並列処理シンポジューム
JSPP '89での発表論文

December, 1988

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

TM 0659　PIMOSの資源管理方式
佐藤裕幸，越村三幸，近山　隆，藤瀬哲朗，松尾正浩，
和田久美子
TM 0660　マルチPSIにおける並列構文解析プログラムPAXの実現
及び評価
寿崎かすみ，佐藤裕幸，瀧　和男，杉村領一，赤坂宏二，
山崎重一郎，弘田　直人
TM 0671　マルチPSIにおけるデバッグ機能・メンテナンス機能
杉野栄二，古市昌一，稲村　雄，瀧　和男

PIMOSの資源管理方式

佐藤裕幸[1]、越村三幸[1]、近山隆[1]、藤瀬哲朗[2]、松尾正浩[2]、和田久美子[3]
1: (財)新世代コンピュータ技術開発機構[*] 2: (株)三菱総合研究所 3: 沖電気工業(株)

概要

ICOTでは、知識情報処理を行うに当たって必要とされる計算能力を提供するために、並列推論マシンの研究開発を行っている。この並列推論マシンの能力を最大限に引き出すためには、高度に並列に動作するプログラムを効率的に制御するオペレーティング・システム(OS)が必要不可欠である。PIMOSは、この目的に沿って設計された並列推論マシン用のOSである。本論文は、このPIMOSにおける資源管理方式について述べたものである。

PIMOSは、並列論理型言語KL1で記述されている。このKL1は、FGHCを基本にしており、オブジェクト・レベルのプログラムの実行を制御/監視するための機能が拡張されている。PIMOSでは、この機能を利用して、ユーザが利用するさまざまな資源をタスクと呼ばれる計算単位ごとに管理している。タスクは任意にネストすることができ、子孫タスクもそれを生成したタスクの中で使用されている資源として扱われる。これらの資源は、タスクの階層に沿った木構造の機構によって、分散して管理され、ユーザはストリームを介して通信することにより、これらの資源にアクセスすることができる。

また、ユーザ-PIMOS間に保護のためのフィルタを置くことにより、システム全体を保護している。このフィルタは、ユーザの誤りがPIMOSに浸透することを防ぐだけでなく、ユーザの不正な侵入からもシステムを保護している。

現在PIMOSは、マルチPSI第2版上で開発されており、既に、その最初の版が稼働している。我々は、並列論理型言語KL1によるOSの実現により、その有用性を確認できたと考えている。

1 はじめに

第5世代コンピュータ・プロジェクトでは、高度な知識情報処理を行うに当たって必要とされる計算能力を提供するために、並列推論マシン(PIM)[1, 2]の研究が進められている。また、並列ソフトウェアの研究を促進させるためのシステムであるマルチPSI[3, 4]の開発も進められている。これらの並列推論マシンの能力を最大限に引き出すためには、高並列に動作するプログラムを効率的に制御するオペレーティング・システム(OS)が必要不可欠である。

PIMOS(Parellel Inference Machine Operating System)[5, 6, 7]は、この目的のために設計されてたものであり、PIMやマルチPSIなどの並列推論マシン用のOSである。このPIMOSは、以下のような特徴を持っている。

KL1で記述された純粋なロジックOS: PIMOSは、並列論理型言語KL1(核言語第1版)を用いて、副作用などの超論理的な機能を用いずに記述されている。オペレーティング・システムなどを記述する際にこの超論理的な機能を用いなくて済むように、KL1は設計されている。そのため、従来のシステムではオペレーティングシステムで実現しなければならなかった機能の幾つかのものが言語レベル以下で実現されている。

単一OS: PIMOSは並列処理マシン用のOSではあるが、複数のOSの集合体ではない。プロセサごとに独立して動作するOSが協調して全体の機能を果たすのではなく、全体として一体であるシステムの並列動作可能な部分を並列に実行するものである。

フロントエンド・プロセサを接続: PIMOSが扱う並列推論マシンには、フロントエンド・プロセサ(FEP)が接続されている。このFEPは、物理的な入出力操作を行うのであるが、それらの低レベルの操作がPIMOSから見えないように設計されている。マルチPSIでは、FEPとして、PSI-II[8]を使用している。

本格的なOSとして必要な機能を網羅: PIMOSは実験的なシステムではなく、本格的なOSとして実用に耐えるだけの基本機能を網羅している。現在、工程の関係上、ウインドウ・システムやファイル・システムなどの機能を、FEP上のOS(SIMPOS)に依存しているが、これらもKL1で記述できる仕様となっている。

使い勝手の良いシステム: ハードウェアや並列アルゴリズムの実験/評価が円滑に行えるように、使い勝手の良いシステムとなっている。そのためには、ユーザが誤った使い方をした時に、システム全体を保護することも重要な要素となっている。

単一ユーザ/複数タスク: 複数タスクを許すことはシステムの使い勝手の上から必須であるし、並列処理マシンの能力を最大限に引き出すためにも当然必要な機能である。複数ユーザを考えるとユーザ間の競合関係を調整する必要が生じる。これを並列環境の下で制御するのは、非常に困難な問題なので、当面は

[*] 〒108 東京都港区三田1-4-28 三田国際ビル21階
TEL: 03-456-3069 NET: hiroyuki%icot.junet

本格的には扱わないこととし、将来の課題として残されている。

クロスシステムの充実: 実験的なハードウェアのためのOSであるので、ハーウェアはいつでも利用可能とは限らない。従って、ソフトウェアの研究開発を促進するために、PIMOSではKL1による並列プログラムの開発環境を、PSIや汎用計算機などの、より利用しやすい計算機上に提供している。

PIMOSは計算機資源(CPU資源、メモリ資源、入出力機器)の管理を行い、ユーザの過ちからシステム全体を守ることが、もっとも基本となる機能である。このためには、計算機資源をそのままユーザ(応用プログラム)に見せることはできないので、計算機資源に保護のための枠組を付加し、仮想化してから提供することが必要になる。こうした管理機能がPIMOSの中核となる部分である。

本論文では、このPIMOSの管理機構について、

第2章: プロセスによる通信方式

第3章: 言語処理系で提供する資源管理機構

第4章: 資源木による資源の管理方式

第5章: ユーザの誤りからのシステムの保護方式

を中心に報告する。

2 プロセス

ユーザ・プログラムが消費するCPU時間、メモリ、入出力装置などの資源を、それぞれのプログラムが並列に動作する環境の下で管理するのは、容易なことではない。例えば、OSがあるデータに対する処理を行っている最中に、そのデータの状態をユーザ・プログラムによって変更されることを防ぎたい場合がよくある。そのような場合、従来の逐次計算機上では、「OSがそのデータを処理している最中は、ユーザ・プログラムを動作させない」という実行順序によってデータの変更を防ぐことができた。しかし、PIMOSが動作するような並列環境では、並列性を犠牲にせずに、そのような実行順序の制御はできない。つまり、OSがそのデータを処理している最中に、全てのユーザ・プログラムの実行を停止させることになり、これは明らかに並列性を犠牲にすることになる。従って、従来の逐次計算機と同様の方式を採用することはできない。

そこでPIMOSでは、「プロセス」[9]と呼ばれるプログラミング・スタイルを導入することにより、この問題を解決している。このプロセスは、管理しているデータと通信用の変数(ストリーム)を保持している。プロセスの外からこのデータにアクセスする場合は、このストリームにメッセージを送るという形でしか行えない。従って、そのストリームにアクセスすることができないプログラムは、このデータにアクセスすることはできない。つまり、データへのアクセス・パスを1つにすることで、複数のプログラムからの競合を防いでいる。

PIMOSが管理する全ての資源は、このプロセスを用いて、ストリームを介して通信することによって管理されている。つまり、このシステムでの資源へのアクセスは、従来のシステムのスーパバイザ・コールのような、資源を集中管理しているモジュールへの通信によって行われるのではなく、それぞれの資源を管理しているプロセスへストリームを介して通信することによって行われる。従って、PIMOSでは、これらのストリームを管理することが、資源を管理することに相当している。

ストリームを介したメッセージの送受信は、具体的には、共有変数の具体化によって行われる。例えば、ユーザが、ある入力デバイスから文字列を読み込む場合は、以下のようなプログラムになる。

```
?- pimos(Req), user(Req).
user(Req) :-
   Req = [getb(N,String)|ReqT], ...
pimos([getb(N,String)|ReqT]) :-
   readFromKbd(N, KBDString),
   KBDString = String,
   pimos(ReqT).
```

ユーザとPIMOSとは、共有変数Reqをストリームとして利用することで通信を行う。ユーザは、そこにN文字読み込みたいと言う要求getb/2を送る。PIMOSは、そのメッセージを受け取ると、readFromKbd/2により必要なN文字分の文字列を読み込み、Stringを読み込まれた文字列に具体化することによりユーザに返す。そして、ReqTにより次の通信が行われる。

このpimos/1のようなプログラミング・スタイルをプロセスと呼んでおり、メッセージによってのみ、自分の管理している資源(この例ではキーボード)への操作を許している。

ユーザとPIMOSとの通信は、最初に与えられた唯一の共有変数(ストリーム)Reqを基に、共有変数を増やして行くことで進んで行く。例えば、ユーザが新しいウインドウを生成し、そこに対して入出力を行う場合は、以下のようになる。

```
?- pimos(Req), user(Req).
user(Req) :-
   Req = [create_window(Window)|ReqT],
   Window = [getb(N, String)|WinT],
        .....
```

新たに生成されたウインドウに対する入出力操作は、PIMOSから与えられた通信用変数Window及びそこから派生した変数を介してのみ行える。つまり、これらの変数を持っていない他のプログラムは、このウインドウにアクセスすることはできない。従って、PIMOSではこれらの通信用の変数をユーザに与えることが、資源にアクセスすることのできる権利(capability)を与えることに相当しており、この機構により、ユーザの不正なシステムへの介入を防いでいる。

3 荘園 (資源管理, 例外処理) 機能

PIMOS の記述言語である KL1 は Flat GHC [10] を基にしているため、全てのゴールは AND 関係になっている。従って、このままでは、ユーザ・プログラムが(上記の例の user(Req) から呼ばれたサブ・ゴールの内の1つでも)失敗した場合は、PIMOS も失敗し、システムダウンを起こしてしまう。これは、PIMOS とユーザ・プログラムが同じレベルになっているからである。しかし、ユーザ・プログラムと OS とは、オブジェクト・レベルのプログラムと、そのオブジェクト・プログラムを監視 / 制御するメタ・レベルのプログラムの関係になるべきである。

メタ・プログラミングの機能を実現する単純な方法には、インタプリタがあるが、実行効率面で大きな損失が生じてしまう。そのため、PIMOS の記述言語である KL1 では、このメタ・プログラミングの機能が言語レベルで組み込まれており、これを「荘園」と呼んでいる。荘園には、実行制御 / 監視機能, 資源管理機能, 例外処理機能が組み込まれている。

3.1 実行 / 資源管理機能

荘園は以下のような組込述語を用いて生成する。

```
execute(Goal, ExpMask, Control, Report)
```

ここで, 各引数は以下のような意味を持つ。

Goal: 荘園の中で実行すべきゴールを指定するための変数である。

ExpMask: この荘園で、どのような例外を処理するかを決めるためのマスク・パターンである (詳しくは後述)。

Control: 荘園内の実行を制御したり、資源の割り当てを指定するためのストリームである。ここには、以下のような機能を持ったメッセージを流すことができる。

- 荘園内の実行の開始 / 中断 / 再開 / 放棄
- 荘園内で消費できる計算資源の許容量の問い合わせ / 追加
- 荘園内でこれまでに消費された計算資源量の問い合わせ

Report: 荘園内の実行状態が報告されるストリームである。ここには、以下のような2種類の報告が行われる。

事象の報告: 荘園の実行にともなって生じた事象が報告され、これには以下のようなものがある。

- 荘園内の実行の終了
- 荘園内で消費できる計算資源の不足
- 荘園内で発生した例外事象 (オーバフロー, ゴールの失敗など)

受け取りの確認: 制御ストリームからのメッセージを受け取ったことを報告する。荘園の制御 / 監視は、制御用と報告用の2つのストリームによって行うので、この確認の報告がないと、それらのストリームに流れるメッセージ間の同期をとることができない。例えば、制御ストリームから計算資源の許容量を追加した後、報告ストリームから計算資源の不足が報告された場合、

(1) 資源を追加したがそれでも不足してしまったので、更に追加する必要がある
(2) 資源追加のメッセージが届く前に不足を検出したので、その後資源は追加され、今は追加する必要がない

のどちらか分からない。しかし、報告ストリームから資源追加メッセージを受け取ったという報告がなされれば、それと資源不足の報告との前後関係によって、上記のどちらであるかを知ることができる。

荘園内のゴール群は、荘園外とは独立した AND 関係を成している。すなわち、荘園内での失敗は荘園内に閉じたものであり、荘園外のゴールを巻き添えにすることはない。従って、概念上、荘園はインタプリタを機械語レベルで実現したものであると考えて良い。

また、荘園の中で更に荘園を作ることも可能である。この場合、外側の荘園に実行の中断を指令すれば、内側の荘園の実行も中断される。外側の荘園の実行を再開すれば、内側の荘園の実行も再開される。また、内側の荘園が使用できる資源は、外側の荘園から分け与えられるものである。つまり、内側の荘園が消費した資源は、外側の荘園も消費したと扱われる。

このような管理方式を取ることにより、メタ・レベルはオブジェクト・レベル以下にどのようにメタ / オブジェクトの階層構造があるかを意識することなく、全体として管理することが可能となっている。

3.2 例外処理機能

3.2.1 例外の報告

荘園内の実行中に、例外事象が生じると、以下のようなメッセージが報告ストリームに流がれて来る。

```
exception(Info, Goal, NewGoal)
```

ここで各引数は以下のような意味を持つ。

Info: どのような例外が生じたかについての情報

Goal: 例外を起こしたゴールの情報

NewGoal: 例外処理として、例外を起こしたゴールの代わりに実行すべきゴールを指定するための変数

各例外事象は、その種類に応じて適当なタグが割当てられている。荘園がネストしている場合の例外の報告は、例外タグと荘園のマスク・パターンの論理積が 0 でないような最も近い祖先荘園の報告ストリームに流される。これにより、特定の例外だけを取り扱い、他の例外はより外側の祖先荘園に任せるような記述が可能となっている。

3.2.2 例外の生起

例外事象には、言語処理系で検出するものと、ソフトウェア(PIMOSやユーザ)が検出するものがある。両者を同じ枠組で扱うために、以下のような組込み述語が用意されている。

```
raise(Tag, Info, Data)
```

ここで、各引数は以下のような意味を持つ。

Tag: 例外のタグを指定する。

Info: 例外に関する情報を指定する。この引数が完全に(構造体の場合は構造体全体が)具体化されるまで例外の生起は遅延される。

Data: 例外に関する情報を指定する。ただし、この引数は具体化されていなくてもよい。

例外を取り扱うプログラム(例外ハンドラ)は例外を出すプログラムのメタ・プログラムである。このメタ・プログラムが永遠に変数の具体化を待ち続けることを防ぐためには、具体化されていることが保証された例外情報が必要である。組込み述語 raise の引数 Info はこのような目的に用いるためのものである。一方、引数 Data は、メタ・プログラムが中身を読まなくても良いデータを渡すのに用い、それらは NewGoal の中で読まれる例外からの復帰のための情報に用いられる。

この例外発生機能は、オブジェクト・プログラムとメタ・プログラム(つまりユーザと OS)の通信にも使用できる[1]。

3.2.3 例外後の実行の継続

ある荘園内で例外が発生した場合は、その例外を起こしたゴールの実行だけが中断され、その他のゴールの実行は継続される。例外の発生により荘園の実行を中断または放棄する場合には、その荘園の制御ストリームからメッセージを流すことで行う。また、例外を起こしたゴールは、NewGoal の具体化を待って、元のゴールの代わりに NewGoal を実行する。このゴールがあるため、NewGoal を具体化しないうちに荘園内の実行が終了してしまうことはない。

この荘園の機能を使った、ユーザと PIMOS との通信は以下のようになる。

```
?- pimos(Req, Cnt, Rep),
   execute(user(Req), Cnt, Rep).
```

PIMOS は、変数 Cnt によりユーザ・プログラムの実行を制御し、変数 Rep により実行の監視及び資源の管理を行う。このように荘園は、制御ストリームと報告ストリームをインタフェースとした、言語レベルで提供されているプロセスと考えることができる。

[1]実際に PIMOS では、ユーザが最初に OS との通信を行うためのストリームを獲得する方法として、この例外生起機能を利用している。

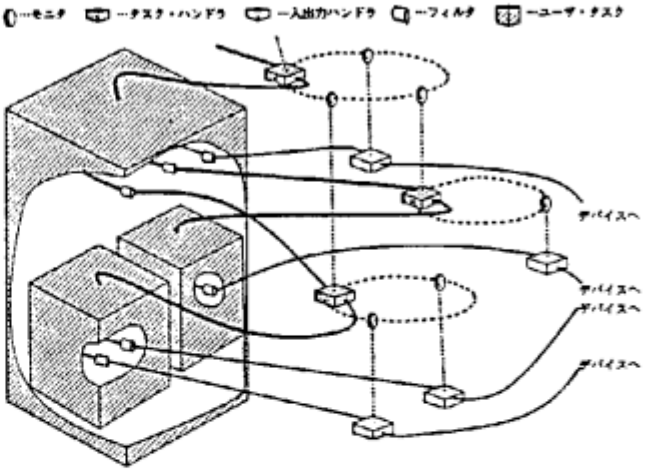

図 1: 資源木とユーザ・タスクの構造

4 資源木による資源の管理

OS が管理すべき資源には、荘園で管理されるような計算時間やメモリのほかに、入出力装置などがある。前者のような資源を「言語定義資源」と呼び、後者のような資源を「OS 定義資源」と呼んでいる。このように、PIMOS がユーザの消費する全ての資源を管理するためには、荘園の機能だけでは不十分であり、入出力装置などの OS 定義資源も管理する機構を導入しなければならない。

4.1 タスク

これらの言語定義資源や OS 定義資源の管理を行うために、PIMOS では、「タスク」と呼ばれる資源管理の単位を導入している。このタスクは、荘園の機能を用いて実現されており、荘園と同様に任意にネストすることができる。このタスク内で生成された子タスクも、親タスク内で使用されている資源の1つとして管理される。

ユーザがタスクを生成すると、タスク制御ストリームとタスク報告ストリームが返される。ユーザは、タスク制御ストリームにメッセージを送ることにより、タスク内の実行を制御したり、タスク内で使用されている資源に対して問い合わせることができる。また、タスク報告ストリームから知らされる情報により、タスクの実行状態を監視したり、例外処理を行なうことができる。

4.2 資源木

タスクの中で使用されている全ての資源(言語定義資源及び OS 定義資源)は、「資源木」と呼ばれるストリームでお互いに結合されたプロセスの木構造によって、タスクの階層ごとに管理されている。

タスクの階層構造及び資源木の構造の例を図 1に示す。この例では、3つのタスクが存在しており、それぞれのタスクは以下のような OS 定義資源を使用している。

親タスク: 1つの入出力資源と「子タスク 1」と「子タスク 2」

子タスク 1: 2つの入出力資源

子タスク 2: 1 つの入出力資源

この図の右側の部分を資源木と呼び、ユーザ・タスクとは別の領域に置かれる。資源木の各ノードは、タスクの階層にそって木構造になっており、各タスク内の資源は、それぞれが輪構造で結ばれている。以下に、資源木を構成する各要素について説明する。

タスク・ハンドラ: タスク内の資源を管理しているプロセスであり、ユーザがそれら資源を操作する時に、ここにメッセージが流れて来る。また、祖先タスクからもここで管理している資源に対するメッセージがモニタ (後述) を経由して流れて来ることもある。ここでは、タスクを実現している荘園の制御ストリームと報告ストリーム (図ではタスクの上につながる太実線) を保持することで、タスクの実行を制御し、実行状態を管理している。

また、このタスク内で使用されている入出力装置や子タスクなどの OS 定義資源は、それぞれ対応するモニタを輪状に繋げること (これを資源ループと呼ぶ) で管理されている。これらの資源に対する操作メッセージは、モニタを通して子資源のハンドラに送られる。

入出力ハンドラ: 入出力装置を資源としてユーザに見せるためのプロセスであり、一般の入出力要求に応じる機能と、この資源に対するメタな問い合わせ (例えば、どのような種類の資源なのかなど) に答える機能を持っている。前者のようなメッセージはユーザからフィルタを通して送られ、更にデバイスに近いプロセス (デバイス・ドライバ) に再送される。また、後者のようなメッセージはモニタを通して送られ、このプロセスで処理される。

モニタ: タスク・ハンドラからのメッセージを自分が保持しているハンドラに送るかどうかを決める。そのメカニズムについては、後述する。

フィルタ: ユーザの誤りからシステム全体を保護するためのプロセスである。ユーザと PIMOS が通信する場合には、必ずこの種の保護フィルタが付けられる。この保護フィルタはユーザ・タスク上で実行されるが、プログラム・コードとしては PIMOS が提供するものである。なお、保護フィルタの詳細については、5 章で記述する。

4.3 OS 資源の管理方式

PIMOS が管理する総ての OS 定義資源には、各タスク内でユニークな ID (資源 ID) が付けられており、資源 ID 列によって特定の資源を指定することができる。例えば、[3,4,2] であれば、タスク内の 3 番目の資源内の 4 番目の資源内の 2 番目の資源を表わしている。

ユーザは、タスク制御ストリームを通して、メッセージを送ることで OS 定義資源に対する制御を行える。この時、資源 ID をメッセージに付けるとその資源に対する操作となる。モニタは、それぞれ自分が管理している OS 定義資源の ID を保持している。そして、自分の所に流れてきたメッセージに付いている ID と自分の ID を比べ、一致すればそのメッセージを (資源 ID 列の残りを付けて) 自分のハンドラに送り、一致しなければとなりのモニタに再送する。

タスク・ハンドラは、親モニタからメッセージを受け取ると、それが自分宛であれば自分で処理し、子孫資源宛であれば自分の資源ループに再送する。資源ループに送ったメッセージがもう一方のストリームからそのまま返ってきた場合は、指定した ID に相当する資源が存在しなかったことになり、エラーとして扱われる。

タスク・ハンドラは、タスクを実現している荘園の報告ストリームを監視している。そこから実行の終了または放棄が報告されると、そのタスク内で使用されていた未解放の全ての資源 (資源ループ) に対して、タスク・ハンドラが解放要求を送ることにより、資源を自動的に解放する。

OS 定義資源を解放する時には、モニタが自分の兄からのストリームと弟へのストリームをユニファイすることにより、輪の中から消滅する。ただし、資源木内のそれぞれのプロセスが並列に動作しているので、資源の解放には注意を要するが、その詳細については省略する。

4.4 従来 OS との比較

従来の OS では、タスク (システムによってはプロセスと呼ぶこともある) は、1 つの平坦なテーブルで集中的に管理されている場合が多い。また、入出力装置への操作も、例えば、「タスク内の入出力資源テーブルの N 番目に登録されている出力装置に出力する」という形で行われることが多い。従って、これらの資源に対する操作を行うたびに管理テーブルへアクセスすることになる。

従来の逐次計算機システムの場合は、逐次的に実行されるため、この管理テーブルへのアクセス集中がそれほど問題になっていなかった。一方、並列計算機の場合は、独立した資源に対する操作を並列に行うことができるが、メモリ共有型の並列計算機の場合には、資源へのアクセス頻度などを考えると、集中管理によるネックがそれほど問題となっていない。

しかし、PIMOS が対象としているネットワーク型でしかも並列度の高い並列計算機システムの場合は、集中管理によりプロセサ間の通信量が増えてしまうので、並列性を犠牲にすることによる。そのため PIMOS では、資源のアクセスが、「資源へのストリームに直接メッセージを送る」という形になっており、独立した資源への操作は、それぞれ並列に行えるようになっている。

また、「タスク内で使用している全ての資源の状態を返す」といった処理の場合、タスク・ハンドラは、資源ループへその旨のメッセージを送るだけなので、すぐにユーザからの次のメッセージに対する処理を行える。そしてモニタは、そのメッセージを受け取ると、資源ハンドラにメッセージを送ると共に、次のモニタに同じメッセージを送る。このように、メッセージが各資源ハンドラに分散されるため、それぞれ独立して各資源の状態を返すことができる。

5 通信の保護機構

5.1 問題点

ユーザ・プログラムの実行をメタなレベルから監視する機構は、KL1 の述語機能により行える。しかし、PIMOS-ユーザ間の通信では、KL1 に個々のユニフィケーションを制御 / 監視する機能がないため、単純に通信を行ってしまうと、システムダウンに陥ることがある。

ユーザが指定した長さ (文字数) の文字列をキーボードから読み込む場合を考えてみる。

```
user(Req) :- true |
   Req = [getb(N,String)|ReqT],...
pimos([getb(N,String)|ReqT], Cnt, Rep) :-
      true |
   readFromKbd(N, KBDString),
   KBDString = String,...          --- (a)
```

ユーザは、通信用ストリーム Req に入力要求メッセージ getb/2 を送る。PIMOS はメッセージ getb/2 を受け取ると、 readFromKbd/2 により必要な N 文字分の文字列を読み込む。この時、readFromKbd/2 は、変数 N が具体化されるのを待つ。そして、変数 String を読み込まれた文字列 KBDString に具体化し、ユーザに返す。しかしこのプログラムには、以下のような 2 種類の問題点が含まれている。

問題点 1: ユーザが以下のように (誤って) 文字列が入るべき変数をアトムに具体化してしまったとする。

```
String = foo                       --- (b)
```

並列環境の下では、効率を落とさずにゴール (a) と (b) の実行順序を規定することはできない。そのため、(a) が先に実行されれば、(b) が失敗するし、(b) が先に実行されれば、(a) が失敗する。前者の場合であればユーザ・タスク内で失敗が起きるだけであり、問題はないが、後者の場合だと、PIMOS 側で失敗が起こるので、システムダウンに陥ってしまう。

KL1 では、ユニフィケーションが成功するかどうかをチェックするような (テスト・アンド・ユニファイ) 機能を持っていない[2]。例えば、上記の String が未定義変数かどうかのチェックができたとしても、そのチェックとユニフィケーションとの間に、String が他の値に具体化されてしまう可能性があるからである。

問題点 2: タスク側が文字数設定用共有変数 N に値を設定するのを忘れたか、または設定する前にタスクの実行が放棄された場合、PIMOS 側の N の具体化を待つプロセス (readFromKbd/2) が永遠に待ち続け、デッドロックしてしまう[3]。

[2] もし、KL1 がテスト・アンド・ユニファイの機能を持った Concurrent Prolog [11] を基にしているのであれば、問題とはならない。

[3] これは、たとえテスト・アンド・ユニファイの機能があっても問題となる。

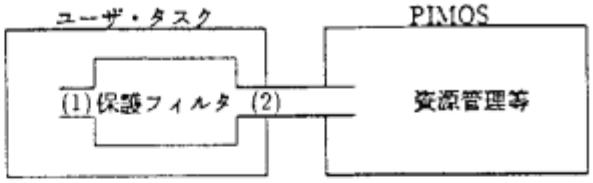

図 2: 保護フィルタ付き通信

5.2 保護フィルタ

これらの問題点を解決するために、PIMOS-ユーザ・タスク間のストリームに保護のためのフィルタ・プロセス (保護フィルタ) を設け、そのプロセスをユーザ・タスク内で実行することにした (図 2)。

タスクは PIMOS とつながっているストリームに直接メッセージを送らずに、保護フィルタへのストリーム (図 2 の (1)) へメッセージを送る。このフィルタはユーザのメッセージを絶対に失敗しない形式に変換し、また、PIMOS 側で待ち続けないことが保証されるまで待ってから、PIMOS へのストリーム (図 2 の (2)) へ変換されたメッセージを送信する。

このように PIMOS 側のストリームをユーザに直接見せないため、ユーザの誤りが PIMOS 側へ波及するのを防ぐだけではなく、ユーザからの悪意を持った PIMOS への侵入を保護フィルタが監視して防ぐことができる。つまり、保護フィルタは、ストリームの capability を制限していることになる。

この保護フィルタの具体的な仕事は以下のようなことである。

- ユーザが値を設定すべきデータ部分は、その値が設定されたことを確認してから PIMOS 側へ送る (問題点 2 の解決)。これにより、PIMOS にメッセージが送られた時には、それらの値は確定していることが分かる。ただし、ストリームの場合だけは、そのヘッド (第 1 要素) の具体化しか待たない。そうしないと、そのストリームを閉じるまで、全ての要求が PIMOS 側に流れないことになるからである。

- PIMOS 側から返されるデータ部分は、必ず未定義状態であるような別の変数を送る (問題点 1 の解決)。そして、PIMOS から値が返された後で、ユーザが指定した変数を返ってきた値に具体化する。これにより、PIMOS 内での値を返すユニフィケーションは、必ず未定義変数とのユニフィケーションとなり、失敗することはない。

例えば、上記のキーボードの例に対する保護フィルタ・プログラムは以下のようになる。

```
pfilter([getb(N,String)|T], OS) :- wait(N) |
     OS = [getb(N,OSString)|OST],
     waitAndUnify(OSString, String),
     pfilter(T, OST).
waitAndUnify(OSV, USERV) :- wait(OSV) |
     OSV = USERV.
```

保護フィルタ(pfilter/2)は、読み込む文字数 N に値が設定されるまで待ってから PIMOS へ要求を送信する。また、読み込まれた文字列が設定される変数 String の代わりに、未定義変数 OSString を PIMOS に送る。そして、その変数の値が設定されるのを待ってから、ユーザの変数 String へ文字列が設定される。従って、ユーザが変数 String を誤った値に具体化たとしても、ユーザ・タスク内で実行されるフィルタ(OSV = USERV)が失敗するだけで、PIMOS には影響はない。

このフィルタは、ユーザ・プログラムを起動する時に、PIMOS により以下のように挿入される。

```
?- pimos(OSReq, Cnt, Rep),
   execute( ( user(Req),
              pfilter(Req, OSReq) ),
            Cnt, Rep).
```

また、ストリーム Req から新たに派生した通信用変数のためのフィルタは、pfilter/2 の中で生成される。例えば、ウインドウを生成するメッセージのための保護フィルタのプログラムは、以下のようになる。

```
pfilter([create_win(Name,Win)|T], OS) :-
        wait(Name) |
  OS = [create_win(Name,OSWin)|OST],
  pfilter_win(Win, OSWin),
  pfilter(T, OST).
```

ユーザは新たに生成されたウインドウに対するメッセージを Win に送るが、それは PIMOS と直接つながれておらず、pfilter_win/2 により保護される。

5.3 保護フィルタ・プログラムの生成

この保護フィルタを生成するためには、以下の項目からなるユーザ-PIMOS 間の通信プロトコルを定義する必要がある。

- 通信メッセージのどの部分をユーザが設定するのか
- 通信メッセージのどの部分を PIMOS が設定するのか
- それらの値はどのようなプロトコルか(システム定義または再帰的に定義されたもの)

この通信プロトコルは、従来の OS でもそうであるが、OS がユーザの要求に答えるために、所詮、定義しなければならない項目である。

この通信プロトコルが定義されれば、保護フィルタのプログラムは、機械的に生成することができる。つまり、ユーザ-PIMOS 間の通信プロトコルを定義するための記述を設け、その定義からトランスレータを用いて保護フィルタ・プログラムに変換することができる。

例えば、キーボードに関する通信プロトコルは、図 3 のように定義できる。この例で、- は PIMOS が値を設定することを意味し、stream/1 . atomic は、システムにより定義されたプロトコルである。また、stream/1 は、パラメタ付きのプロトコルである。

```
kbd = stream(kbd_command).
kbd_command = {
   getb(length, -buffer) ;
   getl(-line) ;
   getc(-character) }.
length  = integer.   buffer    = string.
line    = string.    character = integer.
integer = atomic.    string    = atomic.
```

図 3: プロトコル定義の例

```
'pfilter$kbd'([getb(Length,Buffer)|T], OS) :-
          wait(Length) |
     OS = [getb(Length,OSBuffer)|OST],
     waitAtomic(OSBuffer, Buffer),
     'pfilter$kbd'(T, OST).
'pfilter$kbd'([getl(Line)|T], OS) :-
     OS = [getl(OSLine)|OST],
     waitAtomic(OSLine, Line),
     'pfilter$kbd'(T, OST).
'pfilter$kbd'([getc(Character)|T], OS) :-
     OS = [getc(OSCharacter)|OST],
     waitAtomic(OSCharacter, Character),
     'pfilter$kbd'(T, OST).
waitAtomic(X, Y) :- wait(X) |
     Y = X.
```

図 4: 保護フィルタ・プログラム

kbd は、kbd_command で定義されているプロトコルのデータが流れるストリームである。kbd_command には、3 種類のメッセージがある。例えば、メッセージ getb/2 は、ユーザが第 1 引数に length で定義されるデータ(最終的には atomic)を設定し、PIMOS が第 2 引数に buffer で定義されるデータ(最終的には atomic[4])を設定する。

このプロトコル定義は、図 4 のような保護フィルタのプログラムに変換される。

5.4 資源として扱うプロトコル

以前述べたように、ストリームに対する保護フィルタは、そのヘッドの具体化しか待たない。従って、ストリームは特別なプロトコルとして扱われ、システム定義となっている。

このプロトコルでは、ストリームが閉じられる前にユーザ・タスクの実行が放棄された場合に、PIMOS 側にストリームの具体化を待つプロセスが残ってしまうよ

[4]ストリングはその要素として未定義変数を含むことができないので、アトミックとして扱われる。

りに思われる[3]。従って、ユーザ・タスクの実行が放棄された時に、これらのストリームを閉じなければならない。このためには、ストリーム・プロトコルである全ての通信チャネル(共有変数)を資源として登録する必要がある。

このストリームの他にも、以下のようなプロトコルは、資源として扱わないと不具合が生じてしまう。

```
example = {-message(atomic)}
```

このプロトコルでは、PIMOS が message(Data) の形をしたデータを設定するが、その内側の Data は、ユーザが設定する。このプロトコルの保護フィルタ・プログラムは、以下のようになる。

```
'pfilter$example'(User, message(OSData)) :-
    User = message(UserData),
    waitAtomic(UserData, OSData).
```

この例で、ユーザが UerData を具体化せずにタスクの実行を放棄した場合、PIMOS 側に OSData の具体化を待つプロセスが残ってしまう。従って、PIMOS が設定するデータの内側にユーザの設定するデータが存在する場合は、その内側のデータを資源として扱わなければならない[4]。

6 おわりに

我々は、本論文で述べたプロセスによる資源の分散管理方式を用いて、マルチ PSI 第 2 版上で PIMOS の開発を行っており、現在、その最初の版が稼働している。

現在までの開発で得られた成果として、並列論理型言語 KL1 で OS を「記述できる」という点だけではなく、「記述しやすい」という点が挙げられよう。従来の手続き型の言語で OS を記述する際に最も注意を払う点は、同期の問題である。この同期は、データに対して行われる(例えば、必要なデータが揃うまで待つ)場合が多いが、それを実行順序を制御することで(つまり、実行順序の同期を用いて)行われてきた。従って、OS の中で最もバグが出やすくデバッグしにくい部分である。一方 KL1 の場合は、言語の基本機能として、データに対する同期機構が備わっているため、OS のようなプログラムを楽に記述できるし、バグも少なくなっている。

現在の PIMOS は、入出力機能を FEP 上の OS に依存している部分が多いが、今後、ファイル・システムなどを並列推論マシンの本体上に移行していく予定である。こういった部分に関しても、並列実行による高速化及び KL1 による記述の評価が行えると考えている。

3 章で、保護フィルタを自動生成できることを述べたが、プロトコル定義言語を拡張することにより、資源ハンドラやモニタ等も自動生成することが可能である。従って、これらの部分も自動生成を考えている。

並列マシンの OS にとって最も重要な課題は、負荷の分散である。現在の PIMOS では、負荷分散を総べてユーザに任せており、OS はユーザに近いプロセサで仕事をしている。この負荷分散を何らかの形で自動化する必要があるが、完全自動にするのは、かなり困難な研究課題である。

謝辞

本研究に関して有益な助言を頂いた ICOT 第 4 研究室、協力会社の方々及び沖電気工業の宮崎氏に深く感謝する。マルチ PSI への実装に際しては、KL1 処理系及びクロス・システムの担当者の方々に協力して頂いた。また、本研究の機会を与えて下さった ICOT 第 4 研究室の内田室長及び渕所長に感謝する。

参考文献

[1] A. Goto and S. Uchida, *Toward a High Performance Parallel Inference Machine – The Intermediate Stage Plan of PIM –*, Technical Report TR-201, ICOT, 1986.

[2] 後藤他: 並列推論マシン PIM– 中期構想 –, 第 33 回情処全国大会,3B-5,1986-10.

[3] K. Taki, *The parallel software research and development tool: Multi-PSI system*, Technical Report TR-237, ICOT, 1986.

[4] 瀧他: Multi-PSI システムの概要, 第 32 回情処全国大会, 3Q-8, 1986-3.

[5] T. Chikayama et al., *Overview of the Parallel Inference Machine Operating System (PIMOS)*, In Proc. of FGCS'88, Vol.1, pp.230-251, 1988.

[6] 佐藤他: PIMOS の概要 – 並列推論マシン用オペレーティング・システムの構築 –, 第 34 回情処全国大会, 2P-8, 1987-3.

[7] 佐藤他: 並列論理型 OS–PIMOS(1)– 資源管理方式 –, 第 35 回情処全国大会,4D-3,1987-10.

[8] H. Nakashima and K. Nakajima, *Hardware architecture of the sequential inference machine: PSI-II*, Technical Report TR-265, ICOT, 1987.

[9] E. Shapiro and A. Takeuchi, *Object Oriented Programming in Concurrent Prolog*, New Generation Computing, Springer Verlag Vol.1, No.1, pp.25-48, 1983.

[10] K. Ueda, *Guarded Horn Clauses*, Technical Report TR-103, ICOT, 1985.

[11] E. Shapiro and A. Takeuchi, *A Subset of Concurrent Prolog and Its Interpreter*, Technical Report TR-003, ICOT, 1983.

[3]ユーザがストリームを閉じ忘れた場合は、ユーザからの要求を待つフィルタがユーザ・タスク内に残っているので、ユーザ・タスクは正常終了することはない。その場合は、ユーザの問題であってPIMOS の問題ではない。

[4]現在の PIMOS では、このような通信プロトコルは存在していない。