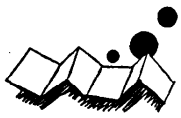
解　説



高級言語による並列処理の記述†

古谷　立美††

1. まえがき

計算機の処理には並列プロセスが相互作用を行いつつ仕事を進める応用が少なくない．例えば OS，リアルタイムシステム，シミュレーション，マルチプロセッサ等はその代表である．このような分野のプログラミングはいまだにアセンブリ言語や中級インプリメンテーション言語で行われる場合が多く，最も高級言語化の遅れている分野である．しかし，近年ソフトウェア工学が進歩し，抽象化や構造化といった概念を利用して並列プログラムを構造的に記述できる高級言語が提案されるに至った．並列プログラムの高級言語化によって期待できることの第1は，記述が簡明になり，プログラムの生産性，読み易さ，保守のし易さ等が向上すること，第2として並列プログラムでよく起こる時間に依存した再現性のない誤りをプログラム開発時やコンパイル時に発見できる可能性が増すことである．

本稿では先ず並列プロセス記述の従来のアプローチについて簡単に触れ，次に現在提案またはインプリメントされている並列プログラム記述のための代表的言語概念，およびそれらの高級言語への導入を Concurrent Pascal と Ada を例にとり示す．

2. プロセス間の相互作用

並列プロセス同志で行う相互作用には次のようなものがある．

（1）　通信：プロセス間のデータ転送．

（2）　同期：プロセス間の実行順序を決定するために行う相互作用でさらに2つに分けることができる．

（i）　あるプロセスのある部分は，ほかのプロセスのある部分が終了した後でないと実行してはいけないというふうに2つのプロセス間の状態変化の時間的順序を決めるために行う相互作用（決定性同期）．（ii）複数のプロセスが入出力装置のような共有資源や共有変数に同時にアクセスすると混乱が起こる．そこで，プロセス間の状態変化に（i）のような順序関係は無いが共有資源へのアクセスは一時に1プロセスと限る，いわゆる相互排除を実現する相互作用（非決定性同期）．

従来の並列プログラムのアプローチは，これらの相互作用を基本命令（プリミティブ）で記述しようというもので，各場合に応じて種々のプリミティブが提案されている．（1）の通信には，送信プロセスからの send 命令と受信プロセスの receive 命令の間で行われる通信が代表的である．決定性同期用プリミティブには，自分のプロセスを止める Block 命令と Block で止っているほかのプロセスを再開させる Wakeup 命令等が知られている．相互排除を実現するプリミティブとしては LOCK，UNLOCK が最も良く知られている．LOCK と UNLOCK はプロセスの相互排除を必要とする部分の始めと終り（**図-1**（b）の P(s) 部に LOCK，V(s) 部に UNLOCK）に置かれ，あるプロセスが LOCK 命令を発して相互排除部に入っているときは，ほかのプロセスは LOCK 命令から中に入れず，UNLOCK 命令が出されるまで待つ．また Dijkstra は，同期（i），（ii）の両方を解決するセマフォア変数 s とそれに対する P・V 操作を提案した[1]．ここで P・V 操作を使った同期の実現を示す．P と Vは次のような操作である（**図-1**（a））．

P(s)：s から1を引き $s<0$ なら，この命令を発したプロセスは待ち（WAIT）になり s に付随したキューに入る．$s\geqq 0$ なら次の命令に進む．

V(s)：s に1を加え $s\leqq 0$ なら s に付随したキューに入っているプロセスをキューから出し再実行させる（SIGNAL）．

図-1（a）は P・V 操作で決定性同期を実現しているもので，（b）は非決定性同期（相互排除）を行うための P・V 操作の配置である．

これらの同期プリミティブがプロセス間の同期を実

† Concurrent Programming by High Level Languages by Tatsumi FURUYA (Electrotechnical Laboratory, Computer division).

†† 電子技術総合研究所電子計算機部

現することはわかっていただけると思うが，これらのプリミティブは共有資源（たとえばセマフォア変数）を使って実現されるため，複数のプロセスで同時に実行されると矛盾が起こる可能性が有することに気付かれたと思う．関連のあるプリミティブの実行は相互排除されねばならない．ここで並列プログラムを実際の計算機で走らせる方法を考えてみる．まず1台の計算機で実現する場合をみると，何らかの形で複数プロセスが並列に動いているようにシミュレートする必要がある．考えられる方法としては，決定性同期のプリミティブ等で，あるプロセスが実行できなくなったとき，実行可能プロセスの処理に移るといういわゆるコルーチン的処理で進める方式と，各プロセスを TSS 方式で処理し，並列プロセスをシミュレートする方法がある．いずれにしても1台のプロセッサの場合には，並列処理用プリミティブが複数のプロセッサで同時に実行されることは無いので，プリミティブの処理が割込みやタイムスライスで中断されないように制御（特権モードなど）すればよい．一方マルチプロセッサでは，関連のあるプリミティブが複数のプロセッサで実行されるのを制御する機構が必要である．IBM 370 の Test & Set 命令は，マルチプロセッサ下で LOCK 命令を実現しているし，特権モードのプロセッサを用意して関連のある同期プリミティブが同時に実行されるのを防ぐ方法も考えられる．

3. 高レベルの言語概念と高級言語

この章では，同期プリミティブが発展しモニタとメッセージパッシングという2つの代表的な並列処理記述用言語概念に整理された背景と，それらの概念およびそれらを用いた高級言語の並列処理記述例を示す．

3.1 プリミティブからプログラミングの言語概念へ[2),3)]

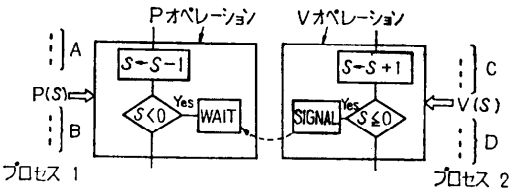
前章に示したプリミティブは，プロセス間の相互作用を記述する最も低レベルの機能で，プログラミング言語に対応させればアセンブリ言語に相当しよう．並列プロセスの記述では各プロセスがほかのプロセスの詳細な動きについての知識が無くても済むことが望まれる．前記プリミティブもプログラムを処理時間の影響から開放するものであったが，高級言語にとってはより高レベルの言語概念の導入が望まれる．プログラミング言語の概念として必要なことを整理すると次のようである．

1．頻繁に使われる一般性のある考え方を表現すること．

2．概念の意味と規則は正確に定義されていること．

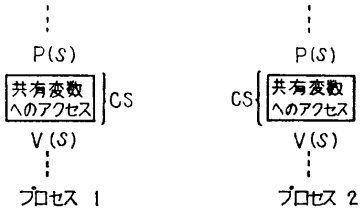
3．簡潔な記法で表現されていること．

4．安全性が高く，効率良い実行が可能．など．



Sの初期値が0なら B部は C部終了後でないと処理されない

（a）決定性同期



Sの初期値が1なら，CS部へは1時に1プロセスしかない．

（b）非決定性同期（相互排除）

図-1 P・V オペレーションによる同期の実現

これらによってプログラムは簡潔で理解し易くなり，誤りはできるだけコンパイル時に検出できるようになる．本節では言語概念の導出をメッセージバッファプログラム（MBP）の記述を例にとってふり返る．MBP とは，生産者と消費者という2つのプロセスがバッファを介してメッセージの伝達を行う並列プログラムの一典型である．生産者プロセスはメッセージを生産してはバッファへ送る操作をくり返し，消費者プロセスはメッセージをバッファから取出しては消費するという操作をくり返す．生産者プロセスはバッファへメッセージを送るとき，バッファが“満”状態なら空くのを待ち，消費者プロセスはバッファからメッセージを取るとき，バッファが“空”状態ならバッファへのメッセージ到着を待つというものである．

並列プロセス記述に高レベルの概念を導入しようとするはしりは Dijkstra による **parbegin, parend**[1)] (**cobegin, coend** が使われることも多い) とクリティカルセクション (CS) である．**parbegin, parend** は対であり，その間にはさまれている1つ下のレベルのブロックが並列実行されることを示す．図-2 でいえば，生産者プロセスと消費者プロセスが並列に走る．複数プロセスが同一資源に同時アクセスすると混乱が

```
var sv: shared record
            buffer: Buffertype;   } 共有変数
            count: 0..N           }
        end
parbegin
 生産者プロセス
  loop produce (m);                    ←メッセージの生産
  region sv when count≠N do            } 条件付 CS.
    begin                              } メッセージをバッファへ入れる
      deposit (m);                     }
      count:=count+1                   }
    end
  end
 消費者プロセス
  loop
  region sv when count≠0 do            } 条件付 CS.
    begin                              } メッセージをバッファから取出す
      fetch (m);                       }
      count:=count-1                   }
    end;
  consume (m)                          ←メッセージの消費
  end
parend
```

図-2 条件付 CS によるメッセージバッファのプログラム

生じる．CS とは各プロセスの中で共有資源へアクセスする部分を明確に示し，それらが同時に動作しないように制御された部分で，先に示した LOCK, UNLOCK や P・V 操作による相互排除はその実現例である．1972 年 Hoare は CS の始めに条件を付け，その条件が真のときのみ CS を実行し，偽のときは待たせるという“条件付 CS”を提案した[4]．図-2 はメッセージバッファのプログラムを条件付 CS で記述したものである．このプログラムではまず SV という変数は buffer と count という 2 つの変数からできている (PASCAL の record) ことを宣言している．buffer とは N 個分のメッセージを収容できるバッファであり FIFO に制御され，deposit 命令でメッセージを格納し，fetch 命令でメッセージを取出すものとしている．count はメッセージがバッファにいくつ入っているかを示す変数でバッファの状態(“空”または“満”)を知るのに使われる．**parbegin** と **parend** の間には 2 つのプロセスがあり，それぞれ並列に無限ループを繰返す．生産者プロセスについてみると，produce (m) でメッセージを作り，条件付 CS と記した **region** 以下でメッセージをバッファに送る．**region** SV **when** C **do begin** S **end** という構文は，SV が共有変数であり，S がクリティカルセクションで，C は S を実行するか否かを決める条件であることを示す．故に生産者プロセスではバッファに空が有る (count≠N) とき buffer にメッセージを入れ，count に 1 を加えている．一方バッファが“満” (count=N) なら条件付 CS は実行されずに空ができるのを待つ．消費者プロセスは逆で，count≠0 のとき buffer からメッセージを取出す．SV を S に付記する記法は，SV が S でのみアクセスされることをコンパイラがチェックするのに役立つ[5]．図-2 でわかるとおり，条件付 CS は並列プロセス記述の上では大変うまくいく．しかしこれを 1 台のプロセッサで実現しようとすると条件テスト **when** を繰返す必要があり，効率が悪い．そこでプロセスを待たせるキューを導入し，条件を満たさぬプロセスをキューに入れ，満たされた時点で走らせれば効率良く実現できることは想像がつく．1972～73 年 Hoare と Brinch Hansen はプロセスキューを導入し，さらに共有資源とそれに対する操作は 1 つのプログラムモジュールにまとめた方が良いという Dijkstra の薦め[6]を入れ，次節に示すモニタの概念を生み出した．

3.2 モニタと Concurrent Pascal

3.2.1 モニタ[7),8)]

モニタは複数のプロセスからアクセスされる共有資源とそれに対する操作からなるプログラムモジュールである．共有変数へのアクセスはモニタ内の操作（モニタプロシージャ）に限られ，共有変数へアクセスしたいプロセスはモニタ名と操作名でモニタプロシージャをコールする．**図-3** はモニタの機構を説明するための概念図であり，メッセージバッファをモデルにしている．モニタ内には共有変数にアクセスするモニタプロシージャがいくつかあるが，それらは一時に最大 1 つしかアクセスされないように制御されている．すなわち，モニタに入口を 1 つ設け，2 つ以上のプロセスの制御がモニタに入るのを防いでいる．もし 2 つ以上アクセス要求が来たときは入口のキューに入れる．モニタ内にはプロセスを入れるプロセスキューを定義でき，条件を表わす共有変数と組合わせて先に示した条件付 CS を効率良く実現する．モニタ内に入ったプロセスは条件を調べ条件が満たされなければ delay 命令を出すことができる．delay を出したプロセスは，delay 命令で指定するプロセスキューに入る．プロセスがプロセスキューに入るとモニタを出たことになり，モニタ入口で待っているプロセスはモニタ内に入れることになる．またモニタ内でプロセスは continue 命令を発してプロセスキューに入っているプロセスをモニタ内に復帰させることができる．continue 命令ではプロセスキューに入っているプロセスが無ければ何の操作もしない．continue 命令でほかのプロセスを起動したプロセスは，一時ブロックされ，起動されたプロセスがモニタをぬけてから continue を発した

プロセスが再開する．

なお Hoare は delay のかわりに wait, continue のかわりに signal という命令を使用している．

モニタは並列プログラムを記述する言語概念として広く認められ，Concurrent Pascal, Modula[9),10)], CSP/K[11)] 等の高級言語に取入れられている．

3.2.2 Concurrent Pascal(CP)[8),12),13)]

CP は OS を高級言語で記述するために Per Brinch Hansen が 1974 年ごろ開発した言語である．この言語は OS のように相互作用のある並列プロセスからなるプログラムを小さなコンポーネントの階層構造で作り，各コンポーネント間はほぼ独立に検証ができて，その検証もできるだけコンパイラで行えるように工夫されている．Hansen は CP で SOLO という OS を書き，OS が高級言語で簡単に書けることを実際に示した．CP は Pascal を基本にし，モニタによる並列プログラム記述機構を加えてできている．Pascal を基本にしている点は，**3.3.2** 節の Ada と共通であり，ブロック構造と豊富なデータタイプを持ちシンプルでエレガントなプログラムが可能だからである．一方 Pascal で不足な点は，(1) 並列プロセスの概念とそれらの間の相互作用を記述する能力，(2) Pascal ではブロックにローカルに宣言された資源(例えば変数)は制御がブロックをぬけると消えるが，OS 等の応用では例えばファイルのように，制御がブロックを出てしまっても保存しておかねばならぬ資源がある等である．CP ではこれらの不足を Simula[14)]のクラスの概念を導入して補い，かつ構造的なプログラムができるように整理されている．Simula のクラスとは簡単に言うとデータとそれに対する操作を 1 つのブロックにしたもので，データへのアクセスはクラス内の操作に限られる．クラスとプロシージャの違いはプロシージャではローカルに宣言された資源はプロシージャの終了と共に消えるがクラスでは保持される点である．そのほかの特徴としてはクラスは同じ構造を持ち名前の異なるコピーを自由に作れること，クラス内でサブクラスが定義でき階層構造が作り易い等がある．

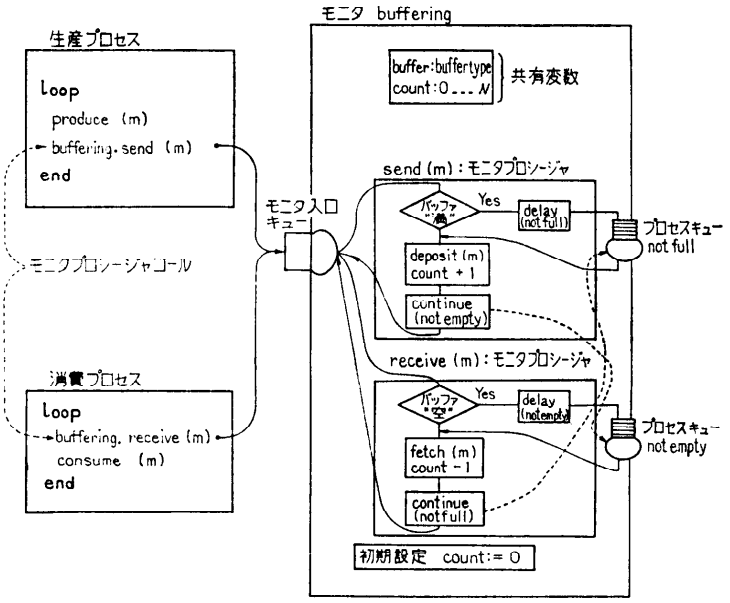


図-3 モニタの概念図

CP では，このクラスの概念に基づいて以下に示す 3 つのシステムタイプを導入した．CP プログラムの構造はこれらのシステムタイプで示すことができる．

(1) プロセス：データとそれを操作するプログラムから成り，普通のプロセスに相当する．

(2) モニタ：データとそれを操作するプロシージャから成り，その操作はプロセスからのモニタプロシージャコールとしてアクセスされる．

(3) クラス：データとそれに対する操作であるが，モニタのように複数のプロセスからアクセスされることはなく，モニタまたはプロセスに付属した操作である．

CP プログラムの構造はこれらのシステムタイプとそれらの間のアクセス関係を示す矢印で示すことができる．**図-4** は MBP の構造であり，生産者プロセスと消費者プロセスがモニタであるバッファにアクセスしている．CP ではダイナミックにプロセスを生成し

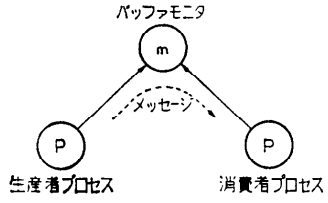


図-4 メッセージバッファのプログラム構造

たり消去したりすることを認めないので，プログラム構造は終始変ることはない．そしてシステムタイプのうち主体的に動作するのはプロセスだけであり，モニタとクラスはコールされるだけなので，プロセスを起動させればプログラムが動作することになる．図-5 は MBP を CP でプログラムしたものである．前節までの説明では buffer をバッファタイプとして定義し，deposit と fetch という操作を使っていたが，このプログラムではこれらを具体化し，より細いマシン命令のイメージに近い形に落している．ここでは buffer は N 個の文字アレイとし，バッファ内のメッセージの先頭を示すポインタ (head) とメッセージの末尾を示すポインタ (tail) を使っている．これによって deposit は tail の指すスロットへのメッセージの格納と tail のインクリメントで実現され，fetch は head の指す先頭メッセージの取出しと head のインクリメントで実現できる．こうするとメッセージが格納される領域はアレイ中を一方向に動くことになるが，N スロットの次は第1スロットにもどるようにサイクリックに管理(tail=tail **mode** size+1) することで無限に使用できる (図-6)．

一般にプログラムはタイプ宣言，変数宣言，プログラムボディの3つの部分から成る．図-5 の (i) の部分はタイプ宣言，(ii) が変数宣言，(iii) がプログラムボディに相当する．(i) では buffering, producer, consumer という3つのタイプを宣言している．ⓜ は buffering というタイプがモニタであることを示し，共有変数と2つの操作 (send と receive) および初期操作より成る．producer と consumer というタイプは共にプロセスタイプで，buffering タイプの資源にアクセスする．(ii) は各タイプの実体を作る部分で，Pascal 等で **var** a : real として実数タイプの a という変数を宣言することに相当する．ここでは buf, pro, con という3つのシステム構成要素が宣言されているが，buf 1: buffering 等とすれば buf と同じ構造のシステムタイプのものをいくつでもつくることができる．(iii) はプログラムボディである．**init** buf ではモニタの初期操作部を処理して共有変数の初期設定を行う．プロセスに対する **init** 命令はプロセスに起動をかけるもので，**init** pro (…), con (…) で，pro と con の2つのプロセスが並列実行を開始する．プロセスからは b. send と b. receive によってモニタ内の send と receive にアクセスしている．send と receive は先に示したモニタの規則により，

```
type buffering=monitor                                          ⎫
    const size=10;                                              ⎪
    var notfull, notempty: queue;                               ⎪
        buffer: array (.1..size.) of char;   } 共有変数          ⎪
        head, tail, count: integer                              ⎪
    procedure entry send (m: char);                             ⎪
      begin                                                     ⎪
        if count=size then delay(notfull);                      ⎪
        buffer (.tail.) :=m;                                    ⎪
        tail :=tail mod size+1;          } receive 操作         ⎪
        count :=count+1;                                        ⎪
        continue (notempty);                                    ⎪ ⓜ
      end;                                                      ⎪
    procedure entry receive (var m: char);                      ⎪
      begin                                                     ⎪
        if count=0 then delay (notempty);                       ⎪
        m :=buffer (.tail.);                                    ⎪
        head :=head mod size+1;          } send 操作            ⎪
        count :=count-1                                         ⎪
        continue (notfull)                                      ⎪
      end                                                       ⎪
    begin head :=1; tail :=1; count:=0;  } 初期操作             ⎪
    end                                                         ⎭ (i)
type producer=process (b: buffering);                           ⎫
      var i: char;                                              ⎪
      cycle                                                     ⎪ (p1)
        produce (i);                                            ⎪
        b. send (i);                                            ⎪
      end;                                                      ⎭
type consumer=process (b: buffering);                           ⎫
      var o: char;                                              ⎪
      cycle.                                                    ⎪ (p2)
        b. receive (o);                                         ⎪
        consume (o);                                            ⎪
      end;                                                      ⎭
var buf: buffering; pro: producer; con: consumer;                  } (ii)
begin                                                           ⎫
  init buf, pro (buf), con (buf), end.                          ⎭ (iii)
```

図-5　CP によるメッセージバッファプログラム

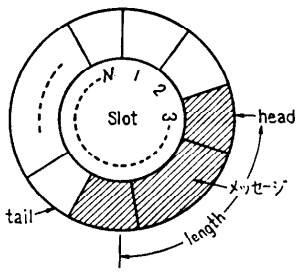


図-6　バッファの実現

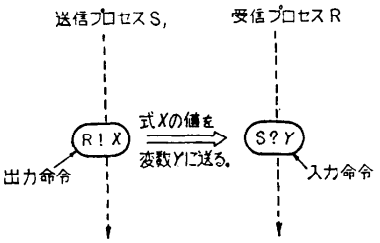
同時にアクセスされることは無い．またプロセスキューによるプロセスの delay, continue も先に示したとおりである．例えば生産者プロセスはメッセージをバッファに送るとき send を呼ぶが，バッファが“満”なら delay で待ちになり，消費者プロセスからの receive でバッファに空ができ continue 命令を出され

ると再起する.

Brinch Hansen は CP プログラムを PDP 11/45 上で走らせるシステムを作ったが，そこでは並列プロセスを TSS 方式でシミュレートしている．また筆者等[15),16)]はマルチプロセッサ上で走らせるシステムを開発した.

3.3 メッセージパッシングと Ada

従来の計算機システムは，並列プロセスがメモリを共有する形で発展して来た．**3.2** 節に示したモニタもそのような状況が背景にある．しかし近年マイクロコンピュータが容易に入手できるようになるに伴って共有メモリを持たずにプロセッサ間通信だけで並列処理を行うマイクロコンピュータネットワークの構成が実現可能となった．このようなシステム構成はリアルタイムシステムの応用等でよく用いられる．この種の分散処理のプログラミングには，プロセス間の同期をメッセージの交換で実現するメッセージパッシングの考え方が向いている．メッセージパッシングの考え方は新しいものではなく，Conway[17)]等がプロセス間の同期に通信を利用している．しかし従来のメッセージパッシングはメッセージを送る側と受ける側が，通信が行われることを互いにはじめから了解しており，単にメッセージパッシングというイベントの発生を待つだけであった．これを決定性メッセージパッシングと呼ぶが，2章の同期の所で使った決定性とは別の意味と考えた方がよい．1978 年に Hoare[18)]は並列プログラム記述には決定性メッセージパッシングだけでは不足で，あらかじめどんな通信が行われるかを決定できない非決定性メッセージパッシングの概念が必要であることを指摘し，Communicating Sequential Process (CSP) という解を示した．同じころ Brinch Hansen[19)]も同様の思想に基づいて Distributed Process (DP) という言語概念を提案した．**3.3.2** に示す Ada は，CSP, DP の影響を強く受けた言語である.



(a) CSP の通信形態

```
Buffering
  buffer: (1..size) char;                          バッファの定義
  head, tail, count: integer
  head :=1; tail :=1; count :=0;                   初期設定
  cycle count<size, producer ? buffer (tail)→      }(i)
       tail :=tail mod size+1; count :=count+1
  □ count>0; consumer ? more ( )→                  }(ii)
       consumer ! buffer (head); head :=head mod size+1;
       count :=count-1
  end
```

(b) CSP によるメッセージバッファプログラム

図-7 Communicating Sequential Process.

3.3.1 メッセージパッシング

非決定性を含むメッセージパッシングの実現は，メッセージを送る通信命令と，非決定性を実現する Dijkstra のガーデッドコマンド (GC)[20),21)]で実現される．まず通信についてみると，次のようなものがある.

(1) 送信プロセスは，メッセージが受信プロセスで受信できるまで待つ型：CSP では**図-7**(a) に示すように，送信プロセスの（受信プロセス名！式）という構文の出力命令と，受信プロセスの（送信プロセス名？メッセージを入れる変数名）という入力命令の間で通信が行われる．すなわち両プロセスの制御がこれらの命令に達したとき，メッセージの転送が行われる.

(2) 送信プロセスは受信プロセスからの応答が返ってくるまで待つ型：DP では，**図-8** に示すように，送信側が受信側のプロセスのプロシージャをコールするような形で相手を指定し，受信側ではそのルーチンが処理できるようになったとき処理を行い終了後その旨を送信側に返す．送信側は受信側のプロシージャ処理終了を待ち次のステップに進む.

(3) 送信側がメッセージを単に受信プロセスのバッファに送り込む.

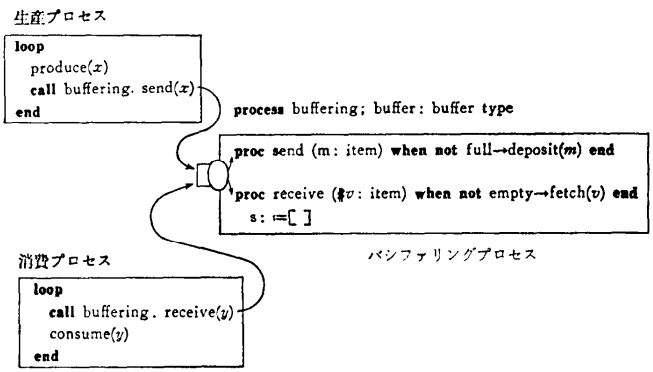


図-8 DP によるメッセージバッファプログラム

次に非決定性を実現するガーデッドコマンド(GC)とガーデッドリージョン(GR)を紹介する．GC とGR はプロセスがプロセスの変数の状態を調べて，いくつかのステートメントのうちから実行するものを選ぶ命令である．GC の構文と意味は次のとおりである．

if B1→S1 ▯ B2→S2 ▯…▯ Bn→Sn **end**

do B1→S1 ▯ B2→S2 ▯…▯ Bn→Sn **end**

ここで Bi→Si の対が GC と呼ばれ，Bi がブール式，Si がステートメントを表わす．**if** 文では B1, B2, ……が調べられ，真の値を持つ Bt のうちの1つ Bm を選び，それと対の Sm を実行して次に進む．もしすべての条件Bが偽の場合プログラムがアボートになる．**do** 文は B のうち真のものがあるうちは次々と対応する S を実行し，真のものが無くなったとき処理を終えて次の命令に進む．もし始めからすべての B が偽なら **do** 文はスキップされる．GC では，真の B が無いときはスキップまたはフォールトになるが，GR では真の値が無いときは真のものが現れるまで待つ．GR の構文と意味は次のとおりである．

when B1→S1 ▯ B2→S2 ▯…▯ Bn→Sn **end**

cycle B1→S1 ▯ B2→S2 ▯…▯ Bn→Sn **end**

when 文は条件の1つが真になるのを待って対応する S を実行する．**cycle** 文は **when** の無限繰返しである．

通信と GC, GR を使った CSP,と DP のメッセージパッシング機構を紹介する．**図-7**（b）はバッファリングプロセスを CSP*で記したものである．

CSP の GC は，条件部の実行が成功したとき実行され，条件部の実行が失敗したとき実行されないという規則になっている．CSP の GC の条件部にはブール式の他入力命令を書くことができる．この場合，入力命令は相手のプロセスの制御が出力命令に達するまで実行が待たされるので，GC は GR のようにふるまう．

バッファリングプロセスでは図-7（b）の（i）か（ii）の条件部の実行が成功するのを待って“→”以下を実行する．（i）については，バッファに空があり(count<size)生産者プロセスからのメッセージが producer? buffer (tail) という入力命令で受取られたときバッファのポインタの変更が行われる．（ii）では count>0 で消費者プロセスからの要求が consumer? more ( ) に来たときメッセージを consumer! buffer (head) で消費者プロセスに送る．（i）と（ii）の両方の条件部の実行が成功しないうちは GR のようにどちらか一方の実行が成功するまで待つし，両方の実行が成功可能なら一方が選ばれて処理される．**cycle** はこの操作が無限にくり返されることを示している．CSP は送信プロセスと受信プロセスが互いに相手を指定し合って通信を行う．しかしこの方法では互いに相手を知っている必要があるし，ほかのプロセスをスケジュールすることができない．DP のプロセスはほかのプロセスからの要求にサービスする共有プロシージャと，イニシャルステートメントから成り，通信は相手のプロセスの共有プロシージャへのコールの形で行われる．サービスを要求されたプロセスは，自分の仕事が終ったり，待ちになったときコールされた共有プロシージャの処理を行って結果を返す．呼んだ方のプロセスは，呼ばれた側の処理が終了するまで待つ．図-8 は DP による MBP でありバッファリングプロセス中の **when** は GR を意味する．

3.3.2 Ada*[22),23)]

米国防省 (DOD) ではエンベッデッド計算機システム (ECS)**のプログラミング用高級言語 Ada を開発した．Adaには並列処理を記述する機能も含まれており，今後この分野のプログラミングに大きな影響を与えると考えられている．

Ada のプログラムは**図-9** に示す4種類のプログラムユニットを基本単位とし，それらのネスト構造で構成される．ネスト構造とはプログラムユニット内で更に別のプログラムユニットを定義するもので，例えば Pascal 等のプロシージャ内で別のプロシージャやファンクションが定義されることに相当する．プログラムユニットのうち並列処理に関係するのがタスクである．タスクは前節までの説明でのプロセスに相当する．

CP のプロセスは始めに起動された後は動的に生成，消去されない静的実体であるのに対し，Ada のタスク

1. サブプログラム
   - (i) プロシージャ
   - (ii) ファンクション
2. モジュール
   - (iii) パッケージ：サブプログラムとデータ資源から成るモジュールで，クラスに相当する．
   - (iv) タスク：並列プログラムのためのもので，タスク間で相互作用を行う．

図-9 Ada プログラムを構成するプログラムユニット

* 記法は説明の都合上 Hoare のものと少し変えている．

*：本稿中の Ada は 1979年5月以降の Preliminary Ada を意味している．

**：計算自体が目的ではなく，システムの一部に組込まれて処理を行うタイプの計算機システム．

はプロシージャ内等で **initate** 文により起動されたり，タスク末で終了するといった動的性格を持つ．MBP でいうと1番外側のプロシージャ内で PRODUCER, CONSUMER, BUFFERING の3つのタスクが定義され，プロシージャボディで3つのタスクを起動することになる．プログラムユニットの構造は，スペシフィケーション部とボディから成る．図-12 がバッファリングタスクである．スペシフィケーション部ではプログラムユニット外部から参照されるもの例えば変数名やサブプログラム名，またタスクの場合ではタスク間通信の入口（**entry** と前置される）などが示される．ボディには本体やスペシフィケーション部で宣言されたサブプログラムの本体等が示される．

タスク間のメッセージパッシングは **accept, select, when** 文によって実現される．先ずメッセージの伝達を行うのが **accept** 文とそれに対するエントリーコールである．**図-10** はタスク S がタスク R にメッセージを転送する様子を示している．送信タスクはエントリーコール（受信タスク名・受信入口）で受信入口を指定し，受信側は **accept** 文でこれを受ける．図-10 の **accept** 文の E1 はメッセージ受入口であり，外部から見える部分なのでスペシフィケーション部に **entry** E1 と宣言される．送信側がコールしても受信側が **accept** に達しなければ送信側は待つし，受信側が **accept** に達してもコールが無ければ受信側が待つ．送信側と受信側のタスクの制御がそれぞれエントリーコールと **accept** に達すると「ランデブーが起こった」といい，次の処理が行われる．先ず送信側からのパラメタが受信側に送られる（入力パラメタは **in** と前置されている）．その後受信側では **accept** に続く **do...end** 間の処理に入り，送信側はこの処理が終るまで一時待つ．最後に出力パラメタ（**out** と示される）があれば，それを送り返し，その後両タスクは独立に走る．

以上の処理は CSP の入出力命令に相当するが，CSP では入力タスクと出力タスクが一対一であった．Ada の **accept** では相手のタスクを指定せず，一対多のメッセージ交換関係が可能になる．その結果各入口には多数のタスクからの要求が到着する可能性があり，入口にはキューを用意する必要がある．

select 文は，ある時点で起こり得るいろいろなイベントを待つ命令で非決定性メッセージパッシングを可能にする．すなわち **accept** 文の集合のうちランデブー可能な **accept** 文を1つ選んで処理を行う．**図-11** は

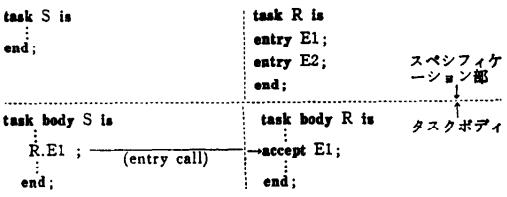


図-10 タスク間通信

```
select__statement ::=
  select
    [when condition⇒]
      select__alternative
  {or [when condition⇒]
       select__alternative}
  [else
    sequence__of__statements]
   end select
select__alternative ::=
     accept__statement [sequence__of__statements]
  | delay__statement [sequence__of__statements]
```

（a） **select** 文の構文

```
select
        accept A;
        :
  or
        accept B;
        :
  or
        accept M;
end select
```

A, B,…M のうちランデブー可能なものから1つ選んで処理する．

（b） select 文の例

図-11 Select 文の構文と例

select 文の構文である．もしどの **accept** もエントリーコールされていなければ，どれかがコールされるのを待つし，複数の **accept** がコールされていれば1つを選んで処理する．さらに**図-11**（a）に示されるように，**accept** 文の前に **when** 文を付加えて **accept** 文の条件を示すことができる．この場合には **when** で示す条件が真で，さらに **when** に続く **accept** がランデブー可能なとき **select** での選択の対象となる．**図-11**（b）のように **when** 文が無いときは条件は常に真と解釈されている．また，すべての **when** 条件が偽ならエラーになる．

図-12 は Ada による MBP であり，**when** はバッファが“満”か“空”かをチェックする条件である．**select** ではまずこれらの条件が調べられ，真のものだけが選択の対象となる．図-12 では COUNT≧SIZE ならバッファが満であるのでバッファへの書込み要求 WRITE は受付られず，COUNT≦0 ならバッファが“空”なのでバッファからの読出し要求 READ は受付けない．もし 0<COUNT<SIZE なら WRITE, READ の両方が **accept** 可となり，コールに応じて処理が行われる．

```
task BUFFERING is                                  ┐
  entry READ (M: out CHARACTER);                   ├ スペシフィケーション部
  entry WRITE (M: in CHARACTER);                   ┘
end
task body BUFFERING is                             ┐
  SIZE: constant INTEGER :=10;                     │
  BUFFER: arrry (1..SIZE) of CHARACTER;            │
  COUNT: INTEGER range 0..SIZE :=0;                │
  HEAD, TAIL: INTEGER range 1..SIZE :=1;           │
begin                                              │
  loop                                             │
    select                                         │
      when COUNT<SIZE⇒                             │
        accept WRITE (M: in CHARACTER) do          │
          BUFFER (TAIL) :=M;                       │ ボディ
        end;                                       │
        TAIL :=TAIL mod SIZE+1;                    │
        COUNT :=COUNT+1;                           │
    or when COUNT>0⇒                               │
        accept READ (M: out CHARACTER) do          │
          M :=BUFFER (HEAD);                       │
        end                                        │
        HEAD :=HEAD mod SIZE+1;                    │
        COUNT :=COUNT-1;                           │
    end select;                                    │
  end loop;                                        │
end BUFFERING;                                     ┘
```

図-12 ADA によるメッセージバッファプログラム

4. むすび

本稿では並列処理記述の概念として基本的で整理されたモニタとメッセージパッシングを紹介し，それらの高級言語への導入を示した．この分野はまだ発展段階にあり，これらの概念や言語に対する興味ある議論[24)-34)]や，これらと異なるアプローチを模索する研究もさかんになりつつあるが，ここでは参考文献を示すに止める．また現在の計算機でこれらの言語を実現しようとすると不充分な点も多く，言語側から計算機アーキテクチャへの影響も今後期待される分野である．

参考文献

1) Dijkstra, E. W.: Cooperating Sequential Process, Programming Languages, F. Genuys(ed.), Academic Press. pp. 43-112 (1968).
2) Brinch Hausen, P., Operating System Principles, Prentice-Hall, (1973).
3) Brinch Hansen, P.: A Keynote Address on Concurrent Programming, IEEE, COMPUTER, pp. 50-56 (May 1979).
4) Hoare, C. A. R.: Towards a Theory of Parallel Programming, Operating Systems Techniques, Academic Press (1972).
5) Brinch Hansen, P.: Structured Multiprogramming, CACM Vol. 15, No. 7, pp. 574-578 (July 1972).
6) Dijkstra, E. W.: Hierarchical Ordering of Sequential Process, Acta Informatica, Vol. 1, pp. 115-138 (1971).
7) Hoare, C. A. R.: Monitors: An Operating System Structuring Concept, CACM Vol. 17, No. 10, pp. 549-557 (Oct. 1974).
8) Brinch Hansen, P.: The Programming Language Concurrent Pascal, IEEE Trans. Software Engineering, Vol. SE-1, No. 2, pp. 199-207 (June 1975).
9) Wirth, N.: Toward a Discipline of Real-time Programming, CACM, Vol, 20, No. 8, pp. 577-583 (Aug, 1977).
10) Wirth, N.: Modula: a Language for Modular Multiprogramming, Software-Practice and Experience 7, 1, pp. 3-35.
11) Holt, R. C. et al.: Structured Concurrent Programming with Operating System Application, Addison-wesley, (1978).
12) Brinch Hansen, P.: The Solo Operating System, Software-Practice and Experience, Vol. 6, No. 2, pp. 141-205 (Apr.-June 1976).
13) Brinch Hansen, P.: The Architecture of Concurrent Programs, Prentice-Hall (1977).
14) Dahl, O. J.: Hierarchical program structure, In Structured Programming, Academic Press (1972).
15) 古谷：マルチプロセッサシステムにおける Concurrent Pascal マシン，情報処理学会論文誌 Vol. 21, No. 2 (1980年3月).
16) Brinch Hansen, P.: Multiprocessor Architectures for Concurrent Programs, ACM 78, Conf. proc. pp. 317-323 (Dec. 1978).
17) Conway, M. E.: Design of a separable transition diagram compiler, CACM, Vol. 6, No. 7, pp. 396-408 (July 1963).
18) Hoare, C. A. R.: Communicating Sequential Processes, CACM, Vol. 21, No. 8, pp. 666-677 (Aug. 1978).
19) Brinch Hansen, P.: Distributed Processes: a Concurrent Programming Concept. CACM Vol. 21, No. 11, pp. 934-941 (Nov. 1978).
20) Dijkstra. E. W.: Guarded Commands, Nondeterminacy and Formal Derivation of Program, CACM. Vol. 18, No. 8, pp. 453-457 (Aug. 1975).
21) Dijkstra, E. W.: A Discipline of Programming, Prentice-Hall (1976).
22) DoD: PRELIMINARY ADA REFERENCE MANUAL, Sigplan Notice 14 (1979).
23) DoD: Rationale for the Design of the ADA Programming Language, Sigplan Notice, Vol.

14 (June 1979).
24) Lister, A. M.: The Problem of Nested Monitor Calls, Operating Systems Review, Vol. 11, No. 3, pp. 5-7 (1977).
25) Haddon, B. K.: Nested Monitor Calls, Operating Systems Review, Vol. 11, No. 4, pp. 18-23 (1977).
26) Löhr, K.P.: Beyond Concurrent Pascal, proc. 6th ACM Symposium on Operating System Principles, pp. 173-180 (Nov. 1977).
27) Dijkstra, E. W.: DoD-1: The Summing Up, Sigplan Notice, 13 (July 1978).
28) 徳田：Ada 実現に関する問題点について，情報処理，Vol. 21, No. 3 (1980年3月).
29) Kieburtz. R. B. and Silberschatz: Comments on Communicating Sequential Processes ACM Trans. on Programming Languages and systems, Vol. 1, No. 2, pp. 218-225 (Oct. 1979).
30) Laner. H. C. and Needham R. M.: On the Duality of Operating System Structures, Operating systems Review 13-2, pp. 3-19 (Apr. 1979).
31) 土居：順路式，情報処理，Vol. 19, No. 8 (1978年8月).
32) Liskov, B.: Primitives for Distributed Computing, Proc. of the 7th Symp. on Operating Systems Principles, 33-42 (Dec. 1979).
33) Campbell, R.: Path Expression in Pascal, Proc. of the 4th Int. Conf. on Software Engineering, pp. 212-219 (Sep. 1979).
34) Yonezawa: Comments on monitors and path-expressions, Journal of Information Processing, Vol. 1, No. 4 (1979).
35) Bryant, R. E. and Dennis, J. B.: Concurrent Programming, TH-115, LSC, MIT (Oct. 1978).
36) 塚本：並列プログラミング，「第5世代コンピュータに関する調査研究」，日本情報処理開発協会 (1980).

(昭和55年5月1日受付)