TM-0234

並列論理型プログラミング言語における プラグマ機能について

神田陽治，鳥居　悟，大原有理，小野越夫
岸下　誠(富士通)

October, 1986

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

並列論理型プログラミング言語におけるプラグマ機能について

Pragma facilities in Parallel Logic Programming Languages

神田陽治（富士通国際情報社会科学研究所）
鳥居　悟・大原有理・小野越夫（富士通研究所）
岸下　誠（富士通ＳＳＬ）・田中二郎（ＩＣＯＴ）

並列論理型言語を使って並列オペレーティングシステムを記述する際に、どのような言語機能があれば並列マシンを活かしうるかを考える。従来はクローズに直接書き込んでいた分散戦略指示子プラグマを分離し、ポリシーとして独立させる。ポリシーの応用としてタスクミックスをじょうずに行うシェルの概略を示す。

1．はじめに

逐次論理型言語Prologを並列論理型言語に拡張しようとする研究は一段落し、引き続いてシステム記述言語に拡張しようとする段階へ焦点が移っている。先行した研究は、CP〔6〕、PARLOG〔3〕、GHC〔11〕といったストリームAND並列論理型言語を生みだした。これら並列論理型言語が期待されている理由には、並列マシンに載る数多くのプロセッサを使い切るだけのプロセスを、必要な同期を組み込みつつ容易に生成できる能力が挙げられる。ゴール列がプロセスとして並列に実行され、ゴール間の共有変数を介して通信が可能で、ガード（と若干の付加仕様）で同期を取る。逐次マシン上で動く並列言語としてはこれで十分であろうが、高並列マシンを活かすシステム記述言語としては一層の工夫を要する。記述したいのは並列マシン用のオペレーティングシステムである。

関心は次の処にある。並列マシン用のオペレーティングシステムを書くために、非列論理型言語に必要な機能は何か。例えば、シェルを書くためにはメタコール call を導入するのが便利である〔2〕。ここに仮定する非列マシンはプロセッサ間通信に局所性がある（プロセッサ間距離が大きくなれば通信コストも増える）ものである〔1, 6〕。並列マシンに特有で重要なものは負荷分散の方法、すなわち分散戦略であり、以下ではここに焦点を当てる。

I　プラグマとは、ポリシーとは

2．並列論理型言語におけるプラグマ機能

プラグマと言えば米国DoDのAda言語を思い出す。そこでのプラグマはコンパイラや実行系にオプション指示を与える注記（annotation）の役目を果たしていた。並列論理型言語におけるプラグマを次のように定義する。プラグマとは分散戦略を指示する注記である。並列論理型プログラムの論理的な意味には全く影響しないが、プログラムを実行する過程に助言を与えうる。「与えうる」と言い「与える」と言わないのは、プラグマがあくまで注記だからである。分散実行系は戦略実現のための操作（mechanism）を準備し、プログラマは操作を組み合わせて戦略（policy）を指定できる。操作と戦略の分離は、歴史的にも古い手法である〔12〕。組み合わせを指定し戦略を決める道具がプラグマというわけである。

プラグマの付け方を論理的な意味とは独立に工夫した方が良い時もあるだろう。論理型言語でのプログラムはクローズの集まりだが、クローズの論理的な意味記述からプラグマ記述を分離し、ポリシーと名付ける。そしてポリシーをクローズに適用してプラグマ付クローズを得、実行する。プラグマをポリシーとしてクローズから分離したことで、二つの利点が生まれる。

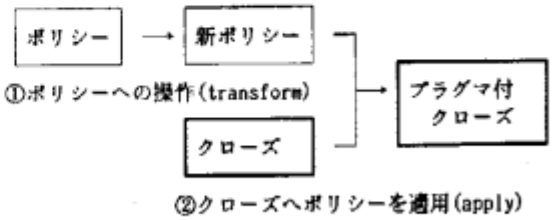
①ポリシーに何らかの操作を施し、新しいポリシーを得た後、クローズに適用してもよい。
②ポリシーとクローズを対に固定にせず、自由な組み合わせで、ポリシーをクローズに適用してもよい。

明らかに利点が生きるかどうかは、うまいポリシーの仕様が作れるかどうかに依っている。どんなポリシーにも施せる汎用な操作にはどのようなものがあるのか。どれほどポリシーをクローズから独立に維持できるものか。

3．オペレーティングシステムにおけるプラグマ機能

プラグマは、並列オペレーティングシステムのための支援機構として役に立つ。並列論理型言語の実行は、解くべきゴールとプログラムとして登録されたクローズのヘッドとの一致を試し、ガードを試し、成功したらボディを次のゴールとして投入する作業の繰り返しである。投入されたゴールは展開され数多くの副ゴールを生みだす。これら一群のゴール全体をタスクと呼ぶことにすると、タスクを数多くのプロセッサにうまく割り当て（タスクミックスを行い）、並列マシンの限界性能を引き出すことが並列オペレーティングシステムに課せられる。プラグマを使えば、タスク内での負荷分散はプログラマが思いのままに決められる。しかしタスクを越えてはできない。

並列オペレーティングシステムは、タスクミックスの現況を勘案しつつ、プログラムに付けられたプラグマを下敷きに分散戦略を立て直し実行する。初期ゴールを投入するプロセッサを決定したり、副ゴールを投げる方向を一斉に組織的に変更したりなどして、広がる版図の形や位置を調節しあうことで良いタスクミックスを達成し、なるべく多くのプロセッサがまんべんなく働けるように工夫する。プラグマのクローズからのポリシーの形での分離は、ここに述べたタスクミックスの問題を解決する。分散戦略をポリシーの形でもとのプログラムから分離すれば、タスクミックスを考慮しての分散戦略の立て直しは、ポリシーへ操作を施すことに当たる（図の①）。新しい分散戦略をプログラムに反映することは、新しく得たポリシーをクローズへ適用することに当たる（図の②）。



II　プラグマとポリシーの設計

4．プラグマの設計

必要なプラグマについてはいまだ議論の余地がある。有効なプラグマが今後発見される可能性もある。プラグマの設計というとき、二つに区分できる。ゴール単位に付けるプラグマと、クローズの選択を決めるガード部述語としてのプラグマである。

4-1　ゴール単位に付すプラグマ(1)

並列に実行されるゴールをプロセッサへの割り付け単位と見立てたとき、「ゴール分布の指定」を行うプラグマを考案したのは、Shapiro である〔7〕。数多くのプロセッサ資源を活かすためにはゴールを散らす必要があるが、一方で通信し合うゴール同志はなるべく近い距離に置きたい。無駄な通信を減らすためである。通信量の削減には、ゴール分布だけではなく「変数セル配置の指定」も大切であることを指摘したのは、玉木である〔8〕。ゴールが展開される際には、展開されるクローズ内の局所変数の一つずつに、値を納める記憶セルが取られる。記憶セルを割り付ける場所を、ゴールの展開を担当しているプロセッサのローカルメモリに限る必要はない。新ゴールが割り付けられる近傍のプロセッサに割り付けても一向に構わず、それで通信量が減らせるなら都合が良い。

Shapiro の方式はタートルグラフィックスを真似る。各ゴールはいま自分がどこにいてどちらの方向に頭を向けているのかを記録していて、「一歩前進」とか「右向け右」といったプラグマ指令に従い、カメさん同様にプロセッサを渡り歩く。タートルグラフィックスと違うのは、同じカメさんが一生歩き続けるのではなくて、新しく展開されたゴール群がプロセッサに割り付けられていくので、生きているゴール数が増減する点にある。プラグマは「@移動」の形をしていて、ゴールに何個でも後置できる。Goal @forward(N)は N歩前進、Goal@right @right は右に二回曲がって「回れ右」の指示である。例えば、

```
do(X) :- size(X,N) |
          process(X,A) @ forward(N), display(A?).
```

では処理の本体process を入力データ長に応じて遠くへ投げる。なお、玉木は変数セル配置の指定をヒューリスティックスの形で述べ、プラグマ記法の開発はしていない。

4-2　ゴール単位に付すプラグマ(2)

加えて我々は、いくつかのプラグマ候補を考慮している〔5,10〕。第一に「ゴール投入時期の指定」である。ゴール分布の指定では、いつゴールをプロセッサに配るかの時期まで指定しない。配るゴールに未確定の変数が含まれていると、後で問い合わせの通信が発生する。ゴール投入を遅らせて、変数が確定した後に送れば通信を減らせる。いわばデータ駆動式AND である。第二に「ゴールの処理優先度の指定」も有効と考える。ゴール分布の指定では、送られた先でのゴールの優先度合の指定まではしない。新ゴールの中には早く処理すべきものと、そうでないものがあるだろう。あるゴールの実行が全体の処理速度を決めるような「律速段階」ゴールであれば、ゴールを早く処理せよと指示したい。早く処理したいゴールはスケジューリングキューの先頭に突っ込み、そうでないものは尻尾に付ければよろしい。

4-3　ガード部に置くプラグマ

さらに我々は「入力データの性質」によって最適なプラグマ付けの方法は違うことに気付いた〔10〕。プログラムにプラグマを一律に張り付けておくだけでは、入力データによっては性能が落ち込む場合がある。一般化すれば、実行状況に応じてプラグマ付けを切り換える方法が必要である。これは、変更されうるプラグマ付けの全方法が前以ってわかっているなら、ガードにより行える。

それには、プラグマ付けだけが異なるクローズを複数個用意し、ガード部には入力データの性質などを調べる述語を加えて、望むプラグマ付けを持ったクローズを選択させる。これらクローズは論理的な意味は同じでも、実行性能に関しては違う意味を持つ。プラグマ付けの動的変更を行うための、有用な一群のガード述語を発明することが課題である。入力データの性質を調べる他に、プロセッサに関する情報（負荷量、機能、ネットワーク座標）なども考慮の対象にできる。例で示そう。

```
do(X) :- pe(load,heavy),size(X,N) |
         process(X,A) @ forward(N), output(A?).
do(X) :- pe(load,light) |
         process(X,A) ,                output(A?).
```

では、現在のプロセッサの負荷状況で分散戦略を切り換えている。pe(Item,Value)がゴールを展開中のプロセッサに関する情報を調べる述語である。また、

```
output(A) :- pe(tty,has)    | write(A).
output(A) :- pe(tty,hasnot) | output(A)@forward.
```

では、現在のプロセッサがＩＯ機能を備えていなければ転送している。この方針で、PARLOGにリアルタイム機能を持たせようとする研究もある〔4〕。例えば、

```
output(A) :- Limit is 20 sec, after(Limit) | true.
```

では、after(Time) は20秒後に真となり、選択された場合出力動作をキャンセルする。

5．ポリシーの設計

プラグマは直接クローズのテキスト記述の一部として書き込まれるので、分散戦略の決定がクローズに散在してしまう上に、実行よりかなり以前に決定してしまわなくてはならない。簡単に体系的に書き直す目的で、一種の構造エディタと言えるプラグマエディタを設計することも一法であるが、戦略決定を急ぐ点には変わりがない。

5-1　ポリシーの仕様

この解決には、判断を保留できるようにクローズに出てくるプラグマを変数化すれば良い。始めにプログラムはプラグマ変数を持つ雛形のクローズとして作成する。後にプラグマ変数に具体値を設定して、より分散戦略のはっきりしたクローズに具体化する。プラグマ変数をどう具体化するかの指示をまとめたものが、ポリシーというわけである。もちろんプラグマをすべて変数化する必要はない。問題を解く上でそうである方が良いとわかっているプラグマは直接雛形クローズに書き込む。例えば「律速段階」のゴールの優先度は常に高いままに固定した方がよいと考えるなら、変数化はしない。

ゴールの配置と変数セルの配置については、Shapiro のタートル方式をそのまま用いればよい。新ゴールと変数セルごとに、特定のゴール位置からの相対位置をタートルコマンドを使って指定する。例示すれば、

〔right,forward(2)〕

なるタートルコマンドは、ゴールあるいは変数セルを右二つめのプロセッサに割り付けよの指示である。この指定法であると、変数セルを新ゴールとは独立に割り付けられる。ゴールの優先度についてもタートルコマンドで、

〔urgent〕

のように指定できる。データ駆動投入やガード部述語による場合分けの指定についても、タートルコマンドを拡張する形で導入可能であるが、工夫の余地が残る。

5-2　ポリシーの汎用化

プラグマ変数の名前とその具体値をペアにして並べたままでは、ポリシーとクローズは独立したものにならない。独立度を高めるために、クローズ側ではプラグマ変数に順位をつけて登録しておき、ポリシーでは「n番めのプラグマ変数にこの値を設定せよ」という形で相対位置指定する。これはちょうど関数呼び出しにおいて、実引数と仮引数の対応を引数位置で付けるのと同じであり、組み合わせの汎用化が図れる。約束事として、クローズ側では分散するのが重要であると考えられる順にプラグマ変数を並べておく。ポリシー側でもそれを考慮して具体値を供給する。ただし相対化により、汎用化と引き替えに精密性を失ったわけで、相対指定法の妥当性は検討を要する。

```
apply(Policy,Clauses,NewClauses):-
  % ポリシーはタートルコマンドの並び。
  % クローズのプラグマ変数をタートルコマンドで、
  % 順に埋めて行く。
```

5-3　ポリシーへの操作の汎用化

ポリシーはタートルコマンドの並びであるから、考えられる操作はコマンドを系統的に書き直す操作であろう。ポリシーに操作を加えて希望の性能になるように調整するという趣旨から言うと、予測できない性能変化を生じる書換えは意味がない。差し当たっては、コマンドのright とleftを一斉に交換するとか、forward(N)の引数N を一斉に整数倍するなどの操作を思い付く。これらの操作を施せば、ゴール分布を裏返したり、拡大したりできる。

```
transform(Policy,Operations,NewPolicy):-
  % Operationsの指示でポリシーを系統的に変更する。
  % Operationsの例: [exchange(right,left)]
```

III　タスクミックスをじょうずに行うシェル

6．ポリシーからの性能予測

タスクミックスをうまく行おうとするなら、これから実行しようとするタスクの実行に関するもろもろの性能が、ある程度正しく予測できなくてはならない。予測情報を元にタスクを組み合わせるからである。これは疑いもなく難しい。前回までの実行のログから性能を推測するとか、数理生物学の手法を援用するなどを思い付くが、予測問題については将来への課題としたい。

6-1　実行性能のプラグマ化

予測の困難を避けるためには、性能を前以て型付けて分類しておき、プログラマに一つを選んでもらう。何のことはなく、実行性能に関するプラグマを新設したことに他ならない。一見すると問題をポリシーからプラグマへ戻したようにも思えるが、従来のプラグマより抽象化の進んだ高次のプラグマになっているから、色々な応用が開ける可能性もあるわけで、タスクミックスはその一例である。

実行性能といっても色々である。ゴール分布、変数セル分布、律速ゴールの推定などがある。以下ではタスクミックスを行うのに最小限必要なゴール分布のみを考えることとし、ゴール分布の形（半直線状、円状など）を指定する実行性能プラグマを「実行パタン」と呼ぶ。プログラマは、プログラムとポリシーを組み合わせる際に、同時に実行パタンを指定しなければならない。

7．タスクミックスシェル

ポリシーの応用として、タスクミックスをじょうずに行うシェルの概要を示す。シェルは、オペレーティングシステムのユーザインターフェスである。

7-1　簡単なシェル

簡単なシェルは次のように書ける。run コマンドはゴールを投入する。loadコマンドはクローズを登録する。

```
shell( [] ).
shell( [run(Goal) | Cmds] ) :-
  call(Goal),shell(Cmds?).
shell( [load(Clause) | Cmds] ) :-
  assert(Clause) | shell(Cmds?).
```

並列マシンでは、プログラムの各プロセッサへのロードの問題は課題とされつつも考慮されて来なかった。しかし、マルチタスクを扱う並列オペレーティングシステムでは無視できない問題である。先に掲げたシェルにロード機能を加えたのは、この点を強調したいがためである。

プログラムロードの実現には、簡単には、すべてのプロセッサにあらかじめプログラムをロードしておく方法か（prefetch）、必要になることがわかった時点でロードする方法（on-demand fetch)がある。前者ではゴール分布には偏りがあるため、結局は必要なかったプロセッサにまでロードしてしまう率が無視できないし、後者ではプロセス単位が小さいために、参照の局所性が保てないと実行時間は大幅に延びてしまう。

7-2　タスクミックスシェル

分散環境の下でタスクミックスを行うシェルの概要を示す。ひとつの実装の形で示すが、言うまでもなく大事なのは考え方であり、詳細は本質ではない。

```
shell( [] ).
```

(a)各種情報のリンク

```
shell( [link(Module,
          [Procedure,Policy,Pattern] ) | Cmds] ):-
  record(Module, [Procedure,Policy,Pattern] ) |
  shell(Cmds?).
```

リンク命令はモジュール名(Module)の名の下に、プログラムクローズ(Procedure)、ポリシー(Policy)、実行パタン(Pattern)の三組を記憶する。記憶はrecordが行う。この段階ではまだポリシーはクローズには適用されない。

(b)プログラムのロード

```
shell( [load(Module,Status) | Cmds] ) :-
  tactics(Module,Status,Policy),
  clauses(Module,Clauses),
  apply(Policy?,Clauses?,AppliedClauses),
  load(AppliedClauses?) | shell(Cmds?).
```

ロード命令は、プログラムクローズを各プロセッサにロードする。この過程でタスクミックスの戦略をtactics が練り、ゴールの投入情報(Status)と新ポリシー(Policy)を計算する。次いで、clauses が複製したプログラムクローズにapply が新ポリシーを適用しプラグマ付クローズを得、loadがロードするという段取である。

(c)ゴールの実行

```
shell( [run(Goal,Status) | Cmds] ) :-
      compose(Goal,Status,ReadyGoal),
      call(ReadyGoal?), shell(Cmds?).
```

ゴール(Goal)を、実行情報(Status)を基に実行する。実行情報には少なくとも、このゴールを投入するプロセッサの位置指定、タートルの向きが含まれている。ゴールへの実行情報の埋めこみはcompose が行い、callが実行する。

(d)ロード&実行

```
shell( [task(Module,Goal) | Cmds] ) :-
  shell( [load(Module,Status),
          run(Goal,Status?) | Cmds? ] ).
```

通常、ロードと実行は続けて行う。タスク命令は、便宜的な命令である。

(e)タスクミックスの計算

```
tactics(Module,NewStatus,NewPolicy):-
    lock(taskmix,TaskMix),
    presume(Module,Pattern),
    plan(TaskMix?,NewTaskMix,Pattern?,NewPattern),
    generate(NewPattern?,NewStatus,Operations),
    transform(Module,Operations?,NewPolicy)  |
    unlock(taskmix,NewTaskMix?).
```

ここに、lockとunlockはグローバル変数TaskMix を排他制御する。presume はリンク命令で記憶した実行パタンを引き出して来る。（本当は、この段階でプログラムクローズとポリシーから実行パタンを算出したい。）

planがタスクミックスの調整を行う。与えられたパタンに手を加え、タスクの混みぐあいをプロセッサに均等に割り振るように調節する。例えば、直線状に伸びる二つのタスクの実行パタンは並行に伸びるように、三角形に広がる二つのタスクの実行パタンは向い合うように配置するなどする。補助述語generateは、新しく割り振られたパタンが実行パタンとなるように、実行情報(Status)とポリシーへの操作(Operations)を割り出す。次いでtransform が、ポリシーを操作して新しいポリシーを求める。

7-3　タスクミックスを考慮したプログラムロード

実行パタンが与えられたなら、プログラムのロードも効率的にできる。実行パタンによってゴールが渡りそうなプロセッサだけにロードしておき、他は必要になった時点でロードする方法を取る。この方法によれば、タスクミックスシェルは副次的に、プログラムメモリの使用効率の改善も果たす。

さらに各プロセッサにロードする過程で、ガード部の述語で静的に評価できるものはしてしまう。例えば、4.3 に出てきた I/O機能を調べるpe(tty,-) が好例である。結局、loadには新たに、実行パタンも引数として与える。

```
load(Pattern,Clauses):-
    % Pattern に載る全てのプロセッサに、Clauses を
    % ロードする。このとき静的に評価できるガード部
    % 述語の評価もしてしまう。
```

8．プラグマをめぐる課題

プラグマ機能に関しては他にも、いろいろ課題が残っている。プラグマデバッガの開発や、プラグマ付けのヒューリスティックスの開発などの話である。プラグマが備わっていたとしても、使いこなす技術が難しければ誰も使わない。プラグマが付いてはいても期待する実行性能が出ないならば、バグである。プラグマデバッガは、プラグマをいろいろ変更してみて、性能をチューニングするために使う道具である。チューニングを早く終えるためには、最初にある程度良いプラグマ付けを施しておく必要があるが、ヒューリスティックスはこれに役立つ。

また、オペレーティングシステムと物理マシンの間に仮想マシンを入れると、より高度な機能を盛り込める可能性が出てくる。保護機構、信頼性向上などの機能である。仮想マシンについては Shapiroらや近山の研究がある。

Shapiro らの方法は、仮想マシンと物理マシンの対応をプログラムで付ける〔9〕。物理マシンを底に、仮想マシンは何段でも重ねられる。ユーザは解くべき問題に最適と思われる仮想マシンを選択し、その上でプログラミングする。仮想マシンの形でマシンレベル間の対応付けがパックされ、ユーザが希望のものを選択する点は、我々のポリシーの考えと相通ずる処である。近山の方法は、Shapiro のプラグマの拡張である〔1〕。ゴールには、割り当てプロセッサ、タートルの向きに加えて、版図（勢力範囲）なる属性が与えられる。実行が進むにつれ各ゴールは世界（プロセッサ群）の覇権を争い、版図に分割していく。物理マシンは位置関係が世界地図に相似になるようにゴールを物理プロセッサに割り当てるが、その際に各物理プロセッサの負荷が一様になるようにも努める。

9．まとめ

並列論理型言語のプラグマ機能について述べた。並列オペレーティングシステムを含むシステム記述言語とするためには、プラグマをポリシーとしてクローズから分離させるべきとし、応用としてマルチタスク下でのタスクミックスを行う分散シェルの概略を示した。

現在、CPのインタプリタ（6）を下敷きに、マルチタスクの分散環境をシミュレートする分散版CPインタプリタを作成し、以上に述べた仕様に「準ずる」タスクミックスシェルが曲がりなりにも動いている。しかし、速度の点でも完成度の点でも満足の行くものではなく、一層の改良が必要である。他にも、ポリシーの仕様の洗練、性能の予測やタスクミックスの高度化など残した課題は多い。

謝辞

本研究は、第５世代コンピュータ・プロジェクトの一環として実施中です。研究の機会を提供して頂いた、富士通国際研の北川会長ならびに榎本所長、ICOT第１研究室に深謝いたします。

参考文献

〔1〕 Chikayama, T.: Load Balancing in Very Large Multi-Processor Systems, In Proc. of 4th Japanese-Swedish Workshop on FGCS, 1986.

〔2〕 Clark, K. and Gregory S.: Notes on Systems Programming in PARLOG, In Proc. of Int. Conf. on FGCS, 1985, pp.299-306.

〔3〕 Clark, K. and Gregory S.: PARLOG: Parallel Programming in Logic, ACM Trans. Program. Lang. and Syst., Vol.8, No.1(1986), pp.1-49.

〔4〕 Bishiewy, N.: Extended PARLOG: Logic Programming of Real Time Systems, In Proc. of 4th Japanese-Swedish Workshop on FGCS, 1986.

〔5〕 大原有理、鳥居悟、小野越夫、岸下誠、田中二郎、宮崎敏彦: FGHCソフトウェアシミュレータの試作、the Logic Program. Conf.'86,1986,pp.93-101.

〔6〕 Shapiro, E.: A subset of Concurrent Prolog and its Interpreter, Tech. rep. TR-003,ICOT,1983.

〔7〕 Shapiro, E.: Systolic Programming: A Paradigm of Parallel Processing, In Proc. of Int. Conf. on FGCS, 1985, pp.458-470.

〔8〕 Tamaki, H: A Distributed Unification Scheme for Systolic Logic Programs, In Proc. of Int. Conf. on Parall. Process., 1985, pp.552-559.

〔9〕 Taylor, S.,Av-Ron, E. and Shapiro, E.: A Layered Method for Process and Code Mapping, Dept. of Comp. Sc.,Weizmann Inst. of Sc., Rehovot Israel,1985.

〔10〕 鳥居　悟、大原有理、小野越夫: 核言語KL1 分散処理系－ソフトウェアシミュレータによる評価－、情報処理学会第33回全国大会論文集、1986, 6D-1.

〔11〕 Ueda, K.: Guarded Horn Clauses, Tech. rep. TR-103, ICOT,1985.

〔12〕 Wulf, W. et al.: HYDRA: The Kernel of a Multi-processor Operating System, Comm. ACM, vol.17, no.6(1974), pp.337-345.