Vol. 24 No. 1 情報処理学会論文誌 Jan. 1983

分散型システム記述用言語 Concurrent C の設計とその処理系の実現†

安藤 誠††* 辻野嘉宏††
荒木俊郎†† 都倉信樹††

近年のハードウェア技術の発展により，複数のプロセッサから構成されたシステムの実現が容易となってきた．本論文では，疎結合分散型システムのためのシステム・プログラム記述用言語 Concurrent C とその処理系の実現方法について述べる．Concurrent C は，システム・プログラム記述用言語Cに並行処理機能を付加した言語であり，複数プロセッサから成るシステム全体に対して，一体化したシステム・プログラムの開発が可能である．Concurrent C のプログラムは，プロセスとプログラム・ユニットから構成され，プロセスはプログラムの一つの機能単位であり，プログラム・ユニットは機能的に関連したプロセスを集めた機能モジュールである．これらのプログラム構造のもとに，動的プロセス生成文 (activate 文)，プロセス間通信 (send 文，receive 文)，共有変数機構（モニタと制御式）などが，言語Cに付加拡張されている．

1. まえがき

疎結合分散型システムは，ネットワークとネットワークを介して互いに通信する複数のプロセッサから構成されている．各プロセッサは，それぞれ，そのプロセッサのみが参照できる主記憶および，そのプロセッサのみが使用できる入出力装置をもっている．このような疎結合分散型システム上でのシステム・プログラム(たとえば，オペレーティング・システムなど)の開発は，各プロセッサごとに1個のプログラムを作成し，それぞれの間をまた別のプログラムで結合する形で行われてきた．しかし，この方法では，システム全体の保守性やプログラムの読みやすさが損われる．また，複数プロセッサにまたがるプログラムのデバッグを，物理的な分散型システムの構築以前に行う場合，シミュレート用のプログラムが作成されるが，デバッグ後に，このプログラムをシステム上で実現するためには，多くの変更が要求される．さらに，従来の言語では，言語上プロセッサの概念が導入されていないため，言語から物理的なシステム構成は見えず，プロセスの動くプロセッサを動的に指定することなどは不可能である．

以上のような疎結合分散型システムのためのシステム・プログラミングの問題点を解決するには，従来から提案されてきた並行処理言語[1),3),5),9),11),16),17)]および機能[4),6),7)]では不十分である．そこで，これらの問題点を踏まえて，システム記述用言語 C[8)] に疎結合分散型システム向き並行処理機能を付加拡張して，言語 Concurrent C を設計した．以下にその特徴を示す．

（1） 複数プロセッサから成るシステム全体に対して一体化したシステム・プログラムを記述できる．

（2） 機能モジュールとして，プログラム・ユニットというプログラム構造を導入する．このプログラム・ユニットは，それがロードされるプロセッサを仮定せずに作成することができる．

（3） プロセッサの概念が言語上明確に現れており，プロセスの動き始めるプロセッサを指定することが可能である．また，プログラム・ユニットが仮想的にプロセッサを表現している．

（4） システム記述用言語としての言語Cと上位互換性をもち，Cのもつシステム記述性の良さを保持し，拡張並行処理機能もCと調和した実行効率の高いものである．

このような特徴をもつ言語 Concurrent C の処理系を NOVA 3 と MP/100(micro NOVA) 結合システムのもとで作成した．本処理系は，コンパイラ・システムと実行時環境を支えるカーネルとから構成されている．

2. 言語 Concurrent C

以下，Concurrent C の概要について述べる（詳細

† Concurrent C : A Programming Language for Distributed Multiprocessor Systems—Design and Implementation— by MAKOTO ANDO, YOSHIHIRO TSUJINO, TOSHIRO ARAKI and NOBUKI TOKURA (Department of Information and Computer Sciences, Faculty of Engineering Science, Osaka University).

†† 大阪大学基礎工学部情報工学科
* 現在 松下電器産業株式会社技術本部

は，文献12),13) 参照)．

2.1 Concurrent C のプログラム構造

Concurrent C のプログラムは，いくつかのプログラム・ユニットとプロセスから構成され，動的に生成されたプロセスの集りによって，目的とした処理が行われる．プログラム・ユニットは，Cの通常の関数と変数，およびプロセス関数（予約語 *process** の付いた関数）の定義の集りで，一つの機能モジュールであり，仮想的にプロセッサを表す．このプログラム・ユニットは，プロセッサにロードされる単位となり，一つのプロセッサには複数のプログラム・ユニットが存在可能である．通常，プログラム・ユニットを越えて名前は参照できない．一方，プロセスは，プロセス関数名およびプロセッサを指定して動的に生成されるが，プロセッサを越えて動くことはできない．プロセス間は，メッセージ転送方式により通信を行うが，とくに，同一プロセッサ内の後述の条件（2.2 節(6)参照）を満たすプロセス間では効率の面から共有変数により通信することを可能としている．

2.2 Concurrent C の並行処理機能

2.1 節で述べた基本的プログラム構造のもとに，以下の6点の言語機能がCに比べて付加拡張されている．構文図およびプログラム例を付録に示す．

（1） 動的プロセス生成 (activate 文)：プロセス生成時に，プロセスが実行開始するプロセス関数とプロセッサをともに指定できる．プロセス生成後，新プロセスのプロセス識別子が，activate 文を実行したプロセスに渡される．プロセッサの指定は整数型の変数または定数により行われ，実行時に生成プロセスの動くプロセッサを指定可能である．

（2） メッセージ転送によるプロセス間通信 (send 文，receive 文)：メッセージの送信 (send 文) と受信 (receive 文) の対応は，互いに相手プロセスをプロセス識別子により指定することや，送信の際にメッセージに付加したタグを受信側で指定することにより，とられる．この対応以外に，非対称的な送受信の対応も許される．すなわち，送信が必ず相手プロセスの指定を必要とするのに対し，受信側は送信プロセスの指定は必要としない．この対応方式により，任意のユーザに資源を与えるライブラリ・ルーチンの作成が容易となる．この送受信の send 文，receive 文による方法は，現実のネットワーク上のメッセージ通信の動作に一致した基本的なものである．

（3） メッセージの複数受信待ち (select 文)：複数受信待ちを行う select 文は，C の switch 文とほぼ同じ構文をもつ（構文図は付録参照）．case ラベルが選択されるのは，論理式が真で，かつ，その受信部の receive 節にメッセージが受信されたときである．いずれかのラベルが選択されれば，複数受信待ちが終了する．

（4） メッセージの送受信時の時間切れ処理の指定 (send 文，select 文で可能)：send 文では，指定された秒数の間に receive 文と同期がとれなければ，時間切れの際の処理を行う．一方，select 文では，指定された秒数の間に，いずれの case ラベルも選択されなかったとき，時間切れ処理が実行される．

（5） プログラム・ユニット間での名前の可視性の指定 (export 宣言，import 宣言)：プログラム・ユニット間での名前の輸出，輸入を行うために2種類の宣言 (export 宣言，import 宣言)* を設けた．これらの宣言が可能な名前は，関数名およびプロセス関数名である．export 宣言で，宣言された名前を，他のプログラム・ユニットで参照することを許し，import 宣言により，指定プログラム・ユニットで export 宣言された名前を，このプログラム・ユニットで参照することを許す．一つのプログラム・ユニットで，複数個 export 宣言できる．これにより，プログラム・ユニットという機能モジュールの中に存在する個々の機能（能動的なプロセス，あるいは，受動的な関数）に対して，直接に要求を出すことが可能である**．

（6） 共有変数機構(モニタと制御式)：疎結合分散型システムでは，プロセッサ間の共有メモリは存在しないので，異プロセッサ間のプロセスの通信は，（2）で述べたメッセージ転送である．しかし，同一プロセッサ内のプロセス間では，Concurrent Pascal[3] により提案された相互排除機構をもつモニタによる共有変数のほうが効率的である．無制限な共有変数の導入は，プログラムの信頼性を低下させるので，Path Pascal[9] のパス式 (path expression) に似た制御式 (control expression) を導入した．これにより，モニタの内部にある関数（モニタ外からよばれ，モニタ内の共有変数の参照を行う関数）の実行順を制御する機能を付加し，さらに，変数を共有するプロセス群のグループ化も可能にした．

---

* 予約語 *process* は，言語Cの type specifier として扱われる．プロセス関数の構文は，付録に示す．

* 予約語 import, export は，C の storage class specifier として扱われる．

** 現在，関数の import, export は，実現されていない．

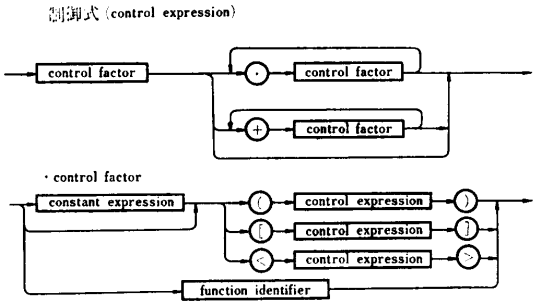


図 1 制御式の構文図
Fig. 1 The syntax chart of a control expression.

表 1 同じ集合を表す制御式とシャフル表現
Table 1 The relation between control expression and shuffle expression.

制御式	シャフル表現
$e_1 \cdot e_2 \cdot \cdots \cdot e_n$	$(e_1 e_2 \cdots e_n)$
$e_1+e_2+\cdots+e_n$	$(e_1+e_2+\cdots+e_n)$
$n(e),(e)$	(e)
$n[e],[e]$	(ee^*)
$n\langle e\rangle$	$\mathrm{MIN}e(\sum_{k=1}^{n}(ee^*)^{\odot k})$
$\langle e\rangle$	$\mathrm{MIN}e(e\odot e^{\circledast})$

$L(\mathrm{MIN}e(\cdots e\cdots))=h(\mathrm{Min}\,(L(\cdots[e]\cdots)))$
ただし,
$h(L)=\{x_1x_2\cdots x_k \mid x_1y_1x_2y_2\cdots x_ky_k\in L,$
$x_1x_2\cdots x_k\in F^*,\ y_1y_2\cdots y_k\in\{[\ ,\ ]\}^*\}$
$\mathrm{Min}\,(L)=\{x\in L \mid {}^{\forall}y\in L:(x=yz, z\neq\lambda)\}$
(F：モニタ内関数名の集合, $[\ ,\ ]\notin F$)

a) 制御式：**図 1** は制御式の構文である．制御式は実行可能なモニタ内関数列の集合と関数列に対する実行条件を表す．等価な集合を表すシャフル表現 (shuffle expression)[14] との対応を**表 1** に示す．実行条件は以下のとおりである．

(1) 制御式全体の表す任意の関数列を，任意数並行して実行することが許される．この限りにおいては，その制御式 ce をもつモニタ内関数の実行列は，シャフル表現 $(ce)^{\circledast}$ の表す関数列集合の要素であるといえる．しかし，(2)実行時の任意の時点において部分制御式 $n(e), n[e], n\langle e\rangle$ から生成された部分を実行中の関数列の数が，それぞれ $n, n, 1$ 未満となるように関数列の実行順が制御される．なお，(3)制御式中に現れないモニタ内関数は任意の時点で実行できる．

b) プロセスのグループ化：プログラム・ユニットの外に輸出 (export) しているプロセス関数名で生成されるプロセスを主プロセス (chief process) とよび，主プロセスとその子孫（プロセスの生成により親子関

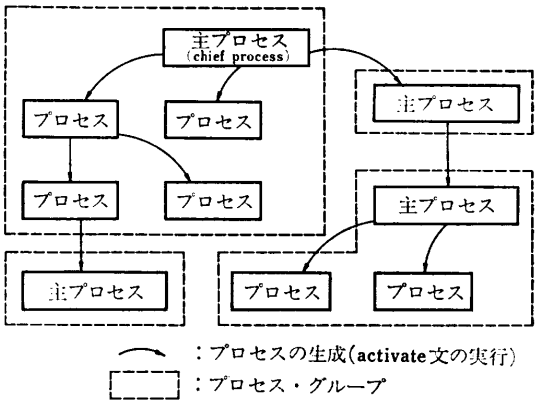


図 2 プロセスの関係とプロセスのグループ化
Fig. 2 Relationship of processes and process grouping.

表 2 追加Cコード表
Table 2 Additional C codes (for concurrent facilities).

ニーモニック	機能
ACT	プロセスを生成する
SND	メッセージを送信する
REC	メッセージを受信する
WAT	複数メッセージの選択的待ちを行う
MIN	最短時間をもつ時間切れ処理を探す
TAS	モニタ内の相互排除のためのP操作
RST	モニタ内の相互排除のためのV操作，かつ，制御式条件のためのV操作
MRS	モニタ内の相互排除のためのV操作，かつ制御式条件不成立によるP操作
LDM	モニタ・データ領域を主記憶にロードする

ソース・プログラム
import process proc 1( ) from "PU_i";
activate pid 1=proc 1 (i,j) on k;
Cコード・プログラム
LAD pid 1　pid 1 のアドレスをスタックに積む.
MST†　スタックに印を付ける.
LOD i　i の値をスタックに積む.
LOD j　j の値をスタックに積む.
LOD k　プロセッサ番号 k をスタックに積む.
LAD $l1$　プログラム・ユニット名のあるアドレスをスタックに積む.
ACT $l2$　プロセスを生成し，その識別子をスタックに積む.
STU　識別子 pid 1 をへ格納する.
$l1$：プログラム・ユニット名 (文字列) "PU_i"
$l2$：プロセス関数名 (文字列) "proc 1"

† …MST で，スタックに印を付けることで，ACT, SND 等の実行時ルーチンの処理後，どこにスタックポインタを戻せばよいかがわかる.

図 3 activate 文の変換例
Fig. 3 The translation of an activate statement.

ソース・プログラム

send (i,j) to pid for (time) ST_1;

Cコード・プログラム

MST		スタックに印を付ける.
LOD	i	iの値をスタックに積む.
LOD	j	jの値をスタックに積む.
LOD	pid	プロセス識別子 pid をスタックに積む.
LDC	0	タグ指名がないので，0をスタックに積む.
LOD	time	時間切れ指定 time の値をスタックに積む.
SND	$l1$	メッセージの送信を行う．時間切れでなければ，$l1$ から実行.
	ST_1 のCコード	時間切れの際の処理
$l1$:	次の文のCコード	

図 4 send 文の変換例

Fig. 4 The translation of a send statement.

ソース・プログラム

receive $(p1,a)$ from pid. request;

Cコード・プログラム

	MST	スタックに印を付ける.
	UJP $l1$	次に，from 以下の評価をするため，$l1$へ飛ぶ.
$l2$:	LAD $p1$	$p1$ のアドレスをスタックに積む.
	LDC 2	$p1$ のサイズ（バイト数）をスタックに積む.
	LAD a	aのアドレスをスタックに積む.
	LDC 1	aのサイズ（バイト数）をスタックに積む.
	UJP $l3$	$l3$ へ飛ぶ.
$l1$:	LOD pid	プロセス識別子 pidを スタックに積む.
	LOD request	タグ request をスタックに積む.
	UJP $l2$	次に，パラメタの評価をするため，$l2$へ飛ぶ.
$l3$:	REC	メッセージの到着を待つ.

図 5 receive 文の変換例

Fig. 5 The translation of a receive statement.

ソース・プログラム

(BE は Boolean Expression, ST は STatement)

```
select {
  case BE1 receive(i) from pid1. ok : ST1; break;
  case BE2 receive(j) from pid2 : ST2; break;
  default for (time1) : ST3; break;
  default for (time2) : ST4; break;
```

Cコード・プログラム

	MST	スタックに印を付ける.
	BE_1 のCコード	BE_1 の評価を行う.
	FJP $l4$	偽なら，次の case 節($l4$)へ飛ぶ.
	UJP $l2$	真なら，from 以下の評価のため $l2$ へ飛ぶ.
$l1$:	LAD i	iのアドレスをスタックに積む.
	LDC 2	iのサイズ（バイト数）をスタックに積む.
	UJP $l3$	$l3$ へ飛ぶ.
$l2$:	LOD pid1	プロセス識別子 pid1 をスタックに積む.
	LOD ok	タグ ok をスタックに積む.
	LAD $l1$	対応する送信が行われたときに，実行を開始する番地（$l1$）をスタックに積む.
	UJP $l4$	次の case 節（$l4$）へ飛ぶ.
$l3$:	REC	メッセージを受信する.
	ST_1 のCコード	受信した後で，文 ST_1 を実行する.
	UJP $l14$	select 文の次へ飛ぶ.
$l4$:	BE_2 のCコード	BE_2 の評価を行う.
	FJP $l12$	偽なら，$l12$ へ飛ぶ.
	UJP $l6$	真なら，from 以下を評価するため，$l6$ へ飛ぶ.
$l5$:	LAD j	jのアドレスをスタックに積む.
	LDC 2	jのサイズ（バイト数）をスタックに積む.
	UJP $l7$	$l7$ へ飛ぶ.
$l6$:	LOD pid2	プロセス識別子 pid2 をスタックに積む.
	LDC 0	タグ指定がないので，0をスタックに積む.
	LAD $l5$	対応する送信が行われたときに，実行を開始する番地（$l5$）をスタックに積む.
	UJP $l12$	$l12$ へ飛ぶ.
$l7$:	REC	メッセージを受信する.
	ST_2 のCコード	受信した後で，文 ST_2 を実行する.
	UJP $l14$	
$l8$:	LOD time1	制限時間 time1 をスタックに積む.
	LAD $l9$	時間切れ時の処理開始番地（$l9$）をスタックに積む.
	MIN	スタックに積まれた2組の時間のうち，短い方の時間とその処理開始番地をスタックに残す.
	UJP $l10$	次の default 節（$l10$）へ飛ぶ.
$l9$:	ST_3 のCコード	time 1 時間切れ時に，文 ST_3 を実行する.
	UJP $l14$	
$l10$:	LOD time2	制限時間 time2 をスタックに積む.
	LAD $l11$	時間切れ時の処理開始番地（$l11$）をスタックに積む.
	MIN	
	UJP $l13$	$l13$ へ飛ぶ.
$l11$:	ST_4 のCコード	time 2 時間切れ時に，文 ST_4 を実行する.
	UJP $l14$	
$l12$:	LDC MAXTIME	時間切れ指定なしのための最大時間 MAXTIME をスタックに積む.
	LAD $l14$	最大時間の時間切れ時の処理開始番地（$l14$）をスタックに積む.
	UJP $l8$	最初の default 節（$l8$）へ飛ぶ.
$l13$:	WAT	複数の受信待ちを登録して，対応する送信を待つ（詳細は，5.3.2節 WAT ルーチン参照）.
$l14$:	次の文のCコード	

図 6 select 文の変換例

Fig. 6 The translation of a select statement.

係が決まる）が一つのグループとなる（**図2**参照）．このグループ内のプロセスがモニタ内の変数を共有する*．

3. 並行処理機能のコンパイル法

Concurrent C のコンパイラ・システムは，プリ・プロセッサ，フェーズⅠ，リンカ，フェーズⅡ，ポスト・プロセッサから構成され，フェーズⅠでは仮想スタック計算機の命令であるCコードを出力し，フェーズⅡでそれを特定の計算機の目的プログラムに変換する（詳細は文献2）参照）．この作成では，すでに作成したCコンパイラ[10]の一部を利用した．追加された機能として，activate 文，send 文，receive 文，select 文，モニタ等があげられる．それぞれについて，フェーズⅠでどのようなCコード列（追加されたCコードは**表2**に示す）を出力しているかを例示する．モニタに関しては，とくにモニタ内関数の実行順を制御する制御式について述べる．

3.1 activate 文

図3に，プロセッサK上に，プログラム・ユニット“PU$_i$”の中の proc 1 というプロセス関数から実行を始めるプロセスを一つ生成し，そのプロセスにパラメタとして i, j の値を渡し，生成したプロセスのプロセス識別子を long 型変数 pid 1 に格納するという activate 文をCコードに変換した例を示す．

3.2 send 文

プロセス識別子が pid のプロセスへ i, j というパラメタを送り，もし time 秒以内に対応する受信が実行されなければ，ST$_1$ の処理をするという send 文の変換例を**図4**に示す．

3.3 receive 文

図5にメッセージの送信側のプロセス識別子が pid で，付加されたタグが request であるメッセージが送られてきた場合に，それを整数型変数 $p1$ と文字型変数 a に格納する receive 文の変換例を示す．

3.4 select 文

図6の select 文は，case 節に書かれた論理式が成立している receive 節のみについて，指定されたプロセス識別子からの指定されたタグを伴うメッセージが送られてくるのを，複数受信待ちするものである．default 節の time 1，time 2 の短いほうの時間以内に，いずれかの受け口のもとにメッセージが送られれば，その receive 節の後の ST$_i$ を実行する．そうでないときは，default 節の後の ST$_j$ を実行する．

3.5 制御式

図7に，プロセス間でのメッセージのバッファリングを行う制御式のCコードへの変換例を示す[15]．制御式 1 (init. [put. get]) は，バッファの初期化 (init) の後に，任意回のバッファへの書込み (put)，読み出し (get) が可能であることを示し，とくに，バッファの個数を1個に制限している．この制御式に対し，**図8**のようにカウンタを設け，モニタ内関数の実行条件をカウンタの状態とカウンタに対する処理に置き換えたものを，Cコードの形で，モニタ内関数のコードの直前に出力し（図7では，例として get についての制御式のCコードを示す），関数の本体の実行前に実行する．

```
ソース・プログラム
monitor (1 (init. [put. get])) {
                制御式
  int  commom;   } 共有変数
               ⋮
  init( ) {……}
  put (in) int in; {……}
  get ( ) {……}
}
```

```
get についての制御式Cコード・プログラム
get : TAS  sem     モニタ内の相互排除のためのP操作.
      LOD  C4
      LDC  0       カウンタ C4 が正（すなわち，get ルーチンの実行可能を示す）なら，l1 へ飛ぶ.
      RGT
      TJP  l1
      MRS  sem     モニタ内の相互排除のためのV操作，かつ制御式条件不成立によるP操作.
      UJP  get
l1 :  LAD  C4
      POD          カウンタ C4 の値を1減らす.
      POP
      LAD  C3
      POI          カウンタ C3 の値を1増やす.
      POP
      [get ルーチンのCコード]
      RST  sem     モニタ内の相互排除のためのV操作，かつ制御式条件のためのV操作.
      RET          関数 get からの復帰
```

図7 モニタの制御式の変換例
Fig. 7 The translation of a control expression.

```
1 ( init. [put. get] )
    |     |     |   |
   C1    C2    C3  C4
```

C_1 : '(…)' 内の表す関数列を実行中の系列の個数.
C_2 : init まで実行した系列の個数.
C_3 : '[…]' 内の表す関数列を反復実行可能な系列の個数.
C_4 : put まで実行した系列の個数.

図8 1 (init. [put. get]) に置かれるカウンタ
Fig. 8 The counters of 1 (init. [put. get]).

* モニタ内の変数のインスタンスは，グループごとに作成される．モニタを含むプログラム・ユニットでは，関数名の輸出を許さないという条件を付加して，不慮の変数の共有を避けている．

4. Concurrent C の実行環境

Concurrent C のプログラム起動時には，システムの生成したプロセスが指定されたプログラム・ユニットの main 関数から実行を開始する．以後，新しいプロセスが動的に生成される．プロセスの生成は，他のプロセッサ上に行うこともできる．すべてのプロセスが終了したときに，プログラムは終了する．

4.1 プログラム・ユニットの構造

プログラム・ユニットは，複数のコンパイル・ユニットをリンクすることにより構成され，実行時，下記の三つの領域に分けられる．

1) データ領域：プログラム・ユニット中で定義された静的変数の領域である．Concurrent C の複数のプロセス間では，これらの変数は共有されないので，プロセスが生成されるたびに，そのプロセスのコードのあるプログラム・ユニットのデータ領域が，新プロセスに新しく割り付けられる*．

2) モニタ・データ領域：モニタ内で定義されたストレージ・クラスが extern の静的変数の領域である．これらの変数は，グループ化されたプロセス群によって共有される．したがって，モニタ・データ領域は，グループの主プロセスが生成されたときに，新しく割り付けられる．

3) コード領域：複数のプロセスが同一コードを共有するほうが，主記憶の利用効率もよいので，コード領域は，再入可能 (reentrant) とした．

これらの3領域は，それぞれ別々のタイミングで主記憶に割り付けられたり，解放されたりする．ここで，Concurrent C の実行時環境として，NOVA 3 のように少ない主記憶しかない場合には，3領域とも主記憶・補助記憶間でスワップ・イン，スワップ・アウトできることが要求される．このためには，各領域がロードされる位置に依存しない (position independent) 性質を満たさなければならない．そこで，各領域内のアドレスの評価は，各領域の先頭からのオフセットの形で行われる．

4.2 プロセスの実行時の状態

実行中のプロセスは，それぞれ**図9**に示すような三つの領域と8種のレジスタ（**表3**参照）をもつ．データ領域の一部である実行時スタックは，必要に応じて伸びるので，データ領域の大きさは可変である．

* 実行時には，動的変数の領域としてのスタックが付け加えられて，プロセスのデータ領域となる．

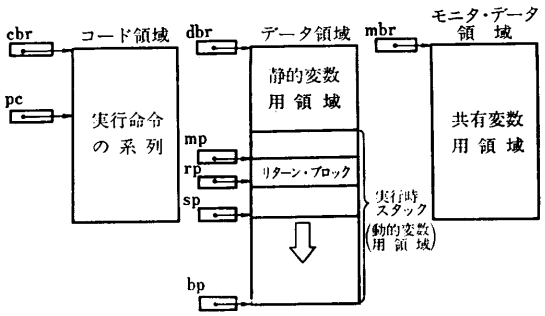


図 9 プロセスの実行時の状況
Fig. 9 Process execution environment.

表 3 レジスタの種類
Table 3 Registers.

レジスタ	機能
プログラム・カウンタ (pc)	次に実行すべきコードを指すポインタ
スタック・ポインタ (sp)	現在の実行時スタックの先頭を指すポインタ
マーク・ポインタ (mp)	実行中の関数の局所的変数の参照に用いられるベース・レジスタ．mp の指す領域の先頭は関数のリターン・ブロック† である
ボトム・ポインタ (bp)	実行時スタック領域の最後（データ領域の最後でもある）を指すポインタ
ブロック・ポインタ (rp)	実行時スタックの先頭に最も近いリターン・ブロックを指すポインタ
データ・ベース・レジスタ (dbr)	プロセスごとの静的変数を参照するのに用いられるベース・レジスタ
モニタ・データ・ベース・レジスタ (mbr)	モニタ内の静的変数を参照するのに用いられるベース・レジスタ
コード・ベース・レジスタ (cbr)	プロセスの動くプログラム・ユニットのコード領域の主記憶上の場所を知るのに用いられるベース・レジスタ

† …関数の復帰に必要な情報を置く領域，関数呼出し前の pc(関数からの戻り番地)，mp，rp，関数のリターン値が置かれる．

スケジューリング・アルゴリズムにより，次に実行されるプロセスがPと決まると，スケジューラが，Pの3領域を，主記憶になければロードする（ロードの際に主記憶が不足すれば，スワッピングアルゴリズムにより，プロセスをスワップ・アウトして，ロードの領域を確保する）．さらに，レジスタ類に実アドレスを設定して，制御をPに渡す．その後，プロセスの実行は次のように行われる．(1)静的変数のロードとストアに対しては，dbr あるいは mbr と，コード中にある変数の論理アドレス ((モニタ・) データ領域の先頭からのオフセット) を加えることにより，実アドレスを計算する（モニタ・データ領域の変数の最上位ビットに1を立てることにより，二つのデータ領域の参照を区別する）．(2)変数のアドレスを評価した値

は，論理アドレスである．したがって，スタックに積まれる値も，変数に格納される値も論理アドレスである．(3)静的変数の参照時には，プロセスの実行中に他のプロセスの領域を壊さないように，参照範囲を検査している．(4)関数の呼出しの際には，cbr とコード中のその関数の先頭アドレス（コード領域の先頭からのオフセット）を加えることにより，実アドレスを計算して呼出しが実行される．このとき，関数のリターン・ブロック中の戻り番地には，コード領域中の論理アドレスが入れられている．(5)ジャンプの際にも，cbr と飛び先の論理アドレスの加算により，実アドレスを計算して飛び先が決定される．(6)算術演算は，Cで行われていたのと同様である．(7)並行処理機能（プロセス生成，メッセージ送受信など）は，スケジューリング，メモリ管理などが必要となるので，カーネルで処理する（5章参照）．(1)～(7)の処理は，それぞれのCコードに対応する実行時ルーチンを呼び出すことにより行われる．

(2),(4)に示したように，実行時にも，コードおよび（モニタ・）データ領域が，スワップ・アウト，スワップ・インによるロード位置の変化の影響を受けないように配慮されている．

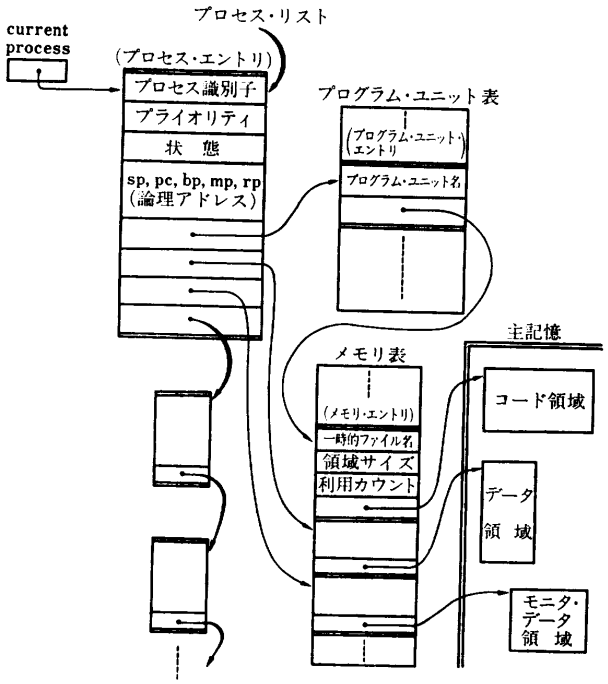


図 10 プロセス実行時に必要なデータ構造
Fig. 10 The data structure in the process execution time.

4.3 プロセス実行時に必要なデータ構造

一つのプロセッサ内には，複数のプロセスが存在し，スケジューラによって制御が切り替えられる．これらの複数のプロセスをスケジューラによって管理するために，プロセス・リスト (**図 10**) がある．その一つのエントリ（プロセス・エントリ）の各項目には，プロセスの状態や，一度スケジューリングで中断されたプロセスを再開するために必要な情報などが入れられている．各プロセスの各領域へは，直接のポインタではなく，メモリ表へのエントリや，プログラム・ユニット表のエントリへのポインタになっている．これにより，メモリ・エントリを見ることで主記憶の管理の大半が可能となる．また，プロセスがどのプログラム・ユニットに属しているかを調べたり，一つのプロセッサ上に同じプログラム・ユニットが二つ以上ロードされるのを避けるため，コードの場合に，メモリ・エントリとの間に，プログラム・ユニット・エントリが設けられている．

5. Concurrent C のカーネル

プロセスの実行時の環境を支えるのが，カーネルである．

5.1 ヘッド・ルーチン

Concurrent C の環境への移行を行う．最初に動き出して欲しい main 関数をもつプログラム・ユニットの名前とその関数へのパラメタを，コマンド・ラインに入力すると，ヘッド・ルーチンが，プロセスを一つ生成して，上記の main 関数から実行を開始させる．

5.2 プロセスの生成

Cコード ACT に対応する実行時ルーチン (ACT ルーチン) のカーネルにおける処理手順を示す．①プロセス・エントリ作成，②データ領域ファイルに，activate 文中のパラメタを付加して，新プロセス用の一時的データ・ファイル作成，③生成を行ったプロセスのモニタ領域を継承（生成されるプロセスが主プロセスなら，実行開始時に，新モニタ領域が割付けられる），④プロセッサに新プロセスのコードのあるプログラム・ユニットがなければ，コード領域をプログラム・ユニット・エントリに登録，すでに存在する場合，利用カウントを1増やす，⑤生成された新プロセスが実行開始できるように，レジスタ類を設定（プロセスの3領域は，スケジューラにより，このプロセスに制御が渡されるまでに，ロードされる），⑥生成されたプロセスの識別子を，生成を行ったプロセスのス

タックの先頭に積む．ただし，他プロセッサ上にプロセスを生成する場合は，生成に必要な制御情報を送信する．プロセス生成後，プロセス識別子が返信される．

5.3 メッセージの送受信

プロセス間のメッセージの送信は SND ルーチン，受信は REC ルーチン，複数メッセージの選択受信待ちは WAT ルーチンで処理される．

5.3.1 送信と受信 (SND, REC ルーチン)

送信が，対応する受信より以前に行われた場合，メッセージと（送受信両方の）プロセス識別子等を送信表 (send table) に登録する．時間切れ処理があれば，処理開始番地とその時間を時間切れ表 (time-up table) に登録する．逆に，受信のほうが先に実行された場合，プロセス識別子等を受信表 (receive table) に登録する．送信と受信の対応がつくまでは，プロセスはサスペンド状態である．対応がついたとき，メッセージを順に receive 文の変数に格納し，プロセスをレディ状態にする．他プロセッサのプロセスへの送信では，相手プロセッサの送信表に登録する．ただし，送信の時間切れ処理がある場合，処理開始番地は自プロセッサの時間切れ表に，時間切れ指定時間は相手プロセッサの時間切れ表に登録する．メッセージの送受信における大部分の処理は，受信プロセスが存在するプロセッサ側で行われる．

5.3.2 複数受信待ち (WAT ルーチン)

スタックに積まれた受信情報 (3.4 節参照) を受信表に登録し，時間切れ時の処理開始番地とその時間を時間切れ表に登録する．いずれかの受信に対応する送信が実行されると，その receive 節のパラメタの評価が行われ，送信されたメッセージが渡される（受信が送信より後に行われた場合と同様である）．このとき，対応しなかった受信情報（複数個ありうる）は，受信表から削除する．

5.3.3 時間切れ時の処理

スケジューリングが行われるたびに，時間切れ表の時間を検査する．時間切れプロセスがあれば，そのプロセスの状態をサスペンドからレディに変え，時間切れ時の処理開始番地から，プロセスを実行させる．

5.4 プロセスのスケジューリング

プロセス・リスト（図 10 参照）からラウンド・ロビン方式でレディ状態のプロセスを選ぶ．選ばれたプロセスのコード，データ，およびモニタ・データ領域が主記憶になければ，補助記憶からロードする（実行時ローディング機能）．スケジューリング・ポイントは，新しく追加された並行処理に関わる実行時ルーチンの中に埋め込まれている．

5.5 メモリの管理

主記憶の制限により，スケジューリングされて実行するプロセスの，三つの領域の一部あるいは全部がロードできない場合，サスペンド状態のプロセスから，主記憶よりスワップ・アウトしていく．必要があれば，レディ状態のプロセスでもスワップ・アウトする（状態は，レディのままである）．スワップ・アウト時の補助記憶上の一時的ファイル名，およびロード時の主記憶上の物理アドレスなどは，メモリ・エントリに入れられている（図 10）．

5.6 他プロセッサとの通信

他プロセッサとの通信は，ACT, SND ルーチンなどで用いられる．プロセッサ間の通信情報の転送は，転送先プロセス名，通信情報の種類，その中身をパラメタにして，送信を行う関数をよぶ．一方，受信済カウンタ（他プロセッサからのメッセージ数を数える）を定期的にみて，通信情報が到着していれば，その解析を行う．送受信は，割込み処理部で処理される．

5.7 カーネルの記述

NOVA 3 と MP/100 のカーネルは，プロセッサ間の通信処理以外，まったく同一である．カーネルは，そのほとんどが，C のプログラムで作成されている．

6. あとがき

本論文では，疎結合分散型システム記述用言語 Concurrent C を提案し，その処理系（コンパイラ・システムとカーネル）の実現について述べた．現在，NOVA 3-MP/100 結合システムで実動している本処理系を用いて，Concurrent C の有効性の評価を行っている．この使用経験から得られる言語自身の評価および実現した処理系の実行効率などの評価については他の機会に譲る．

分散型システムには，さまざまな種類のプロセッサの存在が予想され，それゆえに Concurrent C のコンパイラは，移植性が高くなければならない．今回作成したコンパイラ・システムでは，この点も考慮に入れ，フェーズⅡ以降を異機種ごとに作成すればよい構成になっている．

また，システム記述用言語として Concurrent C は，オペレーティング・システムの支援のない環境での実行環境（カーネル）の実現が要求される．このカーネルは，大半が C で記述され，C のクロス・コンパイ

ラ等を作成すれば，裸の複合プロセッサ・システムにおいても，容易に Concurrent C を用いたシステム・プログラミングが可能である．

謝辞 共同作成者である市井博雄氏（現在，日本電気），小田垣秀雄氏（現在，住友電工），香西省治氏，ならびにハードウェアに関してご協力いただいた竹村治雄氏に感謝いたします．

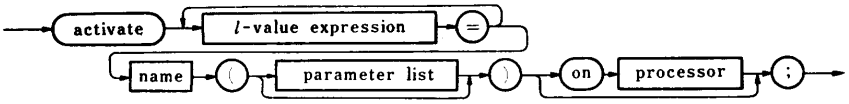
参考文献

1) Ambler, A. L.: GYPSY: A Language for Specification and Implementation of Verifiable Programs, Proc. ACM Conf. on Language Design for Reliable Software, pp. 1-10 (Mar. 1977).
2) 安藤，辻野，荒木，都倉：分散型システム記述用言語 Concurrent C の処理系の試作，情報処理学会ソフトウェア工学研究会資料，22-13 (Feb. 1982).
3) Brinch H. P.: The Architecture of Concurrent Programs, Prentice-Hall, New Jersey (1977).
4) Brinch H. P.: Distributed Processes: A Concurrent Programming Concept, *Comm. ACM*, Vol. 21, No. 11, pp. 934-941 (Nov. 1978).
5) DoD: *Reference Manual for the ADA Programming Language*, United States Department of Defense (Jul. 1980).
6) Feldman, J. A.: High Level Programming for Distributed Computing, *Comm. ACM*, Vol. 22, No. 6, pp. 353-368 (Jun. 1979).
7) Hoare, C. A. R.: Communicating Sequential Processes, *Comm. ACM*, Vol. 21, No. 8, pp. 666-677 (Aug. 1978).
8) Kernighan, B. W. and Ritchie, D. M.: The C Programming Language, Prentice-Hall, New Jersey (1978).
9) Kolstad, R. B. and Campbell, R. H.: Path Pascal User Manual, *SIGPLAN NOTICES*, Vol. 15, No. 9, pp. 15-24 (Sep. 1980).
10) 黒田，辻野，萩原，荒木，都倉：システム記述用言語Cのポータブル・コンパイラの作成，情報処理学会論文誌，Vol. 21, No. 6, pp. 461-468 (Nov. 1980).
11) Schutz, H. A.: On the Design of a Language for Programming Real-Time Concurrent Processes, *IEEE Trans. Softw. Eng.*, Vol. SE-5, No. 3, pp. 248-255 (May 1979).
12) 辻野，安藤，荒木，都倉：分散型システム記述用言語 Concurrent C，信学技報（電子計算機），EC 81-13 (Jun 1981).
13) Tsujino, Y., Ando, M., Araki, T. and Tokura, N.: Concurrent C: The Programming Language for Distributed Multiprocessor Systems, *Programming Languages Group Memo*, No. 81-04, Dept. of Information and Computer Sciences, Osaka Univ. (Sep. 1981).
14) 辻野，荒木，都倉：シャフルを付け加えた正規表現，信学論（D），Vol. J64-D, No. 6, pp. 541-542 (Jun. 1981).
15) 辻野，香西，荒木，都倉：Concurrent C のモニタにおける制御式について，情報処理学会第 24 回全国大会，7L-6 (Mar. 1982).
16) Welsh, J. and Bustard, D. W.: Pascal-Plus-Another Language for Modular Multiprogramming, *Softw. Pract. Exper.*, Vol. 9, No. 11, pp. 947-957 (Nov. 1979).
17) Wirth, N.: Modula: A Language for Modular Multiprogramming, *Softw. Pract. Exper.*, Vol. 7, No. 1, pp. 3-35 (Jan.-Feb. 1977).

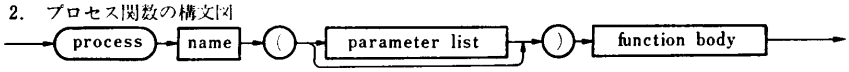
（昭和 57 年 3 月 5 日受付）
（昭和 57 年 6 月 15 日採録）

付録 A 言語Cに付加拡張された並行処理機能の構文図

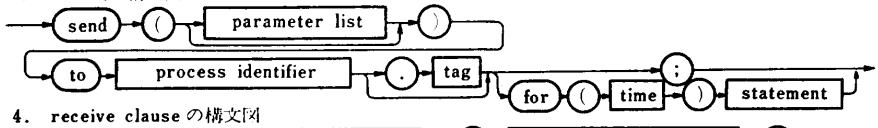
1. activate 文の構文図



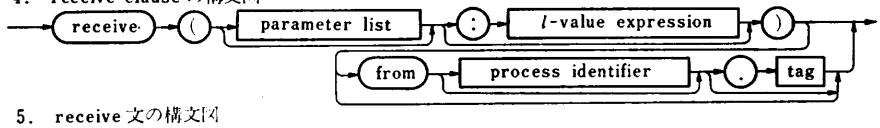
2. プロセス関数の構文図



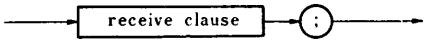
3. send 文の構文図



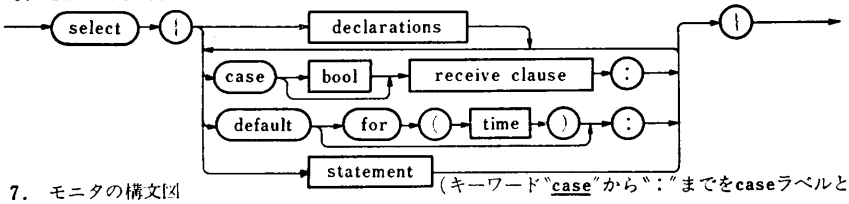
4. receive clause の構文図



5. receive 文の構文図

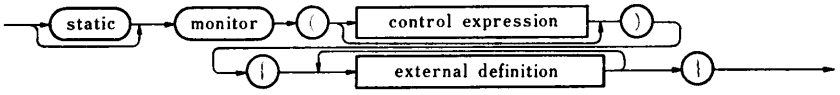


6. select 文の構文図

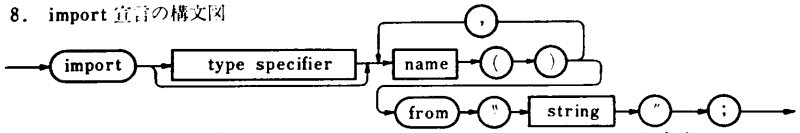


(キーワード"case"から":"までをcaseラベルと呼ぶ.)

7. モニタの構文図

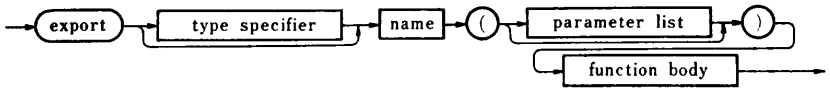


8. import 宣言の構文図



(stringはプログラム・ユニット名.)

9. export 宣言の構文図



付録 B プログラム例：リングバッファ

```
Sending processes
        ⋮
    send (data) to bufferprocess;
        ⋮
Receiving processes
        ⋮
    send ( ) to bufferprocess;
    receive (data) from bufferprocess;
        ⋮
process ringbuffer ( )
{   char buffer [SIZE];
    int in=0, out=0;
    long receiver /* process identifier */;
    for ( ; ; )
      select {
        case in !=out+SIZE
            receive (buffer [in++%SIZE]):
          break;
        case in !=out
            receive( : receiver) :
          send (buffer [out++% SIZE]) to receiver;
      }
}
```

(a) プロセスを用いたリングバッファ

```
Sending processes
        ⋮
    sen(data);
        ⋮
Receiving processes
        ⋮
    data =rec ( );
        ⋮
monitor (SIZE (sen. rec))
{   char buffer [SIZE];
    int    in=0, out=0;
  sen ( c )
    char c;
  { buffer [in++% SIZE]=c; }
  char rec( )
    { return (buffer [out++% SIZE]); }
}
```

(b) モニタを用いたリングバッファ