非同期処理について（I）*

丸　山　　武**

1. はじめに

計算機プログラムの分野で，非同期処理の概念が重要な意味を持ち始めた．その背景には

- 処理装置の機能分離と多重化．
- 多重プログラム処理を行なう管理プログラム・システムの発展．
- オンライン，リアルタイム応用の発展．

などの要因があげられよう．

計算処理をつかさどる中央処理装置（Central Processing Unit，以下 CPU と略す）と，入出力データ転送制御を受け持つデータチャネル（Data Channel，以下 DCH と略す．IOP＝Input Output Processor と呼ぶ方が適切かもしれない）の分離は，新しいことではない．CPU は，入出力命令（の列，チャネル・プログラム，channel program）を DCH にわたして，それ以降の処理を続行する．おいてけぼりされた DCH は，入出力動作を実行して，完了したら CPU に割り込みをかけて，その完了を告げる（**図1**参照）．このようにして，計算処理と入出力の同時並行動作が，可能となったわけである．

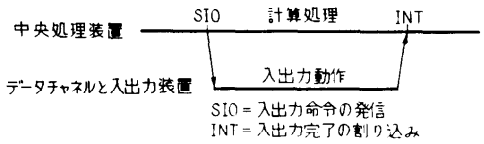


図1　計算処理と入出力動作の並進

さらに，CPU 自身を多重化したシステムも出現してきている．多重 CPU 系は，いくつかの型に分類されるようであるが，以下では，主として対称型の多重 CPU 系を考える．すなわち，2個以上の CPU が，主記憶を共用し，固定的主従関係を持たず，ほぼ対等の機能を備えて動作するシステムである．このようなシステムにおいて，1つのプログラムを2分して，それぞれの CPU に分業させる場合，2分された作業は，非同期的に（それぞれかってに）並進するのであるから，非同期処理の技法を，入出力に限定せず，ある程度一般的に確立しておかなければならない（これは，もちろん，入出力インタフェースが最も高い完成度をみせた非同期系の典型であることを否定するものではない）．

オンライン，リアルタイム応用に計算機システムが用いられる場合，たとえば，各端末は他の端末とは独立に，ランダムにデータを送り込んでくる．これを受信して，元帳の参照・更新などを行なって，返事を端末に送信する，というのが1つの型であろうが，1トランザクション処理中にも，他トランザクションが入力されてくるのだから，これに対しても，適当に応答できなければならない．この場合，計算機システムは動的な社会機構の一構成要素をなすわけで，一方，現実の社会機構は，まぎれもない非同期系であるのだから，計算機システムに非同期処理が要求されるのは，いわば，必然といえるのかも知れない．

以下，非同期処理に関する基本機能，プログラム技法について述べたい．

2. 待　合　せ

2.1 イベント

日常の生活の中でも「待つ」ことは多い．電車の到着を待つ，先方の返事を待つ，硬貨を投入して牛乳のパックが出てくるのを待つ，などなど．計算機プログラムの中では，入出力命令を発してその完了を待つ，というのが一番多い待合せであろう．そのほかにも，操作員に問合せをタイプアウトして，その返事がタイプインされるのを待つ，などがある．いずれにしろ，待ち合わせる相手は，完全には自分の自由にならないのが世の常である．このように「相手のある」事象は，非同期的に（相手の都合によって）発生する．計算機プログラムのなかでは，そのような事象を，とくにイベント（event）と呼んでいる．

先方の返事というイベントを待ち合わせる場合に

---

* A Guide to Asynchronous Processing (Part I), by Takesi Maruyama (FUJITSU LIMITED)

** 富士通(株)・ソフトウェア技術部

は，返事がくるまで，ただひたすらに待ちつづけるというのはまれで，自分の方の仕事をかたずけるのが普通だ．計算機プログラムの方でも同様で，相手の都合にわずらわされずに，自分の所用をできるだけすませてしまうのが，うまいプログラムなのであろう．だが，いずれ，相手の返事を待つこと以外に，何もすることがないという事態に直面する．操作員からの返事がこない限り，右へ進むのか，左へ進むのか決められないというような事態である．

この場合，イベントの発生が，プログラム進行の条件となるわけで，イベントが発生しない限り，プログラムを続行することはできない．システム内に，自分のほかにもプログラムがいるような，多重プログラム・システムの場合には，CPU を別の独立なプログラムにあけわたすことが，CPU の使用効率の面から要求される．

2.2 WAIT

CPU をあけわたすべき相手のプログラムの選択や，いったんあけわたした後でイベントが発生してから，再びそのプログラムに CPU を割り当てること，などの作業は管理プログラムの役割であるが，少なくとも一般大衆プログラムは，「しかじかのイベントが発生するまで CPU をあけわたす」ことを，管理プログラムに向けて宣言しなければならない．そのために，WAIT（待合せ）というようなマクロ命令が用意されている．

ここに，マクロ命令というのは，金物に備わっている実行命令ではなく，管理プログラムが実現する，やや大きな命令のことで，実際には，管理プログラム・ルーチンを呼び出すことに帰着する例が多い．

WAIT マクロ命令では，待ち合わせるイベントを，そのパラメタとして添えるのであるが，問題はイベントの指定方法にある．WAIT するということは，逆に，指定したイベントが発生したら，再び，WAIT マクロ命令のつぎの命令群を続行するということであるから，「何を」（これが，すなわち，イベントであるが）待ち合わせるのか機械的にわかるようになっていなければならない．そのために，WAIT すべきイベントは，それ以前に定義宣言しておくのが普通である．たとえば，データを読み込むときには，読込み作業の終了をイベントとすることを宣言する．

イベントに対して，イベント制御ブロック（event control block，以下 ecb と略す）を設けて管理する．イベントの発生や待合せを，この ecb に表示しておくのである．システムによっても違うが，ecb をプログラム変数として，普通の変数なみに確保する場合には，ecb の番地でもって，イベントそのものを識別することができる．

データ読込み作業の終了をイベントとして，定義宣言するためには，対応する ecb の番地を，データ読込みマクロ［仮に READ（読込み）とする］命令の1つのパラメタとして添えてやればよい：

READ (....., ecb)

ここに，「……」は，ecb 以外のパラメタが存在するという気持である．同様に，この読込みが完了するのを待ち合わせるには，同じ ecb の番地をパラメタとして添えた WAIT マクロ命令を発すればよい：

WAIT (ecb)

2.3 POST

WAIT マクロは，イベントの発生を待ち合わせるためのものであるが，反対に，イベントの発生も POST（通知）というようなマクロ命令を介して宣言する：

POST (ecb)

先にあげたデータ読込みの完了を表わすイベントに対して，POST マクロを発するのは，管理プログラムのなかの入出力制御，とくに，入出力割込み処理（後処理）ルーチンあたりであろう．

POST マクロの具体的作業は，指定された ecb にイベントの発生を表示して，これを待ち合わせているプログラムがあれば，その待合せを解除することである．

ついでながら，FORTRAN, ALGOL, COBOL などのプログラム用言語における入出力命令文は，このような ecb を意識して WAIT するようなことは不要になっている．PL/I でも，ecb が顔を出すのは，バッファリングしない特別の場合だけである．実は，ecb の存在は，言語処理プログラムごとに用意される入出力制御ルーチンや，管理プログラムのなかの入出力制御ルーチンのなかにかくれているである．

2.4 交代バッファ技法

1つの例として，簡単な交代バッファ技法を用いたデータ読込み手順を考えてみよう．2個のバッファ

area [0], area [1]

を用意しておいて，一方のバッファに対するデータ読み込みと，他方のバッファに対する計算処理を並進させる．すなわち，area [i] にデータを読み込みながら，area [j] 中のデータを計算処理するものとする．ここに，i, j は一方が0で，他方が1の値をとるように

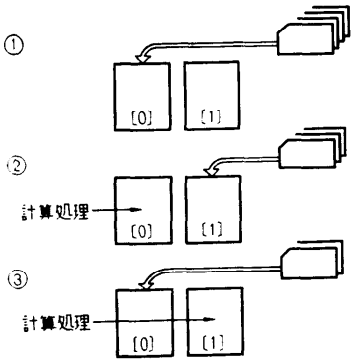


図2　交代バッファ技法図解

制御する（**図2**参照）．ALGOL 風にこのプログラムを書いてみよう：

```
        READ (....., area [0], ecb);
        i: =0;
        j: =1;
    w: WAIT (ecb);
        k: =j;
        j: =i;
        i: =k;
        READ (....., area [i], ecb);
        compute (area [j]);
        goto w;
```

ただし，computeは，計算処理手続きを表わすものとする．また，このプログラムでは，入力データは無限にあって尽きることがないものとしている．実際には，「入力データが尽きた」場合の処理を含めなければならない（**図3**は，このプログラムの進行のタイミングを示すものである）．

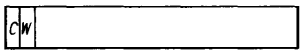
2.5　ecb の制御

さて，WAIT マクロ命令を発したとき，指定されたイベントについて

- すでに発生している……POST が WAIT に先行した．
- まだ発生していない……WAIT が POST に先行した．
- ちょうど発生したところである……WAIT と POST がほぼ同時に発せられた．

という 3 とおりのケースが考えられる．WAIT と POST の順序関係のすべてについて，合理的に処理することが必要である．

図3　交代バッファ技法例題のタイミング



C = 0はイベントが，まだ発生していないことを示す．
C = 1はイベントが，すでに発生していることを示す．
W = 0はWAITマクロが，まだ発せられていないことを示す．
W = 1はWAITマクロが，すでに発せられていることを示す．

図4　ecb の構成

ecb には，**図4**のように，制御用情報として，c，w の2ビットを設ける．これらのビットは，WAIT，POST の際，つぎのように扱われる．

WAIT マクロが発せられたとき，WAIT ルーチンは指定された ecb の c ビットを調べる．$c=1$ であれば，イベントはすでに発生しているのだから「待ち合わせる」必要はない．WAIT マクロのつぎの命令にもどるだけ．もし，$c=0$ であれば，イベントはまだ発生していないのだから，これを「待ち合わせる」必要がある．$w=1$ にして，CPU を他プログラムにあけわたす．

一方，入出力動作の完了割込みなど，イベントが発生して POST マクロが発せられたとき，POST ルーチンは指定された ecb の w ビットを調べる．$w=0$ であれば，$c=1$ に設定しておしまい．$w=1$ であれば，このイベントを待ち合わせているプログラムがあるはずである．そのプログラムを捜しだして，それを続行させる（CPU 割当て対象とする）．

おもしろいのは，WAIT マクロを発したとき，ちょうどイベントが発生するケースである．いま，このイベントは入出力動作の完了であるものとしよう．入出力動作の完了は割込みによって通知される．すなわち，通常のプログラム処理を中断して，入出力動作完了割込み受付けのルーチンに制御が強制的に移される．このルーチンは若干の後処理（エラーチェック，つぎの入出力命令の発信など）の後，POST マクロを発する．いま，ちょうど WAIT マクロが発せられて，WAIT ルーチンが，ecb の c ビットを調べてみて $c=0$ である

ことを知り，$w=1$ に設定しようとしたところで，入出力割込みが発生して，それが運悪く，まさに WAIT マクロで待ち合わせようとしたイベントの発生に一致したものとしよう．POST ルーチンが，その ecb の w ビットを調べると，まだ $w=0$ であるから，$c=1$ に設定しておしまい．割込み処理を終わって中断地点，すなわち，WAIT ルーチンにもどっても，こちらの方は $c=0$ と信じこんでいるから，さきほどの続きで，$w=1$ に設定しておしまい．結局，プログラムは，実はイベントは発生ずみなのに，永遠に待ち続けるということになる．これは「すれ違い」と呼ばれる現象であるが，割合簡単に避けることができるのである．イベント発生は入出力動作の完了に限らないが，いずれも割込によるものであるから，WAIT ルーチンを走っているときには，割込みを受け付けなければよい．たいていの計算機には，「割込み禁止」の機能が備わっているから，WAIT ルーチンにはいるところで，割込み禁止の状態に設定しておき，出て行くところで禁止を解除することにする．割込み禁止期間に，割込み原因が発生しても，実際の割込みは，禁止が解除されるまで延期される．こうして，WAIT ルーチンと POST ルーチンが入りまじって走ることはなくなり，WAIT が POST に先行するか，もしくは，POST が WAIT に先行するか，どちらか一方のケースしか起こりえないことになる．

ただし，多重 CPU 系の場合には，他 CPU の割込みを禁止するような方法はとれず，別の手段が必要になるが，これについては，後で触れよう．

非同期系のプログラムをデバッグしているとき，初めスナップ・ショット・ダンプをとりながら走らせている間はうまく動作していたのに，これでよしと思って，スナップ・ショット・ダンプをとりはらって走らせてみると，うまく動作しない，ということがよくある．ダンプをとりながら走らせたときには，POST が WAIT に先行し，ダンプをとりはらうと，この関係が逆になるなど，タイミングが変わって，バグが表面に顔を出した結果ということが多い．

3. タ ス ク

3.1 タスクの概念

プログラムが，イベントの発生を待つことのほか，何もすることがなくなったような場合，もし，そのイベントが，まだ発生していなければ，CPU を他のプログラムにあけわたすべきことは，すでに述べた．管理プログラムの側からみれば，WAIT マクロを発したプログラムから CPU をとりあげて，別の（WAIT していない）プログラムにわたすことである．そこでは，CPU を割り当てる対象を，明確にしておく必要がある．いままで，ばくぜんと，CPU をプログラムに割り当てるといってきたが，プログラムという語は，慣用にすぎてあいまいなので，CPU 割当て対象の単位としては，「タスク（task）」という別の語を用いる人が多くなってきた．人によっては「プロセス（process）」などともいうようであるが，以下では，タスクということにしよう．管理プログラムは，CPU をあるタスクからとりあげて，他のタスクに割り当てるということをひっきりなしに繰り返している．この操作をタスク切換え（task switch）と呼ぶ．タスク切換えの頻度が高すぎると，CPU を切換え作業にばかり費やすことになってむだであるが，逆に低くすぎると，動的状況の変化に対処できないので，システム全体の使用効率を悪くする．なかなかむずかしいところである．

タスクという語は，日本語の「仕事」を表わすのであるが，似たような語に「ジョブ（job）」というのがある．こちらの方は，利用者側からみた「仕事」の単位を意味して使われるようである．利用者が計算機室に持ち込んでくるのがジョブであって，CPU が実行するのがタスクという具合に解される．

ジョブは，たとえば

FORTRAN 翻訳，
ライブラリとの結合編集，
実行

というように，いくつかの順序づけられた処理段階，つまり，「ジョブ・ステップ（job step）」に分けられる．各ジョブ・ステップは，1 タスクとして，CPU で実行されることが多いが，システムによっては，1 ジョブ・ステップを 2 個以上のタスクから構成することを認めているものもある．その場合には，同じジョブ・ステップに属する 2 個以上のタスクに対して，CPU が少しずつ，時分割的に分け与えられる．したがって，これらのタスクは，仮に単一 CPU 系であっても，並行に作業が進行するようにみえる．これは従来のあたりまえのプログラムには，みられなかった新しい機能である．すなわち，これまでの普通のプログラムの実行は，割込みなどの目に見えない（割り込まれた側のプログラムにとっては意識する必要のない）事件を別にすれば，逐次的に 1 命令ずつ行なわれるも

のであった．しかし，ふりかえって考えてみるに，プログラムは，一般に役割を異にするいくつかの副作業からなっていて，この副作業のなかには，必ずしも逐次的に実行する必要がなく，並行に実行する方が時間的にも得策であるし，プログラムの処理の上からも，より自然であるようなものもないではない．

人工的でつまらない例ではあるが，入力データを読み込んで，計算処理し，結果を出力印刷するプログラムを考えてみよう．これに対して，先に述べた交代バッファ技法を入力・出力の両方に適用することもできるが，強引に

読込み部分，
計算部分，
印刷部分

の3個の副作業に分割し，部分部分をそれぞれタスクとして構成してみる．読込み，計算，印刷は非同期的に進行することになろう．つまり

データ1の読込み→計算→印刷→データ2の読込み→計算→印刷→データ3の読込み→計算→印刷→データ4の読み込み→……

という具合に1つ1つ物事が進行するのではなく

- 読込みタスクは，データを読み込めるだけ読み込んでは，これを計算タスクにわたして行く．計算タスクが前のデータの計算を終了するのを待たずに，とにかく，読み込んだデータをわたして，つぎのデータの読込みに進む．
- 計算タスクは，わたされたデータを，自分のペースで計算処理して行き，計算結果を印刷タスクにわたす．印刷タスクが前の計算結果の印刷を終了するのを待たずに，とにかく，計算結果をわたして，つぎの計算にとりかかる．
- 印刷タスクは，わたされた計算結果をただ黙々と印刷出力する．

というように，多少の相互連絡はとりながらも，あまり内政干渉せずに，おおむね，ばらばらに作業を進めて行くことになる．これは，ちょうど3人の人間が，重なり合わないように，担当範囲を決めて，共同作業を進めるようなものである．

さて，プログラムやシステムを，多重タスクとして構成する場合，タスクに対して，2とおりの見方があるようである．1つは

- タスクを役割の決った部分処理系とみる．

見方であって，特定の機能を備えた装置のようにみなす．先にあげた読込みタスク，計算タスク，印刷タスクは，部分処理系としてのタスクの例である．また，管理プログラムのやや独立した機能，たとえば，システム入力制御や，システム出力制御なども，部分処理系としてのタスクとみられる．タスクに対するもう1つの見方は

- タスクを役割の決まっていない擬似 CPU とみる．

見方であって，こちらの方は万能計算機であるから，どんなプログラムでも実行してくれる．管理プログラムからみて，利用者ごとに用意するタスクはこの例であって，実際，MULTICS などでは，pseudo processor（擬似処理装置）と呼んでいるようである．

市役所のサービスのやり方と同じことで，転入手続きのために，あちこち窓口をわたり歩かせるシステムは，部分処理系タスクを沢山用意しているものであり，1つの窓口で，どんな手続きでも申込みに応じて，すっかり内部で処理してくれるシステムは，擬似 CPU としてのタスクをいくつか配置しているものと類推できよう．応用パターンが決っているものでは，前者が採用され，そうでないシステムでは，後者が採用される．

部分処理系としてのタスクは，「窓口」にほかならないから，その前には，要求の待行列ができる．要求は別のタスクから発生して，この待行列に加わる．ところで，新しい要求が到着したとき，窓口の前には，待行列が全然なく，あき状態のこともあろう．この場合窓口タスクは，サービスする相手がいないので，CPU を他のタスクにあけわたして，いねむりをきめこんでいるはずであるから，鈴をならして，たたき起こしてやる必要がある．具体的には，この鈴というのは，窓口タスクごとに決った ecb であって，いねむりというのは，窓口タスクがその ecb を WAIT していることにほかならない．新着要求は，先行する待行列が空のときに限って，その ecb を POST してやる約束にすれば，めでたく，窓口タスクの WAIT は解除されて，サービスを再開してくれることになる．

3.2 多重処理形タスクと SWAIT マクロ

部分処理系タスクであって要求1件に対するサービスがいくつかの大記憶ファイルへのアクセスを含み，しかも，アクセスしようとするデータが，要求ごとに異なるボリューム上に存在するようなものを考えてみよう．1つの要求に対するサービスを終了してから，つぎの要求を受け付けるやり方もあるが，要求発生の頻度が高い場合には，応答速度をあげるために，もう

ひと工夫ほしいところである．余力がある限り，いくつかの要求を同時にサービスして行くような，聖徳太子みたいな処理構造がとれないものだろうか．このようなタスクを，仮に多重処理形タスクと呼ぼう．

いま，要求1件あたりの処理は，つぎの3個のステージからなるものとしよう：

ステージ 1：前処理の後，対応するデータの読込み命令（READ）を発する．

ステージ 2：読込んだデータに若干の処理を施した後で，データをもとの場所にもどす命令(WRITE)を発する．

ステージ 3：後処理

このうち，ステージ1と2は大記憶ファイルへのアクセスを含み，アクセスが完了するまで，ステージ 2,3 には進めない．この間に生ずる待ちを利用して，つぎの要求を受け付けることにしよう．

サービス中の要求に番号をつけて，$R_1, R_2, R_3, \cdots$ としよう．要求 R_i が処理を受けたステージの番号を S_i で表わす．また，R_i がステージ1と2を終えるときには，ファイル・アクセス終了のイベントを待ち合わせるが，このイベントに対応する ecb を E_i と表わす．さらに，要求の待行列が発生したことを告げる ecb を E_0 と表わすことにしよう．

この多重処理形タスクは，要求 R_i に対して，ステージ S_i の処理を施した後，ecb 群

$E_0, E_1, E_2, E_3, \cdots$

を待ち合わせる．これらのイベントのどれか1つが発生した場合の処理は，つぎのようになる．

- 発生したイベントが E_0 であれば，新着の要求が発生したのであるから，これにステージ1の処理を施した後，ecb 群 $E_0, E_1, E_2, E_3, \cdots$ を待ち合わせる．これには新着要求に割り当てた ecb を含める．
- 発生した ecb が E_i であって，その処理ステージ $S_i=1$ ならば要求 R_i にステージ2の処理を施した後，ecb 群 $E_0, E_1, E_2, E_3, \cdots$ を待ち合わせる．
- 発生した ecb が E_i であって，その処理ステージ $S_i=2$ ならば要求 R_i に最終ステージの処理を施す．これで要求 R_i のサービスは終了したことになる．ecb 群 $E_0, E_1, E_2, E_3, \cdots$ を待ち合わせる．ただし，いま，サービスを終了した要求に割り当てていた ecb E_i は除く．

ecb 群を待ち合わせるたびに，全 ecb を指定し直すのは，厄介だし時間もかかる．そこで，つぎのような新しい SWAIT (Simultaneous WAIT) マクロ：

SWAIT ($ecb_1, ecb_2, ecb_3 \cdots$)

を設けることを考える．これは，$ecb_1, ecb_2, ecb_3, \cdots$ に対応するイベントのうち，どれか1つが発生するのを待ち合わせるもので，1つでも発生すると，待合せは解除され，しかも，待合せを解除した ecb がどれであるかが（その番地が，たとえば，レジスタ経由で）通知される．SWAIT を解除した ecb 以外の ecb はさらに有効であって，同じタスクが再び

SWAIT ($ecb_1', ecb_2', ecb_3', \cdots$)

を発すると，これは ecb 群

$ecb_1, ecb_2, ecb_3, \cdots, ecb_1', ecb_2', ecb_3', \cdots$

（ただし，すでに SWAIT を解除した ecb を除く）

を待ち合わせることになる．つまり，SWAIT は，これ以前の SWAIT で指定した ecb 群のうち，まだ，SWAIT を解除していない ecb をも合わせて待ち合わせるものである．

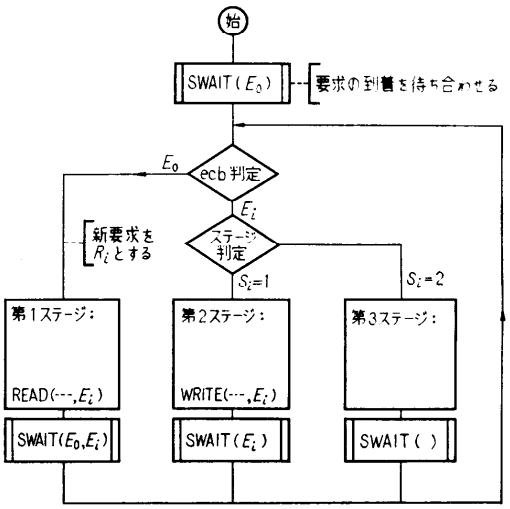


図5 多重処理形タスクの構成

この SWAIT を利用して，この多重処理形タスクの構成は**図5**のようにまとめることができる．これは，ステージの個数を別にすれば，充分一般性を持つものと思われる．ただし，実際には，同時にサービスできる要求の個数は制限されるので，この点を加味する必要がある．

3.3 Conway 系

M. Conway の提案した非同期処理方式（これを，仮に，Conway 系と呼ぶ）は，初期のものであるが，

ecb を中継にタスク制御を行なう WAIT-POST 系と趣が違っていて，制御の分岐・合流というイメージが強烈な点が少しおもしろい．

普通のプログラムの制御の流れは1本であって，一筆書きできる．判断分岐があるにしろ，判断の結果，制御はどちらか一方にだけ流れる．しかし，非同期系では，制御がほんとうに2本に分岐するのである．分岐した2本の流れは並進する．Conway 系では，制御の分岐は，FORK (分岐) マクロ命令：

FORK (l, e, n)

で指定する．これは，特別な変数 e に式 n の正整数値を代入して，かつ，新しいタスク (制御の流れ，支流) として，名札 l の地点から実行を開始するものである．この FORK マクロを発したタスク自身 (本流) は，つぎの命令に進むので，結局，制御は本流・支流の2本に分岐したことになる．初めての分岐のときには，$n=2$ とし，支流がさらに2本に分岐するときには，$n=e+1$ とすれば，変数 e は，並行して進んでいる制御の流れの本数を示すことになる．

こうして分岐した2本の流れは，やがて1本に合流する．WAIT-POST 系では，本流が WAIT して，支流が POST することで合流を実現するのであるが，Conway 系では，合流地点に先にきたものが消滅し最後にきたものだけが生き残るとする．すなわち，合流地点では，本流・支流とも，つぎのような JOIN (合流) マクロ命令：

JOIN (e)

を実行する．これは，e から1を引いて

- $e \geq 1$ なら，実行した制御の流れは消滅する．
- $e=0$ なら，実行した制御の流れは生き残って，この JOIN マクロ命令のつぎに進む．

ものである (**図6**参照)．

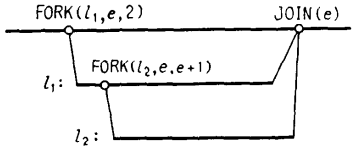


図6　FORK と JOIN

3.4 COBOL の非同期処理機能

COBOL の非同期処理機能を，そのままの形で実現している例はないようである．また，言語仕様上も規定が充分に明確でないなど，いろいろ問題が多いらしい．しかし，いたずらに一般化する方向をとらず，大胆に応用パターンを絞った仕様を意図しているようにみえる点に，筆者は興味を覚える．

COBOL では，非同期的に実行しようとする手続きを，手続きディビジョンの初めの宣言部に，セクションとしてまとめて書いておく．これを「順次実行でない (out-of-line)」手続きと呼ぶ．ここには，大記憶ファイルをランダムに呼出し，その処理順序も，必ずしも呼出し順でないようなデータ処理 (これを，乱呼出し一乱処理手法と呼ぶ) を記述する．この乱一乱手法は，大記憶ファイルがいくつかのボリュームにまたがるような場合，また，入出力管理システムが，いくつかの要求を，システムとして最適になるような順序で，サービスするように並べかえるような場合に，効果が期待される．

順次実行でない手続きに対して，普通の手続きを「順次実行の (in-line)」手続きと呼ぶ．さて，順次実行でない手続きは，順次実行の手続きのなかから，PROCESS (処理) 命令によって起動される：

PROCESS セクション名

こうして起動された順次実行でない手続きの実行は，「サイクル (cycle)」と呼ばれる．起動されたサイクルは並列に進行し，必ずしも起動された順に終了するとは限らない．

おもしろいのは，サイクルを始めるための通信情報領域と，サイクル実行に必要な作業領域を，保存領域 (saved area) レコードと呼ぶ領域にまとめて，順次実行でない手続きにわたす点である．保存領域レコードの構成，用意しておくべき個数は，データディビジョンのファイルセクションに記述しておく．いつも，用意した保存領域レコードのうち1つだけが，順次実行の手続きに対して開かれているが，PROCESS 命令が発せられると，いままで開かれていた保存領域レコードが，起動されたサイクルにわたされ，あいている

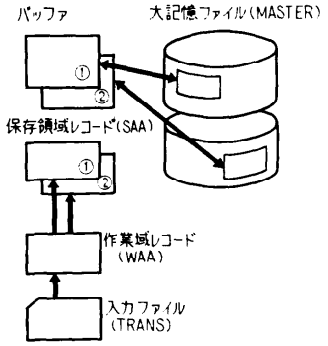


図7　COBOL 例題図解

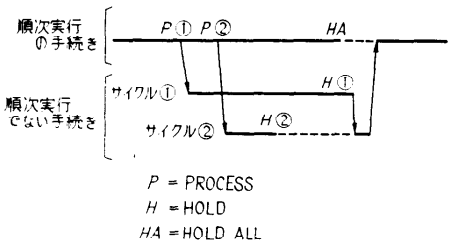


図8 COBOL 例題タイミング

保存領域レコードの1つが新たに選ばれ，前のものに代わって，順次実行の手続きに対して開かれる．したがって，確保しておいた保存領域レコードの個数を越えて，サイクルが起動されることはない．

これは，多重処理形タスクの1種ともみえるが，実現の方法はいろいろありそうである．

COBOL の非同期処理には，個々のサイクルの終了を待ち合わせる機能は用意されていないが，起動した全サイクルの終了を確認する HOLD（待合せ）命令：

HOLD ALL

と，順次実行でない手続きのなかで，起動された順序にサイクルを整列させる命令：

HOLD セクション名

がある．

手続きディビジョンのひな形を，簡単な例によって，以下に示そう．

```
PROCEDURE DIVISION
DECLARATIVES.
MASTER-UPDATE SECTION.
     USE FOR RANDOM PROCESSING.
      :
     READ MASTER; INVALID KEY ~.
      :
     WRITE MASTER-RECORD; INVALID KEY ~.
      :
END DECLARATIVES.
      :
READ-TRANS.
     READ TRANS INTO WAA;
          AT END GO TO FIN.
     PROCESS MASTER-UPDATE
          FROM WAA USING SAA.
     GO TO READ-TRANS.
FIN.
     HOLD ALL.
      :
```

（**図7**，**図8** 参照）．

（昭和44年9月29日受付）