数理解析研究所講究録
470巻 1982年 1-13

同期基本命令の変遷と数理の必要性

慶應義塾大学情報科学研究所
土居範久

I 同期基本命令の変遷

代表的な同期基本命令を列記すると以下の通りである。ただし，(1)〜(4)は特別な命令の助けをかりずに際どい部分(critical section)を構築するためのアルゴリズムである。また，下線を引いた人名は，証明法を与えた人を示す。

(1) Dekker (1968)

(2) Dijkstra (1965)

(3) Knuth (1966) → de Bruijn (1967)

→ Eisenberg-McGuire (1972)

(4) Lamport (1974)

---

(5) fork-join [Conway] (1963)

(6) test and set [IBM] (1965)

(7) lock-unlock [Dennis and Van Horn] (1966)

(8) wakeup-block [Saltzer] (1966)

notify-wait [Spier and Organick] (1969)

(9) post-wait/enq-deq [IBM] (1967)

(10) P-V [Dijkstra] (1968)

--- <u>Habermann</u> (1972), <u>Martin</u> (1981)

(11) message [Brinch Hansen] (1970), [Feldman] (1979)

(12) P-V Multiple [Patil] (1971)

parallel P [Dijkstra] (1971)

P-V chunk [Vantilborgh and van Lamsweerde] (1972)

up-down [Wodon] (1972)

(13) conditional critical region [Hoare] (1971), [Brinch Hansen] (1972)

--- <u>Hoare</u> (1971), <u>Owicki and Gries</u> (1976)

(14) port [Balzer] (1971), [SUE] (1972), [Liskov] (1979)

(15) <u>region</u> r <u>do</u> S <u>await</u> B [Brinch Hansen] (1972)

(16) await-cause [Brinch Hansen] (1973)

(17) monitor [Dijkstra] (1972), [Brinch Hansen] (1973), [Hoare] (1974)

--- <u>Hoare</u> (1974), <u>Howard</u> (1976)

distributed procedure DP [Brinch Hansen] (1978)

(18) actor [Yonezawa] (1977)

(19) serializer [Atkinson and Hewitt] (1979)

--- Atkinson and Hewitt (1979)

(20) path expression [Campbell and Habermann] (1974)

--- Flon and Habermann (1976)

regular path expression [Habermann] (1979)

open path expression [Campbell] (1978)

predicate path expression [Andler] (1979)

--- Andler (1979)

extended simple path expression [土居 and 久良知] (1978)

(21) counter variable [Gerber] (1977)

(22) CSP [Hoare] (1978)

--- Hoare (1978)

Apt, Francez and de Roever (1980)

Levin and Gries (1981), Chen and Hoare (1981)

Hoare (1981)

(23) PDR [Hirose, Saito, Doi et al.] (1978)

(24) rendezvous [DoD] (1980)

(25) inverse P-V [Hirose, Saito, Doi et al.] (1981) [土居] (1981)

ここで，P－V命令以降の同期基本命令は，

- 個々の必要性に応じて
- 共通の問題に対処できるように汎用性を持たせるため
- 効率を重視して
- プログラムを構造化するため

といった理由で開発されてきたように思える。そして，正当性を確認するための方法は，通常，後手にまわっている。

正当性を確認すること，すなわち正当性の証明では，

- 共用データへのアクセスが相互に排除されていること
- バッファに関して生産者－消費者の関係が維持されていること

といったデータの完全性（integrity）の検証が主であり，

- 問題のデータを常に満す不変関係（invariant）
- 操作を施す前に成り立つ事前条件（precondition）

などが用いられる。さらに，並行型プロセスに特有な性質である

- デッドロックに陥ることがないこと
- 個別封鎖がないこと

などに対する証明法が与えられているものもある。

II　数理の必要性

たとえ形式的な証明法が与えられていたとしても，とらえどころが悪いと正しくないものも正しいかの如きことになり得ることもあるが，形式的な証明法がない場合には，一般に何も制約がないので，その余地がさらに広がるのは周知の通りである。特に，並行型のプログラムの場合には，制御の流れが複雑になるので，一層，数理的に要をおさえる必要性が増す。たとえば，以下に示すような間違いをしないですませることができる数理的な「枷（かせ）」はないものだろうか。

Habermann が n（≥2）個のプロセス間で相互排除するためのアルゴリズムを提案し，非形式的にその正当性を確認している。[2]

```
begin local x, y, j
  x ← p                    $ 優先度指示子 p, 配列 IN, プロセスインデクス i
                           $ およびプロセスの数 n は大域的な対象である
  repeat
    IN[i] ← 1
   ┌─────────────────────────────────────────────┐
   │ y ← i − 1 + (if x > i then n else 0 fi)      │
   │ if all j in [x : y] sat IN[j % n] = 0 then IN[i] ← 2 fi │
   └─────────────────────────────────────────────┘
  until (IN[p] = 0 or p = i) and (all j in [i + 1 : n + i − 1] sat IN[j % n] ≠ 2)
  p ← i
  〈際どい部分の文〉
  if some j in [i + 1 : n + i − 1] sat IN[j % n] ≠ 0 then p ← i fi
                                                   (手書き訂正: i → j % n)
  IN[i] ← 0
end
```

検証のために興味ある状態は，プロセス P_i がループ内の **if** 文を成功裡に終わったときにプロセス P_{i-1} がその **if** 文を開始した場合である。 P_{i-1} もまた **if** 文で **true** になったとすると，この二つのプロセスのうちのどちらかが先に次の文でその *IN* を 2 に設定する。この動作は他のプロセスがそのループを終えないようにするのに十分である。これら二つのプロセスは，必然的に自分達の *IN* を 2 に設定する。これら二つのプロセスが互いに際どい部分に入ることをいつまでも妨げることがないことを検証しなければならない。両方が共に他方のプロセスが状態 2 に達したことを検出すると，共にループを再び実行する。二度目のとき，P_i に対しては条件が **false** になり，P_{i-1} に対しては **true** になる。最初のときにプロセス P_i がすばやく振る舞うと，P_{i-1} を打ち負かしすべり込むことができるかもしれないが，いつもそうなるとは限らない。知らせるには遅すぎたので，優先度指示子がプロセスを一度は飛び越すことになるが，そのプロセスは次の回に取り上げてもらえる。

ここで，％は剰余を求める演算子である。

ところが，このアルゴリズムでは次のような問題が生じる。

(1) $p=i$ のとき： $y=i-1$ となり $[i : i-1]$ という範囲指定になる。

(2) 相互実行が起り得る： $p=0$，$IN[p]=0$，$k<q$ とする。

プロセス k	プロセス q
$IN[k] \leftarrow 1$	$IN[q] \leftarrow 1$
	<u>all</u> j <u>in</u> $[0 : q-1]$ <u>sat</u> $IN[j \% n] \neq 0$
	$\{$ $IN[0]=0$
	<u>all</u> j <u>in</u> $[q+1 : n+q-1]$ <u>sat</u> $IN[j \% n] \neq 2$

all j in [0 : k−1]
    sat IN[j % n] = 0

IN[k] ← 2

{ IN[0] = 0
  all j in [k+1 : n+k−1]
      sat IN[j % n] ≠ 2

p ← q
「際どい部分」に入る！

p ← k
「際どい部分」に入る！

これらの事態が生じないように，破線枠内を次のように書きかえてみた。このようにしたところで，前掲の非形式的な証明は，そのまま通用する！

```
y ← i−1+( if x>i then n
            else if x=i then 1 else 0 fi fi )
L: if all j in [x : y] sat IN[j % n]=0 then IN[i]←2
   else goto L fi
```

その結果，

(1) $p=i$ のとき $[i:i-1]$ となるようなことは生じない。

(2) $IN[i]=1$ で際どい部分に入ってしまうようなことはなくなった

のだが，何とデッドロックが生じてしまうことになったのである。

(i) p が i を指しているものとする。i が際どい部分に入りたい状況を考える。このとき，

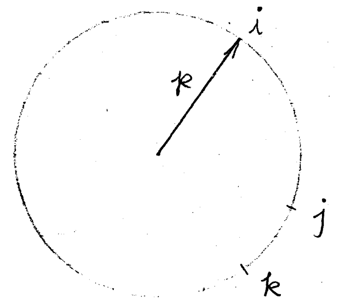
```
L: if all j in [i:i] sat IN[j%n]=0 then
       IN[i] ← 2
   else goto L fi
```

となり，$IN[i]=1$ であることから無限ループに入る。ところで，この状況では p の値は決して変わることはないので，他のプロセスでは

$$(IN[p]=0 \ \underline{or} \ p=i)=false$$

になり，デッドロックに陥る。

(ii) プロセス i が際どい部分を実行中に，プロセス j，k が際どい部分に入りたくなったとする。



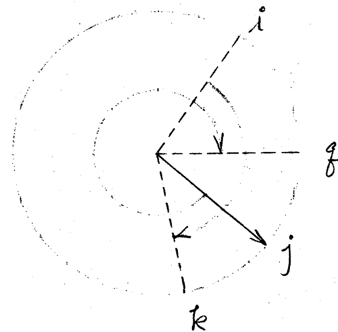
すると，

j は $[i:j-1]$

kは $[i : k-1]$

の範囲内の IN[ ] が 0 になるのを文Lで待つ.

プロセス i が際どい部分の実行を終了すると，p が j を指すように設定される。この結果，j は際どい部分に入れる。



このとき，プロセス q が際どい部分の実行を要求したとする。

すると，

qは $[i : q-1]$

の範囲内の IN[ ] が 0 になるのを Lで待つ。ところが，$[i : k-1]$ には q が居り，$[j : q-1]$ には k がいる。その結果，相互封鎖が起る。

これは，ループの中で x の値を更新していないことに起因する。

以上のことを考慮すると次のように書き直せばよい。

```
repeat
  IN[i] ← 1
L: x ← p
  if x ≠ i then
    y ← i − 1 + (if x > i then n else 0 fi)
    if some j in [x : y] sat IN[j % n] ≠ 0 then goto L fi fi
  IN[i] ← 2
until (IN[p] = 0 or p = i) and (all j in [i + 1 : n + i − 1] sat IN[j % n] ≠ 2)
p ← i
〈際どい部分の文〉
if some j in [i + 1 : n + i − 1] sat IN[j % n] ≠ 0 then p ← j % n fi
IN[i] ← 0
end
```

依然として，前掲の証明は通用する！　ところで，このアルゴリズムは Eisenburg－McGuire のアルゴリズムに他ならない。goto 文を排除すると次のようになる。

```
begin local x, y ; j
  repeat
    IN[i] ← 1
    repeat
      x ← p
      y ← i − 1 + (if x > i then n
                     else if x = i then 1 else 0 fi fi)
    until (all j in [x : y] sat IN[j % n] = 0) or (x = y)
    IN[i] ← 2
  until (IN[p] = 0 or p = i) and
          (all j in [i+1 : n+i−1] sat IN[j % n] ≠ 2)
  p ← i
  〈際どい部分〉
  if some j in [i+1 : n+i−1] sat IN[j % n] ≠ 0 then
     p ← j % n fi
  IN[i] ← 0
end
```

ところで，アルゴリズムを記述する場合には，そこで用いる名前，特に要となるものの役割は明記する必要がある。たとえば，Lamportのアルゴリズムは下記の通りであるが，Lamportはこのアルゴリズムの正当性を証明するのに，配列名choosingを用いている。[3]

```
var choosing, number : array [1..N] of integer (0)
    j : integer
L1: choosing[i] ← 1
    number[i] ← 1 + max (number[1], ..., number[n])
    choosing[i] ← 0
    for j := 1 to N do
      L2: if choosing[j] ≠ 0 then goto L2 fi
      L3: if number[j] ≠ 0 and
                (number[j], j) < (number[i], i) then
             goto L3 fi
    od
    <際どい部分>
    number[i] ← 0
    program_i
    goto L1
```

しかし，アルゴリズムを一見したところでは，choosingは証明のためにだけ用いる擬変数のように映る。しかし，この配列は極めて重要な役割を演じているのである。文L1，choosing ← 0 および文L2 を削除すると，次のような事態が生じる。

(1) プロセス i，j ($i<j$) を考える。

(2) プロセス i，j がほぼ同時に L1 に入る。

(3) max(…) を評価した結果，共に，← の右辺の値が同じになったとする。この値を p とする。

(4) number[j] へ p が代入される。

(5) プロセス j が L3 で検査をする：

number[i] へは，まだ p が代入されていないので，際どい部分に入れる。

(6) number[i] へ p が代入される。

(7) プロセス i が L3 で検査をする：

j については，($p \neq 0$ <u>and</u> $(p, j) < (p, i)$) = false

(8) プロセス i も際どい部分に入ることができる！

また，number[i] への p の代入がさらに遅れると，到着順処理という規則も破られる。

参考文献

[1] Andler, S., "Synchronization Primitives and the Verification of Concurrent Programs", Department of Computer Science, Carnegie-Mellon University (1977).

[2] Habermann, A. N., "Introduction to Operating System Design", Science Research Associates Inc. (1976)
[土居範久訳 "オペレーティングシステムの基礎", 培風館 (1978)].

[3] Lamport, L., "A New Solution of Dijkstra's Concurrent Programming Problem", CACM, Vol. 17, No. 8, 453-455 (1974).