言語Cのライブラリ形式によるコンカレント機能の実現†

辻野嘉宏†† 安藤誠†††*
荒木俊郎†† 都倉信樹††

オペレーティングシステム (OS) や計算機のハードウェアの上に特殊な支援のない環境では，コンカレント処理の可能なプログラミング言語の実現は一般に困難である．筆者らは，複数のプロセスが単一計算機上で仮想的な通信回線を介して通信するコンカレント処理システムを設計した．このシステムは，当研究室で作成したポータブルCコンパイラ (ncc) の実現方法の利用とCのライブラリ形式によるコンカレント処理用基本オペレーションの組込みにより，OSやハードウェアに依存せず，コンパイラを変更することなく容易に実現され，ncc の稼動している計算機へ移植可能である．さらに，この機能を用いて実現した UNIX のパイプライン機能について述べる．

1. まえがき

近年，数々のコンカレント処理記述可能なプログラミング言語が提案され，また，コンカレント処理に必要な機能が研究されている[1)-5)]．しかし，コンカレント機能をもつ言語を導入するには，オペレーティングシステム (OS) や計算機のハードウェアの上に特殊な支援が必要となり，これらのない環境では実現が困難であった．また，そのような言語のコンパイラの作成もかなりの手間を要する．

そこで，筆者らは，既存の言語 (C[6)]) の中に，ライブラリの形式でコンカレント処理を組み込むことによって，コンカレント処理プログラミングを可能とするシステムを設計し，実現した．その使用例として，UNIX のもつパイプライン機能[7)]を作成した．

このコンカレント処理機能システムは，次のような特徴をもつ．実現上の特徴として，

(i) OS, 計算機のハードウェアに独立な形で実現可能である．

(ii) 言語Cのコンパイラ（当研究室で作成したポータブルCコンパイラ[8)] (以下，ncc と略す)）には，まったく変更を加える必要がない．

(iii) 簡単に実現できる．

機能上の特徴としては，

(iv) 単一計算機上のコンカレント処理を想定している．

(v) プロセスは，動的に生成・消滅し，生成によってプロセス間には親子関係が生ずる．

(vi) プロセス群は，仮想的な通信回線を介して通信する．

コンカレント処理プログラミングの下では，複数のプロセスが存在するので，それらのプロセスの生成や制御の切替えの実現に複雑な処理が要求される．

この部分の実現法は，ncc の対象とした仮想スタック計算機の実現法等と深く関連している．そこで，2章では ncc の NOVA 3 上での実現について述べ，3章では設計したコンカレント機能を説明し，4章ではその機能の実現方法について述べる．また，5章では実現したコンカレント機能の使用例の一つとして，パイプライン機能を提供するシステムについて述べる．

2. ncc の NOVA 3 上での実現

ncc は高い移植性をもつCコンパイラであり，計算機に独立な中間言語（Cコード）を生成するフェーズIと，Cコードを実現の計算機の目的プログラムに変換するフェーズII（各計算機ごとに異なる）に分けられる．このCコードは，2.1 節で述べる仮想機械によって受理される．

2.1 仮想機械

Cコードを受理する仮想機械のアーキテクチャを述べる．この機械はスタック計算機であり，1個の実行時スタックと以下に述べる6種のレジスタをもつ．実行時スタック内には，実行時に必要な局所的変数領域や作業領域がある．**図1**に仮想機械の構成図を示す．

1) 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．

2) マークポインタ (mp): 実行中の関数の局所的

† The Concurrent Facility Implemented by C Library by Yoshihiro Tsujino, Makoto Ando, 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)技術本部

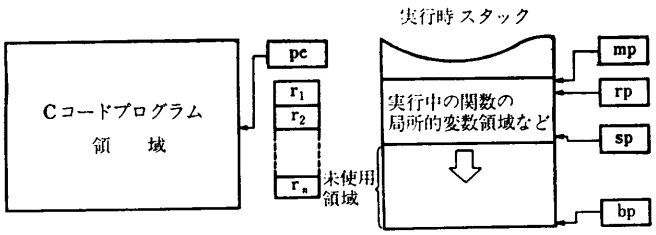

図 1 仮想機械の構成図
Fig. 1 C code machine architecture.

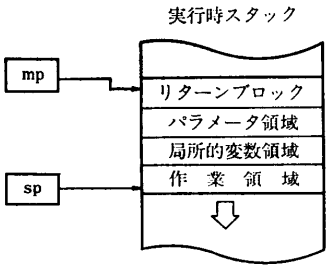

図 2 mp の指す領域の構成
Fig. 2 The area pointed by mp.

変数を参照するときに，ベースレジスタとして使われる．mp の指す領域の先頭は，常にリターンブロック（後述）である．

3) ジェネラルレジスタ：レジスタ変数が割り付けられる．一般には n 個あると考えられる（$r_1, \cdots, r_n$）．

4) プログラムカウンタ（pc）：次に実行すべきCコードを指すポインタ．

5) ボトムポインタ（bp）：実行時スタック領域の終りを指すポインタ．

6) ブロックポインタ（rp）*：実行時スタックの先頭に最も近い位置にあるリターンブロックを指すポインタ．

mp の指す領域は，関数の実行環境を示し，仮想機械の実行上重要な部分である．この領域は，**図 2** に示すように，四つの部分に分けられている．

1) リターンブロック：関数の復帰に必要な情報を置く．たとえば，関数呼び出し前の pc（関数からの戻り番地），mp，rp などの情報である．

2) パラメータ領域：実パラメータを置く．

3) 局所的変数領域：関数内でとられた局所的変数を置く．

4) 作業領域：式を評価する際の中間結果等を置く．

* 通常は，mp と rp は同じリターンブロックを指すが，関数呼び出しを行う場合，リターンブロックを実行時スタックに置き（rp だけが変更される），実パラメータの計算を行い（局所的変数の参照のため mp が使われる），その後に関数呼び出しが行われる（mp も更新され，rp と同じリターンブロックを指す）．

2.2 関数呼び出しの処理

関数の呼び出しと復帰が，仮想スタック計算機上でどのように行われるかを，例と対応するCコードを挙げて示す（**図 3**）．

- MST コード：リターンブロックを実行時スタック上に置き，リターンブロック内に rp を待避して，rp を更新する．
- CFS コード：関数を呼び出す前の実行環境（mp, pc）をリターンブロック内に退避し，mp を更新して，関数を呼び出す．
- RET コード・退避した実行環境（mp, pc, rp, sp）を復活させ，関数の実行結果であるリターン値を，スタックの先頭に置く．実行環境の復活により，関数の呼び出し側に制御が戻る．

2.3 NOVA 3 上での仮想機械の実現法

2.1 節で述べた仮想機械を NOVA 3 では，次のように実現した．

- レジスタ類：sp, pc, rp は NOVA 3 のハードウェアのレジスタを用い，mp, bp は主記憶上に設けた（ジェネラルレジスタはレジスタ数の制限等から実現されていない．）
- リターンブロックに退避される情報：前述した

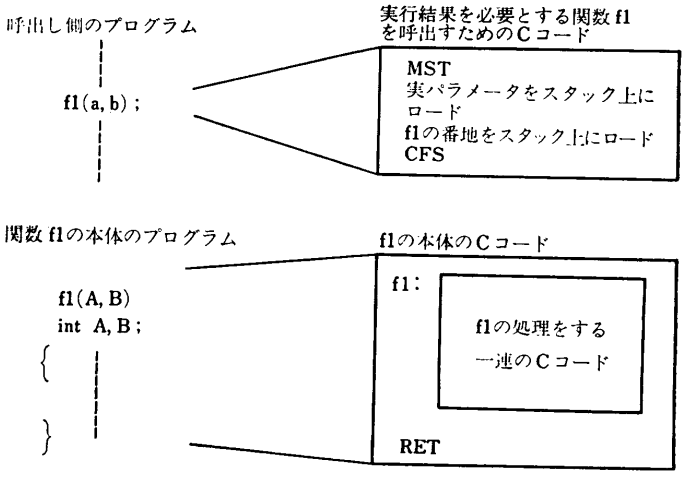

図 3 関数呼出しの実際例とそのCコード
Fig. 3 A function call and corresponding C code.

ように関数呼び出し前の mp, pc, rp や関数のリターン値等を置く.

- Cコードの実行：フェーズIの出力であるCコードは，フェーズIIにより NOVA 3 の命令の系列に変換される．とくに機能の大きなCコードの実行時ルーチンは，目的プログラムを小さくするため，サブルーチンコールの形で実行する．各Cコードの実行時ルーチンは，純手続き(pure procedure)として実現した．

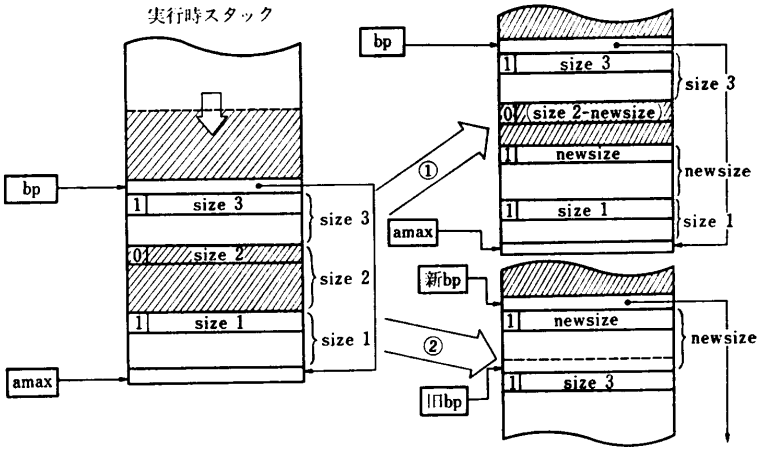

図 4　動的記憶領域割付けの様子．(斜線部が，未使用スタック領域である)
Fig. 4　The dynamic allocation area on the runtime stack.

2.4 システムライブラリ

ncc には，いくつかのシステムライブラリが用意されており，すべてのライブラリルーチンは純手続きになっているが，それらのなかに動的記憶領域割付けライブラリルーチンがある．

これは，利用者からの領域割付け要求 (alloc) に対し，未使用スタック領域の底からその領域を確保し，逆に領域解放要求 (free) が出されると，その領域を再び未使用スタック領域の一部に戻すという機構である．実際には，次のようにして実現している．

- 割り付けられた領域：alloc によって割り付けられた領域は，bp の指す番地と bp の指す番地の指す番地（そのプログラムが利用できる最大番地：以下 amax と略す）との間にとられている．bp の値は，後述のように alloc によって変化するが，bp の指す内容すなわち amax は変化しない．
- 割り付けられた領域のデータ構造：先頭の1語に，領域のサイズと領域の使用中を示す1ビットのフラグをもち，その後に要求されたサイズ分の連続領域が続く．
- 領域割付け：**図4**に示すような状態で，alloc 要求が出されると，bp の指す番地から amax までの未使用領域(フラグが0のところ)のなかで，要求領域の確保できる領域を探す．そのうち，いちばん amax に近い領域から要求領域をとる(図4中の①)．もし，bp から amax までの未使用領域では要求サイズ分の領域が確保できないならば，bp の値をそのサイズ分だけ減らして，領域を確保する（図4中の②)．
- 領域解放：free 要求は，解放したい領域の先頭番地をパラメータとして渡す．この番地とすでに割り付けられた領域の先頭番地とを順に比べて，一致すれば，その解放要求が正当なものであるとして，その領域のフラグを0に変更する*．したがって，free 要求した番地が，bp よりも小さかったり，amax より大きければ，解放要求は不当なものとなる．

3. コンカレント機能の設計

この章では2章で述べた ncc の実現の下で，OS に依存せず容易に実現できるコンカレント機能の設計について述べる．この機能は，一つの計算機上にある複数のプロセスが，設定された通信回線を介して交信し合う機能を支援するものであり，これを行うのに必要な基本オペレーションとして，プロセスの起動・終了・中断に関するものと，プロセス間の通信に関するものが用意されている．Cで書かれたプログラムの中に，このコンカレント機能を導入する方法として，上記の基本オペレーションを関数の形で用意し，それらの関数を呼び出す方式を用いた．

3.1 プロセスの起動・終了・中断

- プロセスの起動：activate (function 名, …)という基本オペレーションは，プロセスを一つ生成し，function 名で指定される関数の最初からそのプロセスの実行が開始される．この起動方法から，プロセス間に親子関係が生ずる．すなわち，activate を発したプロセスは，それにより起動さ

* 解放領域が連続した場合，alloc の際にそれらをまとめて一つの未使用領域とみなす．

れたプロセスの親となる．プロセス間の親子関係だけがプロセスを指定する方法なので，特定のプロセス間で通信回線を設定するには，それらの先祖のプロセスがあらかじめ通信回線を作っておき，その通信回線用送受信識別子 (Send/Receive communication descriptor：以下 S/R cd と略す) をプロセス起動時にパラメータとして子プロセスへ伝達することが必要である．このように，activate 時のパラメータとして，起動したプロセスにさまざまな情報を渡すことも可能である．

- プロセスの終了：プロセスは，terminate ( ) という基本オペレーションを発することにより，陽に終了することができる．プロセスはこれ以外に，通常の関数の復帰によっても終了できる．起動で述べたプロセスの親子関係により，あるプロセスが終了したときにその子孫のプロセス群すべてが強制的に終了させられるという方式を用いる．
- プロセスの中断：プロセスが実行を中断して，制御をいったん放棄する基本オペレーションとして wait ( ) がある．

3.2 プロセス間の通信

- 通信回線の作成：setline(*v*, …) という基本オペレーションにより，1本の通信回線が作られる．このとき，通信回線に対するいくつかの情報をパラメータとして渡すことができ，配列 *v* には通信回線の送信用ならびに受信用識別子が戻される (それぞれ *v*[0], *v*[1] に戻される)．この setline は，起動のところで述べた先祖のプロセスの通信回線作成の際に用いられる．setline では，通信回線の利用プロセスに関して，まだいっさい指定されていない．
- 通信回線を用いた通信方法：通常のファイルに対する標準入出力手続きを用いて，通信回線からデータを読み取ったり，データを書き出したりする．この場合，ファイルを指定するために用いる入出力ファイル識別子のかわりに，S/R cd が用いられる．この S/R cd は，一般のファイル識別子と区別できる必要がある*．
- 共有変数：言語Cの変数の有効範囲の規則により，プロセス間で共有変数をもつことも可能である．しかし，コンパイル単位内の変数を static 属性とすることにより，その単位を超えたプロセス間では，変数を共有できないようにもできる．

* ファイル識別子と S/R cd は，ともに整数型である．

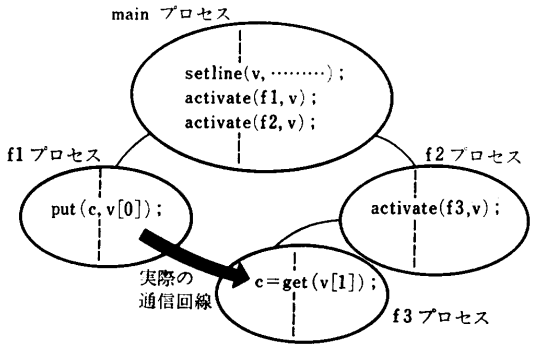

図 5 プロセスの親子関係と通信
Fig. 5 Inter-process communication.

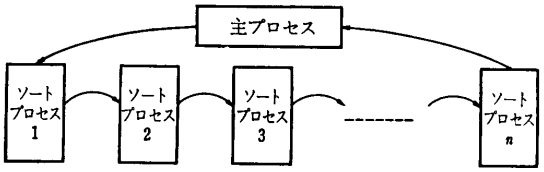

図 6 ソーティングプログラムの構成
Fig. 6 A sorting program.

3.3 プロセスの親子関係と通信

プロセスの親子関係と通信の例を概念的に示すと，**図5**のようになる．これは，"main という親プロセスが，あらかじめ子孫 f1, f3 のために通信回線を setline を用いて作っておき，プロセスを activate するときに，パラメータとして，その通信回線の S/R cd を渡していく，実際に通信回線を使用するプロセスは，put・get の際に，渡された S/R cd を用いて通信する" というモデルである．

3.4 設計したコンカレント機能を用いた記述例

3.1, 3.2 節で述べた基本オペレーションを用いてコンカレントに処理を行うプログラム例として，"*n* 個の入力データをソートしその結果を出力する" ものを考える．**図6**にその構成を示す．

主プロセスは，入力ファイルから *n* 個のデータを取り出し順にソートプロセス1に送る．すべてのデータを送り終えると，EOF を送る．次に *n* 個のデータをソートプロセス *n* から受け取って順に出力する．この出力結果は降順に並ぶ．

ソートプロセスは，すべて同じリエントラントなプログラムコードを共有する．いま，ソートプロセス *l* を考えると，ソートプロセス *l*−1 からデータを受け取って，内部に貯えられたデータの値と比較して大きいほうをソートプロセス *l*+1 に送り，小さいほうは内部に貯えておく．最初，内部に貯えられたデータが

```
main( )
 {
  extern  sort( ) ;
  int     fdline, v[2], nextfd, i,
          fdin = open("inputfile",READ),
          fdout = open("outputfile",WRITE) ;

  setline(v,SIZE) ;
  for(fdline = v[1] ; (i = get(fdin))! = EOF ; put(i,fdline))
   {
      nextfd = v[0] ;
      setline(v,SIZE) ;
      activate(sort,nextfd,v[1]) ;
   }
  put(EOF,fdline) ;
  while((i = get(v[0]))! = EOF) put(i,fdout) ;
  close(fdin) ;
  close(fdout) ;
 }

sort(sdin,sdout)
  int     sdin, sdout ;
 {
  int     i, j ;

  for(i = get(sdin) ; (j = get(sdin))! = EOF ; )
      put(i>j ? (i,, i = j) : j, sdout) ;
  put(i,sdout) ;
  put(EOF,sdout) ;
 }
```

図 7 設計したコンカレント機能を用いてソーティングをプログラムした例

Fig. 7 A sorting program realized by using the concurrent facility.

注 $(e_1,,e_1)$: 式 e_1, 式 e_2 の順に評価し，結果の値は式 e_1 の値とする．

ないので，一つ先読みする．EOF のデータが来ると，内部データと EOF を続けて送って，プロセスは終了する．

以上のモデルは，**図7**のように簡潔にプログラム化できる．すべてのソートプロセスは，主プロセスの発した activate (sort, …) によって起動され，関数 sort のコードを共有している．

4. コンカレント機能の実現

3章で示した機能を実現するのに必要となる処理を，プロセス・スワッパ，プロセス・イニシャライザ，スケジューラ，通信回線に分けて示す．この実現には，OS・計算機のハードウェアに依存しないこと，ncc の変更は行わないこと，簡単にできることを目標に行った．とくに，プロセスの切替えと起動には，関数呼び出しと復帰の機構を利用した．また，複数のプロセスそれぞれに実行時スタックをもたすために，動的記憶領域割付けライブラリルーチンを用いた．

4.1 プロセス・スワッパ

プロセスが実行を中断して，他のプロセスに制御を渡す必要のある時点，すなわち，プロセスの切替えを行う個所をスケジューリングポイントとよぶ．スワッパは，このスケジューリングポイントで，他のプロセスに制御を移す処理をする．スケジューリングポイントは，具体的には，プロセスが実行を続けられなくなった時点（主記憶上のバッファの形で実現された通信回線（4.4 節参照）において，空バッファからデータを取り出そうとしたり，一杯のバッファにデータを入力しようとした場合，また，プロセスの実行終了の場合）や，陽に実行を中断しようとした時点（wait( ) という基本オペレーションを出した場合）が相当する．

スワッパの処理は，関数の呼び出しと復帰を用いて，次のように実現できる．まず，スケジューリングポイントで，プロセスはスワッパを呼び出す．スワッパの最初で，呼び出したプロセスの実行環境に関する情報（必要な情報は，mp, bp, rp の3語）を，そのプロセスの実行時スタックに退避し，退避後のスタックポインタをスワッパのもつプロセス表に登録する（**図8**）．

次にスケジューラを用いて，次に制御を渡すプロセスを決定し，プロセス表からそのプロセスのスタックポインタを取り出して，以前に退避してあった3語の情報を復活する．

最後に，スワッパから復帰すると，新しく制御を渡されたプロセスは，以前にスワッパを呼び出したところから再開することになる．

以上のようにして，プロセスの中断と再開がスワッパの呼び出しと復帰により実現される（**図9**）．

基本オペレーション wait( ) は，実際には，スワッパの呼び出しを行っている．

4.2 プロセス・イニシャライザ

プロセスを起動するときに，イニシャライザによ

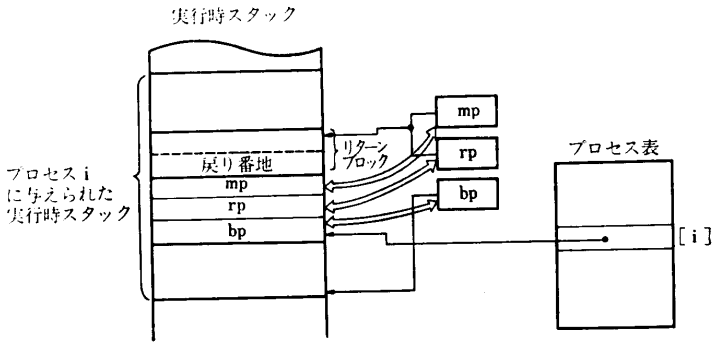

図 8 プロセス切替え時の実行時スタックにある退避情報とプロセス表

Fig. 8 The data structure for process switching time.

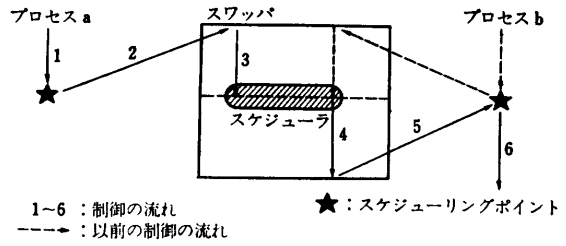

図 9 プロセスの切替え
Fig. 9 Process switching.

り，プロセスの実行に必要な実行時スタックを動的記憶領域割付けライブラリ中の alloc を用いて確保する．プロセスの初期化を行うプロセスは，activate を発したプロセス（親プロセス）であり，この親プロセスの実行時スタック内の動的割付け領域から新しく生成されるプロセス（子プロセス）の実行時スタックがとられる（**図 10**）．親プロセスが終了すれば，その実行時スタック領域を解放するので，子プロセスの実行時スタックも自動的に解放される．したがって，親が終了すれば，すべての子孫のプロセスが終了することになる．

実行時スタックの領域を確保した時点で，そのスタックを**図 11** に示すように初期化する．これは，スワッパにより初めてプロセスに制御が渡されたとき，そのプロセスの実行する関数の最初から実行できるよう，関数の実行に入る前で中断していたかのように設定が行われる．また，プロセスの起動時に渡したいパラメータ群も実行時スタック上に置かれる．

プロセスを起動するには，activate (function 名, 起動プロセスに渡すパラメータ群) を用いる．

4.3 スケジューラ

スケジューラは，あるプロセスが実行を中断したとき，他のコンカレント機能支援ルーチンのデータ構造，たとえば，プロセス表や通信用バッファの状況(データの個数)などを参照して，次の制御を渡すプロセスを選ぶ．一方，他のコンカレント機能支援ルーチンは，スケジューラのもつデータ構造の参照が必要ないように作成されている．このため，具体的に最適なスケジューリングアルゴリズムをもつスケジューラを，必要なデータ構造の参照だけにより，自由に作成し，システムに付加することができる．

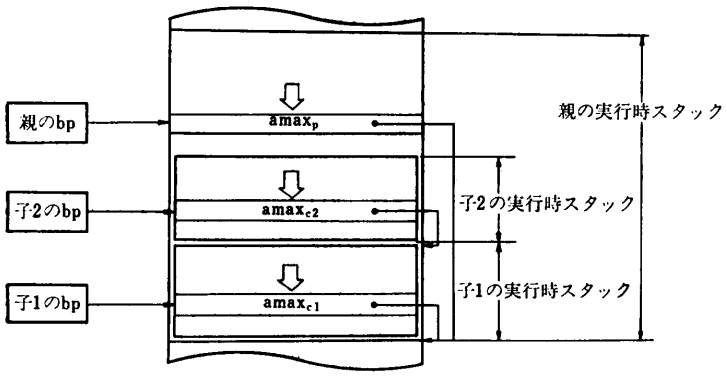

図 10 初期化の際の実行時スタックの割付け
Fig. 10 The initial memory allocation for the runtime stack.

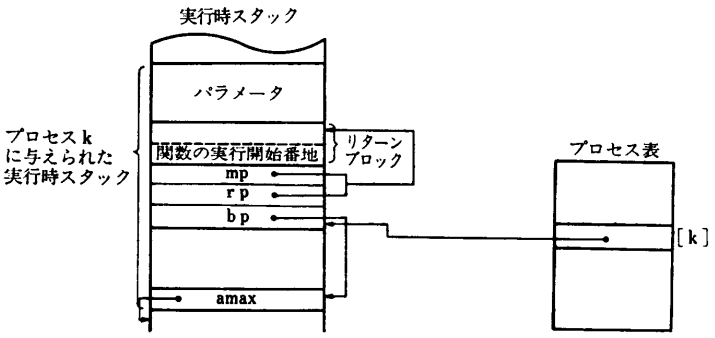

図 11 プロセス・イニシャライザにより設定される実行時スタックの情報とプロセス表
Fig. 11 The data structure on process initialization time.

4.4 通信回線

通信回線は，主記憶上の固定領域を通信用バッファとして利用することにより実現する．このバッファに対してデータのやり取りを行う．実際に通信用バッファと入出力を行うのは，通信用基本オペレーションを支援するライブラリルーチンであり，4.1 節で述べたスケジューリングポイントは，プロセスの使うこのオペレーションの中に隠されていて，ユーザプログラムからは，プロセスがコンカレントに動いているように見える．通信用回線を設定する際に，バッファの大きさを指定でき，パラメータとして渡す．すなわち，

setline (*v*, バッファの大きさ)

となる．

このバッファ領域は，setline を発したプロセスの実行時スタックから割り付けられるので，実際に通信を行うプロセス群の先祖のプロセスの実行時スタック

にとられていることになる．したがって，プロセス・イニシャライザで述べた親プロセスの終了による領域の解放の問題が，この場合も生じて，親が終了すれば，子孫は通信ができなくなる．そこで，親が終了すれば，その子孫はすべて終了するという立場をとる．

5. パイプライン機能の実現

複数の小さなソフトウェアツールを組み合わせて，複雑な処理を行う手法であるパイプライン機能を，4章で述べたコンカレント機能の一使用例として示す．

パイプラインとは，複数個のコマンドを縦棒でつなぎ合わせたものをいい，縦棒の左のコマンドの標準出力が右のコマンドの標準入力となり，各コマンドはそれぞれ一つのプロセスとしてコンカレントに動く．各コマンドにはパラメータを与えることも可能であり，パイプラインに対する入出力ファイルも指定できる．パイプラインの利用者は，コマンド間の作業用ファイルを作成することなく，さまざまなコマンドをつなぎ合わせて，かなり複雑な処理を手軽に構成できる．

このパイプライン機能は，4章で実現したコンカレント機能に，パイプラインに最適なスケジューラの作成を行うことにより実現できる．

5.1 パイプラインの構成

全体の構成は**図 12** に示すように大きく二つに分けられる．一つは，最初に実行されるパイプライン専用のコマンド処理プログラムである．これは，利用者からの入力コマンドを読み込んで，入出力ファイル名とコマンド（すなわち，パイプライン上に並ぶプロセス．これをパイプラインプロセスとよぶ）の順序を調べ，また，各コマンドに対するパラメータ等の情報も調べる．さらに，パイプラインプロセス群，コンカレント機能処理ルーチン，入出力プロセス，使用ライブラリの各相対形式プログラムより，一つの実行形式プログラムを作る．ここで，コンカレント機能処理ルーチンとは，3章で述べたコンカレント用基本オペレーションの実現ルーチンと，それに必要なプロセス・イニシャライザ，プロセス・スワッパ，スケジューラの各ルーチンである．また，入力プロセスとは，入力ファイルから最初のコマンドに対応するプロセスの受信用バッファへのデータ転送を行う専用プロセスである．一方，出力プロセスは，最後のコマンドに対応するプロセスの送信用バッファから出力ファイルへのデータ転送を行う専用プロセスである．この二つのプロセスにより，各コマンドに対応するプロセスは，通信用バッファでの入出力のみを標準入出力とすればよい．

図 12 のもう一つの構成要素は，パイプライン機能を処理するコンカレントプログラムを実行する部分である．前段階のコマンド処理プログラムから，実行形式プログラムにプログラムスワップして，パイプラインプロセス群の主プロセスの最初に制御を渡す．コンカレントプログラムの実行部は，主プロセスと各パイプラインプロセスから成っている．

主プロセスは，次のような仕事を行う．

- コマンドの順に応じて通信回線を設定する．
- 設定により得られた S/R cd ならびにコマンド入力時に与えられたパイプラインプロセスへのパラメータ等を，activate へのパラメータの形で渡して，パイプラインプロセスや入出力プロセスを起動する．

以上により，コマンドによって与えられた特定の二つのプロセス間の通信回線が設定される．

各パイプラインプロセスは，パラメータとして与えられた S/R cd を標準入出力ファイルディスクリプタとして，標準入出力手続きを用いて，バッファを介しデータのやり取りをする．

なお，各プロセスの実行コードはCで書かれている．

5.2 パイプライン機能の実現法

この節では，細部の実現法について述べる．

通信回線は，リングバッファで実現されている．

スケジューラは，4.1 節で述べたスケジューリングポイントと，主プロセスがすべての通信回線を設定完了したときに出す wait( ) オペレーションの時点とにおいて，次のようなスケジューリングアルゴリズム

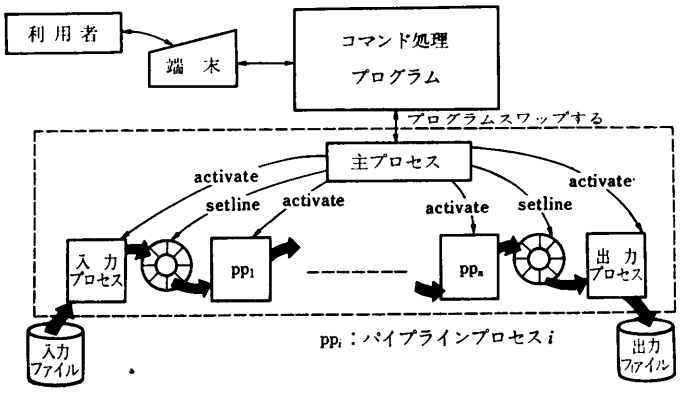

図 12 パイプライン機能の構成
Fig. 12 The pipeline system.

を用いている.

- 空バッファからデータを取り出そうとした時点：そのバッファにデータを置くプロセスに制御を渡す.
- 一杯のバッファに出力しようとした時点：そのバッファからデータを取り出すプロセスに制御を渡す.
- プロセスの終了点：いったん，親プロセスに戻し，ひき続き，スケジューラがプロセス表から終了したプロセスを消し，その実行時スタックを解放する．さらに，出力の終了記号 (EOF) を，そのプロセスの出力バッファに置く．この後，動くことのできるプロセスのいずれかに制御を渡すことができる.
- 次に制御を渡そうとしたプロセスがすでに終了していたとき：未だ終了していない最も入力プロセスに近いプロセスに制御を渡す.
- 主プロセスがすべての通信回線を設定完了した時点：入力プロセスに制御を渡す.

上記のスケジューリングアルゴリズムを採用した理由は，パイプラインのデータの流れが一次元的（入力プロセス側から出力プロセス側へ）なので，パイプラインプロセスの中断原因を除く方向へ制御を渡すことが可能だからである.

5.3 実現したパイプライン機能の評価

4章で述べたコンカレント処理を行う上での諸機能は，ほとんど言語Cで実現された（プロセスの切替え・生成部分では，Cの関数呼び出しと復帰を利用するために，38語のアセンブリコードが必要であったが，約7人日で実現可能であった）．そのため，具体的なパイプラインの実現に必要な部分は，コマンド処理プログラムとパイプラインに最適なスケジューラの作成のみであり，容易に実現できた（約7人日）.

このパイプライン機能，ならびに，各プロセスの実行するコードは，OSに依存せず，上記のように大半がCで書かれているため，nccの稼動している計算機へ移植可能てある.

実現したパイプライン機能とUNIXの機能との比較を以下にあげる.

1) 入力形式は，本方式もUNIXと同様であり，操作上の異和感はない.

2) 縦棒でつながれたコマンドは，UNIXの場合，個々が一つの実行形式プログラムであるのに対し，本方式では，コマンドは相対形式のプログラムである．したがって，入力に並ぶコマンドをリンクして，一つの実行形式プログラムを作成する必要があるので，コマンド実行開始までに時間がかかる.

3) UNIXでは，コマンド間はパイプ用の作業ファイルにより結合されているが，本方式では，主記憶上の一時的バッファで結合されている．したがって，コマンド間のデータ転送は本方式のほうが高速である.

4) すべてのコマンドのプログラムのコードやデータが，本方式ではすべて主記憶になければならないが，UNIXでは，その必要がないので，多数のコマンドを並べたり，サイズの大きなコマンドも並べることができる.

5) 本方式のパイプラインプロセス群の主プロセスの行う仕事を，UNIXではshell[7]が実行している.

6) パイプライン実行のために，本方式では，パイプライン用の支援ルーチン，入出力プロセスを特別に付加する必要があるが，UNIXでは，OSでそれらの環境を支援している.

6. あとがき

本論文で述べたコンカレント機能は，実現上，

- OSに依存していない.
- 言語のコンパイラに変更を加えていない.
- 計算機のハードウェアの援助を必要としない.
- 簡単に作成できる.

という特徴をもち，シーケンシャル・プログラミングしか考えていない計算機でも，コンカレント処理機能をもつシステムを容易に実現できる.

この機能を用いて，5章で述べたようなパイプライン機能が容易に実現できたことは，設計機能が十分有用であることを示している.

しかし，実現したコンカレント機能には，次のような制限がある.

- 単一CPUでの実現のみを考えている.
- プロセスが，複数のイベントを待つような機構は用意されていない.
- プロセス間での変数の共有が許される．とくにextern変数の場合，すべてのプロセスに共有されてしまう.

これらの制限を除くには，コンカレント処理のできるOSの支援と，その機能が取り入れられた言語が必要である.

参考文献

1) DoD: Reference Manual for the Ada Programming Language, United States Department of Defense (Jul. 1980).
2) DoD: Rationale for the Design of the ADA Programming Language, *SIGPLAN NOTICES*, Vol. 14, No. 6 (Jun. 1979).
3) Brinch, H. P.: The Architecture of Concurrent Programs, Prentice-Hall, New Jersey (1977). (田中英彦訳: 並行動作プログラムの構造, 日本コンピュータ協会, 東京 (1980)).
4) Hoare, C. A. R.: Communicating Sequential Processes, *CACM*, Vol. 21, No. 8, pp. 666-677 (Aug. 1978).
5) Brinch, H. P.: Distributed Processes: a Concurrent Programming Concept, *CACM*, Vol. 21, No. 11, pp. 934-941 (Nov. 1978).
6) Kernighan, B. W. and Ritchie, D. M.: *The C Programming Language*, Prentice-Hall, New Jersey (1978) (石田晴久(訳): プログラミング言語C, 共立出版, 東京 (1981)).
7) Bourne, S. R.: The UNIX Shell, *Bell Syst. Tech. J.*, Vol. 57, No. 6, pp. 1971-1990 (Jul.-Aug. 1978).
8) 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).

(昭和56年2月17日受付)
(昭和57年9月 6 日採録)