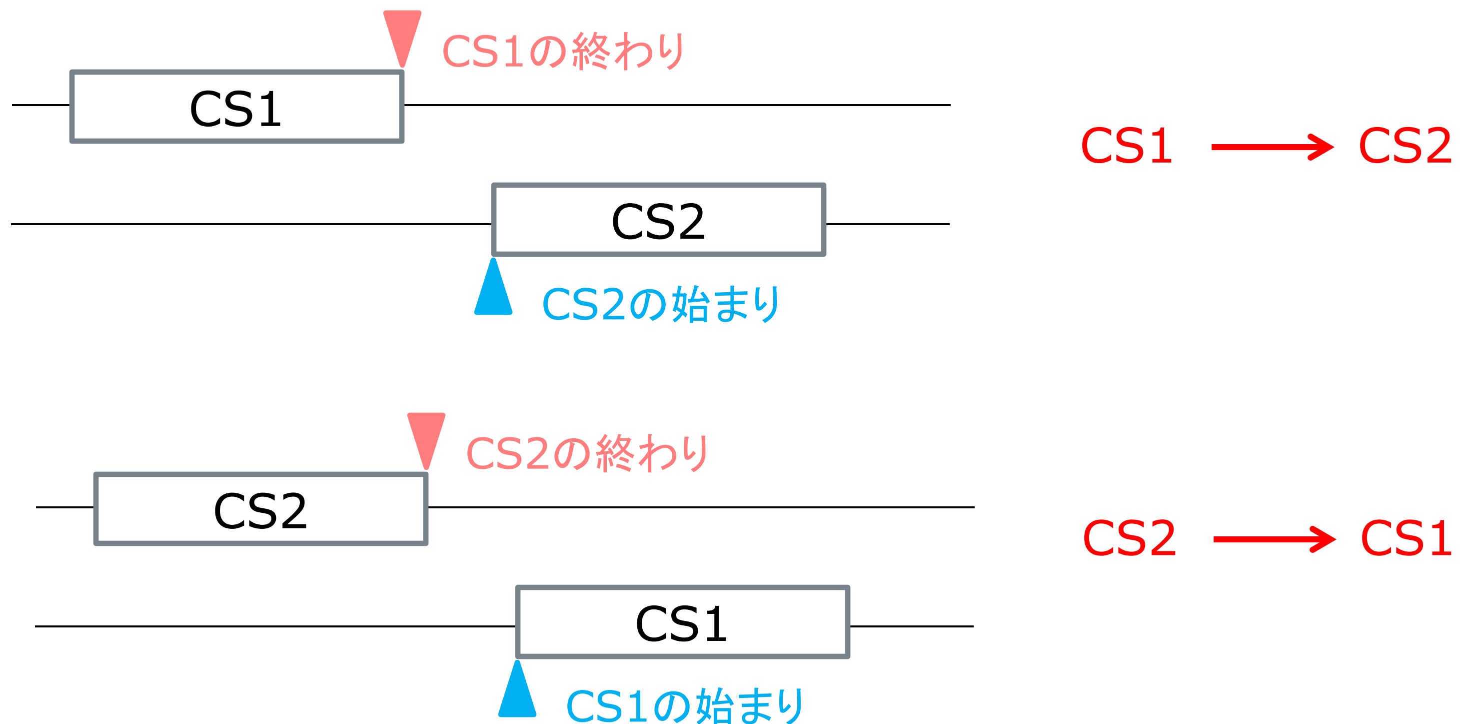
AとBが並列であるということは、

$A \not\rightarrow B$　かつ　$B \not\rightarrow A$

と表現される

CS1とCS2が相互排除であるということ

CS1とCS2が相互排除であるということは、CS1が終わってからCS2が始まるか、あるいは、CS2が終わってからCS1が始まるということである。次の図式は、相互排除である。



CS1とCS2が相互排除であるということ

CS1とCS2が相互排除であるということは

CS1 ⟶ CS2　あるいは　CS2 ⟶ CS1

と表現される

Part III

生産者消費者同期

並列･分散アルゴリズムの基礎

Part III　生産者消費者同期

- ダイクストラのsemaphore
- 生産者消費者同期アルゴリズムを記述する
- Producer-Consumerアルゴリズムの振る舞いを図解する

ダイクストラのsemaphore

基本的な並列プログラミングの問題としての生産者･消費者同期

コンピュータの最初期から、ハードウェア的には、バッファーは使われてきた。

しかし、この問題を初めて並列プログラミングの問題として捉えたのは、ダイクストラだった。

Dijkstra, E.W. Cooperating sequential processes. In *Programming Languages*. F. Genuys, ed. Academic Press, New York, 1968, 43112. Originally appeared as EWD123 (1965).
https://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html

https://pure.tue.nl/ws/files/4279816/344354178746665.pdf

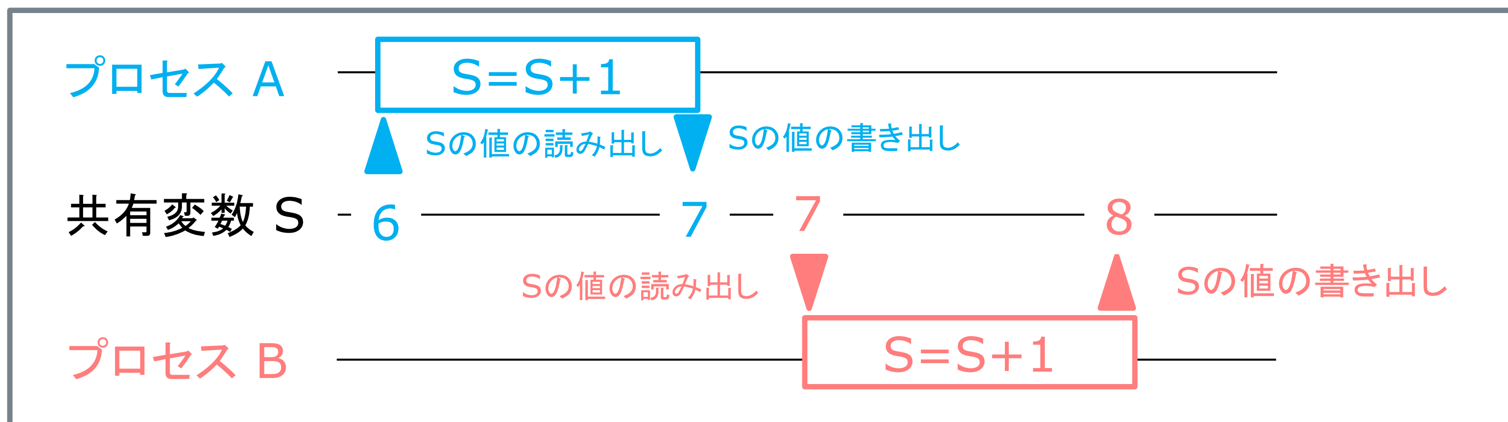
ダイクストラのSemaphore

- Semaphoreは、複数のプロセスによって共有される非負の整数値をとる変数である。（その値は、0 と 1 でもいい　-- Binary semaphore。それ以外を counting semaphore と呼ぶ。）
- Semaphoreには、二つの操作、VとPが定義されている。
- Semaphoreの名前をSとすると、操作 V, P を適用することをV(S), P(S) で表す。
- 操作Vは、semaphoreの値を一つ増やす。
- 操作Pは、semaphoreの値を一つ減らす。
  現在の値が、0なら、操作Pは、他のプロセスが操作Vを行うまでブロックして待機する。
- 操作Vも操作Pもアトミックな操作で、同時に行われることはない。

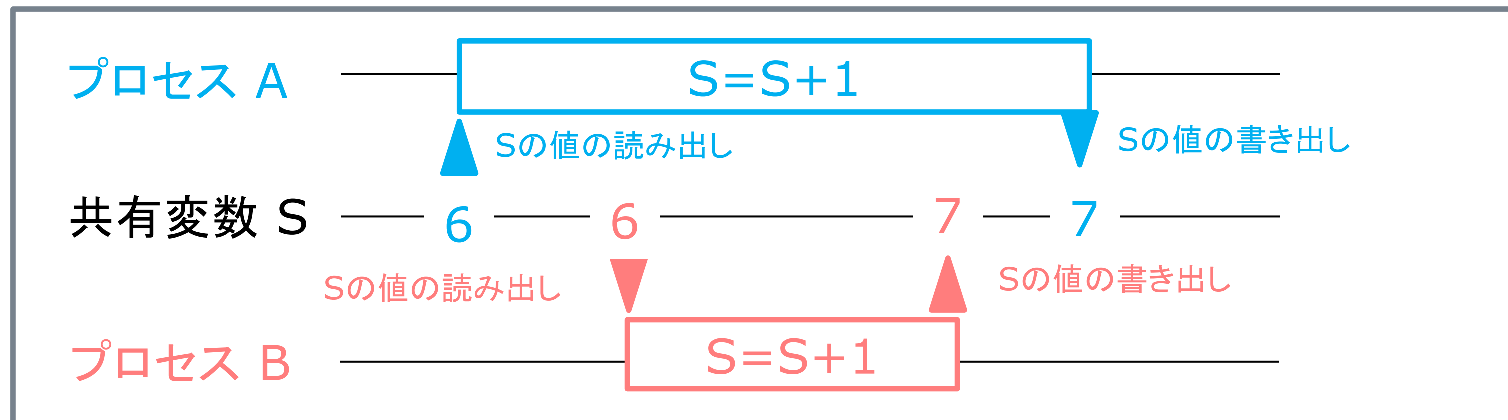
通常のS=S+1とアトミックなV(S)との違い

通常のS=S+1:二つのプロセスが共有変数Sにアクセスした場合

A ⟶ B



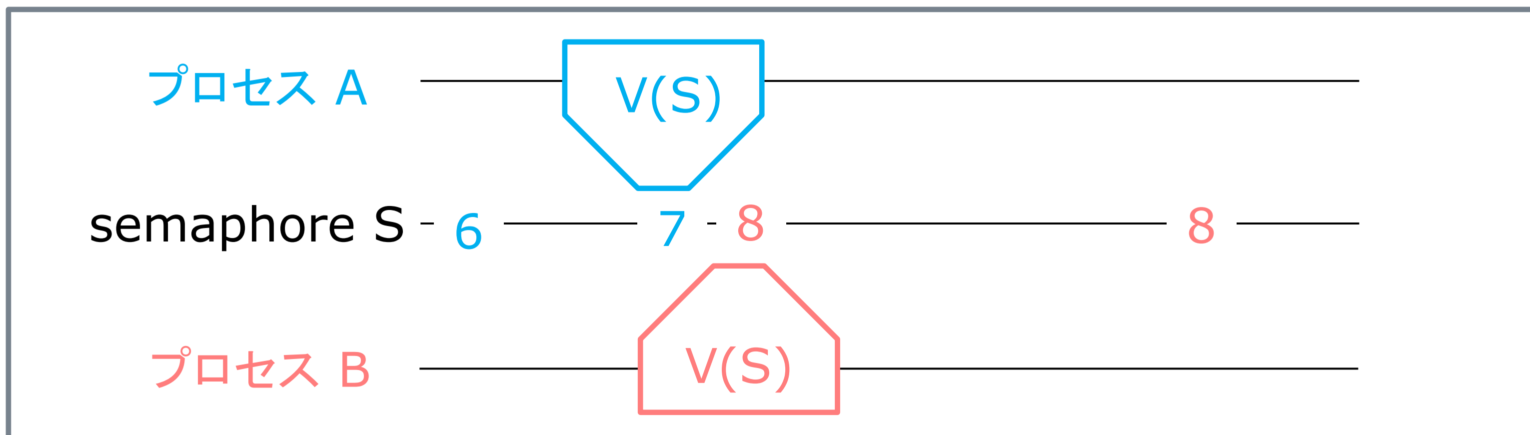
A - -> B



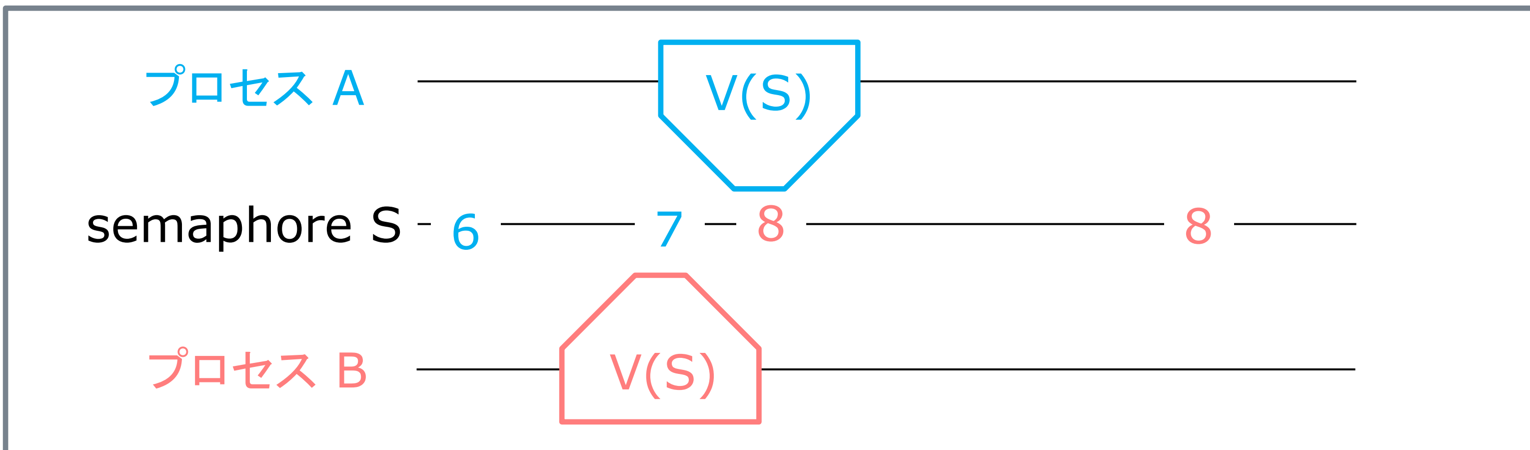
通常のS=S+1とアトミックなV(S)との違い

V(S):二つのプロセスが、semaphore Sにアクセスした場合

A → B



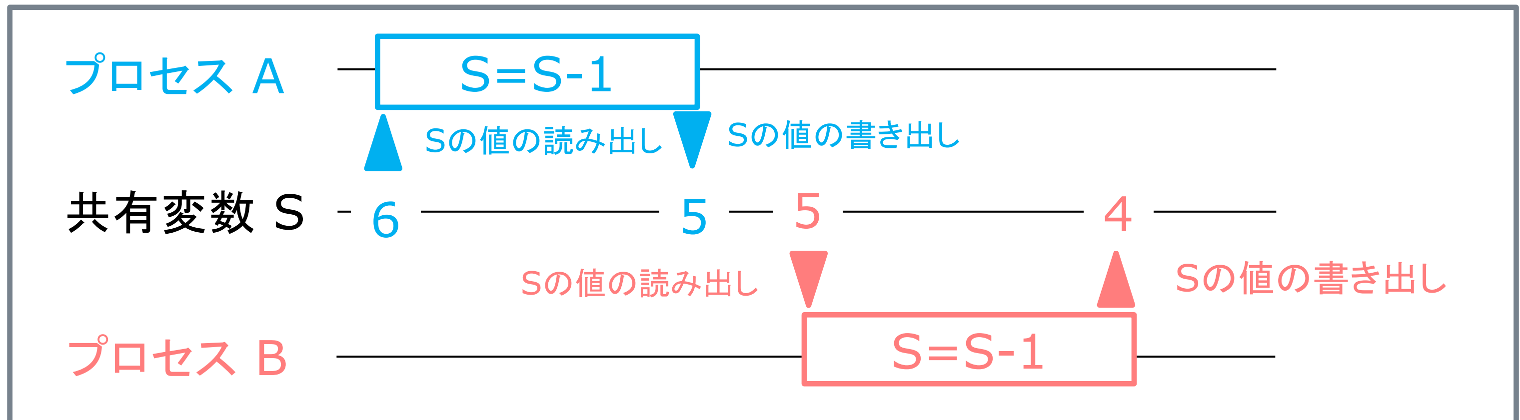
B → A



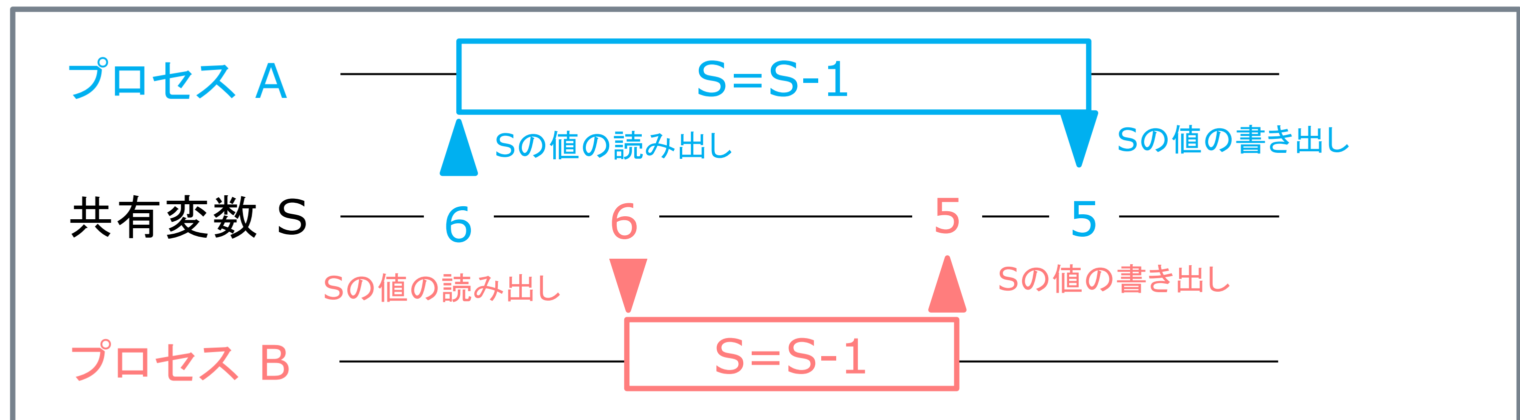
通常のS=S−1とアトミックなP(S)との違い

通常のS=S−1:二つのプロセスが共有変数Sにアクセスした場合

A → B



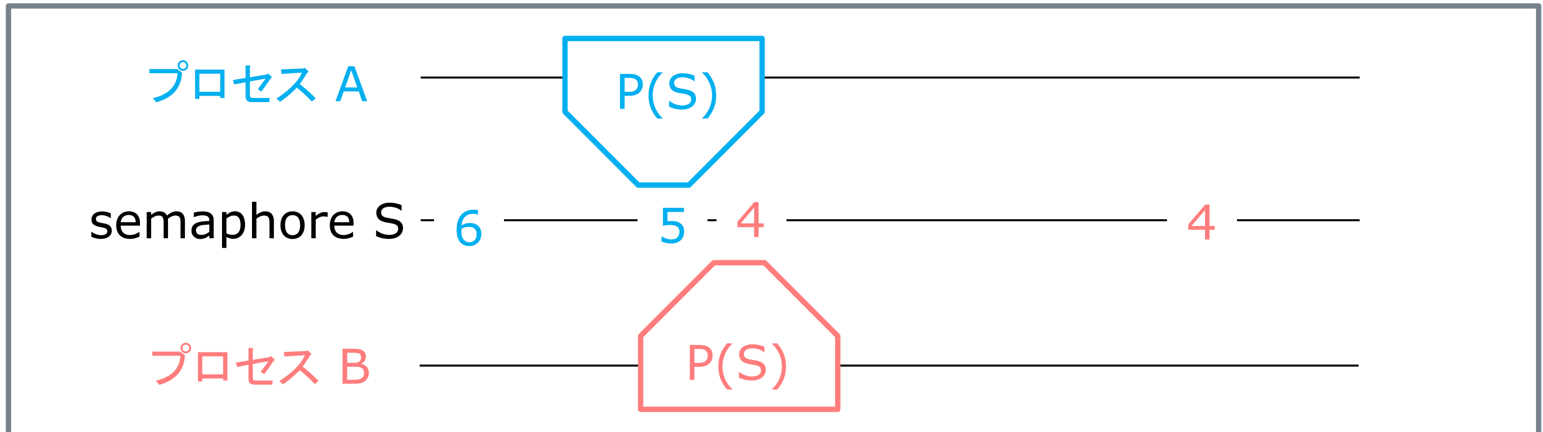
A - -> B



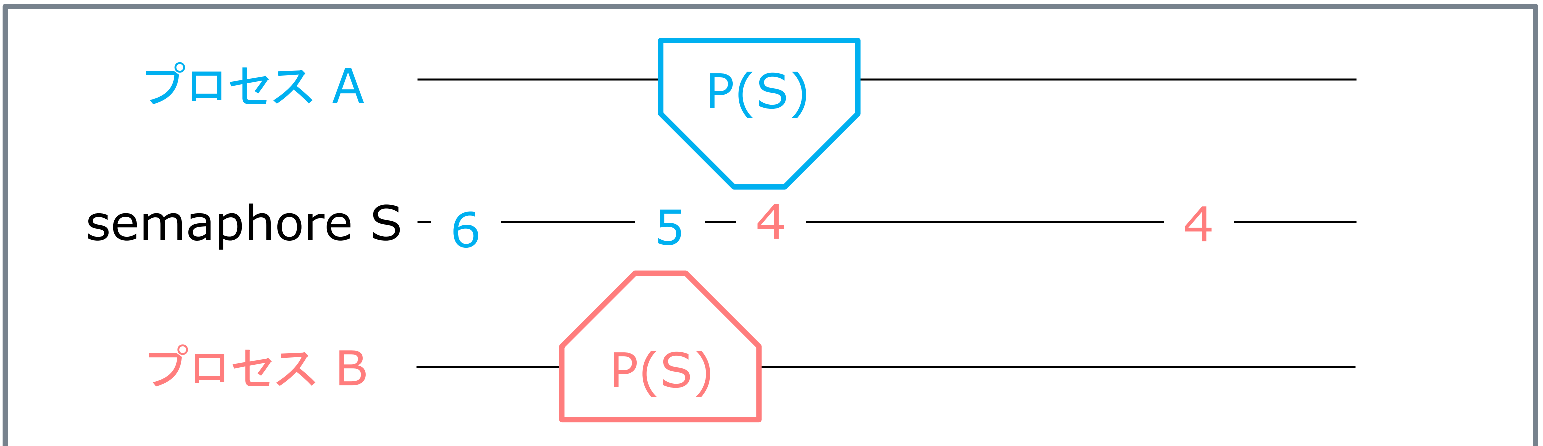
通常のS=S−1とアトミックなP(S)との違い

P(S):二つのプロセスが、semaphore Sにアクセスした場合

A → B



B → A



Semaphoreを使ったダイクストラの mutual exclusionのアルゴリズム

begin integer free; free:= 1;
    parbegin
    process 1: **begin**...............**end**;
    process 2: **begin**...............**end**;
      .
    process N: **begin**...............**end**;
    parend
end

/** with the i-th process of the form: **/

<u>**process**</u> i: **begin**
        Li: **P(free);** critical section i; **V(free);**
            remainder of cycle i; **goto** Li
      end

Semaphoreを使ったダイクストラの mutual exclusionのアルゴリズム

```
begin integer free; free:= 1;
    parbegin
    process 1: begin....P...V.....end;
    process 2: begin....P...V.....end;
      .
    process N: begin.....P...V.....end;
    parend
end


/** with the i-th process of the form: **/

process i: begin
             Li: P(free); critical section i; V(free);
                remainder of cycle i; goto Li
           end
```

Semaphoreを使ったダイクストラの mutual exclusionのアルゴリズム

begin integer free; free:= 1;
    parbegin
    process 1: **begin**....P...V.....**end**;
    process 2: **begin**....P...V.....**end**;
      .
    process N: **begin**.....P...V.....**end**;
    parend
end

"free = 1" は、どのプロセスもcritical sectionに入っていないことを意味し、
"free = 0" は、一つのプロセスが critical sectionに入っていることを意味する。

process i: **begin**
        Li: **P(free);** critical section i; **V(free);**
            remainder of cycle i; **goto** Li
      end

Semaphoreを使ったダイクストラの生産者消費者同期のアルゴリズム

```
begin integer number of queuing portions;
    number of queuing portions:= 0;
    parbegin
    producer: begin
            again 1: produce the next portion;
                     add portion to buffer;
                     V(number of queuing portions);
                     goto again1
              end;
    consumer: begin
            again 2: P(number of queuing portions);
                     take portion from buffer;
                     process portion taken;
                     goto again 2
              end
    parend
end
```

生産者消費者同期アルゴリズムを記述する

生産者消費者同期アルゴリズムを記述する

前回のセッションでは、ダイクストラのsemaphoreを用いた、生産者消費者同期アルゴリズムを見てきた。

今回のセッションでは、この問題へのランポートのアルゴリズムを見ていく。

留意して欲しいのは、ダイクストラの定式化は、ALGOL風の疑似コードを持ちいたものだが、ランポートの定式化は、アルゴリズムを記述する言語の提案を含むものだということである。

Semaphoreを使ったダイクストラの疑似コードによる Producer-consumer synchronization アルゴリズム

```
begin integer number of queuing portions;
    number of queuing portions:= 0;
    parbegin
    producer: begin
            again 1: produce the next portion;
                     add portion to buffer;
                     V(number of queuing portions);
                     goto again1
              end;
    consumer: begin
            again 2: P(number of queuing portions);
                     take portion from buffer;
                     process portion taken;
                     goto again 2
              end
    parend
end
```

ランポートのアルゴリズム記述言語を用いた Producer-consumer synchronization アルゴリズム

```
--algorithm PC {
    variables in = Input,  out = ⟨⟩, buf = ⟨⟩;
    fair process (Producer = 0) {
      P: while (TRUE) {
            await Len(buf) < N;
            buf  := Append(buf, Head(in));
            in   := Tail(in)                   }}
    fair process (Consumer = 1) {
      C: while (TRUE) {
            await Len(buf) > 0;
            out  := Append(out, Head(buf));
            buf  := Tail(buf)                  }}  }
```

FIFO (first-in-first-out) queue

ここでは、生産者消費者同期の問題を、制限付きの（バッファーの容量が有限という意味）FIFO (first-in-first-out) queueの問題として考える。

生産者のプロセスは入力をN個の容量をもつバッファーに読み込み、消費者のプロセスはバッファーから順番に出力する。

ここでは、次の三つの変数が用いられている。

- ***in*** は、まだ読み込まれていない入力データの無限列である。
- ***buf*** はN個の値を保持できるバッファーである。
- ***out*** は、これまで出力されたデータ列である。

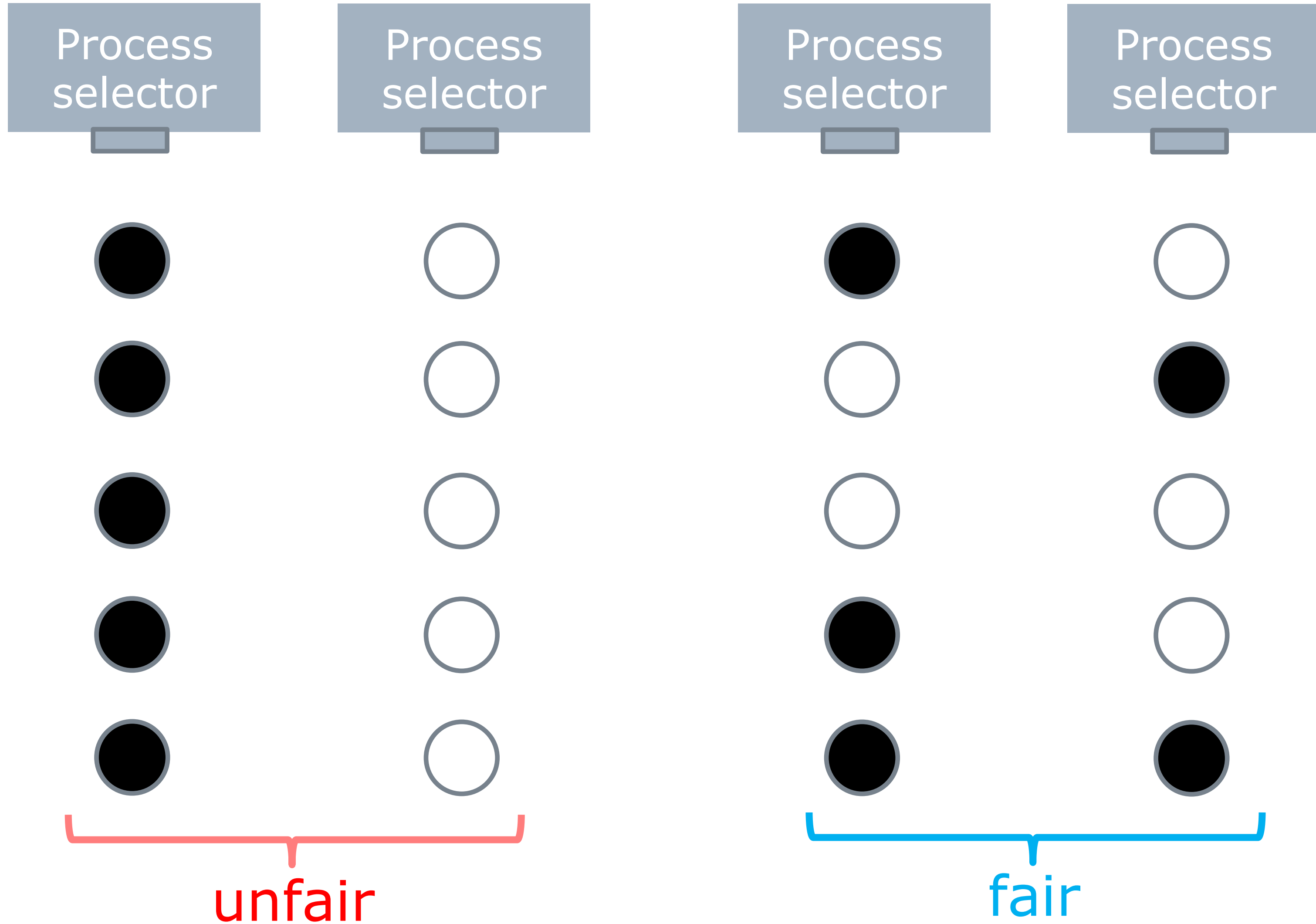
ランポートのアルゴリズムの記法
PlusCal algorithm language

Lamport, L. The PlusCal algorithm language..

https://lamport.azurewebsites.net/pubs/pluscal.pdf

Leslie Lamport. The pluscal algorithm language. http://research.microsoft.com/users/lamport/tla/pluscal.html.

unfairなプロセス、fairなプロセス



await *condition*

await 文は、より大きいアトミックなアクションの中で利用される。
await P を含むステップは、Pが真の時のみ、実行される。

```
await Len(buf) < N;
    buf := Append(buf, Head(in));
    in := Tail(in):
```

```
await Len(buf) > 0;
    buf := Append(out, Head(buf));
    buf := Tail(buf)
```

Finite Sequence

有限シーケンス Finite sequenceは、タプル tuple の別名である。

$< -3, \text{“}xyz\text{”}, \{0, 2\} >$ は長さ 3のシーケンスである。

長さ Nのシーケンスは、1 .. N 1からNまでの整数を定義域とする関数である。

$< -3, \text{“}xyz\text{”}, \{0, 2\} >[1] = -3$
$< -3, \text{“}xyz\text{”}, \{0, 2\} >[2] = \text{“}xyz\text{”}$
$< -3, \text{“}xyz\text{”}, \{0, 2\} >[3] = \{0, 2\}$

Sequences Moduleの演算子

- Tail(<s1, . . . , sn>) ≜ <s2, . . . , sn>.
- Head(seq) ≜ seq[1]
- ∘ (concatenation)
  if seq ≠ <> then seq = $Head(seq) \circ Tail(seq)$
  Any non-empty sequence is the concatenation of the one-element sequence containing only its head, with its tail

  < 3, 2, 1> ∘ < “a”, “b”> = <3, 2, 1, “a”, “b” >
- Append(seq, e) ≜ seq ∘ <e>

Sequences Moduleの演算子 Len, Seq

- Len(seq) は、シーケンスの長さに等しい。
  関数としてのseqの定義域は、1 .. N である。
  1 .. 0 = {} 空集合である。
  これは、空なシークエンス <>の定義行きに等しい。

- Seq(s)は、集合Sの要素からなる全てのシークエンスの集合である。
  Seq({3})={ϕ, <3>, <3,3>, <3, 3, 3>, … }

Sequence操作の例 1
Producer – in から buf へ

$\boldsymbol{in} = < v_1, v_2, v_3, v_4 >$
$\boldsymbol{buf} = <>$

Producer

buf := Append(buf, Head(in))
= Append(<>,Head($< v_1, v_2, v_3, v_4 >$)
= Append(<>.v_1)
$= < v_1 >;$
in := Tail(in)
= Tail($< v_1, v_2, v_3, v_4 >$)
$= < v_2, v_3, v_4 >$

$\boldsymbol{in} = < v_2, v_3, v_4 >$
$\boldsymbol{buf} = < v_1 >$

Sequence操作の例 2
Consumer – buf から out へ

$\boldsymbol{buf} = < v_2, v_3, v_4 >$
$\boldsymbol{out} = < v_1 >$

out := Append(out, Head(buf))
$= \text{Append}(< v_1 >, \text{Head}(< v_2, v_3, v_4 >))$
$= \text{Append}(< v_1 >. v_2)$
$= < v_1, v_2 >$

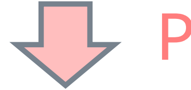
buf := Tail(buf)
$= \text{Tail}(< v_2, v_3, v_4 >)$
$= < v_3, v_4 >$

$\boldsymbol{buf} = < v_3, v_4 >$
$\boldsymbol{out} = < v_1, v_2 >$

システムの振る舞い – 状態の変化の例 1

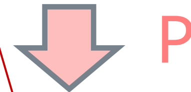
状態 1

$in = < v_1, v_2, v_3, v_4, v_5, v_6, \dots v_i, v_{i+1}, v_{i+2}, \dots >$
$buf = <>$
$out = <>$



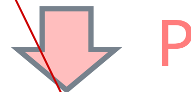
状態 2

$in = < v_2, v_3, v_4, v_5, v_6, \dots v_i, v_{i+1}, v_{i+2}, \dots >$
$buf = < v_1 >$
$out = <>$



状態 3

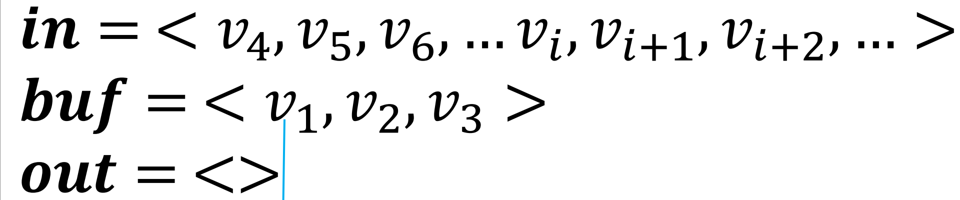
$in = < v_3, v_4, v_5, v_6, \dots v_i, v_{i+1}, v_{i+2}, \dots >$
$buf = < v_1, v_2 >$
$out = <>$



状態 4

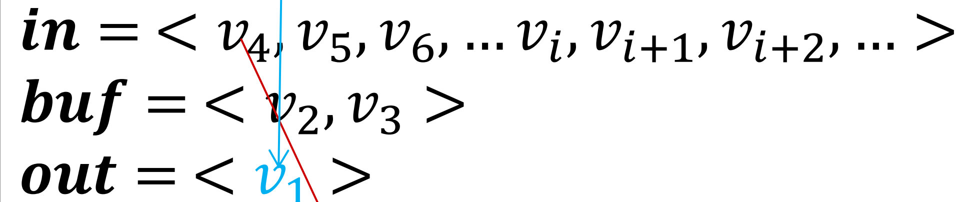
$in = < v_4, v_5, v_6, \dots v_i, v_{i+1}, v_{i+2}, \dots >$
$buf = < v_1, v_2, v_3 >$
$out = <>$

状態 4



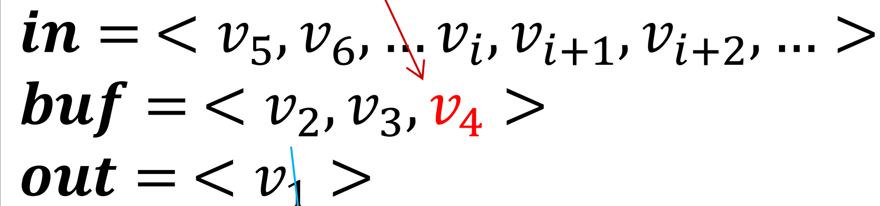
C

状態 5



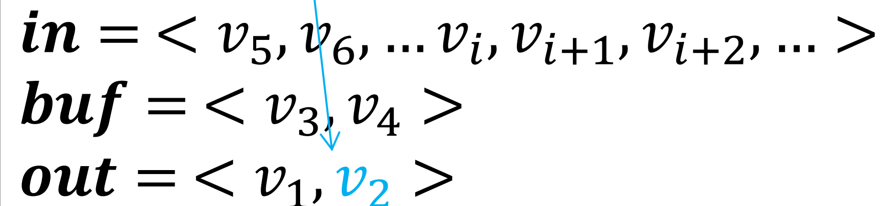
P

状態 6

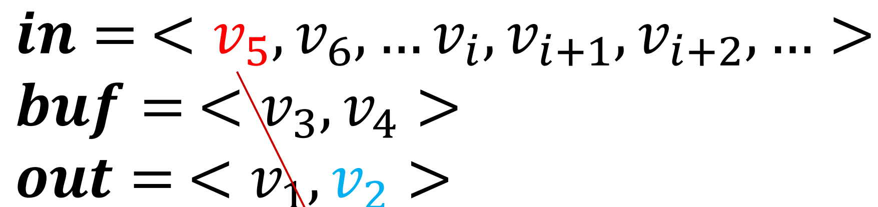


C

状態 7



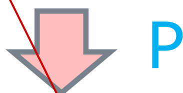
状態 7



$in = < v_5, v_6, \ldots v_i, v_{i+1}, v_{i+2}, \ldots >$

$buf = < v_3, v_4 >$

$out = < v_1, v_2 >$



状態 8

$in = < v_6, \ldots v_i, v_{i+1}, v_{i+2}, \ldots >$

$buf = < v_3, v_4, v_5 >$

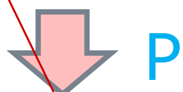
$out = < v_1, v_2 >$

状態 9

$in = < v_6, \ldots v_i, v_{i+1}, v_{i+2}, \ldots >$

$buf = < v_4, v_5 >$

$out = < v_1, v_2, v_3 >$



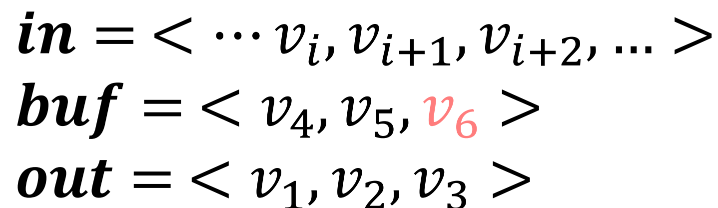
状態 10

$in = < \cdots v_i, v_{i+1}, v_{i+2}, \ldots >$

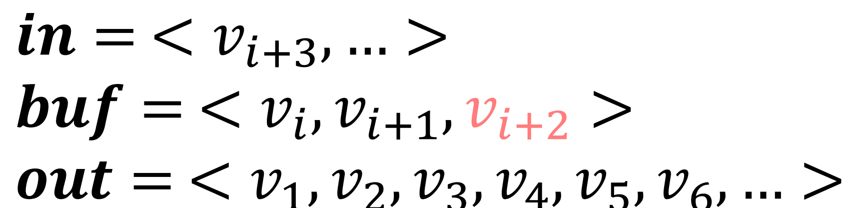
$buf = < v_4, v_5, v_6 >$

$out = < v_1, v_2, v_3 >$

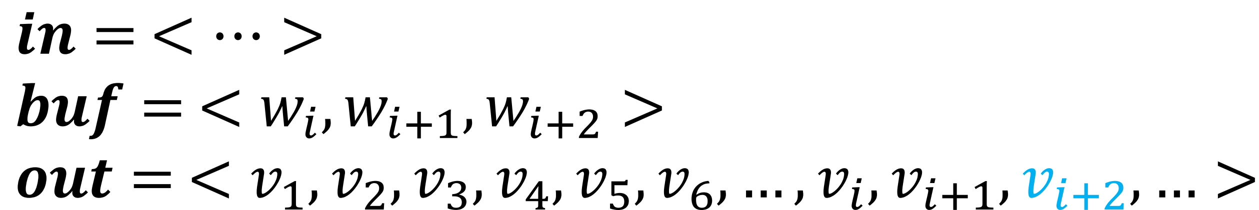
状態 10



状態 X



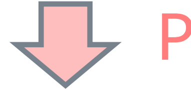
状態 Y



システムの振る舞い – 状態の変化の例 2

状態 1

$in = < v_1, v_2, v_3, v_4, v_5, v_6, \ldots v_i, v_{i+1}, v_{i+2}, \ldots >$
$buf = <>$
$out = <>$

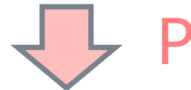


状態 2

$in = < v_2, v_3, v_4, v_5, v_6, \ldots v_i, v_{i+1}, v_{i+2}, \ldots >$
$buf = < v_1 >$
$out = <>$

状態 3

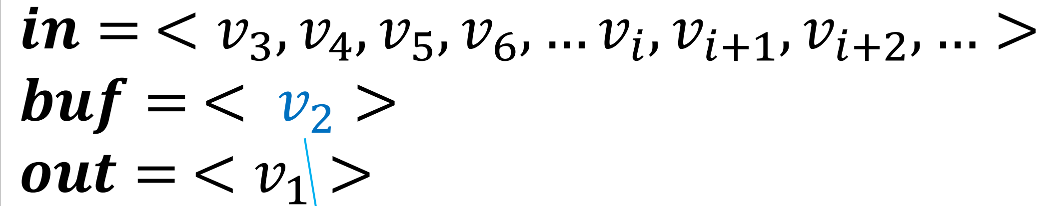
$in = < v_2, v_3, v_4, v_5, v_6, \ldots v_i, v_{i+1}, v_{i+2}, \ldots >$
$buf = < >$
$out = < v_1 >$



状態 4

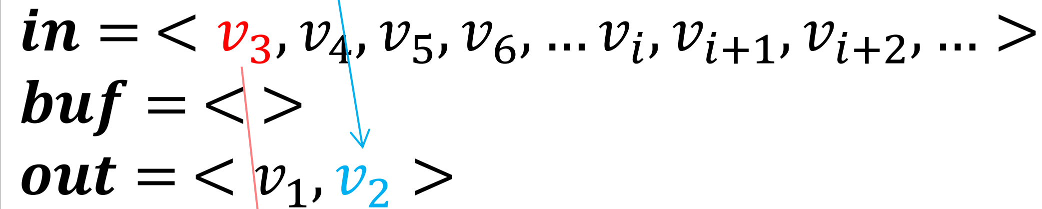
$in = < v_3, v_4, v_5, v_6, \ldots v_i, v_{i+1}, v_{i+2}, \ldots >$
$buf = < v_2 >$
$out = < v_1 >$

状態 4



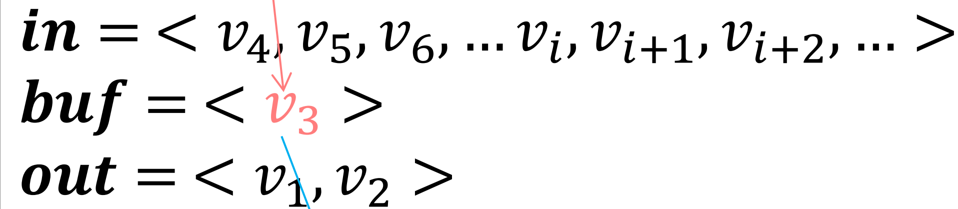
C

状態 5



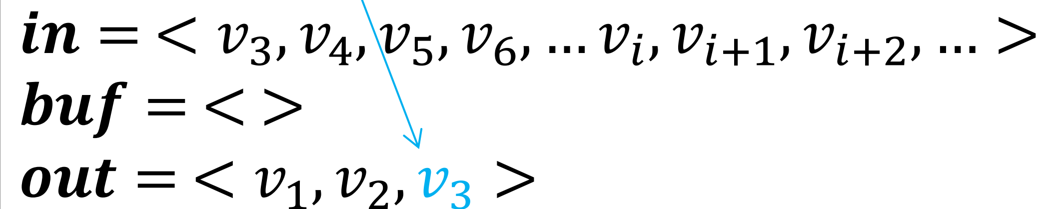
P

状態 6



C

状態 7



状態とシステムの振る舞い

「状態」とは、変数に対する値の割り当てである。

「振る舞い」とは、状態の系列である。

Producer-Consumerアルゴリズムの
振る舞いを図解する

ランポートのアルゴリズム記述言語を用いた Producer-consumer synchronization アルゴリズム

--algorithm PC {

variables $in = Input$, $out = \langle \rangle$, $buf = \langle \rangle$;

fair process ($Producer = 0$) {

P: while (TRUE) {

await $Len(buf) < N$;

$buf := Append(buf, Head(in))$;

$in := Tail(in)$ }}

fair process ($Consumer = 1$) {

C: while (TRUE) {

await $Len(buf) > 0$;

$out := Append(out, Head(buf))$;

$buf := Tail(buf)$ }} }

ランポートのアルゴリズム記述言語を用いた Producer-consumer synchronization アルゴリズム

--algorithm PC {

variables $in = Input,\ out = \langle\rangle,\ buf = \langle\rangle$;

fair process ($Producer = 0$) {

P: while (TRUE) {

await $Len(buf) < N$;

$buf := Append(buf, Head(in))$;

$in := Tail(in)$ }}

fair process ($Consumer = 1$) {

C: while (TRUE) {

await $Len(buf) > 0$;

$out := Append(out, Head(buf))$;

$buf := Tail(buf)$ }} }

Producer と Consumer

Producerプロセス

```
P:while(TRUE) {
    await Len(buf) < N;
    buf := Append(buf, Head(in));
    in := Tail(in): }
```

Consumerプロセス

```
C:while(TRUE) {
    await Len(buf) > 0;
    out := Append(out, Head(buf));
    buf := Tail(buf): }
```

待機か実行か？

Producerプロセス

```
await Len(buf) < N;
buf := Append(buf, Head(in));
in := Tail(in);
```

← $\mathrm{Len(buf)} \geq N$ の時、待機

← $\mathrm{Len(buf)} < N$ の時、実行

Consumerプロセス

```
await Len(buf) > 0;
out := Append(out, Head(buf));
buf := Tail(buf);
```

← $\mathrm{Len(buf)} \leq 0$ の時、待機

← $\mathrm{Len(buf)} > 0$ の時、実行

待機か実行か？

Producerプロセス

P_N

```
await Len(buf) < N;
```

← Len(buf) ≥ N の時、待機

P_i

```
buf := Append(buf, Head(in));
in := Tail(in):
```

← Len(buf) < N の時、実行

Consumerプロセス

C_0

```
await Len(buf) > 0;
```

← Len(buf) ≤ 0 の時、待機

C_j

```
out := Append(out, Head(buf));
buf := Tail(buf):
```

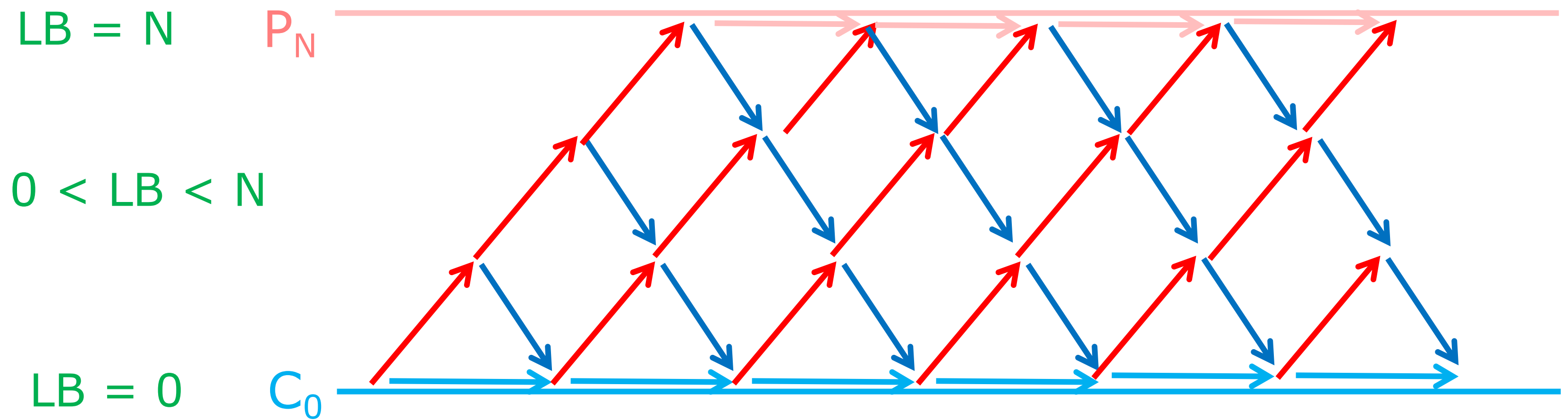
← Len(buf) > 0 の時、実行

バッファーの容量で整理する

LB = Len(buf) とする

	バッファー容量		
Producer	$LB \geq N$	待機　P_N	バッファー full
	$LB < N$	実行　P_i	LB++
Consumer	$LB > 0$	実行　C_j	LB−−
	$LB \leq 0$	待機　C_0	バッファー 空

	バッファー容量		
Producer	$LB \geq N$	待機　P_N	バッファー full
	$LB < N$	実行　P_i	LB++
Consumer	$LB > 0$	実行　C_j	LB−−
	$LB \leq 0$	待機　C_0	バッファー 空



Part IV

アルゴリズムの正しさへのアプローチ

並列・分散アルゴリズムの基礎

Part IV アルゴリズムの正しさへのアプローチ

- アルゴリズムとプログラム
- 状態モデルとアルゴリズムの「不変量」
- イベントの順序関係とランポートの「論理的時計」
- 状態とアクションと時制論理式

アルゴリズムとプログラム

アルゴリズムとプログラム

プログラムとアルゴリズムは似ている。

我々は、歴史的には、アルゴリズムの記述に、疑似的にプログラム言語を利用してきた。

だから、同じアルゴリズムの説明に、異なるプログラム言語が利用されていても -- 例えば、ある場合にはALGOLが、ある場合にはCが利用されていても --大きな違和感を感じることはなかったかもしれない。

なぜ、複数のプログラム言語が、同一のアルゴリズムの記述に利用できるのか？

それは、自然言語では普通のことだ。同じ意味を、大抵の場合、日本語でも英語でもフランス語でも、異なる言語で表現できる。

プログラムの「意味」としてのアルゴリズム

プログラムとアルゴリズムは異なるものだ。

プログラムは、具体的なコンピュータによって具体的に実行されるものだが、アルゴリズムは、その「具体的実行」の背後に存在する抽象的な「意味」に関わるものだ。

そうしたプログラムの「意味」の記述としてのアルゴリズムの記述は、プログラム言語とは異なる形式が必要である。

ある意味当然のことだが、プログラムとアルゴリズムの違いの認識、さらには、アルゴリズムを記述する言語の必要性の認識が生まれたことは重要なことである。

仕様と実装

視点を変えよう。

アルゴリズムとプログラムとの関係は、実践的には、「仕様」とその実装としての「プログラム」の関係としても考えることができる。

アルゴリズムの記述が「仕様」として与えられ、その実装が「プログラム」として与えられる。

ランポートのアルゴリズム記述言語を用いた Producer-consumer synchronization アルゴリズム

```
--algorithm PC {
  variables in = Input,  out = ⟨⟩, buf = ⟨⟩;
  fair process (Producer = 0) {
   P: while (TRUE) {
        await Len(buf) < N;
        buf  := Append(buf, Head(in));
        in   := Tail(in)                }}
  fair process (Consumer = 1) {
   C: while (TRUE) {
        await Len(buf) > 0;
        out  := Append(out, Head(buf));
        buf  := Tail(buf)               }}  }
```

アルゴリズムと「不変性」

この Algorithm *PC* は「仕様」である。bounded FIFO queue のプログラムは、この仕様としてのアルゴリズムを実装しなければならない。

仕様は定義である。ある定義が正しいかと問うことは、形式的には意味がない。

しかしながら、このアルゴリズムのいくつかの性質を証明することによって、我々はこのアルゴリズムがまさにbounded FIFO queueの仕様であることに確信を持つことができる。

我々があるアルゴリズムについて証明する最も重要な性質は、「不変性」という性質である。

状態モデルとアルゴリズムの「不変量」

状態モデル

並列アルゴリズムを、状態の遷移として記述するアプローチを「状態モデル」とよぶ。

このアプローチとは別に、イベントの継起として並列アルゴリズムを記述するモデル「イベント・モデル」があるのだが、それについては後で述べる。

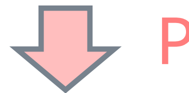
状態モデルでは、

- 状態は、変数への値の割り当てである。
- 状態の遷移の系列を「振る舞い Behavior」と呼ぶ。
- 状態の遷移を引き起こすものを「アクション」と呼ぶ。

状態モデルでのシステムの振る舞いの例
振る舞い＝状態の系列

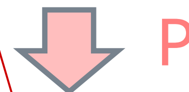
状態 1

$in = < v_1, v_2, v_3, v_4, v_5, v_6, \dots v_i, v_{i+1}, v_{i+2}, \dots >$
$buf = <>$
$out = <>$



状態 2

$in = < v_2, v_3, v_4, v_5, v_6, \dots v_i, v_{i+1}, v_{i+2}, \dots >$
$buf = < v_1 >$
$out = <>$



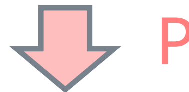
状態 3

$in = < v_3, v_4, v_5, v_6, \dots v_i, v_{i+1}, v_{i+2}, \dots >$
$buf = < v_1, v_2 >$
$out = <>$

状態モデルでのシステムの振る舞いの例
振る舞い＝状態の系列

状態 1

$$in = < v_1, v_2, v_3, v_4, v_5, v_6, \dots v_i, v_{i+1}, v_{i+2}, \dots >$$
$$buf = <>$$
$$out = <>$$



状態 2

$$in = < v_2, v_3, v_4, v_5, v_6, \dots v_i, v_{i+1}, v_{i+2}, \dots >$$
$$buf = < v_1 >$$
$$out = <>$$

状態 3

$$in = < v_2, v_3, v_4, v_5, v_6, \dots v_i, v_{i+1}, v_{i+2}, \dots >$$
$$buf = < >$$
$$out = < v_1 >$$

同一のアルゴリズムでも、振る舞いは多様である

同一のFIFOアルゴリズムでも、可能なシステムの振る舞いは、多様である。それは無数に存在する。

ここでは状態の遷移を引き起こすアクションを並べて、振る舞いを記述しよう。

システムの可能な振る舞いの例 1

P P P C P C P ...

システムの可能な振る舞いの例 2

P C P C P C ...

システムの可能な振る舞いの例 3

P P P C C C P ...

これらの無数の振る舞いを
共通に特徴づけるものが存在する！

このアルゴリズムに基づく振る舞いは無限個存在するのだが、この状態モデルで全てのアクションの実行の全てのステップで成り立つ、状態についての論理式が存在する。

こうした状態についての論理式を、アルゴリズムの「不変式」「不変量」と呼ぶ。

次に見るbounded FIFO queue のアルゴリズムの「不変量」の存在は、このアルゴリズムがN個の容量のバッファを持つbounded FIFO queue の正しい仕様であることを示唆している。

N個の容量のバッファを持つ bounded FIFO queueアルゴリズムの「不変量」

N個の容量のバッファを持つ bounded FIFO queueアルゴリズムの「不変量」は次のものである。

$$(Len(buf) \leq N) \wedge (Input = out \circ buf \circ in)$$

ここに、∘は、リストの結合演算子である。

最初の項 $(Len(buf) \leq N)$ が成り立つのは、バッファーの容量は、高々N個であるという仮定から明らか。

第二項 $(Input = out \circ buf \circ in)$ については、初期状態では、$Input = in$ で $out = buf = \langle\rangle$ で成り立っている。

ある状態 st でこの式が成り立っていると仮定して、その次の状態 st′ でもこの式が成り立っていることを示せば、帰納法でこの式が不変量であることを示すことができる。

$Input = out \circ buf \circ in$

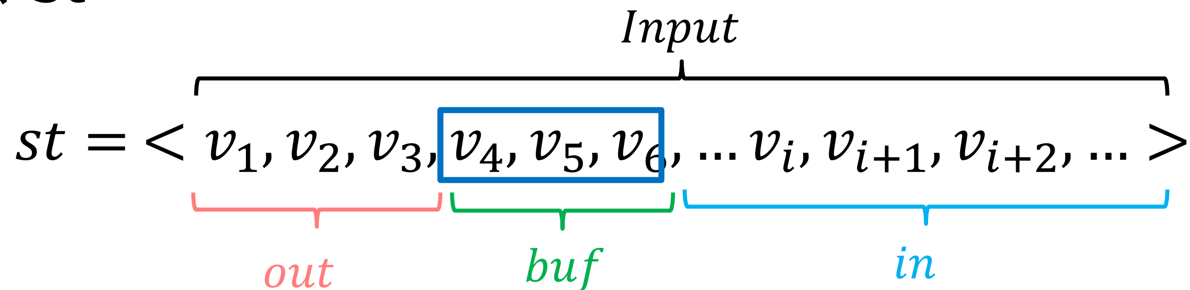
初期状態　$Input = in$ で $out = buf = <>$

$\boldsymbol{Input} = < v_1, v_2, v_3, v_4, v_5, v_6, \ldots v_i, v_{i+1}, v_{i+2}, \ldots >$

$\boldsymbol{in} = < v_1, v_2, v_3, v_4, v_5, v_6, \ldots v_i, v_{i+1}, v_{i+2}, \ldots >$

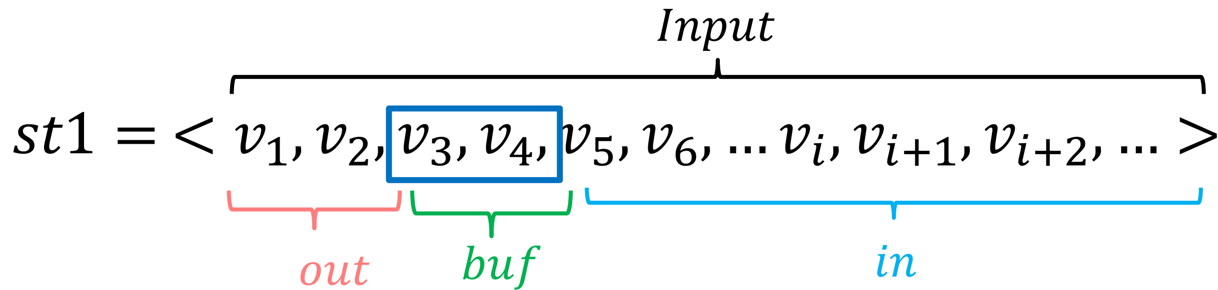
$Input = out \circ buf \circ in$

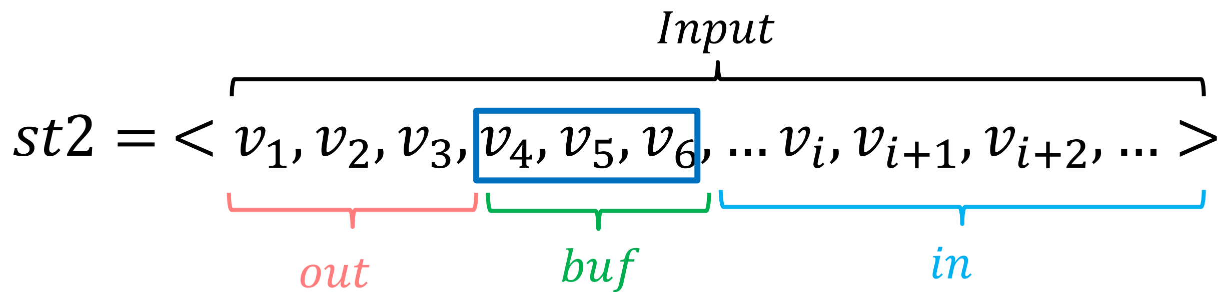
ある状態 st

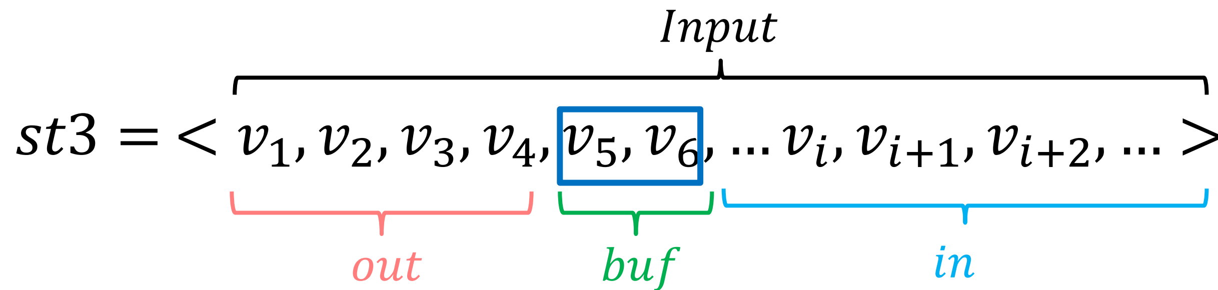


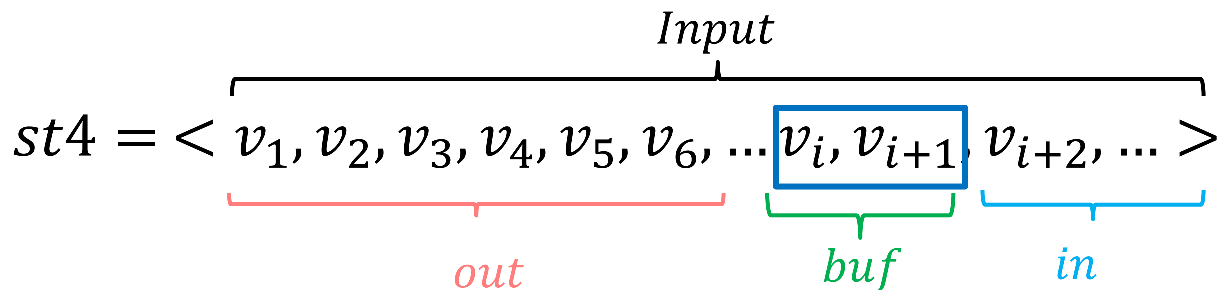
$Input = out \circ buf \circ in$

$$Input = out \circ buf \circ in$$









イベントの順序関係と
ランポートの「論理的時計」

分散システムにおける
時間と時計とイベントの順序付け

このセッションでは、ランポートの次の論文を紹介する。

Lamport, L. (1978). "Time, clocks, and the ordering of events in a distributed system". Communications of the ACM . 21 (7): 558–565. http://research.microsoft.com/users/lamport/pubs/time-clocks.pdf

論文の概要

分散システムで、あるイベントが他のイベントより先に起きるという概念を考察し、それらのイベントの半順序関係が定義される。

イベントたちの全順序付けに利用できる、論理的時計のシステムの同期の分散アルゴリズムが与えられる。

全順序付けを利用することは、同期問題の解決のための一つの方法として提示される。

その後、このアルゴリズムは、物理的時計の同期という特殊な問題に応用され、同時性の欠如が、時計たちをどこまで遠いものにするかについて、一つの限界が導かれる。

はじめに

時間の概念は、我々の考え方にとって基本的である。時間は、イベントたちが生起する順序という、より基本的な概念から導かれるものだ。

我々は、時計が 3時15分をさしていて、それが3時16分を指す前に起きた時、我々は何かが 3時15分に起きたという。

イベントの時間的順序の概念は、システムについての我々の思考に浸み込んでいる。たとえば、飛行機の予約システムでは、予約が一杯になる前まで予約のリクエストを受け入れるようにシステムの仕様を作る。

しかし、イベントの時間的順序というこの概念は、分散システムのイベントを考慮する際には、注意深く検討されねばならないことを見ていこうと思う。

分散システムは、空間的に分離された異なるプロセスの集まりから構成される。また、それはメッセージを交換することでお互いにコミュニケーションする。

ARPA net のような相互接続されたコンピュータのネットワークは分散システムである。単一のコンピュータも、その中で、中央のコントロールユニット、メモリーユニット、入出力チャンネルが分離したプロセスとして走る分散システムとして見ることができる。

あるシステムは、メッセージ送信の遅延が、単一プロセス間の遅延時間と比べて無視できないものであれば、分散システムである。

我々は、まず最初には、空間的に分離したコンピュータたちのシステムに関心を持つだろう。しかし、我々の見解の多くは、もっと一般的なものに適用可能である。

特に、単一プロセッサーのマルチ・プロセスのシステムは、これらの分散システムと同様の問題を抱えている。なぜなら、起こりうるイベントは予測できない順序で生起するからである。

分散システムでは、時には、二つのイベントのどちらかが先に起きるかをいうことが不可能となる。「先に起きる」という関係が、それゆえ唯一のシステムのイベントの半順序関係である。

我々は、人々がこの事実とその意味するところを、十分には理解していないがゆえに、こうした問題がしばしば生まれていることを見つけてきた。

この論文では、「先に起きる」という関係によって定義される半順序関係について考察し、それを全てのイベントの矛盾のない全順序関係に拡大する分散アルゴリズムを与える。

このアルゴリズムは、分散システムを実装する有用なメカニズムを提供することができる。我々は、このアルゴリズムの利用例として、同期問題を解決する単純な方法を提示する。

このアルゴリズムによって得られる順序がユーザーによって受け止められるものと異なっていれば、期待に反して、異常な振る舞いが起こりうる。こうした事態は、現実の物理的時計を導入することで回避することができる。

我々は、これらの時計たちを同期させる単純な方法について述べ、同時性の欠如がそれらをどこまで逸脱させるのかの上限を導出する。

時間の半順序関係

もしイベント aよりイベント bが早い時間に起きたとすれば、ほとんどの人は、あるイベント a はあるイベント b の前に起きたというだろう。彼らはこの定義を時間の物理的理論で正当化するかもしれない。

しかし、もしあるシステムがある仕様に正しく適合していれば、この仕様はそのシステムの内部でイベントとして観測可能なものの言葉で与えられなければならない。

もし、その仕様が物理的時間の言葉で語られていれば、そのシステムは実在する時計を含まなければならない。たとえそれがリアルな時計を含んでいたとしても、こうした時計は完全に正確ではなく、正確な物理時間を維持できないという問題は引き続き残る。

それゆえ、我々は、「先に起きる」という関係を、物理時計を遣うことなく定義しようと思う。

「イベント a が、イベント b より先に起きた」ことを a → b と表そう

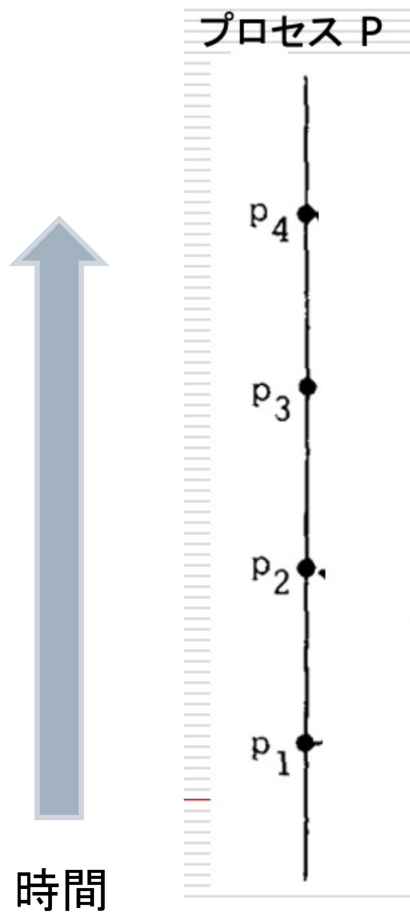
a ⟶ b となるのは、次の場合である

(1)もし、aとbが同じプロセスのイベントで、aがbより先に起きる場合、a ⟶ b である。

(2)もし、a が一つのプロセスからメッセージを送り、他のプロセスのbがそのメッセージを受け取るなら、a ⟶ b である。

(3)もし、 a ⟶ b で b ⟶ c なら a ⟶ c である。

同じプロセスのイベントの場合



$$p_1 \rightarrow p_2$$

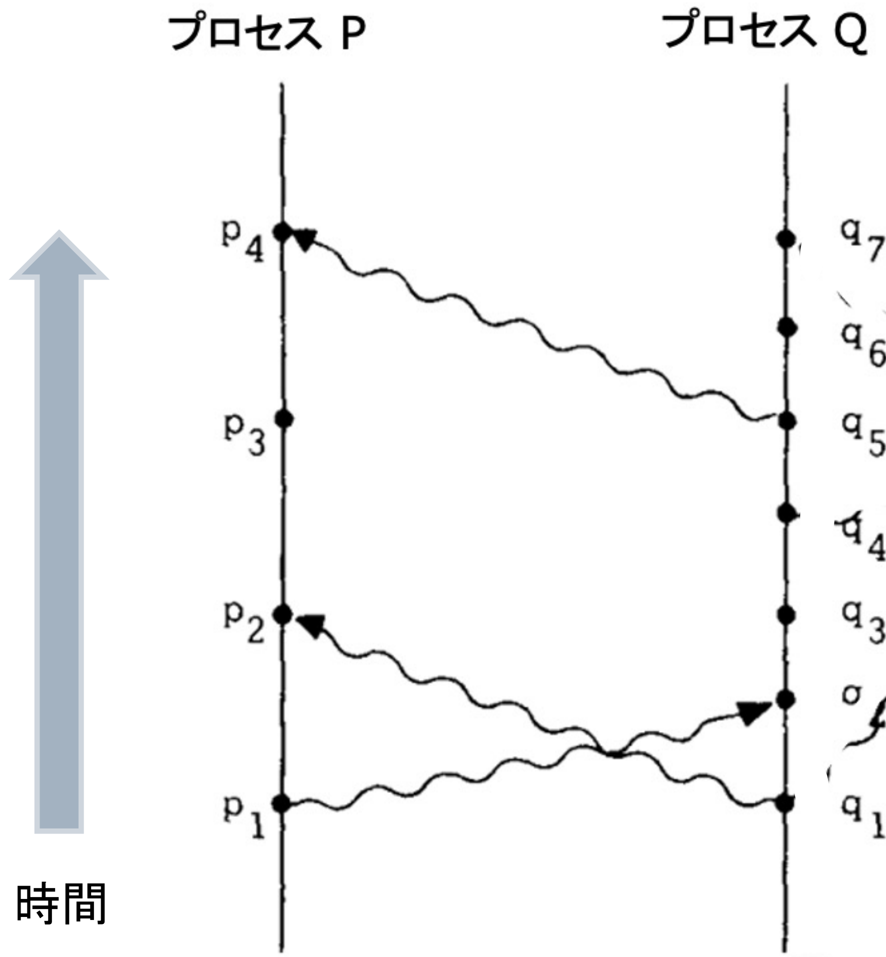
$$p_2 \rightarrow p_3$$

$$p_3 \rightarrow p_4$$

また、先の(3)から、

$$p_1 \rightarrow p_3$$
$$p_1 \rightarrow p_4$$
…

二つのプロセスのメッセージ交換



$p_1 \to q_2$

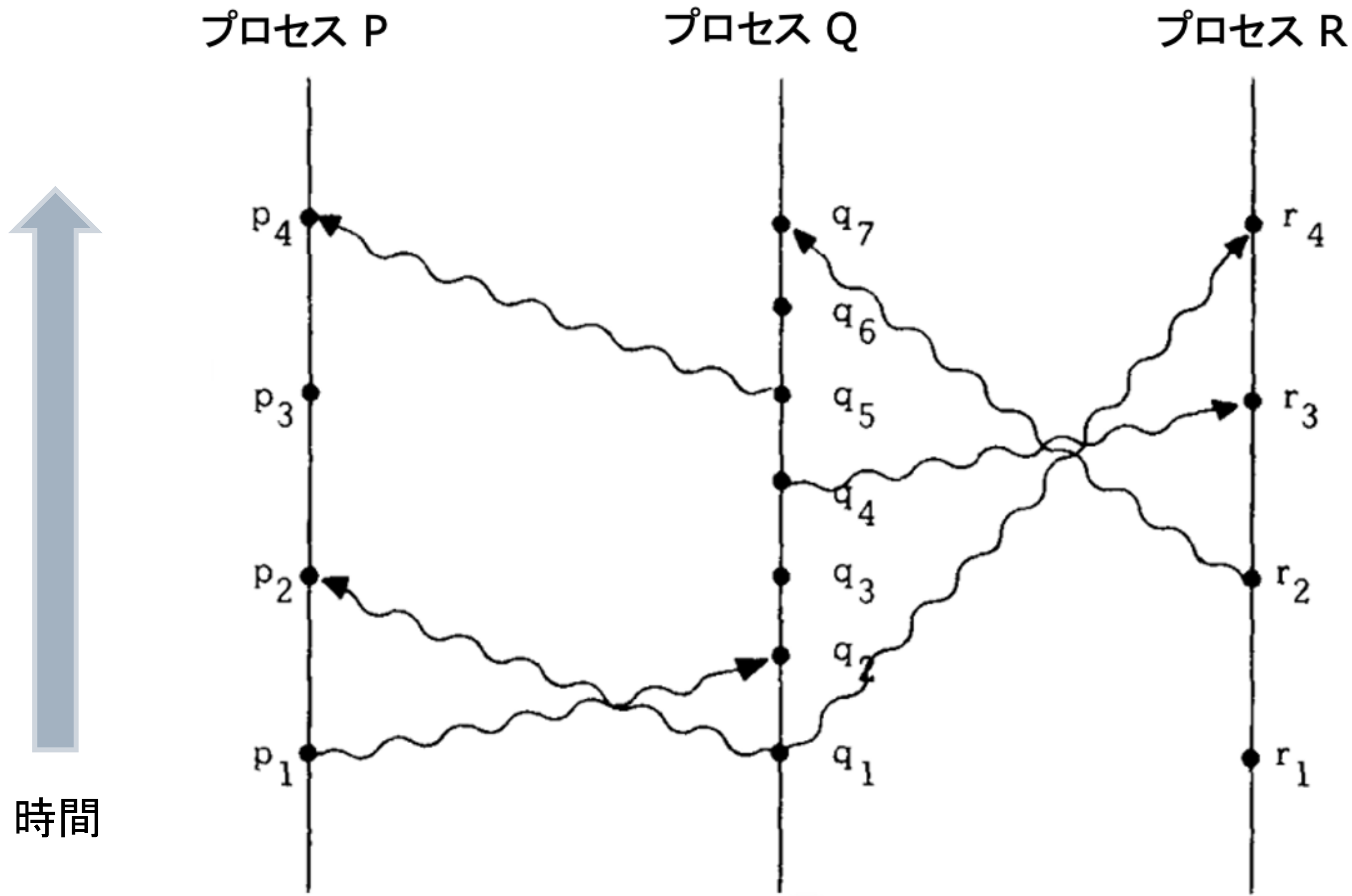
$q_1 \to p_2$

$q_5 \to p_4$

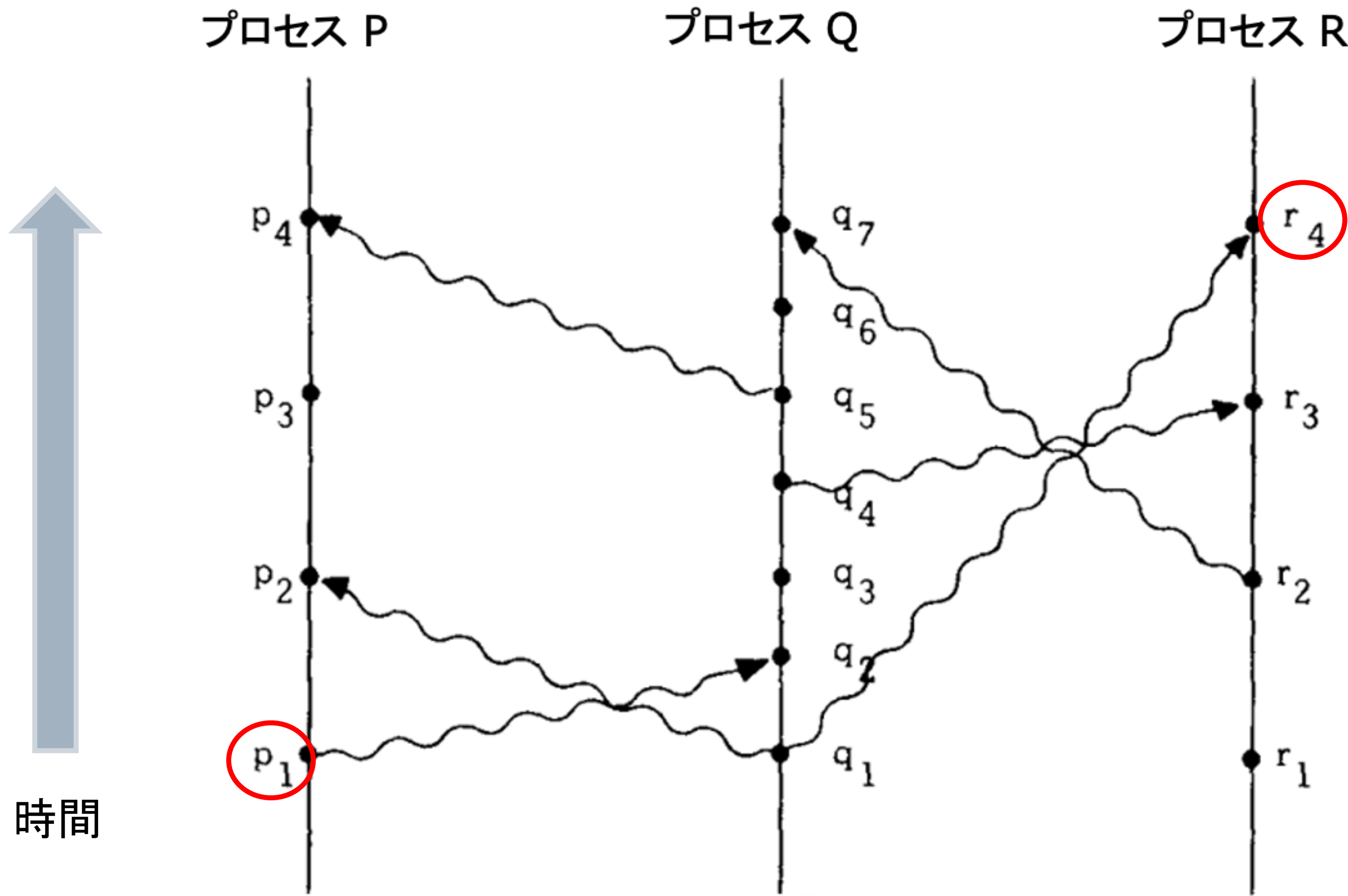
また、先の(3)から、

$q_1 \to p_3$
$p_1 \to q_7$
…

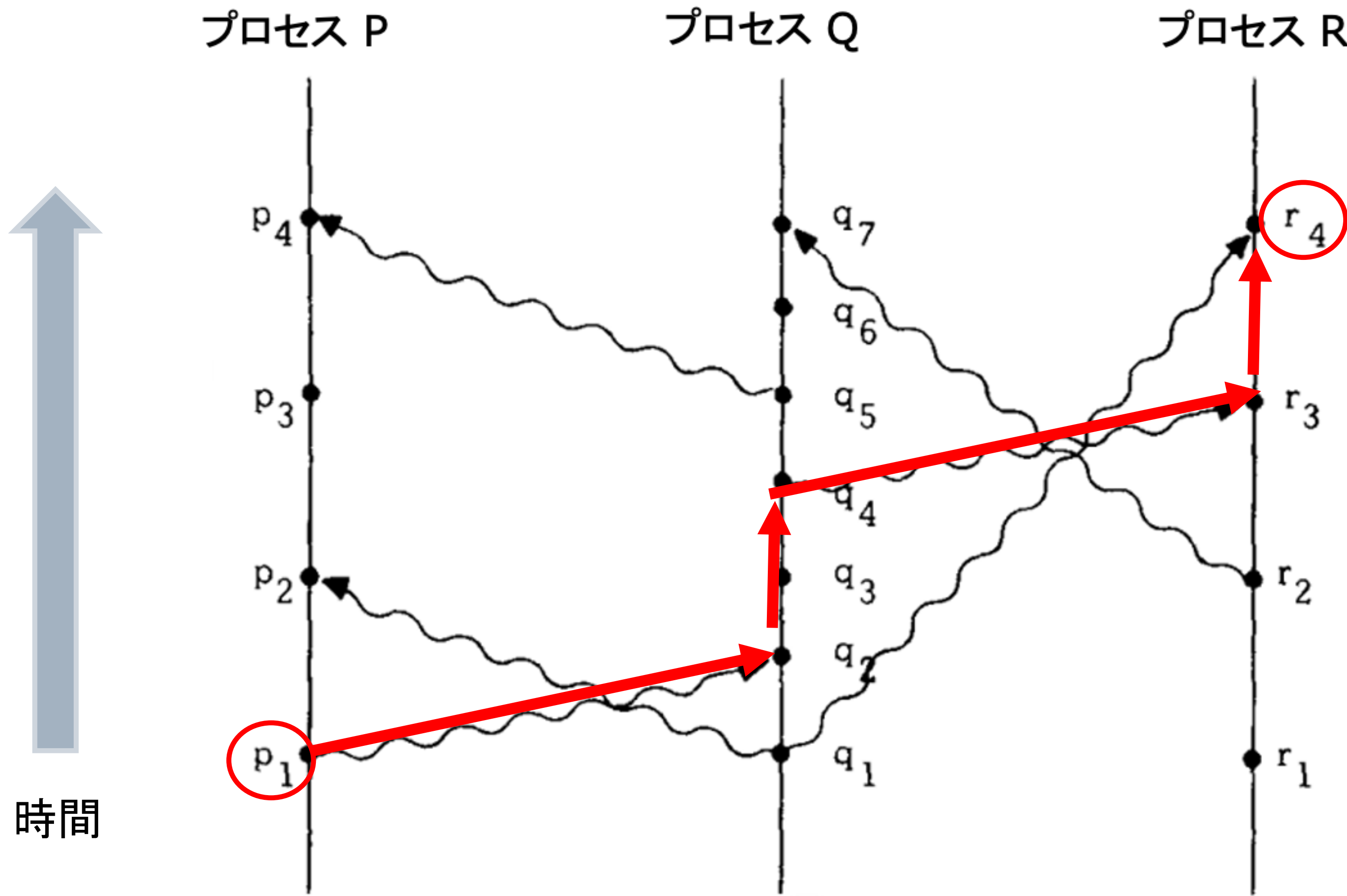
三つのプロセスのメッセージ交換



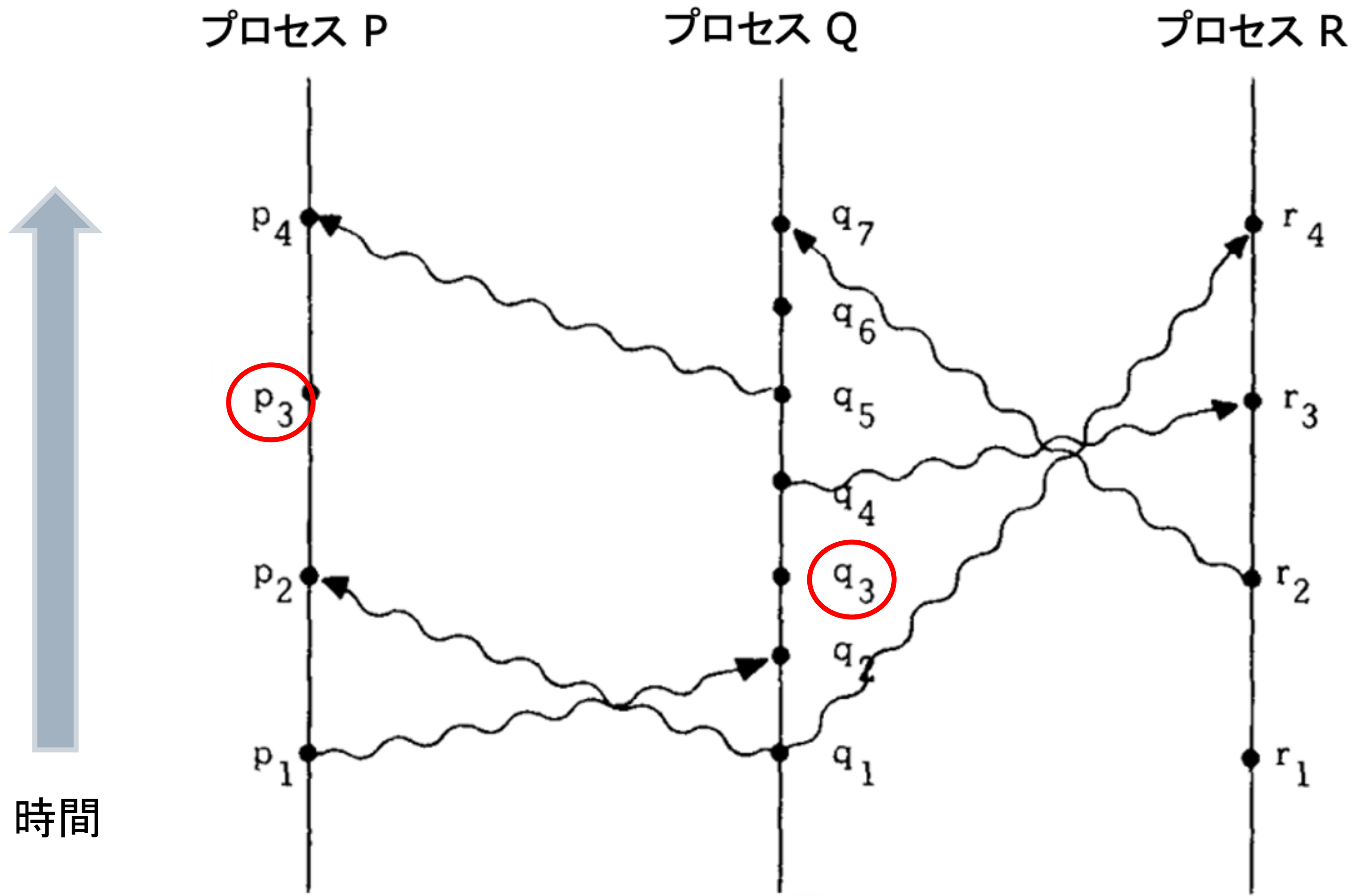
三つのプロセスのメッセージ交換
この時、$p_1 \rightarrow r_4$



三つのプロセスのメッセージ交換
この時、$p_1 \rightarrow r_4$

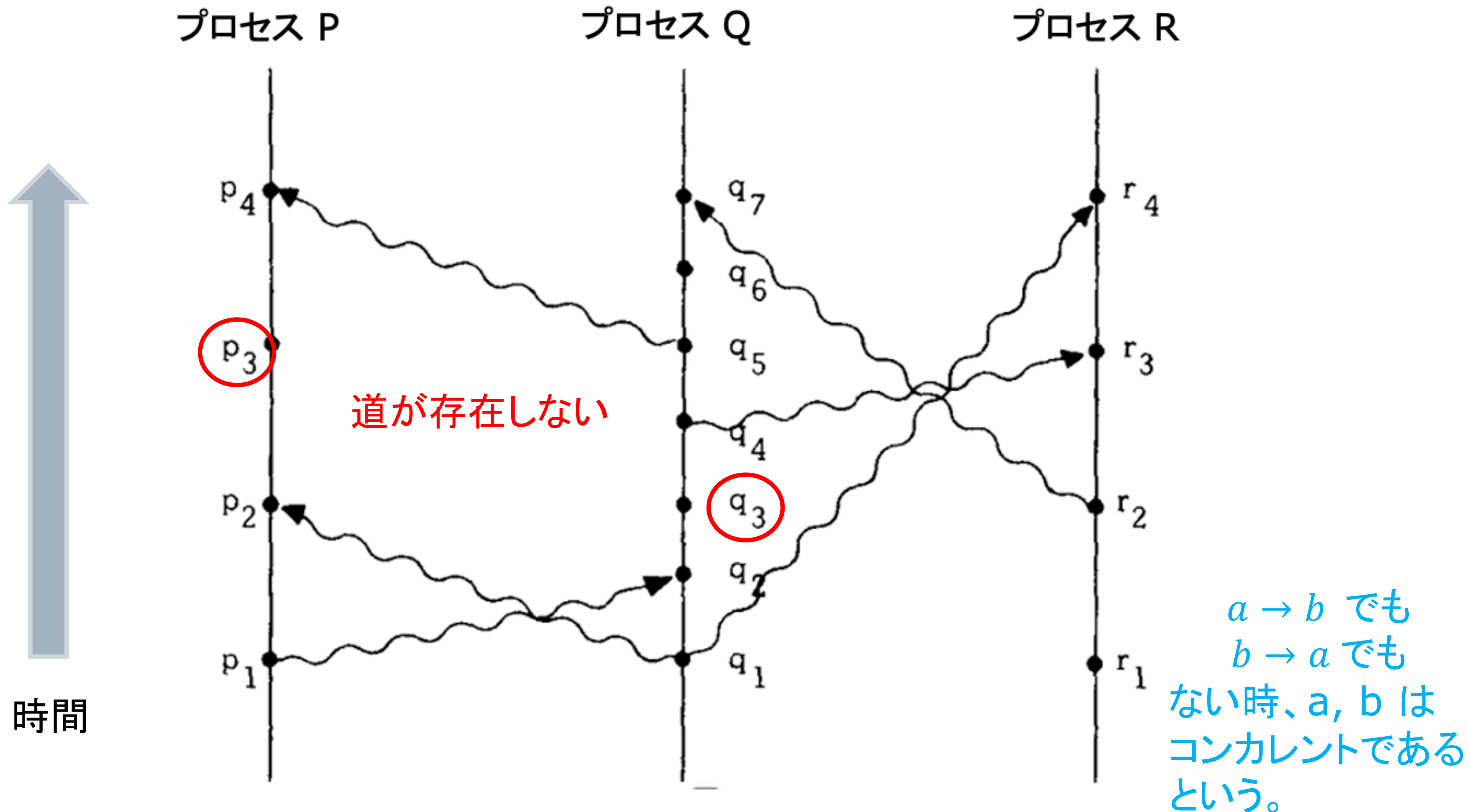


三つのプロセスのメッセージ交換
この時、$p_3 \rightarrow q_3$ でも $q_3 \rightarrow p_3$ でもない



時間

三つのプロセスのメッセージ交換
この時、$p_3 \to q_3$でも$q_3 \to p_3$でもない



$a \to b$ でも
$b \to a$ でも
ない時、a, b は
コンカレントである
という。

論理的時計は、イベントaに
論理的時間 C(a)を割り当てる

C1. もし aとbがプロセス P_i のイベントで、aが bより先に起きるならば、$C_i(a) < C_i(b)$

C2. もし、aがプロセス P_i からメッセージを送り、bがプロセス P_h でメッセージを受け取るなら、$C_i(a) < C_j(b)$

Clock Condition

任意のイベント a, b について、
$a \to b$ なら $C(a) < C(b)$

論理的時計の実装

IR1. それぞれのプロセスP_iは、二つの連続するイベントすべてで時計C_i をインクリメントする。

IR2.

(a) もし、イベント a が、プロセスP_i でメッセージ m をおくるのなら、メッセージ m は、$\mathrm{Tm} = C_i(a)$ という値のタイムスタンプを含む。

(b) メッセージ m を受け取ったプロセスP_j は、C_j の値を、現在の値と受取ったメッセージ中のTmより大きい値にセットする。

全順序化

1. To request the resource, process P_i sends the message T_m:P_i *requests resource* to every other process, and puts that message on its request queue, where T_m is the timestamp of the message.

2. When process P_j receives the message T_m:P_i *requests resource*, it places it on its request queue and sends a (timestamped) acknowledgment message to P_i.[5]

3. To release the resource, process P_i removes any T_m:P_i *requests resource* message from its request queue and sends a (timestamped) P_i *releases resource* message to every other process.

全順序化

4. When process P_j receives a P_i *releases resource* message, it removes any T_m:P_i *requests resource* message from its request queue.

5. Process P_i is granted the resource when the following two conditions are satisfied: (i) There is a T_m:P_i *requests resource* message in its request queue which is ordered before any other request in its queue by the relation $\Rightarrow$. (To define the relation "$\Rightarrow$" for messages, we identify a message with the event of sending it.) (ii) P_i has received a message from every other process time-stamped later than T_m.[6]

状態とアクションと時制論理式

アクションの時制論理

このセッションでは、ランポートのつぎの論文を紹介する。

Leslie Lamport. The temporal logic of actions. Technical Report 79, Digital Equipment Corporation, Systems Research Center, December 1991.

http://lamport.azurewebsites.net/pubs/old-tla-src.pdf

Abstract

Here, a temporal logic is introduced based only on modality "forever",

but with predicates (assertions about a single state) generalized to actions—assertions about pairs of states.

This logic has all the expressive power needed to describe and reason about concurrent programs.

Much of the temporal reasoning required with other logics is replaced by nontemporal reasoning about actions.

States and State Predicate

We assume a **set V of values**.

State variables are primitive terms in the logic. They represent quantities that can change during execution of a program.

A state function is an expression made from state variables. Its meaning is a function from S to V.

A state predicate is a boolean-valued state function.

A state predicate P is valid, written $\models P$,

iff P is true in all states.

Actions

An action is a boolean-valued expression containing primed and unprimed state variables.

Its meaning is a boolean-valued function on $S \times S$, where unprimed state variables are applied to the first component and primed variables to the second component.

For example, the meaning of the action $y' - x > 1$ is defined by
$[[y' - x > 1]](s, t) \equiv [[y]](t) - [[x]](s) > 1$.

Action as a state transition

We think of an action as specifying a set of allowed state transitions.

Action A allows

the transition $s \rightarrow t$ from state s to state t

iff [[A]](s, t) equals true.

$$
\begin{array}{l}
\textbf{var}\ \ \textbf{integer}\ \ x,\, y \qquad\quad = 0; \\
\qquad\ \ \textbf{semaphore}\ \ sem\ \ = 1; \\
\\
\textbf{cobegin}\ \ \textbf{loop}\ \ \alpha_1\!:\ \langle\, \mathbf{P}(sem)\,\rangle; \\
\qquad\qquad\qquad\quad\ \beta_1\!:\ \langle\, x := x + 1\,\rangle; \\
\qquad\qquad\qquad\quad\ \gamma_1\!:\ \langle\, \mathbf{V}(sem)\,\rangle \\
\qquad\qquad\ \ \textbf{end loop} \\
\quad \Box \\
\qquad\qquad\ \ \textbf{loop}\ \ \alpha_2\!:\ \langle\, \mathbf{P}(sem)\,\rangle; \\
\qquad\qquad\qquad\quad\ \beta_2\!:\ \langle\, y := y + 1\,\rangle; \\
\qquad\qquad\qquad\quad\ \gamma_2\!:\ \langle\, \mathbf{V}(sem)\,\rangle \\
\qquad\qquad\ \ \textbf{end loop} \\
\textbf{coend}
\end{array}
$$

samples of actions

$$
\begin{aligned}
\beta_2 \;\triangleq\;\; & pc_2 = \beta \;\wedge\; y' = y + 1 \;\wedge \\
& pc_2' = \gamma \;\wedge\; \textbf{unchanged}\ \{x,\ sem,\ pc_1\}
\end{aligned}
$$

$$
\begin{aligned}
\gamma_2 \;\triangleq\;\; & pc_2 = \gamma \;\wedge\; sem' = sem + 1 \;\wedge \\
& pc_2' = \alpha \;\wedge\; \textbf{unchanged}\ \{x,\ y,\ pc_1\}
\end{aligned}
$$

$$
\begin{aligned}
\alpha_1 \;\triangleq\;\; & pc_1 = \alpha \;\wedge\; sem > 0 \;\wedge \\
& pc_1' = \beta \;\wedge\; sem' = sem - 1 \;\wedge \\
& \textbf{unchanged}\ \{x,\ y,\ pc_2\}
\end{aligned}
$$

Enabled Predicate

$$\begin{aligned}\beta_2 \triangleq\ & pc_2 = \beta \ \wedge\ y' = y + 1 \ \wedge \\ & pc_2' = \gamma \ \wedge\ \textbf{unchanged}\ \{x,\ sem,\ pc_1\}\end{aligned}$$

$$\begin{aligned}\gamma_2 \triangleq\ & pc_2 = \gamma \ \wedge\ sem' = sem + 1 \ \wedge \\ & pc_2' = \alpha \ \wedge\ \textbf{unchanged}\ \{x,\ y,\ pc_1\}\end{aligned}$$

$$\begin{aligned}\alpha_1 \triangleq\ & pc_1 = \alpha \ \wedge\ sem > 0 \ \wedge \\ & pc_1' = \beta \ \wedge\ sem' = sem - 1 \ \wedge \\ & \textbf{unchanged}\ \{x,\ y,\ pc_2\}\end{aligned}$$

$$\begin{aligned}Enabled(\beta_2) &= pc_2 = \beta \\ Enabled(\gamma_2) &= pc_2 = \gamma \\ Enabled(\alpha_1) &= pc_1 = \alpha \wedge sem > 0\end{aligned}$$

Behavior

A *behavior* is an infinite sequence of states; the set of all behaviors is denoted by $\mathbf{S}^\omega$. If σ is the behavior $s_0, s_1, \ldots,$ then σ_i denotes the ith state s_i. The ith *step* of σ is the state transition $\sigma_{i-1} \rightarrow \sigma_i$. It is called a *stuttering step* iff states σ_{i-1} and σ_i are equal.

$$
\begin{aligned}
[\![\neg F]\!](\sigma) &\triangleq \neg[\![F]\!](\sigma) \\
[\![F \wedge G]\!](\sigma) &\triangleq [\![F]\!](\sigma) \wedge [\![G]\!](\sigma)
\end{aligned}
$$

A formula F is valid iff it is true for all behaviors:

$$
[\![\models F]\!] \triangleq \forall \sigma \in \mathbf{S}^\omega : [\![F]\!](\sigma)
$$

operator $\Box$ (usually read "always")

operator $\Diamond$ (read "eventually")

The temporal logic operator $\Box$ (usually read "always") is defined as follows. If σ is a behavior, let σ^{+i} denote the behavior $\sigma_i, \sigma_{i+1}, \ldots$ obtained by cutting off the first i states in the sequence σ. For any formula F and behavior σ,

$$[\![\Box F]\!](\sigma) \;\triangleq\; \forall i \geq 0 : [\![F]\!](\sigma^{+i}) \tag{5}$$

Intuitively, a temporal formula F holds at a certain time iff F holds for the infinite behavior starting at that time. The formula $\Box F$ asserts that F holds at all times—now and in the future.

The operator $\Diamond$ (read "eventually") is defined by $\Diamond F \triangleq \neg\Box\neg F$. Intuitively, $\Diamond F$ asserts that F holds now or at some time in the future.

operator $\leadsto$ (read "leads to"),

The operators $\Box$ and $\Diamond$ can be nested and combined with logical operators to provide more complicated temporal modalities. For example, $\Box\Diamond F$ asserts that at all times, F must be true then or at some future time. In other words, $\Box\Diamond F$ as operator $\leadsto$ (read "leads to"), often. Of particularly interest is the operator $\leadsto$ (read "leads to"), where $F \leadsto G \triangleq \Box(F \Rightarrow \Diamond G)$. Intuitively, $F \leadsto G$ asserts that whenever F is true, G is true then or at some later time.

Temporal Reasoning

Invariance Rule

$$\frac{\{P\}\mathcal{A}\{P\}}{\Box[\mathcal{A}] \;\Rightarrow\; (P \Rightarrow \Box P)}$$

The hypothesis asserts that any A transition with P true in the starting state has P true in the ending state. The conclusion asserts that if every nonstuttering step is an A transition, then P true initially implies that it remains true forever. Observe that the hypothesis is an action, while the conclusion is a temporal formula.

Expressing Programs as Temporal Formulas

A program Π is described by four things:

1. A collection of **state variables**.
2. A **state predicate** $Init_{\Pi}$ specifying the initial state.
3. An **action** N_{Π} specifying the state transitions allowed by the program.
4. A **temporal formula** L_{Π} specifying the program's progress condition.

The program itself is the temporal logic formula Π defined by

$$\Pi \triangleq Init_{\Pi} \wedge \Box[\mathcal{N}_{\Pi}] \wedge L_{\Pi}$$

Reasoning with Fairness

WF Rule

$$\frac{\begin{array}{l}\{P\}\ \mathcal{A}\ \{Q\} \\ \{P\}\ \mathcal{N} \wedge \neg\mathcal{A}\ \{P \vee Q\} \\ P \Rightarrow \mathit{Enabled}(\mathcal{A})\end{array}}{\Box[\mathcal{N}] \wedge \mathrm{WF}(\mathcal{A}) \ \Rightarrow\ (P \leadsto Q)}$$

SF Rule

$$\frac{\begin{array}{l}\{P\}\ \mathcal{A}\ \{Q\} \\ \{P\}\ \mathcal{N} \wedge \neg\mathcal{A}\ \{P \vee Q\} \\ \Box F \wedge \Box[\mathcal{N} \wedge \neg\mathcal{A}] \Rightarrow (P \wedge \neg\mathit{Enabled}(\mathcal{A}) \leadsto \mathit{Enabled}(\mathcal{A}))\end{array}}{\Box F \wedge \Box[\mathcal{N}] \wedge \mathrm{SF}(\mathcal{A}) \ \Rightarrow\ (P \leadsto Q)}$$

Reasoning About Programs

Safety Properties: type-correct

The first safety property one usually proves about a program is that it is *type-correct*, meaning that the values of all variables are of the expected "type". Type-correctness for the program of Figure 1 is expressed by the formula $\Pi \Rightarrow \Box T$, where

$$
\begin{aligned}
T \;\triangleq\;\; & pc_1 \in \{\alpha, \beta, \gamma\} \;\wedge\; x \in \mathbf{Int} \;\wedge\; sem \in \mathbf{Nat} \;\wedge \\
& pc_2 \in \{\alpha, \beta, \gamma\} \;\wedge\; y \in \mathbf{Int}
\end{aligned} \tag{9}
$$

Liveness Properties

we prove

$$\Pi \;\Rightarrow\; (x = n \leadsto x = n + 1)$$

Liveness Properties

A. Control in process 1 is either at α_1, β_1, or γ_1.

B. If control is at γ_1 with x equal to n, then eventually (after executing γ_1) control will be at α_1 with x equal to n.

C. If control is at α_1 with x equal to n, then eventually (after executing α_1) control will be at β_1 with x equal to n.

D. If control is at β_1, then eventually (after executing β_1) x will equal $n + 1$.

Liveness Properties

A. $\Pi \;\Rightarrow\; (x = n) \leadsto$
$((pc_1 = \gamma \wedge x = n) \vee (pc_1 = \alpha \wedge x = n) \vee (pc_1 = \beta \wedge x = n))$

B. $\Pi \;\Rightarrow\; (pc_1 = \gamma \wedge x = n) \leadsto (pc_1 = \alpha \wedge x = n)$

C. $\Pi \;\Rightarrow\; (pc_1 = \alpha \wedge x = n) \leadsto (pc_1 = \beta \wedge x = n)$

D. $\Pi \;\Rightarrow\; (pc_1 = \beta \wedge x = n) \leadsto (x = n + 1)$

General Logic

$$\begin{aligned}
Init_\Phi &\triangleq x = 0 \land y = 0 \\
\mathcal{N}^1_\Phi &\triangleq x' = x + 1 \land y' = y \\
\mathcal{N}^2_\Phi &\triangleq y' = y + 1 \land x' = x \\
\mathcal{N}_\Phi &\triangleq \mathcal{N}^1_\Phi \lor \mathcal{N}^2_\Phi \\
\Phi &\triangleq Init_\Phi \land \Box[\mathcal{N}_\Phi] \land \mathrm{WF}(\mathcal{N}^1_\Phi) \land \mathrm{WF}(\mathcal{N}^2_\Phi)
\end{aligned}$$

$$\begin{aligned}
\Pi &\triangleq Init_\Pi \land \Box[\mathcal{N}_\Pi]_{\mathbf{w}} \land \mathrm{SF}_{\mathbf{w}}(\mathcal{N}^1_\Pi) \land \mathrm{SF}_{\mathbf{w}}(\mathcal{N}^2_\Pi) \\
\Phi &\triangleq Init_\Phi \land \Box[\mathcal{N}_\Phi]_{\{x,y\}} \land \mathrm{WF}_{\{x,y\}}(\mathcal{N}^1_\Phi) \land \mathrm{WF}_{\{x,y\}}(\mathcal{N}^2_\Phi)
\end{aligned}$$

With these definitions, the formula $\Pi \Rightarrow \Phi$ is valid.