TM-1044

VPIM処理方式解説書

今井　明

May, 1991

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03)3456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

VPIM 処理方式解説書

1991年 4月 5日

(財) 新世代コンピュータ技術開発機構

第一研究室 編

目次

はじめに

VPIMの開発は昭和63年夏頃から始まり、すでに二年半が経過した。その間、VPIM自身の開発は言うまでもなく、それと並行して、本資料に含まれていない、(1) PSLの言語仕様の設計、(2) VPIMの開発環境の整備、(3) Symmetry (汎用並列UNIXマシン) でのVPIM仕様確認方式の開発、(4) KL1コンパイラ/アセンブラ/リンカの開発、(5) PSLで書いたVPIMを実機機械語へ変換する方式の開発などを行ないながら、ようやく平成3年2月に PIMOS を動かすための、ひと通りの機能を盛り込んだVPIMバージョン1.0 をリリースすることができた。

このバージョン1.0のリリースの機会に、平成3年2月28日より3月2日まで、湯河原において、リリース説明およびVPIM ソース読み合わせの合宿を行なった。本資料は、この合宿のために作成した解説書を元に、その合宿での議論を反映させて編集し直したものである。

VPIMにおけるKL1処理方式は、VPIM仕様記述言語PSLを用いて記述しているが、本資料は、PSLの文法に馴染みのない人にも理解できるように、できるだけPSLに依存しない形で記述したつもりである。ただし、KL1言語の文法に関する説明を省略しているので、KL1を使ってプログラムを書いた経験がないと、内容を理解することが困難かもしれない。また、本資料では、KL1プログラムの KL1-B へのコンパイル方式についてはあまり深く触れないで、KL1-B 命令の解釈実行に重点を置いて説明する。

本資料の構成は以下の通りである。まず 第1章で抽象機械の構成と仮想ハードウェアのアーキテクチャを説明し、第2章で VPIM で用いているデータ構造について述べる。第3章では、KL1-B 命令列を格納したコードモジュールの構成と、KL1-B命令の一覧を示す。第4章では、VPIM のメモリ管理方式および即時GC方式であるMRB方式について述べる。第5章では、複雑な処理を一度KL1ゴールの形にして処理を単純化させる機構であるDコードについて述べる。第6章では、KL1ゴールのスケジューリングやクラスタ内での自動負荷分散方式について、第7章では、論理型言語の基本操作であるユニフィケーションや、マージ組込述語についてそれぞれ述べる。第8章では、KL1の並列実行過程における多種多様なイベント処理を行なうスリットチェックという機構について説明する。第9章では、クラスタ間のデータ参照方式、メッセージ処理方式について述べ、第10章では、荘園というKL1ゴール群のメタ管理機構について説明する。第11章では、不要なメモリ領域を回収するガーベジコレクション(GC)について、第12章では、KL1プログラムを開発するユーザ向けにサポートした永久中断ゴール報告やトレース・スパイといった機能について解説する。最後に、第13では、ディスクやFEP (フロントエンドプロセッサ)を操作するためのSCSI (Small Computer System Interface)のバス制御やそれを操作するKL1組込述語の実装方式について説明する。

なお、本資料中に不明な点や疑問な点があった場合は、今後の本資料の改定およびVPIM処理方式の

改良の参考にさせて頂きたいので、

vpim@icot.or.jp

まで、メイルを頂きたい。

本資料が、VPIMにおけるKL1言語処理方式を理解するうえで参考になれば光栄である。

1991年4月5日

VPIM処理方式解説書 執筆者一同

第 1 章

VPIM の全体構成概要

執筆担当者: 今井

1.1 概要

本資料は、第五世代コンピュータ計画のプロトタイプハードウェアシステムである並列推論マシン **PIM** (Parallel Inference Machine) の核言語**KL1** (Kernel Language version 1)の実行方式を解説したものである。

PIM の KL1 処理系開発に当たっては、共有メモリ結合部を持つ複数のハードウェアサブモジュールに共通なKL1言語実行方式の仕様となる抽象機械 **VPIM** [1] (Virtual PIM) を開発し、それを実際のハードウェアが持つ命令に変換することとした。そのため、本資料は、このVPIMのKL1実行方式を解説したものになっている。またこのVPIMにおけるKL1処理方式は、VPIM仕様記述言語**PSL** (PIM System descriptive Language)を用いて記述している。

PIM 上のオペレーティングシステム **PIMOS** (Parallel Inference Machine Operating System) は、すべてこのKL1言語を用いて記述されている。そのため、通常の計算機ではオペレーティングシステムで行なわれる「メモリ管理」や「優先度制御」などが VPIM のレベルで実現されている。

1.2 KL1 の実行メカニズム

KL1 のプログラムは、次の形をしたガードつきホーン節の集合である。

$$H \ :- \ G_1,\ G_2,\ \ldots,\ G_m \mid B_1,\ B_2,\ \ldots,\ B_n.(m \geq 0, n \geq 0)$$

ここで、H をヘッド、G_1 から G_m をガードゴール、B_1 から B_n をボディゴールと呼ぶ。'|' をコミットメントバーと呼びそれより左をガード部 (または、受動部)、右をボディ部 (または、能動部) と呼ぶ。

図1-1のように、KL1プログラムはまず抽象機械語**KL1-B**にコンパイルされ、その KL1-B 命令は、

[1] 歴史的理由から Virtual PIM と呼んでいるが、その意味するところは、KL1-B を解釈実行する抽象機械 (Abstract Machine)である。抽象機械VPIMは、ある仮想マシン(Virtual Machine)上のKL1-Bを処理する仕組みを含めたシステムを指すことにする。

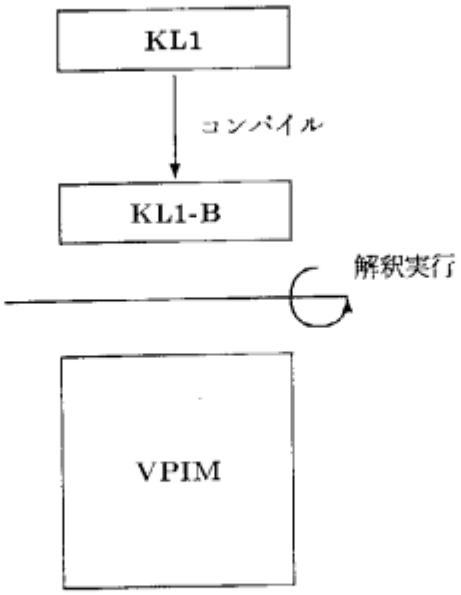


図 1-1: KL1 の実行メカニズム

VPIMによって解釈実行される。KL1-B は、Prolog の WAM (Warren Abstract Machine) に相当するものである。

KL1 の実行は、ヘッドを、ガードゴールおよびボディゴールに置き替えるゴールリダクションである。これは次のようなサイクルで行なわれる。

1. 実行可能なゴールを置いたレディゴールスタックから、ゴールを一つ取り出す。ゴールはゴールレコードという構造に、引数情報などが格納されている。
2. 能動部を実行する節を一つに決定するために、候補節の受動部をテストする。この時読みだした値が具体化していない時は、中断原因になった変数をスタックに積み、次の候補節を試みる。
   すべての候補節の受動部が成功できず、かつ中断する候補節が存在する場合、ゴールの実行を中断し、中断原因の変数が具体化されるまで待ち合わせる。この中断の処理をサスペンド処理と呼び、待ち合わせのためにその変数にゴールをフックする。
3. 受動部のテストに成功した(コミットしたと呼ぶ)節の能動部を実行する。能動部では、新たなゴールが生成される時そのゴールの引数などの環境情報をゴールレコードに書き込み、レディゴールスタックに入れる。
   待ち合わせゴールのある変数に対して、能動部のユニフィケーションが行なわれると、その待ち合わせゴールをレディゴールスタックに入れる。

KL1の実行に必要なデータには、変数、リスト、ゴールレコード、KL1-Bコードモジュールなどがあるが(2章参照)、これらのデータに必要なメモリ領域は、すべてヒープ領域 に確保される。

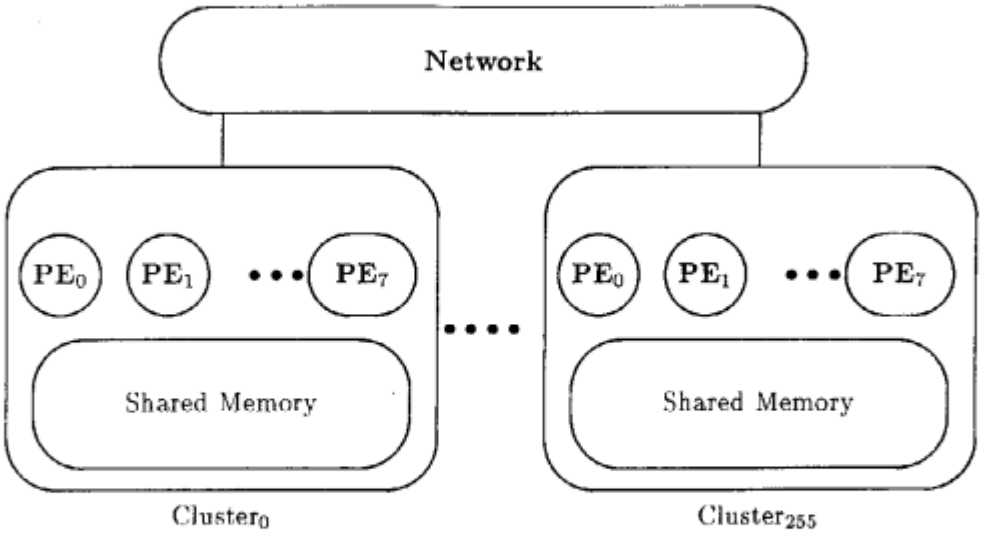


図 1–2: 仮想マシンのアーキテクチャ

1.3 VPIM のアーキテクチャ

1.3.1 アーキテクチャ概要

VPIM が仮定している仮想マシンのアーキテクチャの全体構成を図1–2に示す。 VPIM は、ネットワークによって結合された多数台の**クラスタ** で構成される。各クラスタは、共有バス / 共有メモリによって密に結合された8台程度の要素プロセッサ(**PE**: Processing Element) からなる。

各クラスタは、クラスタ内の全てのPEがメモリを共有する密結合マルチプロセッサ構成となっている。一方、クラスタ間に渡る分散ユニフィケーションはネットワークを介した非同期メッセージ通信によって行う。各クラスタが1PEで構成されると仮定すると、クラスタ間の並列処理は、PIM に先だって開発されたMulti-PSIのような、疎結合マルチプロセッサの並列処理であるとみなすことができる。

VPIM に含まれるクラスタおよびクラスタ内のPEは、

- クラスタ番号 (8ビット: 0 ～ 255)
- クラスタ内PE番号 (3ビット: 0 ～ 7)

を用いて、システム内の全てのPEがユニークに識別できるものとする。クラスタ番号、およびクラスタ内PE番号は、クラスタ間通信やPE間通信において通信相手を特定するために使用するものである。

1.3.2 アドレス空間とタグワード

VPIM としてのアドレス空間は、32ビットのクラスタ内共有メモリ空間のみを考える。PE毎の局所メモリは特に想定せず、PE毎に用いる局所的なメモリ空間はクラスタの共有メモリに確保できるものとする。PIMの実機が局所メモリを持つ場合は、適宜利用することにする。

VPIMのPEは、図1–3のタグワードを処理の基本単位とするタグアーキテクチャを想定する。タグ部

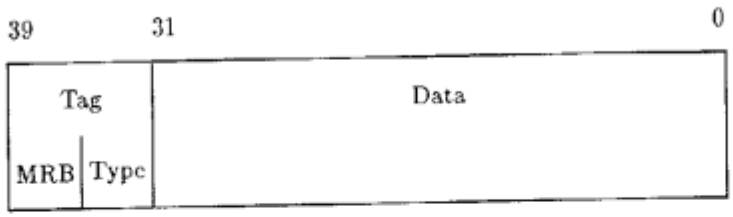


図 1-3: タグワード

は、MRB(4.4節を参照: 1ビット)とデータタイプ(6ビット)からなる[2]。本資料では、データタイプに関しては、そのデータをわかりやすく表現するために、データタイプに関して特定の名前を付けている(2.2節を参照)。

後述の排他的メモリアクセス命令等では、このタグワード単位の読み書きがアトミックな(不可分の)操作として実行できる。

1.3.3 PE 内のレジスタ

VPIMのPEは、下記のようなレジスタを持つものとする。

- 引数レジスタ:
  - KL1コンパイラがKL1-B命令の引数用に割り当てるタグワード単位のデータレジスタ
  - 個数としては、5個(もっとも多く引数を持つ組込述語の引数個数)以上の適当数持つものとする。
- ワークレジスタ:
  - KL1-B命令の実行のために使用するタグワード単位のデータレジスタである。
  - 仮想マシンとしては、十分な数のワークレジスタが使用可能とする。これらのワークレジスタはPIMの実機では個数が限られるため、実機コードへの変換に際して必要に応じて割り当てるものとする。
- 定数レジスタ:
  - ナルレジスタ:読みだし専用であり常にオールゼロが読める
  - ボイドレジスタ:書き込み専用であり、演算結果を捨てるために用いる
  - PE番号レジスタ:PE番号を示すレジスタ(初期化時に設定)
  - クラスタ番号レジスタ:クラスタ番号を示すレジスタ(初期化時に設定)
  - etc.
- 専用レジスタ:
  - プログラムカウンタ (PC)
  - コンディションコードレジスタ (CCR)

[2] 8ビットのタグを持つマシンでは、使われないビットが1ビット残されているので、計測などの目的に自由に使うことができる。

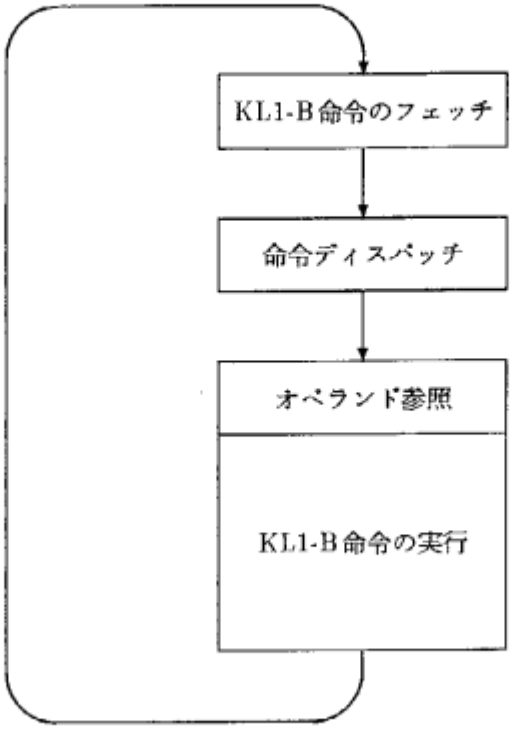


図 1-4: PE の命令実行

◇ スリットチェックレジスタ (SCR)

1.3.4 PE の基本操作

PEの実行は、通常の計算機と同様に、図1-4に示す命令のフェッチ/実行を繰り返す。 ただし、対象とする命令は、機械語KL1-Bの命令であり、タグアーキテクチャによる命令の実行と並列処理のためのPE間通信処理を含む点が通常とは異なる。

KL1-B命令は、レジスタまたは即値定数を引数とする抽象機械語命令であり、命令のオペランドの参照およびそれに従った命令の実行は、実機PIMのマイクロ命令操作に相当する以下のような基本操作の列として定義できる。

- (引数, ワーク, 定数) レジスタ間のデータ (タグワード, データ部, タグ部, MRB) の移動
- (引数, ワーク) レジスタ間のデータ (タグワード, データ部, タグ部, MRB) への即値の設定
- レジスタによってアドレスを指定した共有メモリの読みだし、書き込み
- レジスタおよび即値定数を引数とした算術論理演算
- レジスタ中のタグ部(Type, MRB) データ部の値判定に基づく基本操作列の制御のスイッチ
- KL1-B命令のためのプログラムカウンタ (PC) の操作
- クラスタ内でのPE間通信シグナルの送信およびスリットチェックレジスタの読みだし
- ネットワーク経由のデータの送受信のための操作
- SCSIバスの制御、SCSIバス経由のデータの送受信のための操作

表 1-1: 共有メモリアクセス操作

アクセス操作	説明
read_invalidate	キャッシュにミスし、他のキャッシュからデータを読み出す場合、ソースとなるキャッシュブロックを無効化する。それ以外は通常の読みだし操作。
read_purge	読み出した後、そのデータを含む自分(および他のキャッシュ)のキャッシュブロックを無効化する。
exclusive_read	キャッシュブロックの最後のワードについては read_purge と同じで、それ以外については read_invalidate と同じ。
direct_write	書き込み操作に自分のキャッシュにミスした時、それがブロックの先頭のワードの場合は、共有メモリや他のキャッシュからブロックをフェッチせずに、自分のキャッシュにブロックを割り当てる。先頭のワード以外は、通常の書き込みと同じ。
lock_read	指定されたメモリ番地へのアクセスをロックして読み出す。ロックが競合した場合は、解除されるまで待ってから上記を行う。
write_unlock	データをメモリに書き込んでからロックを解除する。
unlock	ロックを解除する。

これらの操作は、PSLのプリミティブとして用意されている。

仮想マシンとしては、KL1-B 命令の種類数、引数の数には制限はなく、3.2節で述べる KL1-B 命令を幾つかの単純化したものに分解したような命令セットや、幾つかの KL1-B 命令を複合したような命令セットについてもその動作を定義して実行することができる。ただし、前述(3) のように、命令のレジスタ引数個数は仮想マシンの引数レジスタ数以下である必要がある。

1.3.5 共有メモリアクセス

VPIMには、共有メモリ結合のマルチプロセッサに必要な排他アクセス操作を基本操作として備えている。これ加えて、実際のハードウェアにおいて一貫性キャッシュメモリを用いた場合、共有バスのデータ転送トラヒックを抑えるためのメモリアクセス操作を持つ。(表1-1参照)

上記の操作の内、lock_read、write_unlock、unlock 以外は全て最適化のための操作であり、これらの操作を持たないPIMにおいては通常の read/write に置き換えてよい。

排他アクセスは、ハードウェア実装を考慮し、同時に2個所以上のロックを許さず、そのロック期間に実行される操作は、レジスタ演算程度のものに制限している。このため、VPIM が持つ排他アクセス操作は、Compare & Swapに類する排他制御操作を実現するためのプリミティブとして用いる。

一方、荘園/里親レコード(10.5節参照)や輸出入表(9.1節参照)のように、共有データとしてアクセスされるリンク構造などを操作する場合は、複数個所の同時ロックが必要となる。そのような場合は、前

述の排他アクセス操作を用いて、データ構造の一部を利用したロック操作(**ソフトウェアロック** ないし **ソフトロック** と呼ぶ)を行うこととする。

1.3.6 スリットチェック

KL1の並列処理の過程で生じた事象の内，処理系が意図的に行ったもの(PE間のシグナル通信) や，PIMOS との橋渡しに必要なものは，KL1 プログラムとしての処理単位で検出して対応する必要がある。このため，KL1 の処理系においては、リダクションの切れ目で事象を検知し，その処理ルーチンに飛ぶことができる機構として**スリットチェック** 機構を用いる。

VPIMにおいては、スリットチェックに必要な要因レジスタである**スリットチェックレジスタ** (SCR: Slit Check Register)と要因の設定、検出に必要な基本操作を持つ。仮想マシンにおける割り込み / 割り出しに相当する操作は、全てスリットチェックの形態をとる。SCRは一種の割り込み要因フラグの集合(10ないし12個程度)であり、

- 外部からの割り込み(ネットワークからの受信)
- クラスタ内の他のPEからの割り込み
- KL1-B命令実行中における割り出し

等の意味を与えることができる。

1.3.7 クラスタ間ネットワーク

仮想マシンにおけるクラスタ間ネットワークは、各クラスタにおける同一のPE番号を持つPE同士を結ぶ完全結合(すべてのクラスタ同士が接続されている)のネットワークとなっている。つまり、各PEはネットワークを介したデータ送受信のためのポートを持っており、自分と同一PE番号を持つPE間で自由にデータの送受が可能である。

実際のデータの送受信は、タグワード単位でレジスタの内容を送り出す、またはデータを受信する基本操作として用意されている。ネットワーク自体が無限長のバッファを持っており、メッセージ長に制限は無い。また、ネットワークの各リンクは追い越しの無いFIFO(First In First Out:先入れ先出し)となっており、相手先が同一ならば送信順に受信できる。ただし、メッセージ通信の主体であるクラスタ間のリンクが複数あるため、どのリンクを経由したか、または、どのPEが送受したかによって、受信処理されるメッセージとしては順番が保証されない。このため、クラスタ間処理においては、メッセージ間での**追い越し** がありうると想定し、その対策を施している。

第 2 章

データ構造

執筆担当者: 畑澤

本章では、VPIMにおけるデータ構造について説明する。

2.1 データの基本単位

VPIMにおけるデータの基本単位はタグ付きワードである。タグ付きワードは、タグ(Tag)部 8 ビットと値(Value)部 32 ビットのデータから成っている。

この内タグ部は、

Reserved Bit	1bit
MRB	1bit
Type	6bits

で構成されている。MRBは原則としてValue部がポインタの場合にその先のデータが単一参照であるか、多重参照であるかを区別するのに使用する。Typeはデータの種類を表す。

以下では、タグ付きワードを単に'ワード'と呼び、図2-1に示すような形式で表す。

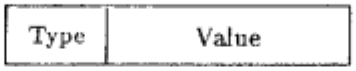


図 2-1: タグ付きワード

2.2 タイプ一覧

表2-1および表2-2に、VPIMにおけるタイプの一覧を示す。種類は、Value部の意味を表し、Pがポインタ、Oがオブジェクト(データの本体)、DCは don't care(Value部は未使用であり、値は不定)である。

表 2 1: VPIM におけるタイプ一覧 (その 1)

タイプ	種類	説明	参照箇所
EOL	DC	リンクの終端	
FLC	P	フリーリストセル	
SLOCK	O	ソフトウェアロック	9章
MARKED	P,O	GC時に使用	11章
ATOM	O	アトム	
INT	O	整数	
FLT	P	浮動小数点数	
LIST	P	リスト	
STRG	P	ストリング	
VECT0	DC	0要素ベクタ	
VECT1	P	1要素ベクタ	
VECT2	P	2要素ベクタ	
VECT3	P	3要素ベクタ	
VECT4	P	4要素ベクタ	
VECT5	P	5要素ベクタ	
VECT6	P	6要素ベクタ	
VECT7	P	7要素ベクタ	
VECT8	P	8要素ベクタ	
VECT	P	ロングベクタ	
REF	P	参照ポインタ	
DREF	P	サスペンドフラグへのポインタ (MRB メンテナンス不要)	7.3節
VOID	DC	void変数 (必ず REF → VOID となる)	
UNDF	DC	未定義変数 (必ず REF → UNDF となる)	

表 2-2: VPIM におけるタイプ一覧 (その 2)

タイプ	種類	説明	参照箇所
HOOK	P	ゴールレコードへのポインタ (single-HOOK)	7.3節
MHOOK	P	サスペンドレコードへのポインタ (multiple-HOOK)	7.3節
RHOOK	P	リプライフックレコードへのポインタ (reply-HOOK)	9章
EUNDF	DC	輸出されたUNDF	9章
EHOOK	P	輸出されたHOOK	9章
EMHOK	P	輸出されたMHOOK	9章
WEXREF	P	白輸入表への外部参照ポインタ	9章
BEXREF	P	黒輸入表への外部参照ポインタ	9章
WEXVAL	P	白輸入表への外部参照ポインタ (輸出側具体化済み)	9章
BEXVAL	P	黒輸入表への外部参照ポインタ (輸出側具体化済み)	9章
RDHOK	P	ReadHook レコードへのポインタ	9章
EXLOCK	O	外部参照ポインタのロック	9章
MGHOK	P	マージャセルへのポインタ (merge-HOOK)	7.6節
WEXMRF		(現在未使用)	
SHREC	P	荘園レコードへのポインタ	10章
FPREC	P	里親レコードへのポインタ	10章
CDESC	O	GC時のメンテナンス不要領域前につけるディスクリプタ	11章
COD	P	コード	3章
MOD	P	モジュール	3章
GOAL	O	ゴールレコード(アリティ), マージャレコード(参照数)で使用	11章
VISIT	O	永久中断ゴール検出のためGC時に使用	11章
DNTC	O	タグ部を使わないデータ	
HLINK	P	ハッシュ表へのポインタ (黒輸出入表で使用, INTで代用可)	11章

2.3 KL1 基本データの構造

以下に、VPIMにおける基本的なKL1データの構造について説明する。

(1) アトム

定数データ。Type は ATOM。Value はアトム番号。(図2-2)

アトム番号は、32ビットを上位8ビットと下位24ビットに分け、

上位8ビット　クラスタ番号またはOS予約アトムを意味する値(0xFF)

下位24ビット　クラスタローカルなアトム番号(1〜)

として使う。特別な場合として、[] はアトム番号 0x00000000 とする。これによって、組込述語 new_atom を使用して各クラスタで生成するアトム番号がシステム全体でユニークであることを保証している。[1]

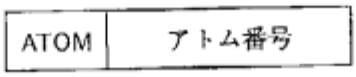


図 2 2: アトム

(2) 整数

符号ビットを含む 32ビットの整数。[2] Type は INT。Value は整数値。(図2-3)

図 2-3: 整数

(3) 浮動小数点数

浮動小数点数。Type は FLT。Value はデータ本体へのポインタ。データ本体は、2ワードの整数からなる。(図2-4)

データ本体には、倍精度浮動小数点数を 64 ビットの IEEE フォーマット (ANSI/IEEE Standard 754-1985) で表し、これを上位32ビットおよび下位32ビットに分けて、それぞれを整数として格納する。[3]

[1] アトム番号のビット数、ビットフィールドの割り付け方は、各実機処理系において必ずしもVPIMと同じにする必要はな

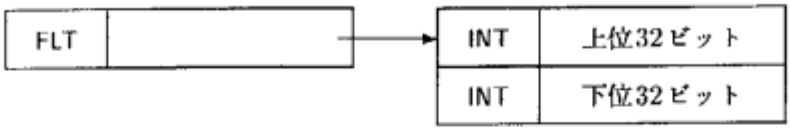


図 2–4: 浮動小数点数

(4) ストリング

要素数 0 以上の符号無し整数から成る 1 次元の配列。要素サイズは 1, 8, 16, 32 ビットのいずれかの幅を選ぶことができる。Type は STRG。Value はストリング本体へのポインタ。ストリング本体は、先頭にディスクリプタを含む1つ以上のワードで構成する。(図2–5)

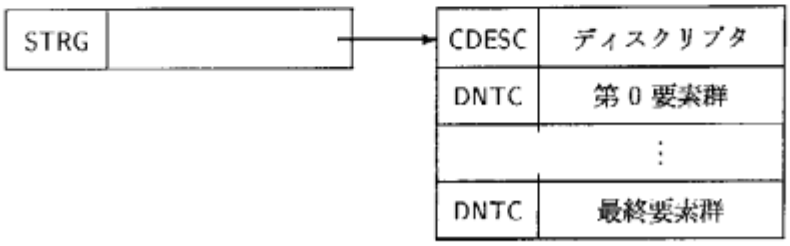


図 2–5: ストリング

ディスクリプタは、Type が CDESC、Value にはストリング・ディスクリプタを、

上位8ビット　ストリング・タイプ

下位24ビット　ストリング要素のワード数

のように格納する。(図2–6)

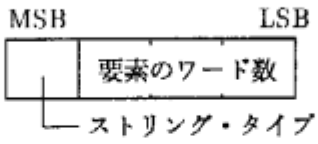


図 2–6: ストリング・ディスクリプタ

ストリング・タイプは、次のように8ビットのうち上位3ビットで要素サイズを、下位5ビットで最終ワード中の未使用要素数を表す。

い。

[2]整数は、必ず32ビットであることがPIMOS側から要請される。例外タグの割り振り等において32ビットあることが前提となっているためである。

[3]浮動小数点データの格納方法は、各実機処理系において必ずしもVPIMと同じにする必要はない。PIM/pでは、CDESCワードに続く64ビットの連続領域を使用して浮動小数点データを格納する。

000XXXXX　1ビットストリング
001XXXXX　8ビットストリング
010XXXXX　16ビットストリング
011XXXXX　32ビットストリング

ここで、XXXXX は最終ワード中の未使用要素数を表す。(1ビットストリングでは0〜31、8ビットストリングでは0〜3)

ディスクリプタに続くワードには、ストリングの要素を1ワード32ビット毎にDNTCタイプで格納する。各ワードの中では、要素を上位ビット側から詰める。最終ワード中の、未使用領域には0を詰める。[4]

なお、ユーザプログラム中のストリングは、16ビットストリングをデフォルトとする。

(5) リスト

Type は LIST。Value は CONS セルへのポインタ。CONS セルの CAR/CDR には任意の KL1 データを入れることができる。[5] (図2-7)

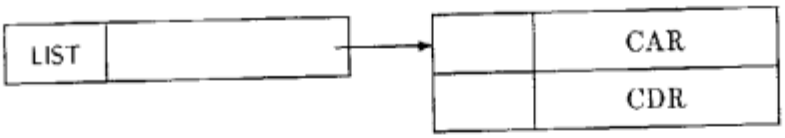


図 2-7: リスト

(6) ベクタ

要素数 0 以上の 1 次元の配列。要素数によって、ナル(null)ベクタ、ショートベクタ、ロングベクタの3種類に分類する。

① ナルベクタ

要素数0のベクタ。Typeは VECT0。Valueは未使用(値は不定)。(図2-8)

[4] ストリング要素データの格納方法は、各実機処理系において必ずしもVPIMと同じにする必要はない。ただし、組込述語 change_element_size/3においては、各要素を上位ビット側から詰めることを前提としているので注意すること。PIM/pでは、各要素のデータをタグ無しで詰めて格納する。

[5] リストのCAR/CDRをメモリに格納する順序は、各実機処理系において必ずしもVPIMと同じにする必要はない。ただし、その場合にはKL1コンパイラ等の設定を合わせて変更する必要がある。

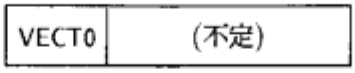


図 2–8: ナルベクタ

② ショートベクタ

要素数1～8のベクタ。Typeは要素数に応じて VECT1～VECT8 。Valueはショートベクタ本体へのポインタ。ショートベクタ本体は、要素数個のワードからなり、各ワードには、ベクタの要素として任意の KL1データを入れることができる。(図2–9)

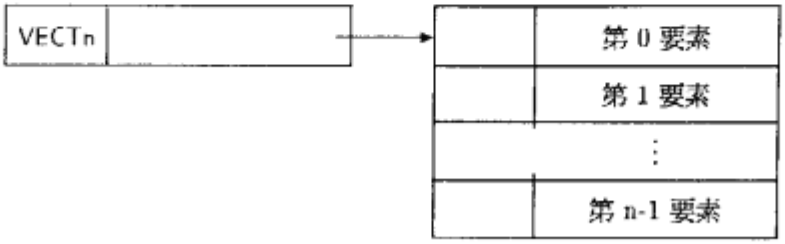


図 2–9: ショートベクタ

③ ロングベクタ

要素数9以上のベクタ。Type は VECT。Value はロングベクタ本体の2番目のワードへのポインタ。[6]ロングベクタ本体は、(要素数+1) 個のワードからなり、1番目のワードには、Type に INT、Value に要素数を格納する。2番目以降の各ワードには、ベクタの要素として任意の KL1データを入れることができる。(図2–10)

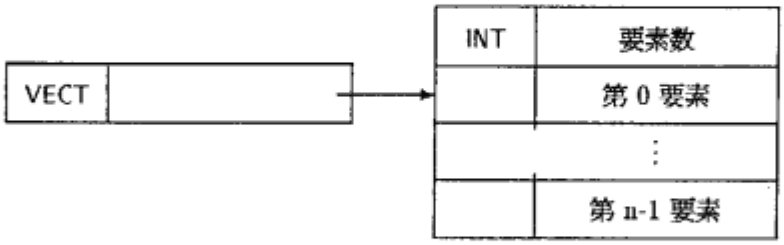


図 2–10: ロングベクタ

[6]ポインタの指す先はショートベクタでもロングベクタでも第0要素であるとしておくことによって、ベクタの要素に対するアクセスを共通化している。

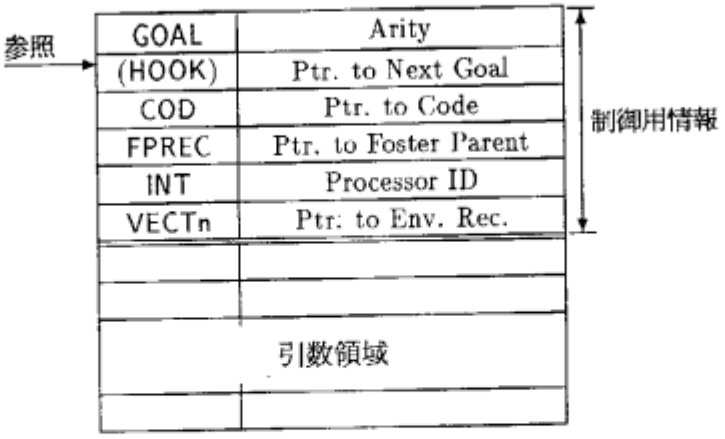


図 2-11: ゴールレコードの形式

2.4 ゴールレコードおよび環境レコードの構造

VPIM ではトレース、スパイ等の機能に対応するために、ゴールの環境情報のうち、

- 変更頻度の少ないもの。
- デバッグ用の情報。

を保持する環境レコードをへのポインタをゴールレコードの保持する。

環境レコードの形式はデバッグの状況によって変化する。ゴールレコードの本体は、この環境レコードへのポインタを持つことになる。

以下に、ゴールレコードおよび各状況での環境レコードの形式について述べる。

(1) ゴールレコード

ゴールレコードを図2-11に示すように、タグ付ワード6個(固定長)の制御用情報と、可変長の引数領域とで構成する。なお、制御用情報の各ポインタは、REFポインタをはさまずに格納されている必要がある。

- 引数個数(Arity) ..... GOAL
  ゴールが持っている引数の個数。タグ部は、一括GC時に永久中断を検出する処理のためにGOAL型(MRBは必ず白)とする。
- 次ゴールへのポインタ(Ptr. to Next Goal) ..... HOOK, MHOOK, EOL
  これはゴールスタック/サスペンドキュー/フリーリストにおいて次のゴールレコード(またはサスペンドレコード)を指すポインタとして使用する。タグ部は、次のゴールレコード/サスペンドレコードがあればHOOK/MHOOK型、なければEOL型である。
- 実行すべきコードへのポインタ(Ptr. to Code) ..... COD
  ゴールが次に実行すべき述語の先頭の命令へのポインタであり、デキュー時に(デレフ無しで)プログラムカウンタPCにセットされる。

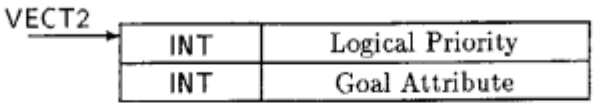


図 2–12: デバッグ情報を持たない環境レコードの形式

- 所属里親へのポインタ(Ptr. to Foster Parent)+限定トレーサスイッチ ..... FPREC

ゴールが所属する里親へのポインタであり、親ゴールから子孫ゴールに受け継がれるていく荘園環境の情報である。このポインタはデキュー時に読み込み、直前に実行したゴールと同じ里親を指しているか検査し、異なる場合に荘園環境の切替処理を行なう。

また、この情報が親から子ゴールへ受け継がれるのを利用して、このエントリのMRBビットを限定トレーサのON/OFFを表すためにも使用する。即ち、MRB白が限定トレーサOFFを、MRB黒が限定トレーサON を意味するものとする。

- プロセッサID(Processor ID) ..... INT

ゴールが属しているPEのクラスタ内番号を保持するのに使用する。この情報は一旦サスペンドしたゴールがリジュームした時に、そのゴールを元のPEに戻すために使用する。従って、この情報はサスペンド時(またはそれ以前)に書き込み、リジューム時に参照することになる。

- 環境レコードへのポインタ(Ptr. to Env. Rec.) ..... VECT(2,3,4,6,8)

ゴールの環境情報のうち変更頻度の少ないものとデバッグ用の情報を保持するための環境レコードを指すポインタである。このレコードの大きさは基本状態では2ワード(ポインタはVECT2型)であるが、デバッグ情報を付加する場合にはその状況に応じて要素数を増やす(ポインタはVECT3, VECT4, VECT6, VECT8型)。そして、この要素数の違い=タグの違いによりゴールがデバッグモードか否かを区別する。環境レコードの形式の詳細については、以下で述べる。

環境レコードは変更が必要でない限り物理的に同じものを親ゴールから子孫ゴールに渡していくので複数のゴールで共有することになる。従って、このポインタは原則的にMRBを黒くしておく。また、環境を変える場合には、MRB黒のベクタに `set_vector_element` を行なう時のように、新たな環境レコードを割り付ける必要がある。

- 引数領域

引数領域は可変長であり、VPIM 1.0版ではフリーリストの種類に対応してそれぞれ10, 26個までの引数を保持することができる。

これ以上の引数を持てるように機能拡張することも容易である。

(2) デバッグ情報を持たない環境レコード

デバッグ用の情報を持たない場合の環境レコードは、図2–12に示すようにタグ付ワード2個で構成する。この時のゴールレコード側のポインタはVECT2 型になる。

- 論理プライオリティ(Logical Priority)

ゴールの論理プライオリティ。 VPIMでは符号無し32 ビット整数の扱いが難しいので符号無

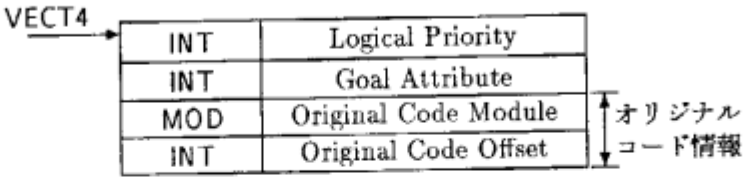


図 2-13: オリジナルコードを持つ環境レコード

し 31 ビット整数で表現する。この情報はプライオリティ指定プラグマ付きのゴールを呼び出す enqueue_with_priority 命令で設定し、プライオリティを計算/参照する組込述語およびリジューム後の enqueue 操作で参照する。

- ゴール属性(Goal Attribute)

レジデント属性等のゴールの属性を表すフラグ群で、現在は以下のように下位2ビットのみを使用している。この情報はレジデント属性操作のプラグマの付いたゴール呼び出しのための enqueue 系命令で設定し、自動負荷分散の処理で参照する。

–bit0 : PE レジデント属性を表す。0→OFF(移動可)、1→ON(固定)。

–bit1 : クラスタレジデント属性を表す。0→OFF(移動可)、1→ON(固定)。

(3) オリジナルコードを持つ環境レコード

オリジナルコード情報は組込述語と enqueue 系命令のサスペンド処理のために D コードゴールを KL1 側に割り出す時に、元々の組込述語等の命令の位置を保持しておくために付加する。この情報は割り出した先の KL1 プログラム内で例外が発生した時に例外メッセージを生成するために使用する。(10.10節参照)

オリジナルコード情報が付く場合の環境レコードは、通常、図2–13に示すようにタグ付ワード4個で構成する。ただし、呼び出し元コード情報やスパイとの関係で別の形式になる場合もある。

この時のゴールレコード側のポインタは VECT4 型になる。

オリジナルコード情報は2ワードの情報で、サスペンドした組込述語等の位置を表す、

- モジュールへのポインタ (MOD 型)。
- モジュール内のオフセット (モジュール先頭からの相対位置、INT 型)。

から成り、環境レコードの第3, 4番目を使用する。

なお、組込述語等の位置とはそれに対応する KL1-B 命令の位置であるが、PIM/p のように KL1-B を複数の低レベル機械語命令列に変換してから実行する場合には、その命令列の始まる位置とする。

(4) 呼び出し元コードを持つ環境レコード

呼び出し元コード情報は KL1 で定義された組込述語を呼び出した時に、そのための enqueue 系命令の位置を指定する情報で、組込述語を定義している KL1 プログラム内で例外が発生した時に、それが元々ユーザープログラムのどこから呼び出されたものかを知るために使用される。

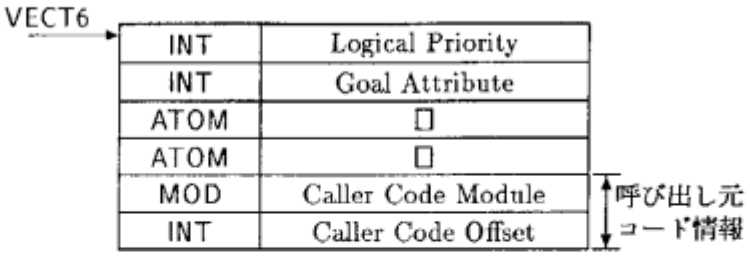


図 2-14: 呼び出し元コードを持つ環境レコード

呼び出し元コード情報が付く場合の環境レコードは、通常、図2-14に示すようにタグ付ワード6個で構成する。ただし、スパイとの関係で別の形式になる場合もある。また、環境レコードの第3,4番目はオリジナルコード情報のために使用される場合がある。

この時のゴールレコード側のポインタはVECT6 型になる。

呼び出し元コード情報は2ワードの情報で、KL1定義の組込述語を呼び出したenqueue系命令の位置を表す、

- モジュールへのポインタ(MOD型)。
- モジュール内のオフセット(モジュール先頭からの相対位置、INT型)。

から成り、環境レコードの第5,6番目を使用する。

なお、PIM/pのようにKL1-Bを複数の低レベル機械語命令列に変換してから実行する場合には、enqueue系命令に相当する機械語命令列の始まる位置とする。この情報を付加する必要がある場合には、KL1コンパイラが付加のためのKL1-B命令をenqueue系命令の直前に出力する。

(5) スパイ用環境レコード

スパイは、荘園に組み込まれた特定のゴールが生成された時点で検出する機能であり、組込述語apply_spyingで生成されたゴールの子孫ゴールが対象になる。(12.5節参照)

スパイのための情報が付く場合の環境レコードは、図2-15に示すようにタグ付ワード8個で構成する。また、環境レコードの第3,4番目はオリジナルコード情報のために、第5,6番目は呼び出し元コード情報のために使用される場合がある。

この時のゴールレコード側のポインタはVECT8 型になる。

スパイのための情報は2つの情報で、スパイID(任意のデータ型)とスパイ対象述語表(ベクタ)から成り、環境レコードの第7,8番目を使用する。

- スパイID(Spy ID)

スパイの処理に付ける識別子で、ユーザーがapply_spying組込述語の引数に与えた(未定義変数を含む)任意のKL1データであり、スパイ例外発生時にメッセージの引数として出力する。

- スパイ対象述語表(Spied Pred. Table)

3の倍数の長さを持つベクタであり、要素としてスパイ対象述語のモジュール名アトム/述語名アトム

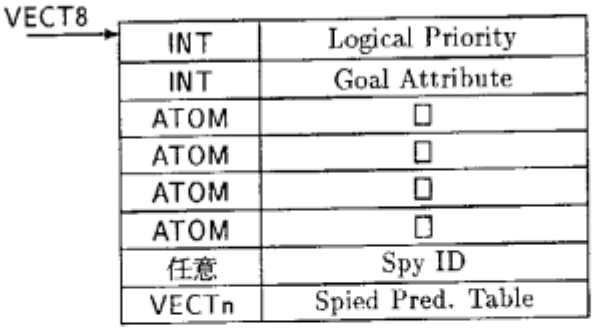


図 2-15: スパイ用環境レコード

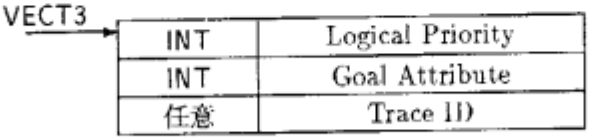


図 2-16: トレース用環境レコード

/引数個数が繰り返し格納されたものである。この要素には必ずアトム/整数が(REF無しで)直接入っていることが保証されている必要がある。なお、パッケージ名は含まれない。

(6) トレース用環境レコード

トレース[7]は述語に組み込まれたゴールリダクションをトレースする機能であり、組込述語apply_tracingで生成されたゴールが対象になる。(12.4節参照)

トレースのための情報が付く場合の環境レコードは、図2-16に示すようにタグ付ワード3個で構成する。この時のゴールレコード側のポインタはVECT3型になる。

トレースのための情報はトレースID(任意のデータ型)の1つだけであり、環境レコードの第3番目を使用する。

- トレースID(Trace ID)

トレースの処理に付ける識別子で、ユーザーが apply_tracing 組込述語の引数に与えた(未定義変数を含む)任意のKL1データであり、トレース例外発生時にメッセージの引数として出力する。

(7) 環境レコードの状態遷移

複数のデバッグ情報を持つ場合も含めて、環境レコードの種類をまとめると、次のようになる。

タイプ	ポインタ	付加されているデバッグ情報の種類
B	VECT2	デバッグ情報がない場合
BO	VECT4	オリジナルコード情報
B-C	VECT6	呼び出し元コード情報

[7]特にCSPによる「限定トレース」と区別したい場合は、「ソフトトレース」と呼ぶことがある。

BOC	VECT6	オリジナルコード情報＋呼び出し元コード情報
B--S	VECT8	スパイのための情報
BO-S	VECT8	オリジナルコード情報＋スパイのための情報
B-CS	VECT8	呼び出し元コード情報＋スパイのための情報
BOCS	VECT8	オリジナルコード情報＋呼び出し元コード情報＋スパイのための情報
BT	VECT3	トレースのための情報

以上の9通りの状態の間で、表2-3～2-6のような遷移をする可能性がある。状態遷移を起こす場面では考えられる全ての組合せを考慮する必要がある。なお、表2-3～2-6中の記号は以下の通りである。

→：親ゴールの環境レコードをそのまま使用する。

新：新しい環境レコードを生成し使用する。

×：トレース／スパイの例外を発生するのでゴールを生成しない。

△：このような状態はあり得ない。

表 2-3: 環境レコードの状態遷移 (その 1)

旧状態	enqueue, execute系命令		
	親の持つ状態を変更しない場合	プライオリティ等の属性を変更する場合	サスペンドする場合→ Dコード
B	→ B	新 B	新 BO
BO	△	△	△
B-C	→ B-C	新 B-C	新 BOC
BOC	△	△	△
B--S	→ B--S	新 B--S	新 BO-S
BO-S	△	△	△
B-CS	→ B-CS	新 B-CS	新 BOCS
BOCS	△	△	△
BT	×	×	△

enqueue, execute系の命令がサスペンドする場合にはenqueue系の組込述語を実行するためのDコードゴールを生成する。

表 2-4: 環境レコードの状態遷移 (その 2)

旧状態	enqueue系組込述語		
	親の持つ状態を変更しない場合	プライオリティ等の属性を変更する場合	サスペンドする場合→ Dコード
B	△	△	△
BO	新 B	新 B	△
B-C	△	△	△
BOC	新 B-C	新 B-C	△
B--S	△	△	△
BO-S	新 B--S	新 B--S	△
B-CS	△	△	△
BOCS	新 B-CS	新 B-CS	△
BT	△	△	△

表 2-5: 環境レコードの状態遷移 (その 3)

旧状態	apply 組込の実行	apply_tracing 組込の実行	apply_spying 組込の実行	create_shoen 系組込の実行
B	→ B	新 BT	新 B--S	新 B
BO	新 B	新 BT	新 B--S	新 B
B-C	新 B	新 BT	新 B--S	新 B
BOC	新 B	新 BT	新 B--S	新 B
B--S	→ B--S	×	×	×
BO-S	新 B--S	×	×	×
B-CS	新 B--S	×	×	×
BOCS	新 B--S	×	×	×
BT	×	×	×	×

これらの組込述語が サスペンドする場合は、一般の組込述語のサスペンド(表2-6参照) と同じ扱いにする。

表 2-6: 環境レコードの状態遷移 (その 4)

旧状態	組込述語をサスペンドする場合→ D コード	KL1 定義の組込述語を呼び出す(enqueue する)場合
B	新 BO	新 B-C
BO	△	△
B-C	新 BOC	→ B-C
BOC	△	△
B--S	新 BO-S	新 B-CS
BO-S	△	△
B-CS	新 BOCS	→ B-CS
BOCS	△	△
BT	△	×

組込述語が D コードで呼び出される (オリジナルコード情報ありの) 場合、
入力引数をガードで wait しているのでサスペンドは発生しない。
トレースモードでは、組込述語のサスペンドは発生しない。

第 3 章

コードモジュール

執筆担当者: 平野

コードモジュールはKL1のオブジェクトプログラムを表現する構造体データのことで、ヒープ上に置かれ、VPIMだけでなくPIMOS等のKL1プログラムレベルからも扱うことができるようになっている。VPIMにおけるコードモジュールの仕様はMulti-PSIのものをベースとして以下の条件を満足するように決めた。

- 各社PIMに移植しやすいように、機種依存部分と共通部分とを分離する。
- PIMOSをなるべく機種独立にするために、PIMOSのコードモジュール操作ライブラリから参照される情報はなるべく共通部分に置くこととする。
- コードモジュールのデマンドローディング機能を導入し易いように、モジュール間の参照パスを一箇所にまとめる。

3.1 コードモジュールの仕様

コードモジュールを図3-1のように、モジュールヘッダ、他モジュール参照テーブル、定数テーブル、エントリーテーブル、および、コード領域の5つの部分で構成することにした。このうち、最後のコード領域は機種依存になるが、それ以外の部分は全機種に共通の仕様とし、全てタグ付ワードで構成することにした。コード領域の手前に定数ディスクリプタ(CDESCタイプ)が置かれているが、これは一括GC時にコード領域のメンテナンスをスキップさせるためである。なお、コードモジュールを指すポインタのタイプはCODである。

3.1.1 モジュールヘッダ

モジュールヘッダはモジュール全体に関する情報を保持する部分であり、タグ付ワード16個(固定長)から成っている。ここは全機種共通とする。

Module Size

このコードモジュール全体の大きさ。単位はタグ付ワード数で自分自身を含む。コード領域にタグが付かない機種の場合にも全体の大きさはタグ付ワード単位に換算しておく。タイプは必ずINTである

MOD→	INT	Module Size
1	INT	GC Maintenance Area Size
2	ATOM/INT	Package Name
3	ATOM	Module Name
4	INT	Number of External Module Reference
5	INT	Number of Constants
6	INT	Number of Entries
7	INT	Start Posiston of Entry Table
8	INT	Hash Mask for Entry Table
9	INT	Compile Information
10	INT	Compile Date/Time
11	INT/ATOM	Version Number(KL1Comp)
12	INT/ATOM	Version Number(PostComp)
13	INT	Version Number(Assembler)
14	INT/ATOM	Version Number(for User)
15	???	Machine Dependent Information
16		
	他モジュール参照テーブル	
	定数テーブル	
	エントリーテーブル	
	CDESC	Code Area Size
	コード領域 (機種依存)	

図 3-1: ヒープ上でのコードモジュールの形式

こと。REFポインタによる間接参照にすることはできない。MRBは任意でよい。

GC Maintenance Area Size

一括GC時にメンテナンスをする領域の大きさ。単位はタグ付ワード数でモジュールヘッダ、他モジュール参照テーブル、定数テーブルおよびエントリーテーブルの4つの部分を合計した大きさを表す。なお、この値は一括GC時時に参照されるものではなく(メンテナンスの要/不要はCDESCディスクリプタで制御される)、VPIMおよびKL1プログラムでコードモジュールをコピーする時にメンテナンスする領域の大きさを求めるために参照される。タイプは必ずINTであること。REFポインタによる間接参照にすることはできない。MRBは任意でよい。

Package Name

コンパイル時に指定されたパッケージ[1]名を格納するスロット。パッケージ名が明示的に指定された場合にはその名前のアトムを、特に指定が無かった場合にはデフォルトとして整数(値は任意)を格納する。タイプは必ずATOMまたはINTであること。REFポインタによる間接参照にすることはできない。MRBは任意でよい。なお、現在この部分の仕様は固まっていないので、当面は整数の0を書き込んでおくこととする。

Module Name

モジュール名アトム。タイプは必ずATOMであること。REFポインタによる間接参照にすることはできない。MRBは任意でよい。

Number of External Module Reference

他モジュール参照テーブルの要素数。このテーブルには後述するモジュール記述子と述語記述子が置かれるが、このスロットにはそれらの個数(ワード数ではない)を合計した値を入れておく。タイプは必ずINTであること。REFポインタによる間接参照にすることはできない。MRBは任意でよい。

Number of Constants

定数テーブルにある定数(構造体定数)の個数、即ち、タグ付ワードの数。タイプは必ずINTであること。REFポインタによる間接参照にすることはできない。MRBは任意でよい。

Number of Entries

外部に公開されている述語の個数、即ち、エントリーテーブルに登録されている述語の数。タイプは必ずINTであること。REFポインタによる間接参照にすることはできない。MRBは任意でよい。

Start Position of Entry Table

エントリーテーブルの開始位置。コードモジュールの先頭を基準にしたタグ付きワード単位での相対アドレスで指定する。なお、この相対アドレスの値はKL1から見ると、module_element等の組込述語で使用するモジュール要素の番号に相当する。タイプは必ずINTであること。REFポインタによる間

[1]パッケージについてはPIMOSマニュアル(第2.5版)の8.2モジュールの管理を参照のこと。

接参照にすることはできない。MRBは任意でよい。

Hash Mask of Entry Table

エントリーテーブルを検索するときに使うハッシュ関数用のマスク値。この値はコンパイラ/アセンブラが外部に公開されている述語の個数を基に決めるもので必ず $2^n - 1$ の値を持つ。タイプは必ずINTであること。REFポインタによる間接参照にすることはできない。MRBは任意でよい。

Compile Information

コンパイル時の環境情報(デバッグオプション等)を格納する予定のスロット。現在は未使用で常に整数0を格納している。ここはPIMOSのコード管理部等から参照され、VPIMからは参照しない。データ型はベクタを予定している。REFポインタによる間接参照にしてもよい。MRBはポインタの場合には黒くしておく必要がある。アトミックデータの場合は任意である。

Compile Date/Time

コンパイルした(正確にはアセンブラによりコードモジュールが作られた)日付と時間を表す整数。年月日時分秒を6+4+5+5+6+6=32bitで表現することにする。なお、年は1990年を原点とすることで1990～2053年を表現することにする。VPIMからは参照しない。タイプはINTであるが、REFポインタによる間接参照にしてもよい。MRBはINTが直接の入る場合には任意で良いが、REFが入る場合は黒とする。

Version Number

コンパイラ本体/ポストコンパイラ/アセンブラの版数、および、ユーザーが指定した版数を格納するスロット。版数は整数で表現される。ユーザー指定以外の3つはコンパイル時に付加されるものでコンパイラの各部分の版数を意味する。これはロード時にコードモジュールの版数がその時のVPIMに適合しているか検査するため等に使用する予定である。なお、ソースプログラムがKL1以外(KL1-B等)の場合にはコンパイラの一部を使用しないので、使用しなかった部分に相当するスロットにはアトム [] を格納しておく。ユーザー指定のものは、このモジュールのプログラムに対してユーザーが指定した版数であり、KL1ソースプログラムにおいて、

:- *version*(整数).

というディレクティブで指示された数値を格納する。この指定が無かった場合にはアトム [] を格納しておく。VPIMからは参照しない。タイプはINTまたはATOMであるが、REFポインタによる間接参照にしてもよい。MRBはINTが直接の入る場合には任意で良いが、REFが入る場合は黒とする。

Machine Dependent Information

機種によりコード管理等のために使用する情報を格納するスロット。内容は機種依存であるが、使用しない場合には整数0を書き込んでおく。ここは、例えば、PIM/pの場合にはコード領域の部分にタグが付かないので、その中のどこにアトムやマクロコール(ランタイムに使用するサブルーチン)アドレスが書かれているかという情報をこの部分にベクタ等の形で保持しておき、アンローダ(ヒープ上のコー

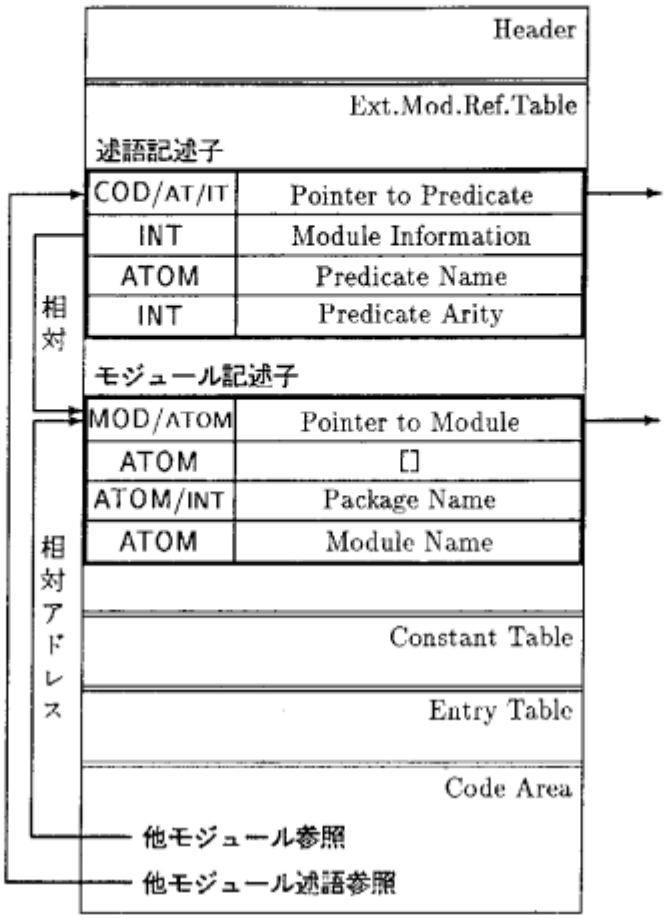


図 3-2: 他モジュール参照テーブル

ドモジュールをファイルに書き出すツール)等から参照する。VPIMからは参照しない。REFポインタによる間接参照にしてもよい。MRBはINTが直接の入る場合には任意で良いが、REFが入る場合は黒とする。

3.1.2 他モジュール参照テーブル

他モジュール参照テーブルはこのモジュールから他のモジュールへの参照を管理するためのもので、モジュール記述子または述語記述子を要素として持つ。これらは参照相手の状態を表す情報(絶対アドレス等)を扱っており、その値が変わる場面、即ち、他のクラスタへの輸出の時、一括GCの時、および、コードモジュールの再リンクの時にはメンテナンスが必要である。しかし、他モジュールへの参照をここだけにまとめてあるので、そのような時でもコード領域内の命令をメンテナンスする必要はないようになっている。

なお、このテーブルにOSやコンパイラ/リンカがバグにより間違ったデータを書き込む可能性があるので、VPIMは参照時(モジュール間呼び出し等)にタイプ等をチェックし、正しくない場合には例外を発生する必要がある。

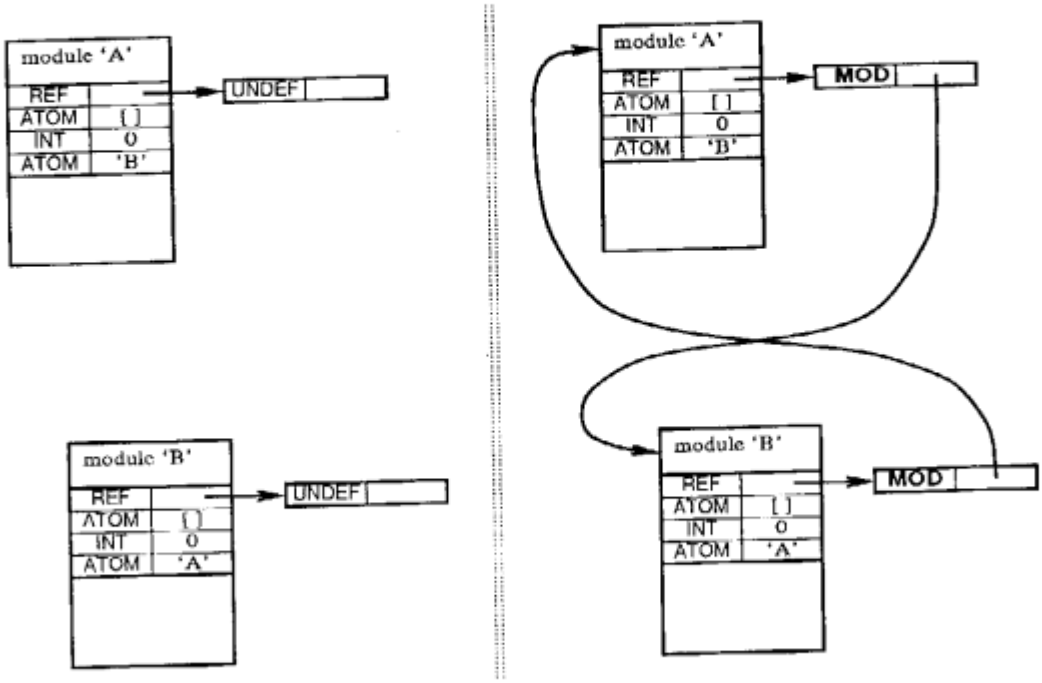


図 3-3: アセンブラが生成した直後 (左) とリンク直後 (右) のモジュール記述子

(1) モジュール記述子

モジュール記述子は、他のモジュールを参照する場合に使う間接ポインタであり、図3-2に示すように、4個のタグ付ワードから成っている。しかし、実際に使用するのは以下の3個であり、2ワード目は述語記述子(必ず整数)との区別のためにアトム [] を書いておく。

Package Name

参照先モジュールの属するパッケージ名を表すアトム。パッケージの指定が無かった場合には整数0によりデフォルトを表す。タイプは必ずATOMまたはINTであること。REFポインタによる間接参照にすることはできない。MRBは任意でよい。

Module Name

参照先モジュールの名前を表すアトム。タイプは必ずATOMであること。REFポインタによる間接参照にすることはできない。MRBは任意でよい。

Pointer to Module

参照先モジュールへのポインタを格納するためのスロット。

アセンブラはコードモジュールを生成する時に、ここを図3-3に示すように未定義変数REF→UNDEFとしておく。そして、リンカが図3-4にように参照先モジュール(MODタイプのポインタ)に具体化するようになっている。従って、リンカの処理が完了する前にプログラムが起動されると、まだ相手モジュールが具体化されていない場合がありうるので、ここを参照する時にはデレファレンス、具体化の

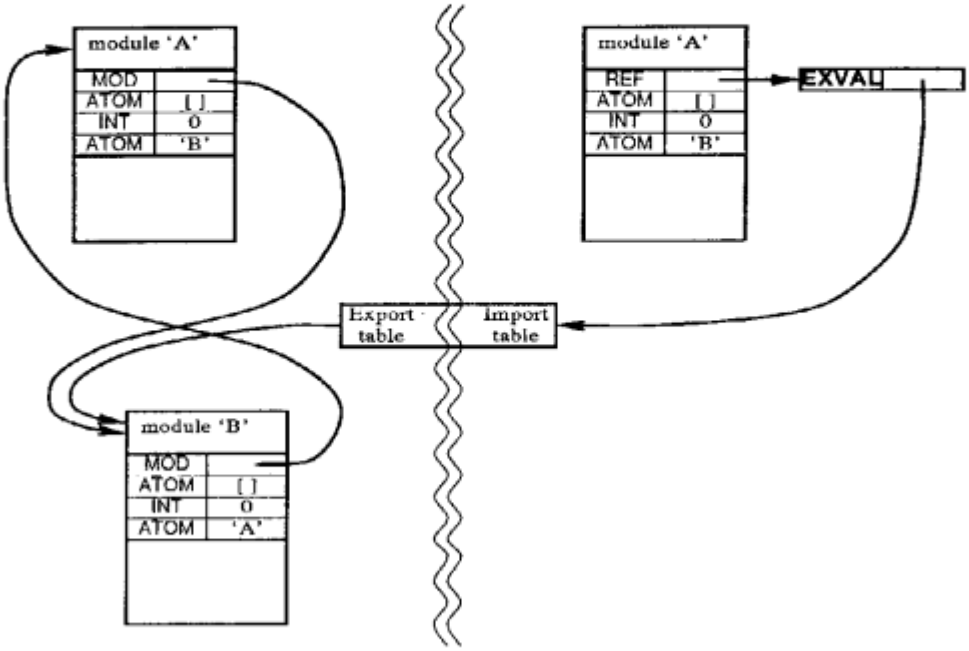


図 3–4: 輸出入 (左→右) の時のモジュール記述子 -1: 輸入直後で参照先はまだ輸入されてない場合

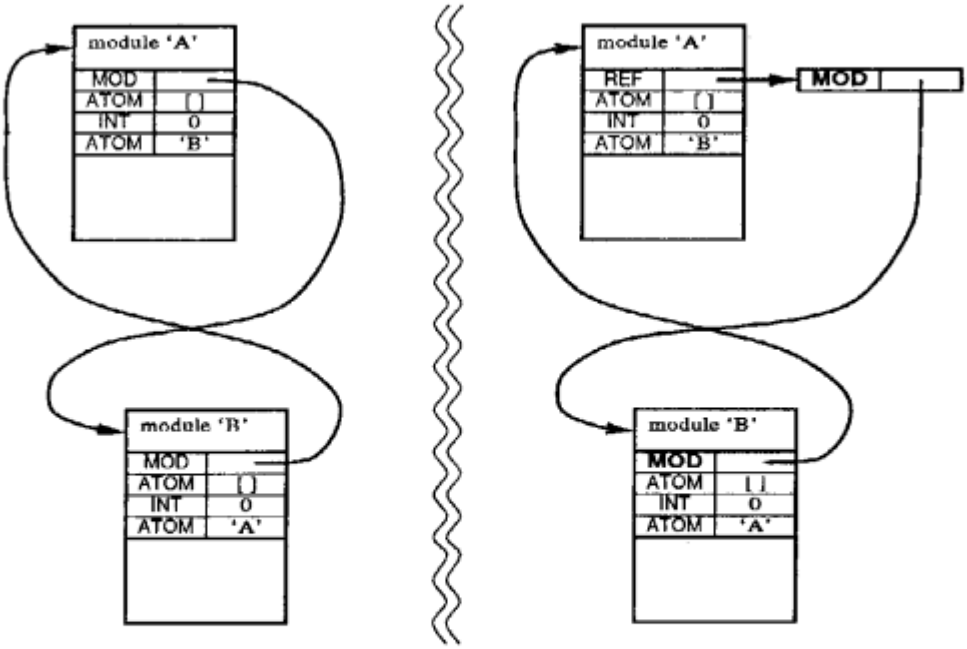


図 3–5: 輸出入 (左→右) の時のモジュール記述子 -2: 参照先も輸入されてきた場合

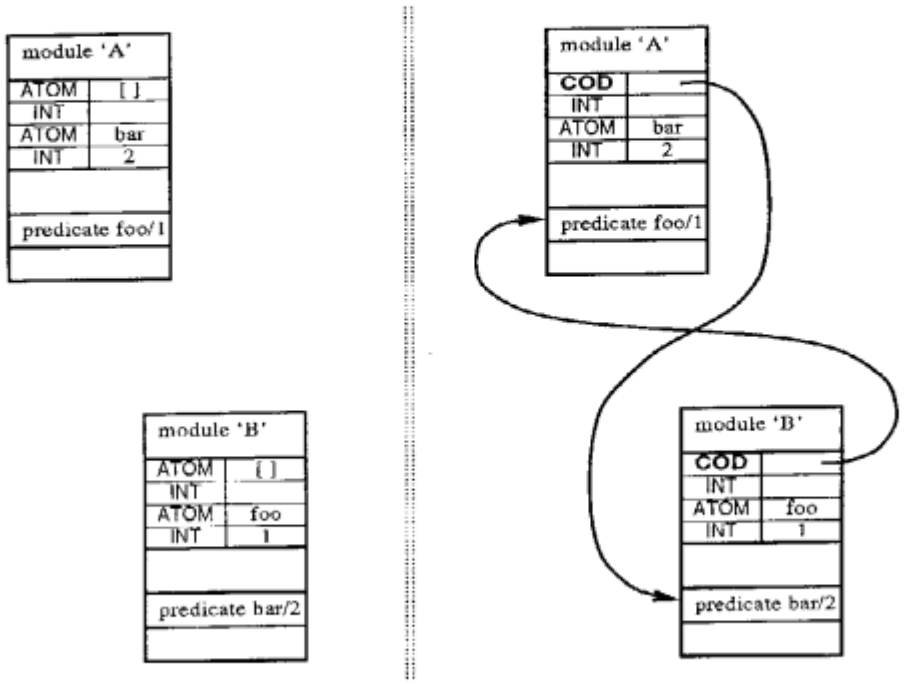


図 3-6: アセンブラが生成した直後 (左) と一旦参照された後 (右) の述語記述子

チェックと未定義ならサスペンドする処理が必要である。MRBは黒くしておく。

また、コードモジュールを他のクラスタに輸出する時には、一般の構造体参照ポインタの場合と同様、図3 4に示すように、ここをREF→EXVAL(リンカにより具体化されていない場合にはEXREF)としてコピーする。そして、輸入先でその参照先モジュールが最初に必要になった時、即ち、ここを読んでみたらMODではなくREF→EXVALだった時に参照先モジュールの実体を輸入し、図3-5に示すように、そこを指すMODポインタに戻すようにする。この時、モジュールを参照しようとしたゴールは参照先の実体が輸入されるまで一旦サスペンドする処理が必要である。

また、リンク時にデマンドローディングを行なうように指定した場合には、ここにアトム □ を具体化しておくことにより、最初に参照した時に例外を発生しPIMOS側にコードモジュールのローディングが必要なことを知らせることが可能[2]になっている。

(2) 述語記述子

述語記述子は、他モジュールの述語を参照する場合に使う間接ポインタであり、図3-2に示すようにタグ付ワード4個から成っている。

Module Information

参照先の述語の属するモジュールに関する情報。具体的にはモジュール記述子へのタグ付きワード単位での自己相対アドレス。タイプは必ずINTであること。REFポインタによる間接参照にすることは

[2]デマンドローディングの手順はPIMOS側との調整が必要であり、まだ決まっていない。

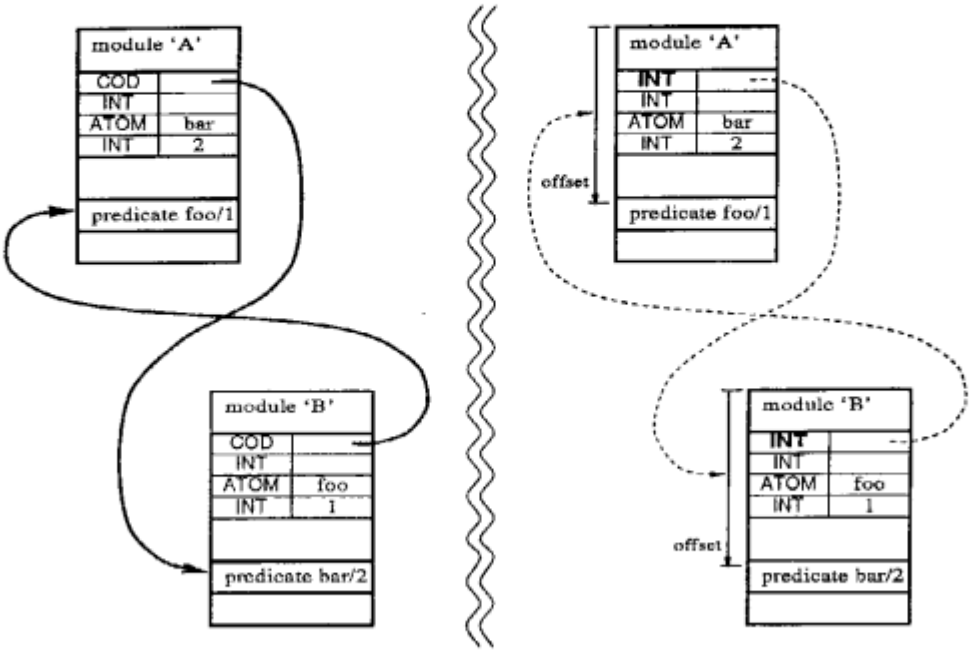


図 3-7: 輸出入 (左→右) の時の述語記述子 -1: 輸入直後でまだ一度も参照したことのない場合

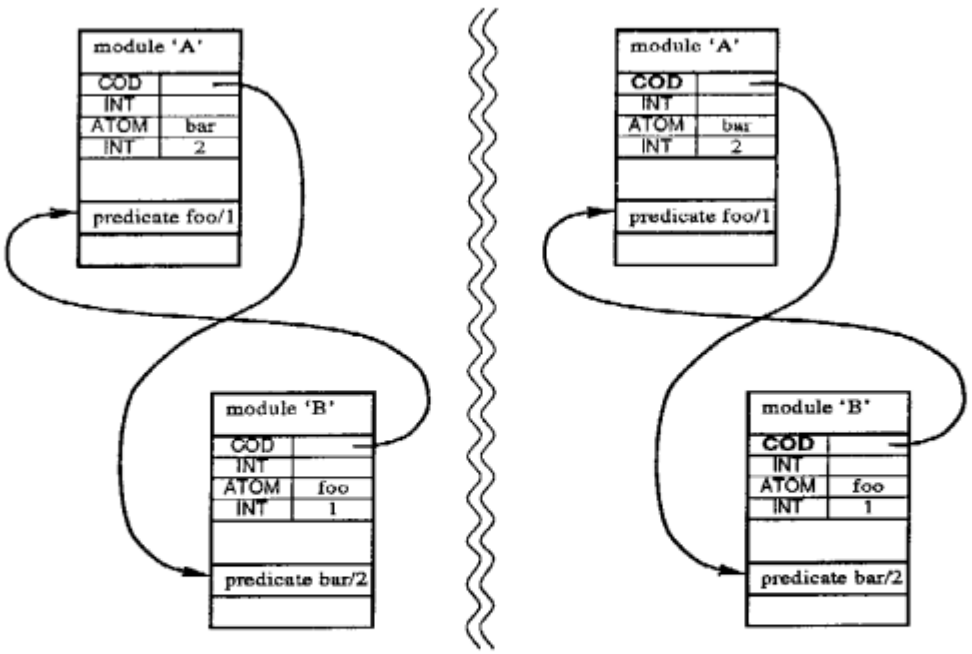


図 3-8: 輸出入 (左→右) の時の述語記述子 -2: 一度以上参照したことがある場合

できない。MRBは任意でよい。なお、ここがINTタイプであることによりモジュール記述子と区別される。

Predicate Name

参照先の述語の名前を表すアトム。タイプは必ずATOMであること。REFポインタによる間接参照にすることはできない。MRBは任意でよい。

Predicate Arity

参照先の述語の引数個数。タイプは必ずINTであること。REFポインタによる間接参照にすることはできない。MRBは任意でよい。

Pointer to Predicate

参照先述語へのポインタを格納するスロット。

ここはVPIMで全てメンテナンスするので、図3-6に示すようにアセンブラ/リンカは初期値としてアトム [] を書き込んでおく。VPIMが最初に参照した時にはここが [] になっているので Module Information スロットで指定されるモジュール記述子を調べ、参照先の述語の属するモジュールの絶対アドレスを求める。そして、そのモジュールのエントリーテーブルを検索することにより参照先の述語のアドレスを求める。この時、求めた述語アドレスをCODタイプのポインタとしてここに書き込んでおく。従って、2回目以降に参照した時には相手述語へのCODポインタが書かれているので、わざわざエントリーテーブルを調べたりせずに高速に相手の絶対アドレスを求めることができる。MRBは任意よい。

また、コードモジュールを他のクラスタに輸出する時には、図3-7に示すように、ここを述語のオフセット、即ち、述語の属するモジュールの先頭からの相対アドレス(単位は機種依存, INTタイプ)に変えてコピーする。そして、輸出先でその述語を最初に参照した時、即ち、このポインタを読んでみたらINTだった時には、Module Information スロットで指定されるモジュール記述子を調べ、参照先の述語の属するモジュールの絶対アドレスを求める。そして、そのモジュールアドレスとオフセットから述語の絶対アドレスを求め、3-8に示すように、CODタイプのポインタに戻す。

なお、上記の処理において参照先の述語の属するモジュールの絶対アドレスを求める時には、リンク中や未輸入のためにモジュールの絶対アドレスが決まっていない場合がありうるので、その場合にはモジュールが具体化されるまで、述語を参照しようとしたゴールをサスペンドする必要がある。

3.1.3 定数テーブル

定数テーブルはKL1のソースコード上に現れる構造体定数(リスト,ベクタ,ストリング等で変数を含まないもの)を置くテーブルである。これらの定数を表現するためにはポインタと本体が必要であるが、このテーブルには図3-9に示すようにポインタ側を登録し、本体側はコードモジュールの外のヒープ上に展開しておく。ポインタのMRBは全て黒くしておく必要がある。なお、この定数データをコード領域から参照する場合には相対アドレス(基準,単位は機種依存)を使用するので、一括GC時にはコード領域内を

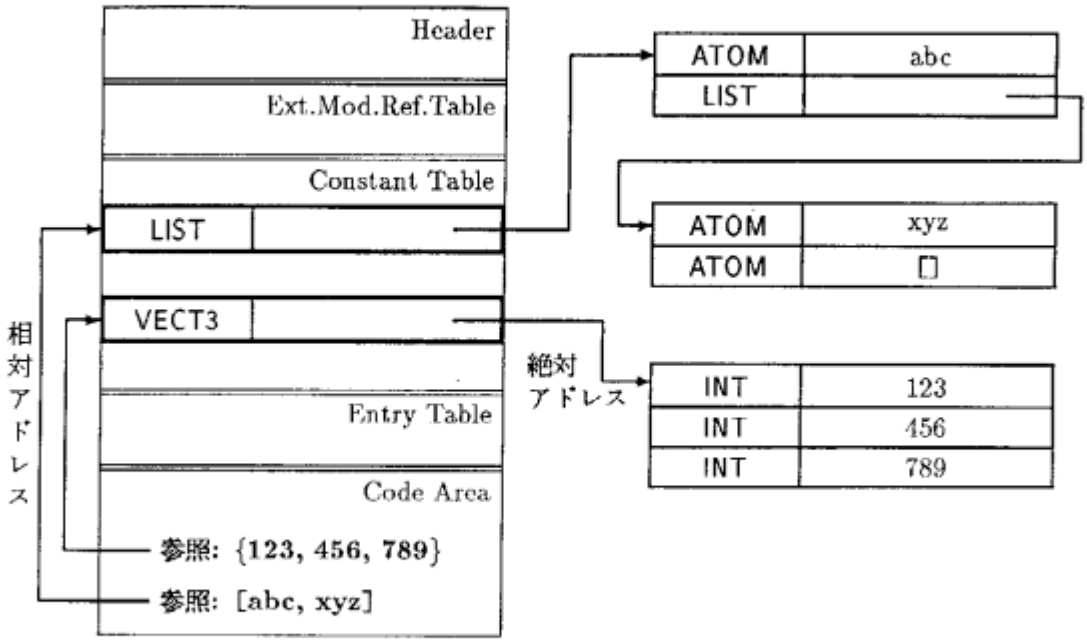


図 3-9: 定数テーブル

メンテナンスする必要がない。

3.1.4 エントリーテーブル

エントリーテーブルは、パブリック宣言により外部に公開された述語を管理するテーブルであり、述語名と引数個数から述語のエントリーアドレス(述語実行の最初に実行する命令の位置)を求めるために使われる。このテーブルは図3-10に示すようにハッシュテーブルと述語名〜述語オフセット対応テーブルの2つの部分から成っている。

ハッシュテーブルは2の冪乗の大きさを持つ配列であり、各要素はそれぞれハッシュ関数($2^n - 1$ のマスク)の値に対応し、述語名〜述語オフセット対応テーブルの検索を開始する位置(モジュール先頭を基準にしたタグ付きワード単位での相対アドレス=KL1から見るとモジュール要素の番号)と、同じハッシュ値を持つ候補の数の2つの情報を持つ。なお、このテーブルの大きさはコンパイラ/アセンブラが外部に公開されている述語の個数を基にして決める。ここの各要素のタイプはINT、MRBは任意とする。

述語名〜述語オフセット対応テーブルはパブリック宣言された述語と同数の要素を持つ配列であり、述語名アトム(ATOMタイプ、MRBは任意)、引数個数(INTタイプ、MRBは任意)および述語のモジュール内オフセット(機種依存のアドレス系でのモジュール先頭からの相対アドレスであり、例外メッセージや組込述語 *module_offset_to_code* の引数に使われるのと同じ仕様のもの。sf INTタイプ、MRBは任意)の3つの情報を持つ。なお、このテーブル内では同じハッシュ値の候補が連続するようにしておく必要がある。また、特定のハッシュ値に対応するエントリがない場合には、別のエントリ(何処でも良い)を指すようにし、検索時には1回比較を行なって(必ず失敗する)から「述語が見つからない」というエラーを発生させるようにする。これは、テーブルの検索では最初の候補でマッチする場合が多いと期

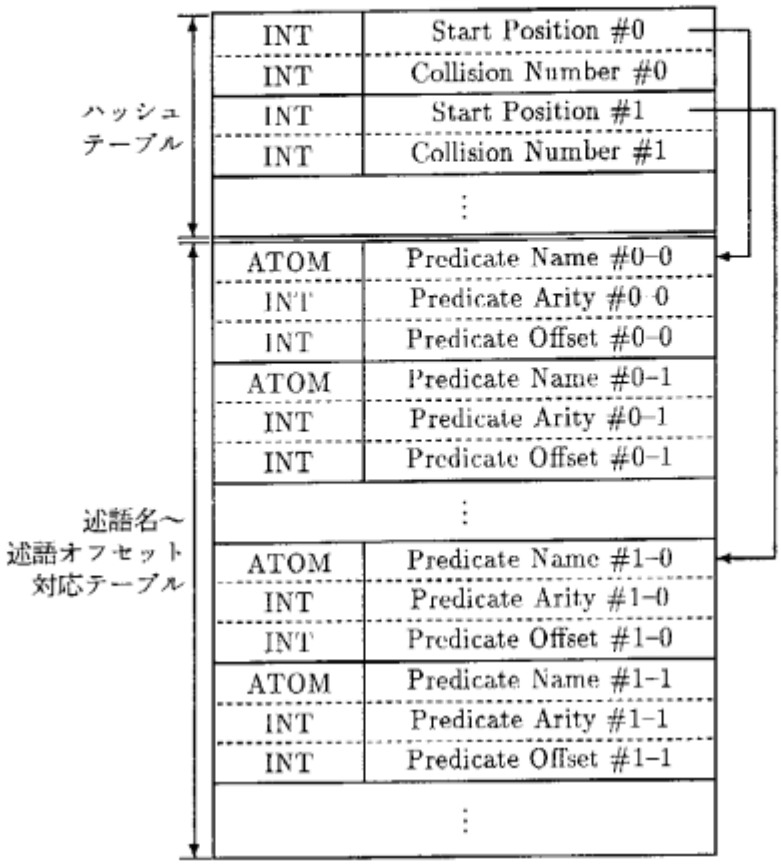


図 3-10: エントリーテーブル

待して、検索を“while ～” 型でなく、“repeat ～ until” 型にしたからである。

3.1.5 コード領域

コード領域は実際の機械命令を置く部分であり特定のハードウェアに依存した仕様になるが、基本的な構造はどの機種でも同じにする必要がある。ここでは、VPIM(PIM/s)におけるコード領域の形式を例にして説明する。なお、VPIM ではコード領域も全てタグ付ワードで構成されており、図3-11に示すように多方向分岐テーブル領域と述語定義領域に分かれている。なお、コード領域の先頭には1ワードの整数 Table Area Size (INTタイプ、MRBは任意)が置かれている。これは多方向分岐テーブル領域の大きさをタグ付きワード数単位で表すものである。

(1) 多方向分岐テーブル

多方向分岐テーブルは、多数の分岐候補の中からレジスタ上のアトムまたは整数の値により1つの分岐先を決める KL1-B 命令で使われるもので、ハッシュ方式とインデックス方式がある。この多方向分岐テーブルを述語定義本体から独立させたのは、ロード/リンク時にメンテナンスのためにテーブルを検索するのを容易にするためである。

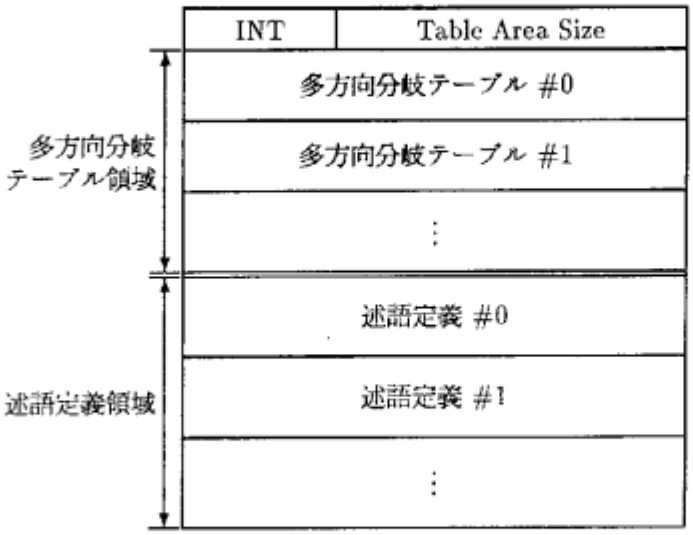


図 3–11: VPIM(PIM/s) のコード領域の形式

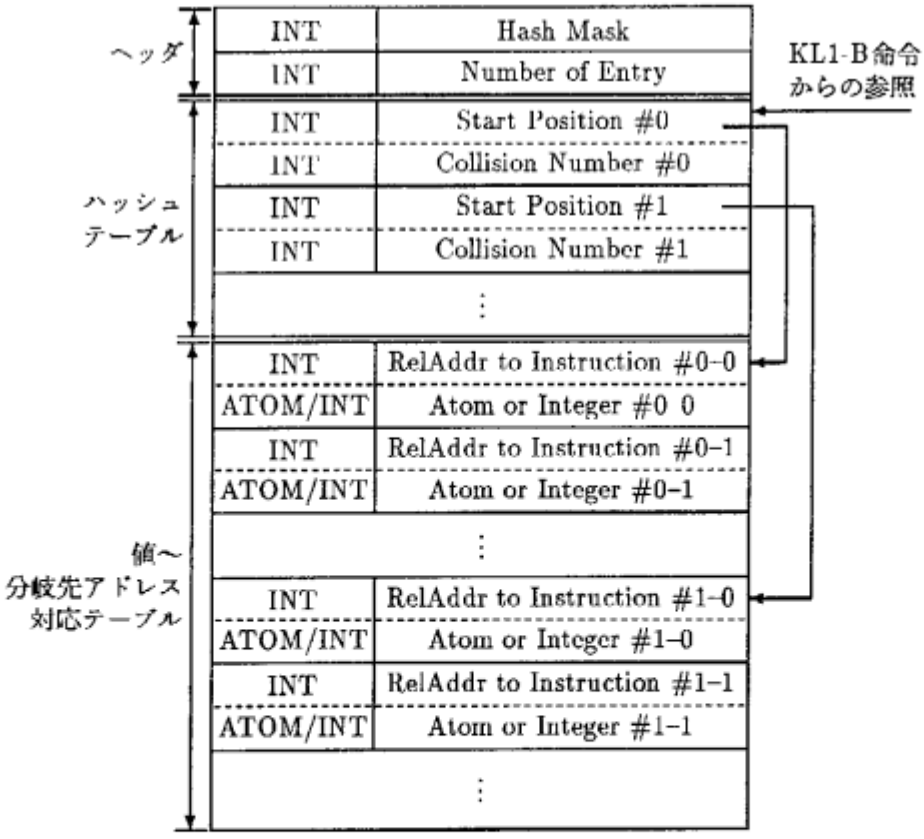


図 3–12: ハッシュ方式の多方向分岐テーブル

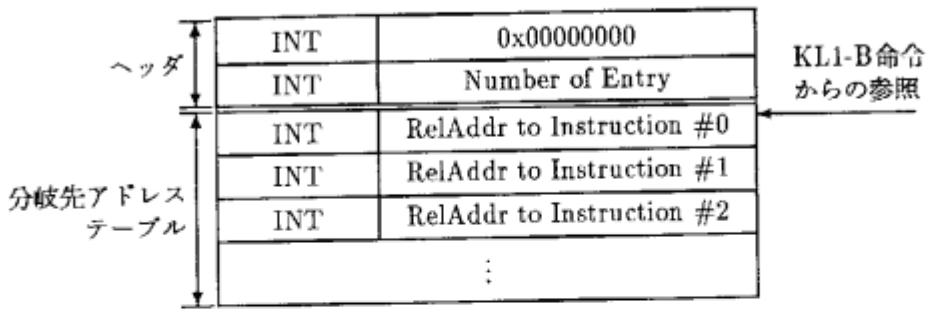


図 3–13: インデックス方式の多方向分岐テーブル

ハッシュ方式の多方向分岐テーブル

ハッシュ方式の多方向分岐テーブルは、KL1-B多方向分岐命令hash_on_integer, hash_on_atomにおいて、整数やアトムの値から分岐先アドレスを求めるのに使われる。このテーブルは図3–12に示すようにヘッダ、ハッシュテーブルと値〜分岐先アドレス対応テーブルの3つの部分から成っている。

ヘッダにはハッシュ関数として使用するマスクの値 Hash Mask と分岐先の候補数 Number of Entry が書いてある。Hash Mask は $2^n - 1$ (ここで、2^n はハッシュテーブルの大きさである)の値を持つ整数で必ず非ゼロである。この値はインデックス方式(必ずゼロ)との区別のためにも使われる。Number of Entry は分岐先の数であるが、この最上位ビットは hash_on_integer と hash_on_atom を区別するために使用している。即ち、ここが0だとhash_on_integer を、1だと hash_on_atom を意味する。このビットはコードモジュールのローダ/アンローダで使用される。ここの各ワードのタイプはINT、MRBは任意とする。

ハッシュテーブルは2の冪乗の大きさを持つ配列であり、各エントリはそれぞれハッシュ関数($2^n - 1$のマスク)の値に対応し、値〜分岐先アドレス対応テーブルの検索を開始する位置 Start Position と、同じハッシュ値を持つ候補の数 Collision Number の2つの情報を持つ。なお、検索開始位置はエントリーテーブルの場合と異なり機種依存の自己相対アドレスで指定する。ここの各要素のタイプはINT、MRBは任意とする。

値〜分岐先アドレス対応テーブルは分岐先の数と同数のエントリを持つ配列であり、分岐先アドレス(INTタイプ、MRBは任意)と整数/アトムの値(INTまたはATOMタイプ、MRBは任意)の2つの情報を持つ。ここの分岐先アドレスも機種依存の自己相対アドレスで指定する。なお、このテーブル内では同じハッシュ値の候補が連続するようにしておく必要がある。また、特定のハッシュ値に対応するエントリがない場合には、別のエントリ(何処でも良い)を指すようにし、検索時には1回比較を行なって(必ず失敗する)から失敗ラベルに分岐するようにする。これは、テーブルの検索では最初の候補でマッチする場合が多いと期待して、検索を "while 〜" 型でなく、"repeat 〜until" 型にしたからである。

インデックス方式の多方向分岐テーブル

インデックス方式の多方向分岐テーブルは、KL1-B多方向分岐命令jump_on_integer において、整数値から分岐先アドレスを求めるのに使われる。このテーブルは図3–13に示すようにヘッダ、分岐先

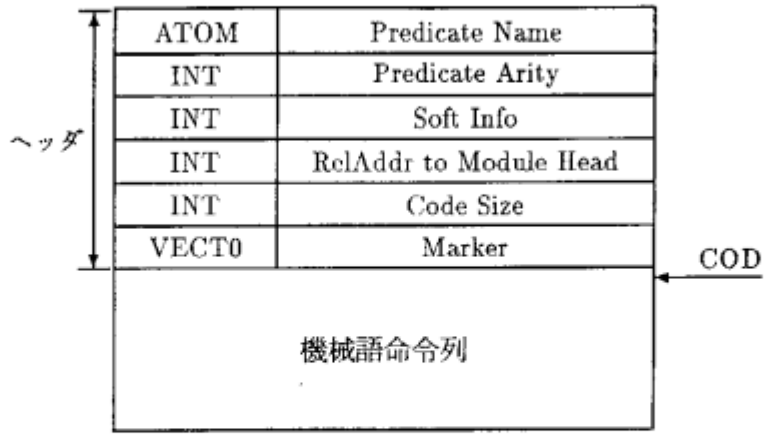


図 3-14: 述語定義

アドレステーブルの2つの部分から成っている。なお、jump_on_integer 命令はキーの候補値がある程度連続する整数の場合に生成される命令で、その範囲が Nl 以上 Nh 未満の場合には、Number of Entry が $Nh - Nl$ に、$bias$ が Nl になる。

ヘッダにはインデックス方式であることを示す整数ゼロ(ハッシュ方式ではHash Mask スロットに相当し必ず非ゼロである)とアドレステーブルの要素数Number of Entry が書いてある。ここの各ワードのタイプはINT、MRBは任意とする。

アドレステーブルは単純な分岐先アドレスの配列になっており、n 番目の要素は整数値が $bias + n$ の時の分岐先アドレスを格納している。この分岐先アドレスは機種依存の自己相対アドレスで指定する。なお、ユーザーが記述した候補が $bias$ から $bias$ + NumberofEntry − 1 までの連続した値でない場合には、次命令のアドレス(この命令は候補にマッチしなかった時は分岐しない)を入れておくことにより穴埋めをしておく必要がある。ここの各要素のタイプはINT、MRBは任意とする。

(2) 述語定義

述語定義は1つのKL1述語を表現するものであり、図3-14に示すように、ヘッダ部と機械語命令列の2つの部分から成る。ヘッダ部はその述語に関する情報を格納する部分であり、後で説明する6つのスロットから成っている。機械語命令列は実際にPIMが実行する命令列が書かれている領域で、VPIM(PIM/s)の場合にはここにもタグが付いている。ここのタイプは基本的にINTであるが、アトムをオペランドとして持つ命令では、そのオペランドの部分のワードはATOMタイプとする。このタイプの違いはコードモジュールのローダ/アンローダで使用される。なお、述語ポインタ(CODタイプ)は常に機械語命令列の先頭を指す。

Predicate Name

述語名のアトム。タイプは必ずATOMであること。REFポインタによる間接参照にすることはできない。MRBは任意でよい。

Predicate Arity

述語の引数個数。タイプは必ずINTであること。REFポインタによる間接参照にすることはできない。MRBは任意でよい。

Soft Info

コンパイル時に設定するソフト用の情報を表す整数。この情報はコンパイル時に埋め込まれるものでデバッガ等のソフトウエアが参照する。VPIMでは参照しない。タイプは必ずINTであること。REFポインタによる間接参照にすることはできない。MRBは任意でよい。

RelAddr to Module Head

モジュールの先頭へのポインタ(機種依存の自己相対アドレス)。例外発生時等に述語アドレスからコードモジュールの先頭を求めるために使われる。タイプは必ずINTであること。REFポインタによる間接参照にすることはできない。MRBは任意でよい。

Code Size

後に続く機械語命令列の大きさ。単位はタグ付ワード数。これを利用することにより、コード領域に連続して配置される述語定義のヘッダ部分だけを効率良く検索することもできる。タイプは必ずINTであること。REFポインタによる間接参照にすることはできない。MRBは任意でよい。

Marker

機械語命令列が次のタグ付ワードから始まることを示すための目印。機械語命令列中には現れないパターンである必要があり、VPIMではタグフィールドで識別できるようにVECT0oを使用する。これは例外発生時等に、実行中の Program Counter の値から述語のヘッダを探す時に使用する。REFポインタによる間接参照にすることはできない。

3.2 KL1-B の命令仕様

ここではVPIM(PIM/s)の機械語であるKL1-Bの個々の命令について仕様を説明する。ここで説明するKL1-Bの仕様はPIM/sの機械語としてのもので、KL1-B/sと呼んでいるものである。

ここの説明で、下線の付いた見出の部分に書かれているのがKL1-B命令の名前、引数一覧とオペコードである。ここに挙げた名前はコンパイラ側で使用している名前であり、VPIMのソースコード上では別の名前になっているものがあるので注意が必要である。なお、ここの引数名は綴り方により、

- 全て大文字 → 即値引数を表す。
- Rの後に小文字列 → レジスタ引数(レジスタ番号)を表す。
- Lの後に小文字列 → ラベル(相対アドレス)を表す。

のように引数の種類を表している。オペコードは16進数で表してある。また、見出の直後にある図はヒープ上での命令フォーマットである。

3.2.1 実行準備の命令

各述語の実行開始時およびコミット時に行なう検査や初期化の命令。

reset_ssp [000]

INT	OP = 000	—	—

サスペンドスタックポインタ(SSPレジスタ)を初期化する。この命令はサスペンドスタックを使用する場合、即ち、サスペンドする可能性があり、単一待ちの最適化ができなかった時に、述語定義の先頭に生成される。

check_heap Pages [001]

INT	OP = 001	Pages

述語の実行に先立ちヒープの残りページ数を調べ、それが引数Pagesで指定されたページ数よりも少なかった場合には、現在の述語の実行を止め、それをゴールスタックに戻し、一括GC要求のフラグを立て、スリットチェックの処理を行なう。その後、ゴールスタックからゴールを取り出し、その実行を行なう。

commit ClauseID [002]

INT	OP = 002	CID

コミット時の処理を行なう。具体的には、どのクローズがコミットされたかを示すID(整数)を専用のレジスタに記録する。この情報はトレーサーにより使用される。

3.2.2 レジスタ上のデータ移動の命令

汎用レジスタ上のデータを移動する命令。

move Rsource, Rdestination [042]

INT	OP = 042	Rsrc.	Rdst.

レジスタRsourceの値をレジスタRdestinationにコピーする。

3.2.3 データ読み込みと具体化検査の命令

メモリ上のデータを汎用レジスタに読み込み、具体化されているか検査を行なう命令。元々のデータのある場所(ゴールレコード/構造体/コードモジュールの定数領域)の違いと、サスペンド時の処理(一般用/単一待ち最適化用)の違い、また、構造体の場合には要素位置の指定のしかた(即値/レジスタ)の違いにより、計8種類用意する。これらはガード部でのみ使用する。

load_wait　　Rgoal, POS, Reg, Lsusp　　[013]

read_wait　　Rstruct, POS, Reg, Lsusp　　[010]

INT	OP = 013	Rgoal	POS'
INT	Reg	Lsusp	—

… ここでは POS' := POS+5 である。

INT	OP = 010	Rstr.	POS
INT	Reg	Lsusp	—

レジスタ Rgoal で指定されたゴールレコードの POS 番目の引数、または、レジスタ Rstruct で指定された構造体の POS 番目の要素の値をレジスタ Reg に読み込み、デレファレンスと書き戻しを行う。なお、構造体の場合には読み込むときに Rstruct の MRB が黒ならば要素と Reg の MRB を黒くする必要がある。この後、もし読み込んだデータが未定義変数か外部参照ならば、サスペンションスタックにその値(正確には REF)を積み、サスペンドラベル Lsusp へ分岐する。ここで、引数/要素の位置は即値の整数(0オリジン)で指定する。

read_ind_wait　　Rstruct, Rpos, Reg, Lsusp　　[019]

INT	OP = 019	Rstr.	Rpos
INT	Reg	Lsusp	—

要素位置がレジスタ Rpos 上の整数で指定される他は read_wait 命令と同じ。

read_cst_wait　　Lconst, Reg, Lsusp　　[016]

INT	OP = 016	Lconst
INT	Reg	Lsusp	—

コードモジュールの定数領域にあるラベル Lconst で指定される構造体定数をレジスタ Reg に読み込み、デレファレンスと書き戻しを行う。この後、もし読み込んだデータが外部参照ならば、サスペンションスタックにその値を積み、サスペンドラベル Lsusp へ分岐する。

load_wait_single	Rgoal, POS, Reg, Lsusp	[014]
read_wait_single	Rstruct, POS, Reg, Lsusp	[011]
read_ind_wait_single	Rstruct, Rpos, Reg, Lsusp	[01A]
read_cst_wait_single	Lconst, Reg, Lsusp	[017]

INT	OP = 014	Rgoal	POS'
INT	Reg	Lsusp	—

… ここでは POS' := POS+5 である。

INT	OP = 011	Rstr.	POS
INT	Reg	Lsusp	—

INT	OP = 01A	Rstr.	Rpos
INT	Reg	Lsusp	—

INT	OP = 017	Lconst	
INT	Reg	Lsusp	—

単一待ち最適化用の命令。読み込んだデータがまだ具体化されていないか外部参照の場合の処理のうち、サスペンションスタックにその値を積む処理が不要な他は、それぞれ、load_wait, read_wait, read_ind_wait, read_cst_wait と同じ。

3.2.4 データ読み込みの命令

メモリ上のデータを汎用レジスタに読み込む命令。元々のデータのある場所(ゴールレコード/構造体/コードモジュールの定数領域)の違いと、構造体の場合には要素位置の指定のしかた(即値/レジスタ)の違いにより、計4種類用意する。これらは基本的にボディ部でのみ使用する。

load	Rgoal, POS, Reg	[015]
read	Rstruct, POS, Reg	[012]

INT	OP = 015		Rgoal	POS'
INT	Reg	—	—	—

… ここでは POS' := POS+5 である。

INT	OP = 012		Rstr.	POS
INT	Reg	—	—	—

レジスタ Rgoal で指定されたゴールレコードの POS 番目の引数、または、レジスタ Rstruct で指定された構造体の POS 番目の要素の値をレジスタ Reg に読み込む。なお、構造体の場合には読み込む時に Rstruct の MRB が黒ならば要素と Reg の MRB を黒くする必要がある。ここで、引数/要素の位置は即値の整数(0オリジン)で指定する。

read_ind Rstruct, Rpos, Reg [01B]

INT	OP = 01B		Rstr.	Rpos
INT	Reg	—	—	—

要素位置がレジスタRpos上の整数で指定される他はread命令と同じ。

read_cst Lconst, Reg [018]

INT	OP = 018		Lconst	
INT	Reg	—	—	—

コードモジュールの定数領域にあるラベルLconstで指定される構造体定数をレジスタRegに読み込む。デレファレンス等は必要なく、たとえ外部参照であってもそのまま読み出す。

3.2.5 モジュール / コード定数読み込みの命令

モジュール定数(MODポインタ)やコード定数(CODポインタ)をレジスタ上に用意する命令。参照するモジュール/コードの種類(自モジュール/他モジュール/組込述語)の違いと、具体化の検査を行なうか否かにより、計8種類用意する。なお、将来コードモジュールのデマンドローディング機能を追加する場合には、このグループの命令にその処理を追加する予定である。

get_module_wait Lmodule, Reg, Lsusp [031]

INT	OP = 031		Lmodule
INT	Reg	Lsusp	—

コードモジュールの他モジュール参照テーブルにあるラベルLmoduleで指定されるモジュール記述子から、モジュール定数をレジスタRegに読み込み、デレファレンスと書き戻しを行う。この時、参照先モジュールがまだ具体化されていなかったり輸入されていない(外部参照になっている)場合があるので、その場合には、サスペンションスタックにモジュールが具体化される予定の変数/外部参照(正確にはそれを指すREF)を積み、サスペンドラベルLsuspへ分岐する。

get_module Lmodule, Reg [030]

INT	OP = 030		Lmodule	
INT	Reg	—	—	—

コードモジュールの他モジュール参照テーブルにあるラベルLmoduleで指定されるモジュール記述子から、モジュール定数をレジスタRegに読み込む。デレファレンス等は必要なく、たとえ未定義変数や外部参照であってもそのまま読み出す。

get_module_self Reg [033]

get_module_builtin Reg [032]

INT	OP = 033/032	Reg	—

自モジュール、または、組込述語モジュール(Dコードモジュール)を指すモジュール定数をレジスタRegにセットする。

get_code_wait Lcode, Reg, Lsusp [035]

INT	OP = 035	Lcode	
INT	Reg	Lsusp	—

コードモジュールの他モジュール参照テーブルにあるラベルLcodeで指定される述語記述子から、コード定数をレジスタRegに読み込む。ここで、その述語を初めて参照する場合には参照先のコード定数(述語の絶対アドレス)が不明なので、参照先述語の属するモジュールのエントリーテーブルを検索し述語名/引数個数からコード定数を生成する[3]必要がある。また、輸入後に初めて参照する場合には、オフセット(モジュール内相対アドレス)の形になっているので、参照先述語の属するモジュールのアドレスからアドレスを計算しコード定数を生成する必要がある。なお、参照先述語の属するモジュールががまだ具体化されていなかったり輸入されていない(外部参照になっている)場合があるので、その場合には、サスペンションスタックにモジュールが具体化される予定の変数/外部参照(正確にはそれを指すREF)を積み、サスペンドラベルLsuspへ分岐する。

get_code Lcode, Reg [034]

INT	OP = 034	Lcode		
INT	Reg	—	—	—

この命令は基本的にはget_code_waitと同じであるが、参照先述語の属するモジュールががまだ具体化されていなかったり輸入されていない(外部参照になっている)場合の処理が異なる。即ち、この場合にはコード定数を求めるためのDコードを生成する。このDコードは、

predicate_to_code(Module, PredName, PredArity, Code)

という組込述語を実行するもので、モジュール(実際はまだ未定義か外部参照)と述語名/引数個数からコード定数を求める。そして、RegにはこのDコードが求めた結果を具体化する変数をセットする。

[3]エントリーテーブルを検索した結果、述語が見つからない場合がありうるが、現在のVPIMではその場合の処理はまだ作成していない。基本的に、ガード部では失敗、ボディ部では例外とする予定である。

get_code_self	Lpred, Reg	[037]
get_code_builtin	ID, Reg	[036]

INT	OP = 037		Lpred	
INT	Reg	—	—	—

INT	OP = 036		—	—
INT	ID			
INT	Reg	—	—	—

自モジュール、または、組込述語モジュール(Dコードモジュール)にある述語のコード定数をレジスタRegにセットする。ここで、自モジュールの述語はラベルLpredで、組込述語は識別番号IDで指定する。

3.2.6 データ書き出しの命令

汎用レジスタ上のデータをメモリに書き出す命令。書き出す先(ゴールレコード/構造体)の違いと、構造体の場合には要素位置の指定のしかた(即値/レジスタ)の違いにより、計3種類用意する。

store	Reg, Rgoal, POS	[020]
write	Reg, Rstruct, POS	[020]

INT	OP = 020		Reg	Rgoal
INT	POS'	—	—	—

… ここでは POS' := POS+5 である。

INT	OP = 020		Reg	Rstr.
INT	POS	—	—	—

レジスタRcg上の値を、レジスタRgoalで指定されたゴールレコードのPOS番目の引数、または、レジスタRstructで指定された構造体のPOS番目の要素に書き出す。ここで、引数/要素の位置は即値の整数(0オリジン)で指定する。なお、このstore命令とwrite命令はPOSの値を調整することにより共通化することができ、VPIMでは同一の機械命令になっている。即ち、ゴールレコードでは引数配列の前に制御用領域があるので、そのワード数をPOSにあらかじめ足し込んでやることにより、write命令でゴール引数を操作している。

write_ind Reg, Rstruct, Rpos [021]

INT	OP = 021		Reg	Rstr.
INT	Rpos	—	—	—

要素位置がレジスタRpos上の整数で指定される他はwrite命令と同じ。

3.2.7 作業用メモリの操作の命令

作業用メモリの要素を操作する命令。作業用としてレジスタが足りなくなった時に当面必要でないデータをメモリ上に追い出すために用いられる。

save Reg, POS [040]

INT	OP = 040	Reg	POS

レジスタReg上のデータを作業用メモリのPOS番目に退避する。ここで、要素位置は即値の整数(0オリジン)で指定する。

restore POS, Reg [041]

INT	OP = 041	POS	Reg

作業用メモリのPOS番目に退避されたデータをレジスタRegに戻す。ここで、要素位置は即値の整数(0オリジン)で指定する。

3.2.8 データ型の検査の命令

汎用レジスタ上のデータ(デレフ、具体化済み)の型を検査する命令。基本的な6種類の型について命令を用意した。これらの命令ではデータが指定された型でない場合には失敗ラベルに分岐する。なお、データ型による多方向分岐命令は用意していない。

is_atom	Reg, Lfail	[050]
is_integer	Reg, Lfail	[051]
is_floating_point	Reg, Lfail	[052]
is_string	Reg, Lfail	[055]
is_list	Reg, Lfail	[053]
is_vector	Reg, Lfail	[054]

INT	OP = 050 ～ 054	Reg	—
INT	Lfail	—	—

レジスタ Reg 上のデータ型が、各々アトム、整数、浮動小数点数、ストリング、リスト、ベクタでなければ失敗ラベル Lfail へ分岐する。

3.2.9 値の検査の命令

汎用レジスタ上のデータ(デレフ、型検査済み)の値/要素数を検査する命令。アトムの値、整数の値、ベクタの要素数のそれぞれについて、2方向分岐命令と多方向分岐命令を用意した。そして、コンパイラでは、分岐の候補数とオプション等の指示から、2方向と多方向のどちらを使うかを決めている。

test_atom	ATOM, Reg, Lfail	[056]
test_integer	INTEGER, Reg, Lfail	[057]
test_arity	ARITY, Reg, Lfail	[058]

INT	OP = 056	—	—
ATOM	ATOM		
INT	Reg	Lfail	—

INT	OP = 057	—	—
INT	INTEGER		
INT	Reg	Lfail	—

INT	OP = 058	ARITY	Reg
INT	Lfail	—	—

2方向分岐命令。レジスタReg上のアトムの値が即値ATOMでない、または、レジスタReg上の整数の値が即値INTEGERでない、または、レジスタRegで指されたベクタの要素数が即値ARITYでないならば、失敗ラベルLfailへ分岐する。

hash_on_atom	Reg, MASK, Ltable	[059]
hash_on_integer	Reg, MASK, Ltable	[05A]

INT	OP = 059/05A	Reg	—
INT	MASK	Ltable	

ハッシュテーブルを使った多方向分岐命令。レジスタReg上のアトム/整数の値がラベルLtableで指定されるテーブルの中にあればそれに対応する命令へ分岐し、その値がテーブルに無い場合には次命令に進む。ここで、MASKはハッシュ関数で使用するマスク値である。Ltableで指定したテーブルの仕様は3.1.5項(1)を参照のこと。

jump_on_integer Reg, SIZE, BIAS, Ltable [05B]

INT	OP = 05B	Reg	—
INT	SIZE	—	—
INT	BIAS		
INT	Ltable	—	—

インデックステーブルによる多方向分岐命令。レジスタ Reg 上のアトム / 整数の値がラベル Ltable で指定されるテーブルの中にあればそれに対応する命令へ分岐し、その値がテーブルに無い場合には次命令に進む。この命令は各候補がある程度連続した値の場合に生成されるものであり、整数値により直接テーブルを引くことにより分岐先を決定する。ここで、SIZE はインデックステーブルの大きさであり、BIAS はそのテーブルを引く前に整数値から減算する定数である。即ち、インデックステーブルは整数値が BIAS 以上 BIAS+SIZE 未満の場合に対応するエントリを持つ。Ltable で指定したテーブルの仕様は 3.1.5項(1) を参照のこと。

3.2.10 任意データの比較の命令

2つの汎用レジスタ上の任意のデータ(デレフ、具体化済み)の比較を行う。

equal Reg1, Reg2, Lsusp, Lfail [060]

INT	OP = 060	Reg1	Reg2
INT	Lsusp	Lfail	

2つのレジスタ Reg1 と Reg2 の上のデータの型と値/要素数を比較し、同じでなければ失敗ラベル Lfail へ分岐する。双方が同じ型、同じ要素数の構造体の場合には、その全要素も再帰的に比較し、1つでも違っていれば失敗とし Lfail へ分岐する。この時、要素に対してはデレファレンスや具体化の検査が必要であり、もし未定義変数があった場合には、比較を中止し、その未定義変数をサスペンションスタックに積み Lsusp へ分岐する。なお、構造体のネストが深過ぎて途中で比較を打ち切った場合にも Lsusp へ分岐する。この場合には suspend 命令がサスペンションスタックが空であることをチェックすることにより、失敗(Reduction Fail) と判定する。

3.2.11 MRB 操作と実時間 GC の命令

MRB の操作と実時間GC(構造体の回収/再利用)を行なう命令。

mark Reg [068]

INT	OP = 068	Reg	—

レジスタ Reg の MRB を黒くする。

collect_list Reg [070]

collect_vector ARITY, Reg [071]

INT	OP = 070	Reg	—

INT	OP = 071	ARITY	Reg

レジスタ Reg の MRB が白であれば、それが指すリスト、または、要素数 ARITY 個のベクタを回収する。

collect_value Reg [073]

INT	OP = 073	Reg	—

レジスタ Reg の MRB が白で構造体、浮動小数点数、ストリング等を指していれば、指されている構造体等を回収する。なお、VPIM では構造体要素の再帰的な回収は行なっていない。

reuse_list Reg [080]

reuse_vector ARITY, Reg [082]

INT	OP = 080	Reg	—

INT	OP = 082	ARITY	Reg

レジスタ Reg の MRB が白であれば、それが指すリスト/ベクタ(VPIM ではショートベクタのみ)を同じ型/要素数の構造体に再利用する。即ち、なにもしない。MRB が黒であれば、新たにリスト、または、要素数 ARITY 個のベクタを割り付け、レジスタ Reg にセットする。

reuse_for_list Reg [081]

reuse_for_vector ARITY, Reg [083]

INT	OP = 081	Reg	—

INT	OP = 083	ARITY	Reg

レジスタ Reg の MRB が白であれば、それが指すリスト / ベクタ (VPIM ではショートベクタのみ) を異なる型 / 要素数の構造体に再利用する。即ち、タグの付け変えを行なう。MRB が黒であれば、新たにリスト、または、要素数 ARITY 個のベクタを割り付け、レジスタ Reg にセットする。

reuse_list_with_elements Reg, FLAGS [084]

reuse_vector_with_elements ARITY, Reg, FLAGS [086]

reuse_for_list_with_elements Reg, FLAGS [085]

reuse_for_vector_with_elements ARITY, Reg, FLAGS [087]

INT	OP = 084/085	Reg	FLAGS

INT	OP = 086/087	ARITY	Reg
INT	FLAGS	—	—	—

これは旧構造体上の要素を含めて構造体の再利用を行なう命令である。レジスタ Reg の MRB が白の場合には、普通の Reuse 系命令と同じであるが、MRB が黒の場合には、MRB が白ならばそのまま利用できる筈だった旧構造体上の要素を新たに割り付けた構造体上にコピーする必要がある。この時に FLAGS の情報を使う。これは 8bit のフラグ群で、各ビットで要素をコピーする必要があるか否かを指示する。即ち、その要素が 1 であれば対応する要素をコピーする必要があり、0 であればコピーの必要がないことを意味する。なお、コピーする前に要素の MRB を黒くする必要がある。

3.2.12 定数書き込みの命令

汎用レジスタに定数を書き込む命令。

put_atom	ATOM, Reg	[090]
put_integer	INTEGER, Reg	[091]

INT	OP = 090	—	—
ATOM	ATOM		
INT	Reg	—	—

INT	OP = 091	—	—
INT	INTEGER		
INT	Reg	—	—

レジスタReg上にアトムATOM、または、整数INTEGERを書き込む。

3.2.13 構造体 / 変数の割り付けの命令

新しい構造体/変数を割り付ける命令。

alloc_list	Reg	[0A0]
alloc_vector	ARITY, Reg	[0A1]

INT	OP = 0A0	Reg	—

INT	OP = 0A1	ARITY	Reg

新たにリスト、または、要素数がARITYのベクタを割り付け、そこへのポインタ(MRBは白)をレジスタRegに書き込む。

alloc_variable	Reg	[0A2]
alloc_void	Reg	[0A3]

INT	OP = 0A2/0A3	Reg	—

新たに一般の変数、または、VOID変数を割り付け、そこへのREFポインタ(MRBは白)をRegに書き込む。

3.2.14 アクティブユニフィケーションの命令

汎用レジスタ上のデータのアクティブユニフィケーションを行う命令。

unify_atom ATOM, Reg [0B0]

unify_integer INTEGER, Reg [0B1]

INT	OP = 0B0	—	—
ATOM	ATOM		
INT	Reg	—	—

INT	OP = 0B1	—	—
INT	INTEGER		
INT	Reg	—	—

レジスタ Reg 上の任意のデータと即値のアトム ATOM、または、整数 INTEGER とのアクティブユニフィケーションを行なう。

unify_bound_value Rbound, Reg [0B2]

INT	OP = 0B2	Rbound	Reg

レジスタ Reg 上の任意のデータとレジスタ Rbound 上の少なくともトップレベルが具体化されている構造体とのアクティブユニフィケーションを行なう。

unify Reg1, Reg2 [0B3]

INT	OP = 0B3	Reg1	Reg2

2つのレジスタ Reg1 と Reg2 の上の任意のデータのアクティブユニフィケーションを行なう。

3.2.15 実行制御の命令

ゴールの実行を制御する命令。

alloc_goal ARITY, Rgoal [0C0]

INT	OP = 0C0	ARITY	Rgoal

新たに ARITY 個の引数を格納できるゴールレコードを割り付け、そこへのポインタを Rgoal に書き込む。

collect_goal ARITY, Rgoal [0C1]

INT	OP = 0C1	ARITY	Rgoal

レジスタ Rgoalで指されるARITY個の引数を格納していたゴールレコードを回収する。

set_caller_code_info Rgoal [0C3]

INT	OP = 0C3	Rgoal	—

レジスタ Rgoalで指されるゴールレコードの環境レコードに呼び出し元コード情報を付加する。なお、このときセットする呼び出し元コードの指す位置はこの次の(通常はEnqueue系)命令である。これはKL1で定義された組込述語を呼び出す時に使用される。

enqueue Rgoal, Rcode, ARITY [0C4]

INT	OP = 0C4		Rgoal	Rcode
INT	ARITY	—	—	—

プラグマが付かなかった場合のエンキュー命令。レジスタ Rgoalで指されるゴールレコードに、レジスタ Rcodeで指定される述語を実行するような制御情報を書き込み、ゴールスタックにエンキューする。引数個数ARITYである。なお、Rcodeにはget_code系命令で求めたコード定数を与える。

enqueue_with_priority Rgoal, Rcode, Rpriority, ARITY [0C5]

INT	OP = 0C5		Rgoal	Rcode
INT	Rprio.	ARITY	—	—

プライオリティ関係のプラグマが付いた場合のエンキュー命令。レジスタ Rgoalで指されるゴールレコードに、レジスタ Rcodeで指定される述語を実行するような制御情報を書き込み、レジスタ Rpriority上の論理プライオリティ(整数)で指定されるゴールスタックにエンキューする。

enqueue_to_cluster	Rgoal, Rcode, Rcluster, ARITY	[0C6]
enqueue_resident_to_cluster	Rgoal, Rcode, Rcluster, ARITY	[0C8]
enqueue_emigrant_to_cluster	Rgoal, Rcode, ARITY	[0CA]

INT	OP = 0C6/0C8		Rgoal	Rcode
INT	Rcls.	ARITY	—	—

INT	OP = 0CA		Rgoal	Rcode
INT	ARITY	—	—	—

クラスタ指定関係のプラグマが付いた場合のエンキュー命令。enqueue_to_clusterではレジスタRgoalで指されるゴールレコードに、レジスタRcodeで指定される述語を実行するような制御情報を書き込み、レジスタRcluster上のクラスタ番号(整数)で指定されるクラスタのゴールスタックにエンキュー、即ち、%throwする。enqueue_resident_to_clusterはゴールにクラスタレジデント属性を付加してからenqueue_to_clusterと同様の処理を行なう。enqueue_emigrant_to_clusterはゴールのクラスタレジデント属性を解除してからenqueueと同様の処理を行なう。

enqueue_to_processor	Rgoal, Rcode, Rprocessor, ARITY	[0C7]
enqueue_resident_to_processor	Rgoal, Rcode, Rprocessor, ARITY	[0C9]
enqueue_emigrant_to_processor	Rgoal, Rcode, ARITY	[0CB]

INT	OP = 0C7/0C9		Rgoal	Rcode
INT	Rprc.	ARITY	—	—

INT	OP = 0CB		Rgoal	Rcode
INT	ARITY	—	—	—

プロセッサ指定関係のプラグマが付いた場合のエンキュー命令。enqueue_to_processorではレジスタRgoalで指されるゴールレコードに、レジスタRcodeで指定される述語を実行するような制御情報を書き込み、レジスタRprocessor上のプロセッサ番号(整数)で指定されるクラスタのゴールスタックにエンキュー、即ち、%throwする。enqueue_resident_to_processorはゴールにプロセッサレジデント属性を付加してからenqueue_to_processorと同様の処理を行なう。enqueue_emigrant_to_processorはゴールのプロセッサレジデント属性を解除してからenqueueと同様の処理を行なう。

execute Rcode, ARITY [0D0]

execute_self Lpred, ARITY [0D1]

INT	OP = 0D0	Rcode	ARITY

INT	OP = 0D1		Lpred	
INT	ARITY	—	—	—

現在のゴールの処理を完了し、CGPで指されるゴールレコードを用い引数(RcodeまたはLpred)で指定された述語の処理に移る。しかし、実際にはスリットチェック等の処理も行なう必要があるので以下のようになる。

1. リダクション数のカウントを行う。
2. スリットチェックフラグを調べ、

2a. もし立っていない場合には、CGPで指されたゴールをすぐ実行する。

2b. もし立っていた場合には、CGPで指されるゴールレコードをすぐ実行することはできないので、enqueue命令と同様の処理によりゴールスタックにエンキューした後、スリットチェック処理を行なう。その処理が終了したら、ゴールスタックから1つのゴール(直前にエンキューしたものとは違うかも知れない)をデキューし、その実行に移る。

proceed [0D2]

INT	OP = 0D2	—	—

現在のゴールの処理を完了し次ゴールの実行に移る。しかし、実際にはスリットチェック等の処理も行なうので以下のようになる。

1. リダクション数のカウントを行う。
2. スリットチェックフラグを調べ、もし立っていればスリットチェック処理を行なう。
3. ゴールスタックから1つのゴールをデキューしその実行に移る。

suspend Lpred, ARITY [0D3]

INT	OP = 0D3		Lpred	
INT	ARITY	—	—	—

現在のゴールの処理を中断し次ゴールの実行に移る。しかし、実際にはスリットチェック等の処理も行なうので以下のようになる。

1. サスペンションスタックの状態を調べ、

1a. そこに未定義変数と外部参照だけが積んである場合にはゴールの中断処理を行なう。これはCGP レジスタで指されるゴールレコードに、ラベルLpredで指定される述語(引数ARITY個)を実行するような制御情報を書き込み変数にフックする処理である。

1b. そこに未定義変数以外のものが含まれている場合には、具体化検査からsuspend命令までに間で変数が具体化されたことを意味するので、今中断しようとしたゴールの再実行を行なう。即ち、ラベルLpredで指定されるアドレスに分岐する。この場合、suspend命令の後半の処理へは進まない。

1c. そこが空の場合には、そのゴールはコミットすることができなかったことを意味するので、リダクションフェイルの例外を報告する。

2. スリットチェックフラグを調べ、もし立っていればスリットチェック処理を行なう。

3. ゴールスタックから1つのゴールをデキューしその実行に移る。

suspend_single Reg, Lpred, ARITY [0D4]

INT	OP = 0D4	Reg	—
INT	Lpred	ARITY	—

現在のゴールの処理を中断し次ゴールの実行に移る。しかし、実際にはスリットチェック等の処理も行なうので以下のようになる。

1. レジスタRegで指定された変数の状態を調べ、

1a. それが未定義変数か外部参照の場合には、ゴールの中断処理を行なう。これはCGP レジスタで指されるゴールレコードに、ラベルLpredで指定される述語(引数ARITY個)を実行するような制御情報を書き込み変数にフックする処理である。

1b. それが具体化されていた場合には、具体化検査からsuspend_single命令までに間で変数が具体化されたことを意味するので、今中断しようとしたゴールの再実行を行なう。即ち、ラベルLpredで指定されるアドレスに分岐する。この場合suspend_single命令の後半の処理へは進まない。

2. スリットチェックフラグを調べ、もし立っていればスリットチェック処理を行なう。

3. ゴールスタックから1つのゴールをデキューしその実行に移る。

fail Lpred, ARITY [0D5]

INT	OP = 0D5		Lpred	
INT	ARITY	—	—	—

現在のゴールが失敗したので例外を発生し次ゴールの実行に移る。しかし、実際にはスリットチェック等の処理も行なうので以下のようになる。

1. CGPで指されるゴールの Reduction Fail の例外を報告する。
2. スリットチェックフラグを調べ、もし立っていればスリットチェック処理を行なう。
3. ゴールスタックから1つのゴールをデキューしその実行に移る。

otherwise Lpred, ARITY [0D6]

INT	OP = 0D6		Lpred	
INT	ARITY	—	—	—

実行順序制御 otherwise の処理を行なう。即ち、サスペンションスタックの状態を調べ、そこが空ならば次命令の処理に進むが、空でない場合には次命令には進まずに、suspend命令と同様のサスペンド処理を行なう。

alternatively [0D7]

INT	OP = 0D7	—	—

実行順序制御 alternatively の処理を行なう。即ち、サスペンションスタックの状態を調べ、そこに外部参照が積んである場合には、それに対する Eager Read を行う。

jump Label [0D8]

INT	OP = 0D8	Label

無条件分岐命令。Label指定された命令へ分岐する。

3.2.16 ガード組込述語の命令

ガード組込述語に対応する命令。基本的に、その組込述語に1対1で対応する命令が用意されている。

builtin_name **Rarg1, Rarg2, … , Label** [2xx]

builtin_name **Rarg1, Rarg2, …** [2xx]

INT	OP = 2xx	Reg1	Reg2
INT	Label	—	—

… 2引数の場合(失敗の可能性あり)

INT	OP = 2xx		Reg1	Reg2
INT	Reg3	Label		—

… 3引数の場合(失敗の可能性あり)

INT	OP = 2xx	Reg1	Reg2

… 2引数の場合(必ず成功する)

INT	OP = 2xx		Reg1	Reg2
INT	Reg3	—	—	—

… 2引数の場合(必ず成功する)

ガード組込述語命令は引数を全てレジスタで指定するようなっている。また、失敗する可能性のあるものと必ず成功するものがあり、失敗する可能性のあるものには失敗ラベルがついている。一般的なデータ型の入力を期待する引数のための型検査は組込述語命令の手前に別の命令が出されるので、組込述語命令内では入力引数の型検査は行なわない。従って、単純な型検査の組込述語の場合には、その処理が全て別の命令で実現されてしまうので、対応する組込述語命令は存在しない。

3.2.17 ボディ組込述語の命令

ボディ組込述語に対応する命令。その組込述語に1対1で対応する命令が用意されている。

b_builtin_name **Rarg1, Rarg2, …** [3xx]

INT	OP = 3xx	Reg1	Reg2

… 2引数の場合

INT	OP = 3xx		Reg1	Reg2
INT	Reg3	—	—	—

… 3引数の場合

ボディ組込述語命令は引数を全てレジスタで指定するようなっている。ボディ組込述語の場合には、ガードと違い、入力引数のデレファレンスや型検査を全て組込述語命令の中で行なう必要がある。

第 4 章

メモリ管理

執筆担当者: 今井

本章では、VPIMにおけるメモリ管理方式について述べる。

4.1 メモリマップ

VPIMでは、クラスタ内のメモリ空間を、大きく「システム共有固定領域」、「プロセッサ局所固定領域」、「ヒープ領域」の3つの領域に分けて管理を行っている。このうち、ガーベジコレクション(GC)の対象となるのはヒープ領域だけである。

VPIMにおけるメモリマップを、図4–1に示す。

VPIMでは、メモリ構成/メモリ操作には、次のような仮定を置いている。

- **メモリはすべて共有メモリである**
  物理的な局所メモリは存在しない。
- **クラスタ内の各 PE は、アドレス空間を共有している**
  PE0の1000番地は、他の全てのPE にとっても1000番地である。
- **アドレス変換機構は仮定しない**
  KL1の並列処理方式では、ハードウェアによる論理 / 物理アドレス変換を特に必要としないため、「物理アドレス = 論理アドレス」である。
- **メモリは0番地から連続的に実装されている**
  使用可能なメモリ空間はアドレスが連続している。途中にメモリが実装されていない領域は存在しない。
- **ワードアドレッシングである**
  1ワードは40ビット(タグ部8ビット,値部32ビット)からなっている。
- **任意のアドレスは、値部32ビットで表現できる**
  アドレスは00000000(hex) 番地から始まり、最大でもFFFFFFFF(hex) 番地までである。すなわち、実装できるメモリは最大4G Wordである。

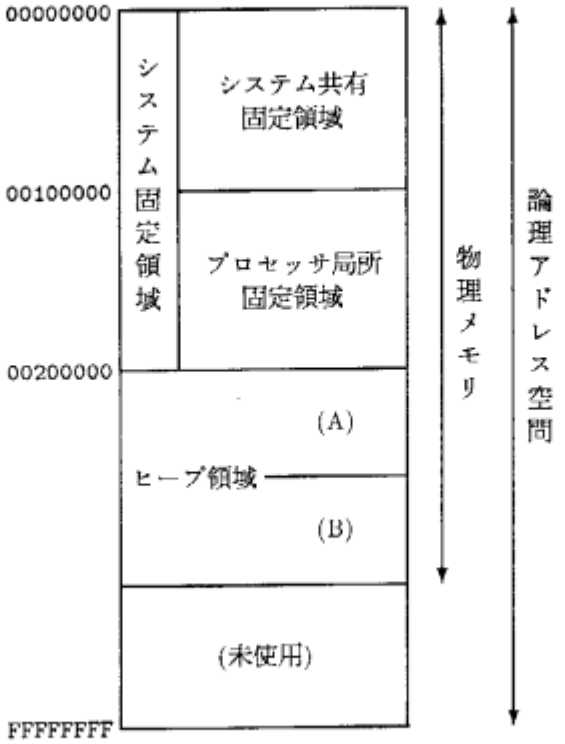


図 4–1: VPIM のメモリマップ

4.2 システム固定領域

システム固定領域は、クラスタ内の一括GC等によって移動しない領域である。システム固定領域とヒープ領域の境界や、システム固定領域内の細かい領域の境界はシステム設計時に定められ、実行時に変更されることはない。

システム固定領域には、各PEが局所メモリとして利用する領域(プロセッサ局所固定領域)と、クラスタ内の全PE が共有する領域(システム共有固定領域)がある。

4.2.1 プロセッサ局所固定領域

プロセッサ局所固定領域 は、クラスタ内の個々のPEがローカルに使用する領域であり、全体の領域を8等分して使う。この領域へのアクセスに関しては、排他制御を必要としない。

プロセッサ局所固定領域をさらに細かく分割し、以下のような情報を置く。

1. メモリ上に割当てられたレジスタ相当のもの

   論理的にはレジスタであるが、物理レジスタ数不足等の理由により、使用頻度の低いものはこの領域に置く。

   また、KL1コンパイラにより、引数レジスタが不足して作業用データを引数レジスタ上に置けなくなった場合には、save命令によって、この領域にデータを退避させる。なお、引数レジスタ上に、再びデータを引き上げる命令はrestore である(3.2節参照)。

2. サスペンドスタック

ガード部の実行で変数が具体化されていなかったためにコミットできなかった場合、その原因となった未定義変数セルへのポインタを置く領域(7.3節参照)。

3. レディゴールスタックのエントリ

実行可能なゴールをそのゴールのプライオリティに応じてリンクして格納するために用いる領域で、実際にはそのリンクの先頭を置く(6.2節参照)。

4. ユニファイスタック

受動部において、値の決まった構造体同士のユニフィケーションを行う場合に、その構造を再帰的に比較してゆくために用いるスタック領域(7.2節参照)。

5. 内部イベントスタック

全PEにシグナルを送ろとした時、既に他のPEによってバリアが設定されていた場合、次のスリットチェックで処理するために用いる。この場合、内部イベントスタックに先送りするイベントをプッシュする。(8章参照)。

等に使用する。

4.2.2 システム共有固定領域

システム共有固定領域は、クラスタ内のPE によって共有される領域であり、以下のような領域に分けられて情報が置かれる。

1. システムコンフィギュレーション領域

クラスタ内のPE台数、システム内のクラスタ数などのシステム構成情報が格納される。

2. メモリ管理用領域

グローバルヒープトップのアドレス、現在使用中のヒープ面(11章参照)の最後のアドレス等が格納される。

3. クラスタ内通信用領域

PE 間でのシグナル通信 (スリットチェックで検出, 8章参照)において、通信内容を補足するフラグやゴールポスト等が格納される。

4. システムコード領域

システムの初期化時にIコードモジュール(5.3節参照)をロードし、実行時にDコードゴールが参照する。さらに実機においては、マイクロ命令(内部命令)から呼び出される処理系ルーチン等も格納される。

5. クラスタ間通信用領域

クラスタ間通信処理に必要な輸出表や輸入表、構造体表などのデータを格納する(9.1節参照)。
輸出表は、クラスタ内のデータを他のクラスタから参照する時に経由する間接テーブルである。データが置かれているアドレスがGC(11章参照)によって変更になっても、GCで移動しないこの領域にある輸出表を経由することで、アドレスが変更されたことを他のクラスタに通知する必要を避けるために設けている。

輸入表は、クラスタ外のデータを参照するための間接テーブルで、一括GC後、この領域を走査することで、参照しなくなったクラスタ外のデータがあれば、その参照が消えたことを輸出したクラスタに通知するために用いる。

輸出入のために、この領域に確保されている領域には、

- ◇ 輸出表エントリ
- ◇ 黒輸出(アドレス)ハッシュ表
- ◇ 白輸入表
- ◇ 黒輸入(ID)ハッシュ表
- ◇ 構造体アドレスハッシュ表
- ◇ 構造体IDハッシュ表

がある。

4.3 ヒープ領域

4.3.1 ヒープ領域に置かれるデータ

ヒープ領域 は、KL1の実行で用いるメモリ領域であり、GCの対象となる連続したメモリ空間である。コピー方式による一括GCを採用しているため(11章参照)、ヒープ領域を二分割し、そのうちの一方のみを交替に使用する。

ヒープ領域には、変数セルのような小さいデータ構造ばかりではなく、コードモジュールのような大きいデータ構造も割当てられる。

1. KL1プログラムが使用するデータ構造
2. KL1のゴール実行環境を格納するゴールレコード
3. ユーザプログラムコード
4. PIMOSプログラムコード

等が置かれる。

4.3.2 ヒープの論理ページ管理

ヒープ領域は、データのサイズ毎のフリーリスト(後述)を用いて管理することにより、実行時のMRB-GC(4.4節参照)によって再利用される。フリーリストの回収・割当てをPE毎に効率的に行うために、各種のフリーリストをPE毎に管理する。

システム立ち上げ当初では、各PEが一斉にフリーリストのための領域をヒープ領域から確保しようとして、未使用のヒープ領域へのアクセスが集中する傾向がある。このため、各PEにおいて新たなフリーリストの生成を効率良く行なえるような工夫が必要である。また、クラスタ全体としてヒープ領域が不足した場合は、処理を中断して一括GCを起動する。この場合、一括GCの起動をゴールリダクションの切れ目で行うためには、ヒープ領域不足が生じる前にヒープ領域の残量が少なくなってきていることを

検知できなければならない。

このような目的のために用意するヒープ領域の管理方式を列挙する。

1. ヒープ領域は、論理的に決められた大きさの**論理ページ** (または、単に**ページ**)を基本単位として各PEに割当てられる。ページの大きさはハードウェアとは独立であり、256ワードとしている。
2. ヒープ領域不足による一括GCの起動はリダクションの切れ目で行う。このため、各PEは一定量以上のページ単位のヒープ領域を**割当ページ** として持っている。
3. ヒープ領域不足により、ページが確保できなかった場合、その事象をスリットチェックの要因として保持し、割当ページから1ページを取り出して、実行中のゴールリダクションを続行する。
4. リダクションの切れ目において割当ページ不足を検出した時は、一括GCを起動する。
5. 各PEが使用するKL1のデータ構造のためのメモリ領域は、MRBによって再利用するためにPE毎のフリーリストで管理される。

(1) 各PEにおける割当ページ管理

各PEは規定数以上の割当ページを持つ。割当ページの大きさは、ページサイズより小さい各種のフリーリストを生成しながら一つのゴールリダクションを実行するのに十分な大きさとする。

各PEはこの目的のために確保される割当ページを、フリーリスト形式で持っている (以下**ページのフリーリスト** と呼ぶ)。

(2) PEにおけるページサイズ以下のデータの生成

PEは、ある大きさのフリーリストが無くなった時、1ページだけヒープを伸ばしてページを獲得し、フリーリストを生成する。もし、ページを獲得できなかったときは、スリットチェック要因をセットし、割当ページの中から1ページを確保し、フリーリストを生成する。

スリットチェック要因は、次のリダクションの切れ目で全PEに報告される。

(3) PEにおけるページサイズ以上のデータの生成

ページサイズ以上のデータは、ボディ組込述語 new_vector, new_string においてのみ生成される。これらの組込述語は、ヒープをページ単位で伸ばして領域を確保し、フリーリスト管理は行なわない。

ヒープ不足によりこの確保に失敗した場合は、スリットチェック要因をセットし、これらのボディ組込述語を実行するためのDコードゴール(5.2節参照)を作りエンキューする。

(4) ヒープのフリーリスト管理

実行時のMRB-GCなどによる即時回収で利用する1ページより小さいデータ領域のフリーリスト管理について述べる。

フリーリストの仕様は、以下の通りである。

終端 EOL (End Of Link)タイプ。

リンク 終端と異なるタグと値であれば良い。デバッグの効率化のために、FLC (Free List Cell)というタグとしている。

初期化 各要素の未使用部分の初期化は行なわない。

まず、フリーリストの要素とするデータの枠組は、VPIMでは2の冪乗サイズで丸めている。即ち、1, 2, 4, 8, 16, 32, 64, 128, 256 の9種類であるこのため、生成するデータ構造がフリーリストの要素の大きさに合わないときは、大きめで一番近いものを使う。例えば、3ワードのデータを割り付けるときは、4ワードのフリーリストを使う。

フリーリストの先頭のセルを指すポインタ(フリーリストトップ)は、レジスタに置く。ただし、PIM実機で物理レジスタ個数の制限などの理由で物理レジスタに置けない場合は、特に使用頻度が高くないもの[1]は、プロセッサ局所固定領域に置けば良い。

ある大きさのフリーリストが空になった場合は、ヒープを伸ばして1ページを獲得し、そのページ内を全てその大きさのフリーリストにする。

なお、以下に述べるようなことは、管理コストが大きくなるので行なわない。

- ある大きさののフリーリストが不足した時に、それ以上のサイズのフリーリストからセルを取ってきて、これを分割して使うこと。
- ある大きさのセルを回収した時、それに隣接したセルが回収済の場合にそれらを併合して大きな領域として再利用すること(バディ(相棒)システム)。

4.4 MRB

4.4.1 概要

KL1では、破壊的代入が許されないこと、バックトラックがないことなどの理由でメモリが急速に消費されるという問題がある。この問題を解決するために、MRB (Multiple-Reference-Bit)方式と呼ぶ低コストの参照数管理を導入している。

MRB方式とは、参照ポインタにMRBと呼ばれる1ビットの多重参照情報を持たせ、MRBがOFF(MRB◦と表記し、「白い」と読む)ならばデータへの単一参照であることを保証し、MRB情報により単一参照であることが分かる参照が消費される時に、参照先のデータ領域を回収するものである。(未定義変数については少し違った扱いをする。)このように、MRBを用いた即時GCをMRB-GCと呼ぶ。

MRB情報は、ベクタ要素の破壊的更新、ストリームマージの最適化(7.6節参照)や輸出入管理(9章参照)の最適化などにも用いられる。

MRB-GCでは、いったん多重参照されるようになったデータはMRBがON(MRB• と表記し、「黒い」と読む)のポインタで指されるので、その後参照されなくなっても回収されない。したがって、クラスタ内で回収されないゴミが段々と溜まって行き、いつかはヒープ領域が使い尽くされてしまう。この

[1] 1,2,16ワード以外が候補として考えられる。

ため、VPIMでは一括GC(11章参照)も実装し、両方式を併用している。

4.4.2 KL1-B 命令における MRB 操作と GC

データに対して、参照数が明らかに変化する場合に KL1-B 命令によりMRB を操作する。

(1) 変数 / 構造体などの作成

ボディ部において、新たに変数セルを割り付けたり、構造体を作ったりする場合である。すなわち、

$$p\ :-\ true \mid q(X),\ r(X,[Car|Cdr]).$$

の例では、ボディ部で新たに割り付けられる変数 X の出現回数が2であるので、このセルに対するポインタの MRB は、∘である。一方、

$$p\ :-\ true \mid q(X),\ r(X,[Car|Cdr]),\ s(X).$$

の例では、X が3回(以上)現れるので、このセルに対するポインタの MRB は、•である。構造体 $[Car|Cdr]$ に対するポインタのMRB は、いずれも参照パスが1本なので∘である。

(2) 参照数の増加

参照数が増加する場合、例えば

$$p(X)\ :-\ true \mid q(X), r(X).$$

のような例では、コミット時点で X の参照数が増えるため、mark命令により X のMRB を ON にしたものを、$q/1, r/1$ に分配する。

(3) 参照数の減少

参照数が減少する場合、例えば

$$p(X)\ :-\ true \mid true.$$

のような例では、コミット時点で X の参照数が減るため、collect_value命令により X のMRB が OFFであれば、X の指すデータは(MRBが白い限り)再帰的に回収できる。ただしVPIMでは、その表層のみの回収に留めている[2]。例外として、リストのCDR部についてのみ、深さが3までは回収を行う。

また、

$$\begin{aligned} p([X|Y]) &:- true \mid q0(X),\ q1(Y). \\ p(\{X,Y,Z\}) &:- true \mid q0(X),\ q1(Y),\ q2(Z). \end{aligned}$$

のような例では、コミット時点でリストセルおよび、3要素ベクタの表層について参照数が減少するので、collect_listないし、collect_vector命令を発行し、その表層の構造を指すポインタのMRBが OFF であれば回収する。

[2]深さ制限がないと、多段にネストした構造の場合にキャッシュミスを繰り返しながら回収を繰り返す可能性があることから、オーバヘッドが大きいため。

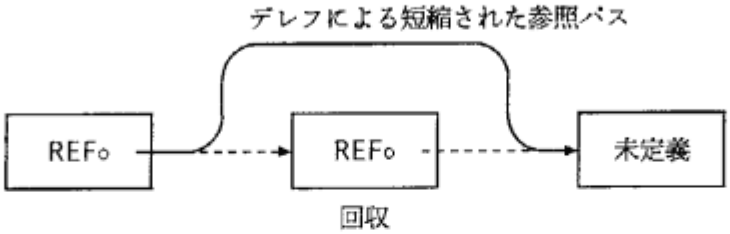


図 4 2: デレファレンスによる間接参照セルの回収

(4) デレファレンスによる回収

間接参照ポインタセル (REFタイプ) をたどり、データの置かれているセルをアクセスする操作を**デレファレンス** (または縮めて**デレフ**) と呼ぶが、この操作においてMRBがOFF である限り、間接参照ポインタセルを回収することができる(図4–2参照)。

デレファレンスにより、全ての間接参照ポインタのMRBが∘であった場合、その参照パスは**「白い参照パス」** であると呼ぶ。また、間接参照ポインタのうち、一つでもMRBが•であった場合、その参照パスは**「黒い参照パス」** であると呼ぶ。

第 5 章

D コード

執筆担当者: 畑澤

本章では、VPIMにおけるDコードについて説明する。

5.1 概要

Dコードは、VPIMの起動時にあらかじめシステムコード領域にロードされる各種のKL1プログラムである。VPIMでは、ユーザプログラムの処理の一部を、KL1ゴールの形で割り出して、実行することがある。この場合、VPIMの処理の都合で生成されたゴールは、あらかじめロードされているKL1プログラムをコードとして参照する。このようなゴールを、Dコードゴールと呼ぶ。

VPIMがDコードゴールを生成して処理を行なうのは、

- ボディ組込述語の処理において、入力引数の具体化を待つ場合。
- 能動部のユニフィケーション処理において、構造体同士のユニフィケーションを、再帰的に行なう場合。

などである。これらについては、次節で詳しく述べる。

Dコードゴール生成の利点は、処理系の処理の一部をKL1ゴールとして割り出すことによって、ユーザゴールと同じレディゴールスタックを使ってスケジューリングを行なうことが出来ることである。また、上述したユニフィケーションの例ではDコードを使わずに処理を行なう場合、スタックを用いて処理を行なうことになるが、KL1ゴールの形で処理を行なえばスタック領域の大きさを気にしなくても良いという利点がある。しかし、Dコードを使った処理はどうしても処理速度が遅くなる。このためDコードゴールによる処理が有効であると考えられるのは、処理系の中で実現する頻度が低い事象(例外処理など)に対する処理である。

5.2 D コードゴールの生成と実行

この節では、VPIMにおいてDコードゴールが生成される代表的な場合について、生成されるタイミング、生成されたゴールの性質などについて説明する。

(1) ボディ組込述語のサスペンド処理

ボディ部の組込述語を実行する際に、入力となる引数が未定義または外部参照の場合、処理を中断してDコードゴールを生成する。この場合、出力引数には未定義変数が渡される。この未定義変数は、Dコードゴールにも渡され、Dコード中で組込述語の実行後にその出力とユニファイされる。

このために、ボディ部の各組込述語に対しては、それぞれに対応するDコードが用意されている。その例を以下に示す。

$$add(A,B,C) : -wait(A),\ wait(B) \mid builtin : add(A,B,C).$$

これらのDコードの特徴は、入力時に具体化していないと実行できない引数に対して wait を行なっていることである。

Dコードゴールの生成以降の処理手順は、以下のようになる。

1. Dコードゴールが生成され、レディゴールスタックに格納される。
2. Dコードゴールがレディゴールスタックから取り出され、ガード部が評価される。
   - もしこの時点で、waitの対象になっていた全引数が具体化されていれば、ボディ部に書かれている組込述語が実行される。
   - waitの対象で具体化されてない引数があれば、このゴールの実行はサスペンドされる。
     この場合には、引数が具体化された時点で通常のユーザゴールと同じリジューム処理が行なわれ、コミット後に組込述語が実行される。

(2) enqueue系命令のサスペンド処理

enqueue系命令の実行において、enqueueするゴールのコードが未定義変数の場合や、プラグマとして指定されるプライオリティ等の情報が未定義変数の場合には、ボディ組込述語の場合と同じようにDコードゴールを生成して、enqueue処理をサスペンドする。

このために、enqueue系命令に対しては、それぞれに対応するDコードが用意されている。これらのDコードでは、コード等の入力引数が具体化したらゴールのenqueue処理を行なう組込述語を実行する。その例を以下に示す。

$$enqueue\_with\_priority(A,B,C) : -wait(B),\ wait(C) \mid$$
$$builtin : enqueue\_with\_priority(A,B,C).$$

この例で、第1引数にはenqueue対象のゴールへのポインタが渡されるが、これについてはwaitしないことを注意しておく。[1]

Dコードゴールの生成以降の処理手順は、ボディ組込述語の場合と同じである。

[1] ゴールへのポインタはHOOK系のタイプになるが、これらは通常未定義変数へのポインタとして扱われるために、waitしても具体化することはない。ゴールへのポインタは、通常のKL1データとしては現れないものである。enqueue系の組込述語は、この意味で特殊なものであり、Dコードとしてしか使われることはない。

(3) ユニファイの再試行

能動部におけるユニフィケーションにおいて、未定義変数への書き込みはComapre & Swapによって行なっている。これに失敗した場合には、以下に示すようなDコードゴールを生成してユニファイの再試行を行なう。

$$dcode\_unify\_retry(X,Y) : -true \mid X = Y.$$

ここで、Xは未定義変数へのポインタであり、YはCompare & Swapで書き換えようとした値である。

Compare & Swapに失敗することは、比較的稀なことと考えられ、これによってVPIMの処理を単純化することが出来る。

(4) 構造体同士のユニファイ

能動部における構造体同士のユニフィケーションにおいては、それらの全ての要素についてユニフィケーションの処理を行なう必要がある。これをVPIMの中で行なう場合にはスタックを用いて再帰的に処理を行なわなければならず、かなり複雑になってしまう。そこで、このような場合にもユニフィケーションのためのDコードゴールを生成することによって、通常のユーザゴールと同じスケジューリングで、処理を行なう。

使用するDコードは、リストについては、

$$dcode\_list\_unifier([X1|X2],[Y1|Y2]) : -true \mid X1 = Y1,\ X2 = Y2.$$

また、ベクタについては、

$$\begin{aligned}
&dcode_vect_unifier(0,V1,V2) : -true \mid true.\\
&dcode_vect_unifier(N,V1,V2) : -N > 0 \mid\\
&\quad builtin : subtract(N,1,N1),\\
&\quad builtin : set_vector_element(V1,N1,Elm1,_,NV1),\\
&\quad builtin : set_vector_element(V2,N1,Elm2,_,NV2),\\
&\quad Elm1 = Elm2,\\
&\quad dcode_vect_unifier(N1,NV1,NV2).
\end{aligned}$$

となっている。

KL1プログラムの実行において、構造体同士のユニフィケーションを能動部で行なうことは稀と考えられるため、多少実行速度が遅くなるがこれによってVPIMの処理を単純化できることは有効であると考えられる。

(5) idle ゴール

現在VPIMでは、各PEにおいて実行すべきゴールがなくなった場合には、idleゴールと呼ばれる特殊なDコードゴールを走らせている。これは、VPIMの起動時に最低のプライオリティでレディゴールス

タックに格納され、レディゴールスタックに他の実行すべきゴールが全くなくなると実行されるものである。

idleゴールのDコードは以下のように定義されている。

$$dcode\_idle\_goal \ :- \ idle \mid builtin : pims\_shut\_down.$$
$$alternatively.$$
$$dcode\_idle\_goal \ :- \ true \mid dcode\_idle\_goal.$$

ここで、第1節の ガード組込述語 idle中では、ゴールの要求を行なった後、スリットチェックレジスタを見ながらループし、次のいずれかになるまで待つ。

1. スリットチェックフラグが立てられたら、フェイルして、第2節を実行。
2. ユーザプログラムの終了を検出した場合コミットして、組込述語 pims_shut_down において終了処理を行う。

第2節では、自分自身を呼び出しているだけであるが、実際には、executeする前にスリットチェック処理を行ない、ここで他PEからゴールが投げられていれば、その処理に移る。idle ゴール自体は、スリットチェック前にenqueueされる。

(6) %throw_goal 受信時の処理

%throw_goalを受信した時には、そのゴールのコードへのポインタ、などを輸入し、ゴールレコードを生成してゴールスタックに入れればよい。但し、この時点ではコードへのポインタが直ちに得られるとは限らないので、実際にはDコードゴール

$$dcode\_apply(Code, Argv) \ :- \ true \ \mid \ builtin : apply(Code, Argv).$$

をエンキューする。

(7) モジュールの実体がない時の処理

輸入した構造体IDが構造体表に登録されていない時は、コードとなる変数セルを1つ割り付け、モジュールへのポインタ(この場合は外部参照セルへのポインタ)とコードのモジュール内オフセットから、コードへのポインタを得るDコードゴール

$$dcode\_read\_module(Mod, Ofst, Code) \ :- \ wait(Mod) \mid$$
$$builtin : module\_offset\_to\_code(Mod, Ofst, Code).$$

をエンキューする。

(8) スパイモードでの他クラスタへのゴール送信

スパイモードの時にenqueue, apply等によりサブゴールを生成する場合には、そのサブゴールも親と同じ述語群をスパイ対象とするスパイモードで生成する必要がある。これは、クラスタ間にまたがる場合も同様で、ゴール送信の処理においてスパイモードのための情報/属性を正しく伝達する必要がある。このため、

$$dcode\_enqueue\_cluster\_with\_spy\_mode(Cluster, Code, Argv, SpiedCodeV, SpyId) \ :-$$
$$integer(Cluster), wait(Code) \mid$$
$$builtin : apply\_spying(Code, Argv, SpiedCodeV, SpyId)@cluster(Cluster).$$

というDコードゴールを用い、送信先クラスタでapply_spyingを実行してもらう。

5.3 D コードモジュール

前節に述べたように、Dコードにはいろいろな種類のものがあるが、これらは全て、Dコードモジュール[2]という一つのモジュール中のpublic述語であるという形式をとっている。

このモジュールは、KL1プログラムにおいて、

$$builtin : 述語名(...) \quad または \quad module\#builtin$$

のように記述することによって参照可能である。

また、KL1で定義された組込述語 もこのモジュール内のpublic述語としてサポートされることになっており、KL1コンパイラへの登録によって通常の処理系内でサポートされた組込述語と同様に扱うことが可能となる。

[2]KL1レベルでのモジュール名からbuiltinモジュールと呼ばれることもある。

第 6 章

クラスタ内ゴールスケジューリング

執筆担当者: 今井

本章では、クラスタ内でのゴールのスケジューリング処理と自動負荷分散機能について説明する。

6.1 概要

クラスタ内で、あるPEが実行可能なゴールがなくなった時に、他のPEとの間でゴールの融通を行うことで、クラスタ内のPEの稼働率を上げることは重要である。VPIMのクラスタ内では、このようなゴールの融通は、(ユーザがプラグマで指定するのではなく)処理系が自動的に行う。以下、この融通の方式を含めてクラスタ内でのゴールのスケジューリングに関して記述する。

6.2 実行可能なゴールの格納方法 (レディゴールスタック)

クラスタ内の各PEは、実行可能なゴール[1]をそれぞれの**プライオリティ** (実行優先度)に応じて図6-1のような(リンクによるLIFOの) **レディゴールスタック** に格納している。

プライオリティには、**論理プライオリティ** と**物理プライオリティ** の二種類がある。物理プライオリティはVPIM内で実行時に実際に区別される4096 (2^{12}) 段階[2]。のプライオリティであり、一方論理プライオリティはKL1ユーザに見えるプライオリティであり、各ゴール毎に 2^{31} の分解能[3]、を持つ。

ゴールがレディゴールスタックに入れられる際には、論理プライオリティは物理プライオリティに変換されるが、論理プライオリティの小さいゴールは(丸め誤差により)同一物理プライオリティに変換されるため、論理プライオリティの大小関係が、実際のスケジューリングに反映されない場合もありうる。

レディゴールスタックのエントリは固定領域に確保され、4ワードで構成される1エントリが、物理プライオリティ種類数連続しているような構成を取る。

---

[1] ここで言う「実行可能」とは、「明らかに中断する」の否定の意である。取り出して実行してみたら「中断してしまう」ことも充分あり得る。明らかに「中断状態」であるゴールは、変数セルにフックしている状態で置かれ、このレディゴールのリンクには入っていない。

[2] Multi-PSIと同様。

[3] Multi-PSIでは 2^{32} であるが、符号なし演算のコストが高いマシンへの移植を考慮し、2^{31} とした。

エントリ内の4ワードはそれぞれ、

- (有効な)下のエントリへのポインタ
- 物理プライオリティの値
- (実行可能な)ゴールレコードへのポインタ
- (有効な)上のエントリへのポインタ

を意味している。有効なエントリを双方向リンクで結ぶ理由は、プライオリティが変化した時に実行すべきゴールの検索を高速化するためである。

6.3 レディゴールスタックの操作

6.3.1 レディゴールスタックを操作するタイミング

レディゴールスタックにゴールを入れるのは、

- enqueue系の命令により、ゴールを生成した時
- サスペンドしていたゴールを再スケジュールする時
  - サスペンド要因変数を具体化した時
  - %start メッセージや、%supply_resource メッセージにより単親が実行可能になった時
- 他のクラスタから %throw_goal メッセージが届いた時
- クラスタ内自動負荷分散でゴールを受けとった時
- 処理系の都合で、一度処理を KL1 ゴールの形 (D コードゴール) に割出して実行する時
- execute系命令の実行中にスリットチェックイベントを検出し、スリットチェック処理前に、次のリダクションで実行しようと思っていたゴールを一時退避する場合。

がある。

レディゴールスタックからゴールを取り出すのは、

- proceed命令により、今まで実行していたゴールが子ゴールを発生せずに終了した時
- execute系命令の実行中にスリットチェックイベントを検出し、スリットチェック処理前にゴールをレディゴールスタックに一時退避していた場合の、スリットチェック処理終了後
- クラスタ内負荷分散要求に応じて、ゴールを投げる時

がある。

6.3.2 レディゴールスタックの詳細操作方法

この操作のために、VPIM に用意しているパーマネントレジスタは以下の通りである。

`D_CurrentGoalStackPtr`

レディゴールスタックエントリのうち、現在実行中のゴールと同じ物理プライオリティを持ったエ

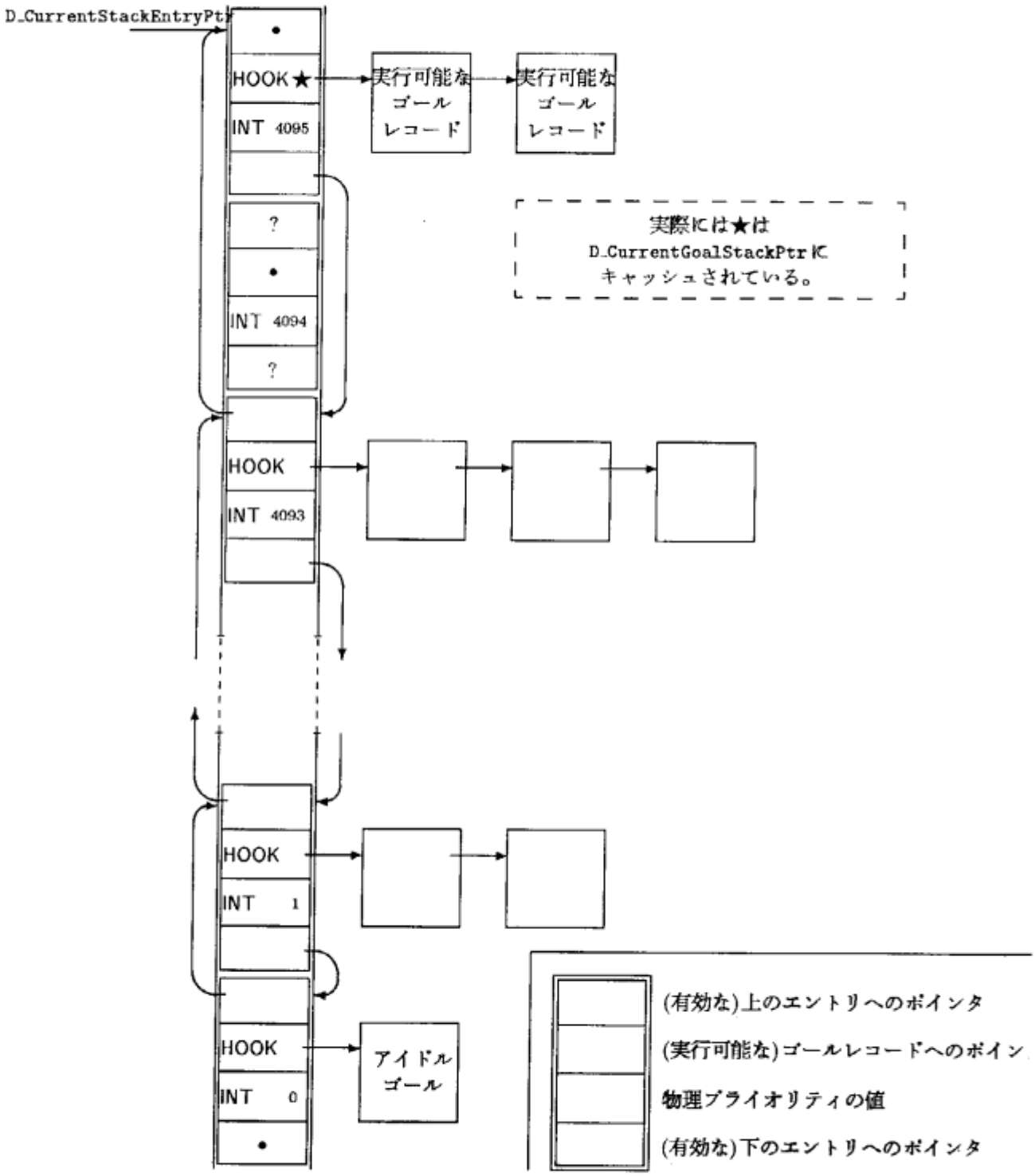


図 6-1: レディゴールスタックの構成

ントリに対する操作を高速化するために、ゴールリンクの先頭を保持したレジスタ。前述のゴールを入れる操作は、物理プライオリティが等しい場合、ただ単にこのレジスタに対する書き換え操作となる。また、このレジスタの値が EOL でない限り、取り出し操作も、このレジスタの書き換え操作となる。

D_CurrentStackEntryPtr

現在実行中のゴールの物理プライオリティのスタックへのポインタを保持している。

D_MaxPriority

現在レディゴールスタックにある、最高の物理プライオリティの値を保持している。値部は物理プライオリティを保持し、タイプ部に、このレジスタの値部が valid か invalid かの意味を持たせている。現在実行中のゴールと物理プライオリティが等しい時がinvalid な状態で、valid な状態となるのは、現在実行中のゴールよりも高い物理プライオリティのゴールをレディゴールスタックに入れてから、D_CurrentStackEntryPtr を更新するまでの間である(必ず「ハイプライオリティゴールの発生」 のソフトウェア割り込みが上がっている。(8 章参照)。

ゴールをレディゴールスタックに入れる操作は、現在実行中のゴールの物理プライオリティ (P_{cur}) と、入れようとしているゴールの物理プライオリティ (P_{push}) を比較し、あるいは、更に必要ならば前述のD_MaxPriority (P_{max}) の値に応じて、次のような操作を行なう。

1. P_{max} の状態に依存しない操作を行なう場合

   (1.a) $P_{push} = P_{cur}$ の場合

   → 単純に、D_CurrentGoalStackPtr にプッシュする。

   (1.b) P_{push} のエントリに既にゴールが繋がっていた場合

   → 単純に、そのエントリにプッシュする。

   (1.c) $P_{push} < P_{cur}$ の場合

   → P_{push} に相当するスタックのエントリは未使用であれば、プライオリティの高い方向へゴールリンクが空でないエントリを探し、エンキューしようとしているエントリの上下のエントリを得て相互に結ぶ。

   → そのエントリにプッシュする。

2. P_{max} がinvalid の場合

   (2.a) $P_{push} > P_{cur}$ の場合

   → P_{push} に相当するスタックのエントリは未使用のはず であるから、それと、P_{cur} のエントリの間を結ぶ。

   → $P_{max} := P_{push}$ として (valid にして) スリットチェック要因を上げる。

   → そのエントリにプッシュする。

3. P_{max} が valid の場合

   (3.a) $P_{push} > P_{max} > P_{cur}$ の場合

→ P_{push} に相当するスタックのエントリは未使用のはずであるから、P_{max} のエントリの間を結び、$P_{max} := P_{push}$ とする。

→ そのエントリにプッシュする。

(3.b) $P_{max} > P_{push} > P_{cur}$

→ P_{push} に相当するスタックのエントリは未使用であれば、プライオリティの低い方向へゴールリンクが空でないエントリを探し、P_{push} のエントリの上下のエントリを得て相互に結ぶ。

(3.c) $P_{max} = P_{push} > P_{cur}$

→ 既に P_{max} をセットしたゴールが繋がっているはずなので、このような状態はあり得ない。

なお、(2.a) の D_CurrentGoalStackPtr, D_CurrentStackEntryPtr の更新および D_MaxPriority の無効化は、スリットチェックで行なう。

6.4 クラスタ内負荷分散

6.4.1 実行すべきゴールがなくなった時のゴール要求処理

各PE の物理プライオリテイ0 (最小値) には、**アイドルゴール** という特別なゴールが必ず繋がっている。これは、実行可能なゴールがなくなった時の処理を簡略化するためである。アイドルゴールは、(コードの定義については、5.2節を参照のこと) 特殊な組込述語kblt_idleを実行するゴールである。

この組込述語kblt_idleは、クラスタ内の他のPE に対し、自分がアイドルになった事を通知し、他のPE からのゴール送信を含む、スリットチェックイベント (8章を参照) を待つ処理をする組込述語である。

すなわち、この組込述語の実行によって、

1. 固定領域に用意したゴール要求の有無を示したビットマップの自分のビットを立てると同時に、クラスタ内の他のPE のスリットチェックレジスタの「ゴール要求ビット」を立てる (図6 2)。
2. 他のPE によりこのイベントが処理され、自分のスリットチェックレジスタの「ゴール受信ビット」が立てられるまで、このビットをポーリングして待つ[4]。

の処理を行う。

6.4.2 ゴール要求を受信したPE の送信処理

スリットチェックのタイミングで、「ゴール要求ビット」が立っていることに気付いたPE は、ゴールを分配するために次のような処理を行う。

1. システム固定領域中のビットマップのパターンをレジスタに読み込むと同時に、他のPE の「ゴール要求ビット」を下ろす (図6-3)。

[4]実際には、何らかのイベント (クラスタ間メッセージの到着など) があるまで待つことになる。

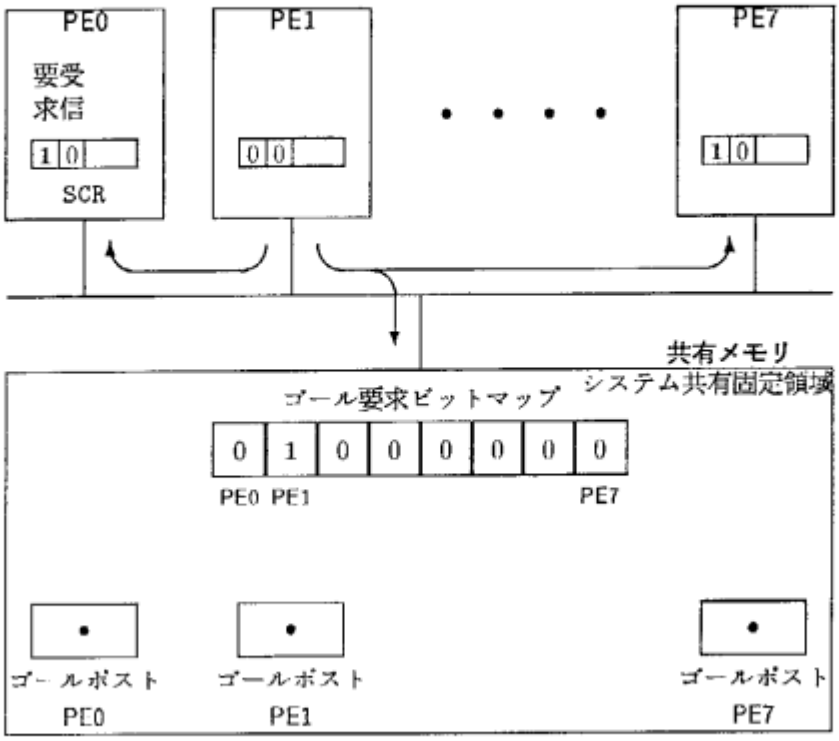


図 6-2: PE1 がアイドルになり、他の PE に通知

2. 自分のレディゴールスタックの中からプライオリティの一番高いゴールを一つ取りだし、ゴール要求を出しているPE のゴールポストに繋ぐと同時に、そのPE のスリットチェックレジスタの「ゴール受信ビット」を立てる(図6-4)。

なお、スリットチェック時にゴール要求をしていたPE は1台とは限らないので、ビットマップ中のビットの立っていたPE 全てに対して、2の処理を行う必要がある。ここで、2の処理を続けているうちに、送信できるゴールがなくなってしまう場合がありうる。この場合は、未送信のPE の代わりにゴール要求を出すようにする。

6.4.3 ゴールを送信されたPE の処理

他のPE によりゴールがポストに繋がれ、「ゴール受信ビット」が立てられると、これを含むイベントを待っていたPE はゴールを送信されたことに気付く。

そこで、ゴールポストから送信されたゴールを取りだして、その物理プライオリティに応じたレディゴールスタックの位置に、ゴールを繋いでゆく。なお、ゴールは複数繋がっている可能性があるので、すべてのゴールをプライオリティに応じてレディゴールスタックに繋ぐ処理を行う。

6.4.4 高プライオリティゴールの要求処理

クラスタ内の各PE は、個別にレディゴールスタックを持っているので、あるPE は高いプライオリティのゴールを潤沢に持っているにもかかわらず、他のPE は低いプライオリティのゴールを実行していると言う状況が起こり得る。

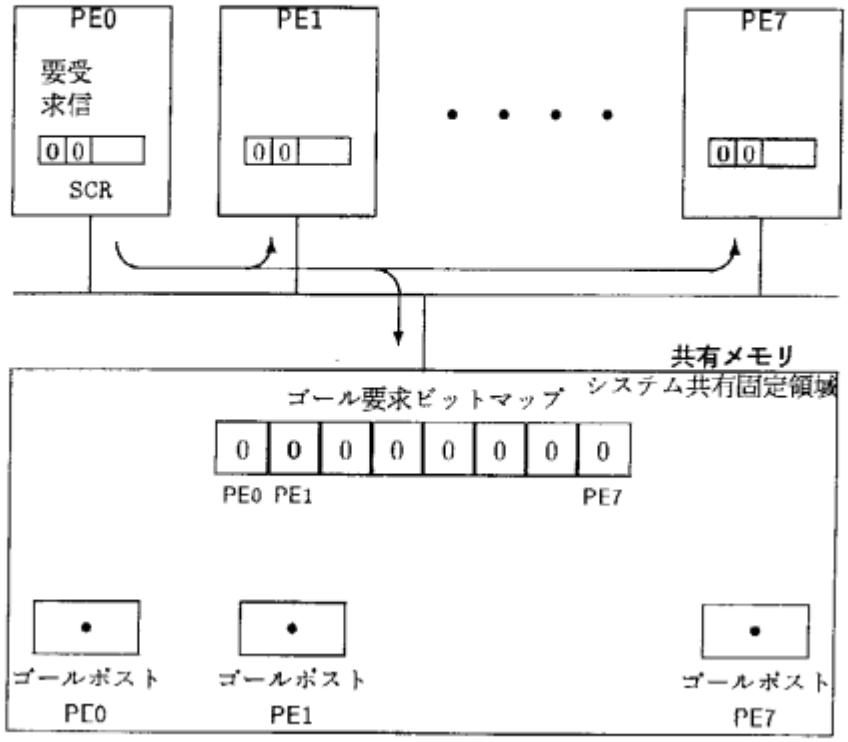


図 6-3: PE0 がスリットチェックでゴール要求に気付く

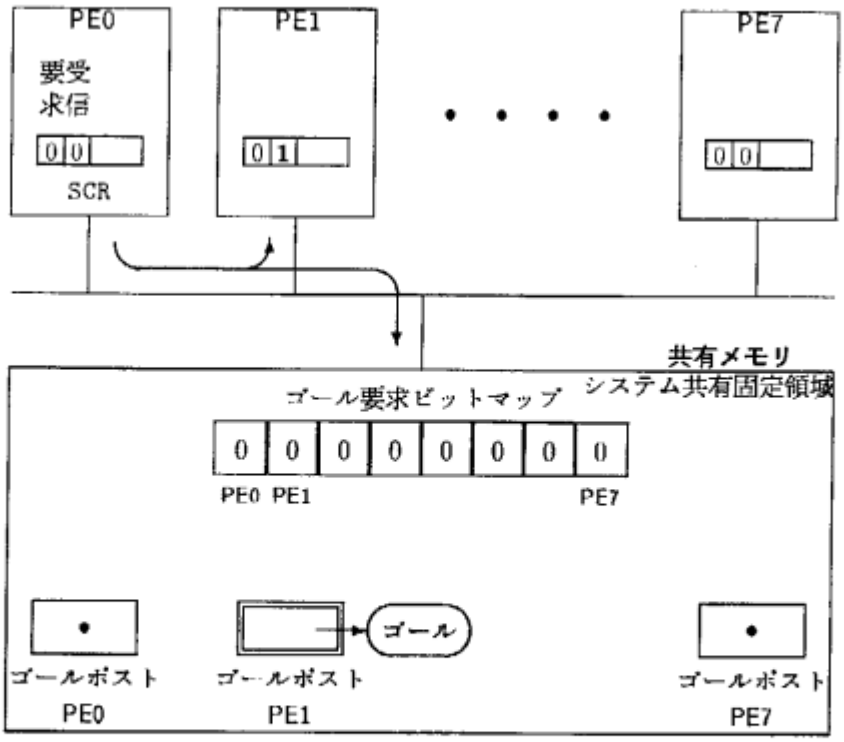


図 6-4: PE0 が PE1 にゴールを送出

そこで、低いコストでできるだけ高いプライオリティのゴールを実行できるように、「**プライオリティ制御の圧力モデル**」を導入して、高いプライオリティのゴールをPE 間で融通する機構を組み込む。圧力モデルでは、次のようなパラメータによってプライオリティ制御を行う。

P 各PE のゴール要求、及び *CLMax* の値の変更要求の強さを表す変数。(Pressure)

UpLimit 圧力がこの定数を上回ると、PE はゴール要求を出し、圧力をリセットする。高プライオリティゴールの要求を間引くために導入した値である。

LowLimit 圧力がこの定数を下回ると、PE は *CLMax* の値を現在自分が実行中のプライオリティ (*PEMax*) に (下方) 修正し、圧力をリセットする。*CLMax* の更新を間引くために導入した値である。

Δt 圧力を更新する量子化時間。タイマの割り込みによる。

各PE は、Δtおきに、圧力を以下のように更新する。

$$P(new) = P(old) + (CLMax - PEMax)$$

例えば、

PE0 が現在実行中のプライオリティ	=	300
PE1 が現在実行中のプライオリティ	=	100
CLMax	=	200

という状況において、

PE0 は Δt おきに、圧力 P が (200 − 300) 加算され (即ち 100 減り)、閾値 *LowLimit* を下回ると、*CLMax* を自分の *PEMax*(即ち300) に上方修正する。

PE1 は、Δtおきに、圧力 P が (200 − 100) 加算され(即ち100増え)、閾値 *UpLimit* を上回ると、PE はゴール要求を出し、圧力をリセットする。

となる。なお、高プライオリティのゴールの要求処理は、実行すべきゴールがなくなった時の処理と同じである。

この処理方式の知られている問題点

1. 高プライオリティゴールの要求を出したにもかかわらず、もらえたゴールがのプライオリティが今まで実行していたものよりも低いことがあり得る。この場合、再び圧力が高まって要求を出すまでは、低いプライオリティのゴールを実行することになる。
2. CLMax を下げるタイミングが難しい。現在は、要求に答えてゴールを送信する時に、そのゴールのプライオリティを CLMax に設定しているが、不当に下げる危険性が高い。

6.4.5 クラスタ内負荷分散に伴うチャイルドカウントのメンテナンス

本節では、クラスタ内負荷分散時に、里親の終了検出のために導入されたゴール数のカウンタである**チャイルドカウント**をどのようにメンテナンスするかについて説明する。また、里親の終結について

は、10.4節にて詳細に説明する。

クラスタ内でその里親に属するゴール(および子荘園)の総数は、里親レコード内に設けたチャイルドカウントにてメンテナンスされる。すなわち、この値が 0 になった時点で里親が終結できることを判定する。しかし、ゴールの生成・消滅の度にクラスタ内の全PEで共有されるこのカウンタを更新するとオーバーヘッドが大きいため、各PE毎にこの値をキャッシュし、普段はこのカウンタをメンテナンスする。この(キャッシュされた)カウンタを**フォークカウンタ**、フォークカウンタに格納される数値を**フォークカウント** と呼ぶ。

フォークカウンタの導入により、クラスタ内のある里親に所属するゴール(および子荘園)の総数は、次のような式で求めることができる。

$$\text{ゴール総数} = \Sigma\,(\text{PE 毎のフォークカウント}) + (\text{里親レコードのチャイルドカウント})$$

フォークカウンタは、実行しようとしたゴールの所属する里親がこれまで実行していたゴールの里親と異なる場合に(アイドルになった場合も含めて)、里親レコードに書き戻される(足し込まれる)。すなわちあるPEは、一時にはひとつの里親のゴール数しかキャッシュしない。

自動負荷分散においては、このキャッシュされた数値を注意深くメンテナンスする必要がある。たとえば、この書き戻しが里親が変わらない限り行なわれないとすると、次のような問題が生じる(図6-5〜6-8)。この例では、PE1 がもらったゴールを終了させた時点で、里親レコードのチャイルドカウントの値が 0 になってしまうため、この里親に属するゴールが PE0 で実行されているにもかかわらず、里親が終了したと判定されてしまう。

これは、上記の式から分かるように、里親レコードのチャイルドカウント値からだけではその里親に属する本当のゴール総数は分からないにもかかわらず、この値を元に一つのPE(この場合はPE1)が里親の終了判定を行おうとしたことが原因である。

しかし逆に、一つのPEがフォークカウントを里親レコードのチャイルドカウントに書き戻すたびにクラスタ内全PEが同期してフォークカウントを里親レコードに書き戻すとオーバヘッドが大きくなってしまう。

このため、自動負荷分散で他のPEにゴールを投げる際、

$$((\text{フォークカウンタ}) > 0) \wedge (\text{投げるゴールが現在実行中の里親に所属})$$

の条件を満たす時には、一旦フォークカウントを里親レコード中のチャイルドカウントに足し込んで反映させてから、そのPEのフォークカウンタを0にクリアした後にゴールを投げることとする。

こうすることによって、PEから里親レコードにフォークカウントを書き戻した時点では里親の正確なゴール総数は分からないが、少なくとも里親のチャイルドカウントが 0 である場合にはその里親にゴールがないことが保証できるため、里親の終了を判定することが出来る。

また、このキャッシュ方式を正常に動作させるため、次のような処理も併用される。

- ゴールがサスペンド後、アクティブユニフィケーションでリジュームされるときは、そのゴールをサスペンドさせた PE に投げて、サスペンドさせたPEで実行させる。

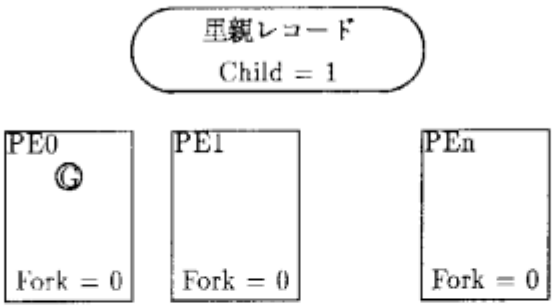


図 6-5: 初期状態

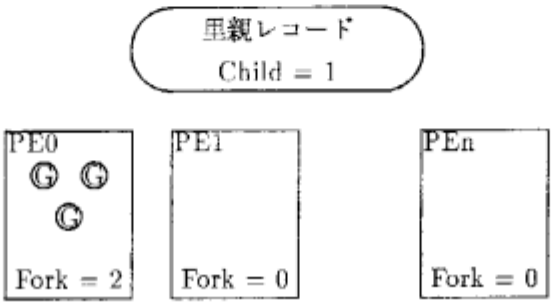


図 6-6: PE0 がゴールを生成

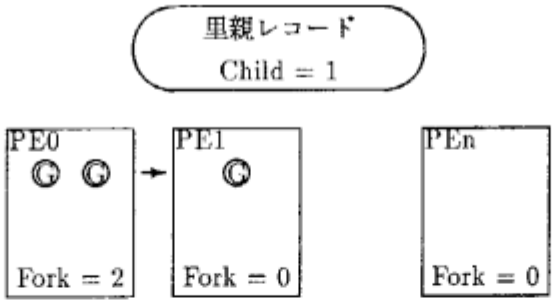


図 6-7: PE0 が PE1 にゴールを分配

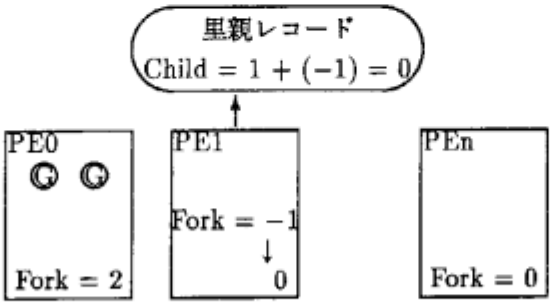


図 6-8: PE1 がゴールの実行を終了し、書き戻した

第 7 章

クラスタ内ユニフィケーション

執筆担当者: 今井

本章では、論理型言語の基本操作であるユニフィケーションの処理方式とストリーム併合の最適化であるマージ組込述語の処理方式について解説する。

7.1 概要

ユニフィケーションは、論理型言語処理系での基本操作である。

KL1 においては、ガード部におけるユニフィケーションと、ボディ部におけるユニフィケーションでそれぞれ処理が異なる。また、ユニフィケーションはゴール間の実行の同期/通信のプリミティブにもなる操作である。

本章では、特にクラスタ内のデータ構造同士のユニフィケーションに話しを絞り、また特にクラスタ内が共有メモリで結合されることで必要となる排他制御を中心に説明する。また、ユニフィケーションによる、ゴールの中断(suspension) / 再開 (resumption) の仕組みについても述べる。

7.2 パッシブユニフィケーション

7.2.1 概要

パッシブユニフィケーション とは、KL1のガード部実行中に行なわれるユニフィケーションである。KL1の文法上、ガード部においては

未定義変数に対し、それを具体化することはできない

という制限があり、これを行なおうとした場合は、ユニフィケーションの中断となる[1]。つまり、パッシブユニフィケーションでは、ユニフィケーション対象の二引数のパターンマッチング(具体化された値同士が同じであるかどうかのチェック)のみを行なう。すなわち、この処理においては、データを読み出してチェックする処理だけであり、値を書き換えるという処理がないため、排他制御は不要である。ただし、後述のサスペンド操作においては、論理的には未定義変数であるが、その具体化を待つゴールがあ

[1]他のクラスタに実体があるデータも未定義扱いとなる。

ることを示すための書き換え処理 (Bind-Hook) があるため、排他制御が必要となる。

7.2.2 アトミックデータのパッシブユニフィケーション

一方がアトミックデータであることがコンパイル時に明らかであったパッシブユニフィケーションは、もう一方のテスト対象を、

1. デレファレンスと具体化チェック (load_wait, read_wait)
2. タイプチェック (is_XXX)
3. 値チェック (test_XXX, hash_on_XXX, jump_on_XXX)

の順で行なう。これらは、複合命令で実行されることもあるし、同様の処理が他のクローズにある場合、重複を避けるためのインデキシング命令により実行されることもある。これらのいずれかに失敗/中断した場合、別のクローズの試行、あるいはサスペンド処理に飛ぶことになる。

7.2.3 構造体データのパッシブユニフィケーション

一方が構造体データであることがコンパイル時に明らかであったパッシブユニフィケーションは、もう一方のテスト対象を、

1. デレファレンスと具体化チェック (load_wait, read_wait)
2. タイプチェック (is_list, is_vector など)
3. (もしあれば) サイズテスト (test_arity)

で行なう。構造体の要素もコンパイル時に決まっている場合は、その要素の種類により、アトミックあるいは構造体の処理をそれぞれ行なう命令が発行される。これらのいずれかに失敗した場合、別のクローズの試行、あるいはサスペンド処理に飛ぶことになる。

7.2.4 変数同士のパッシブユニフィケーション

ユニフィケーション対象の二引数がいずれもコンパイル時にタイプが定まらなかったユニフィケーションでは、

1. 第一引数のデレファレンスと具体化チェック (load_wait)
2. 第二引数のデレファレンスと具体化チェック (load_wait)
3. 引数同士が等しいかどうかをチェック (equal)

の順で処理を行なう。これらのいずれかに失敗した場合、別のクローズの試行、あるいはサスペンド処理に飛ぶことになる。

equal命令において、二つの引数がともに構造体であった場合には、システム固定領域 (プロセッサ局所固定領域) に用意したユニファイスタックを利用して、それらの構造同士の比較を再帰的に行なう。スタックがオーバーフローした時は、equal命令の持つ、サスペンドラベルに飛ぶことになる。

なお、パッシブユニフィケーションでは、二つの引数が同一であっても、それらが具体化されない限り成功しないことにしている。これは、クラスタ間データが絡んでくると、変数同士の同一性を調べることが非常に困難となる場合があるためである。このため、

$$p(X,X,1) : -true \mid true.$$

のような述語を、

$$p(X,X,2).$$

で呼び出した場合は、失敗しないで、中断することになる。

7.2.5 コンパイル方式

パッシブユニフィケーションでは、

$$foo(X,Y) : - \underline{X = Y} \mid ...$$

のように、ユニフィケーション・オペレータ "=" が陽に記述されている場合と、

$$foo(\underline{X,X}) : -true \mid ...$$

のように、ヘッドの引数パターンによって引き起こされる場合があるが、いずれもコンパイラによって、同じパッシブユニフィケーションのための命令(klb_equal) が発行される。

また、

$$foo(X,Y,\underline{Z}) : -add(X,Y,\underline{Z}) \mid ...$$

のように、ガード部における組込述語と、ヘッドの引数が同一の場合、このユニフィケーションは、組込述語の中で行なうのではなく、一度仮の引数に出力された演算結果と、ヘッドの引数とのユニフィケーションを行なう。すなわち、

$$foo(X,Y,Z) : -add(X,Y,\underline{Temp}),\underline{Temp = Z} \mid ...$$

と解釈できる命令をコンパイラが発行する。これは、個々の組込述語の処理を簡略化するためである。

7.3 サスペンド処理

7.3.1 概要

どのクローズもコミットできなかった場合で、かつ、あるユニフィケーションの中断原因となった未定義変数がある場合、サスペンド処理を行なう。これは、サスペンドの原因となった変数が具体化される時に、サスペンドしたゴールをリジュームしてもらうための処理であり、具体的には、その未定義変数からサスペンドしたゴールレコードに対してパスを張る処理を行なう(変数にフックすると呼ぶ)。

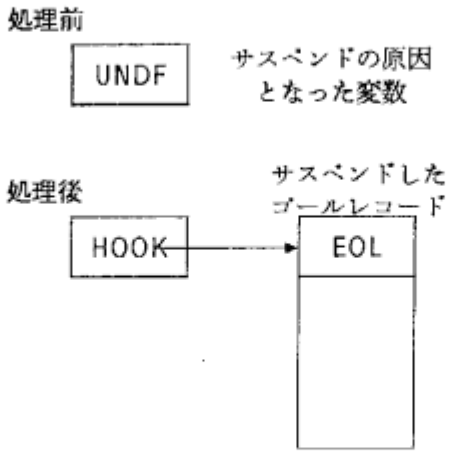


図 7–1: 単一サスペンドの構造

なお、ユニフィケーションの中断の原因となる未定義変数がなかった場合は、(すべての節に失敗したユニフィケーションがあり、コミットできなかったことになるので) Reduction Failureの例外処理となる。

サスペンド処理は、その原因となる変数が特定できる場合は、suspend_single 命令で、また、特定できない場合は、suspend 命令あるいは otherwise 命令によって行なわれる。

7.3.2 単一サスペンド処理

単一サスペンド処理は、サスペンド要因変数がただ一つであった場合に行なう処理であり、具体的には、図7–1のような構造を作る。また、同じ変数を待つゴールが既にあった場合は、そのリンクにプッシュする(図7–2)。

単一サスペンド処理においては、メモリ中の未定義変数を書き換える処理になるため、次のような排他制御を行う。

Step.1 フック原因の変数が UNDFの場合、ゴールレコード中の次リンクスロットを EOLにする。ポインタ (HOOK, MHOOK など) の場合、ゴールレコード中の次リンクスロットを、そのポインタに設定する。

Step.2 Compare & Swap (ptr = 未定義変数へのポインタ, old = 前に読んだ未定義変数の内容, new = ゴールレコードへのポインタ)

Step.3 成功すれば終わり。失敗したら、ゴールレコード中の次リンクスロットを EOL にして、Enqueue する。

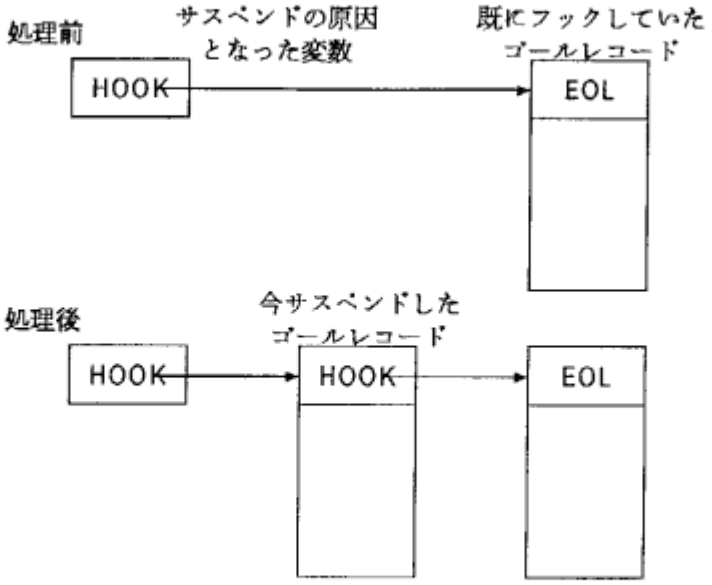


図 7-2: 単一サスペンドの構造 (その 2)

7.3.3 多重サスペンド処理

多重サスペンド処理 は、サスペンド要因が複数ある場合に行う処理であり、次のようなゴールが全て未定義で呼び出された場合などに行なわれる。

$$p(a,B,C,D) : -true \mid ....$$
$$p(1,B,C,D) : -true \mid ....$$
$$p(A,b,C,D) : -true \mid ....$$
$$p(A,B,c,D) : -true \mid ....$$
$$p(A,B,C,d) : -true \mid ....$$

具体的には、図7-3のような構造を作る。ここで、2種類の2ワードのレコードが割り付けられるが、サスペンドリンクを保持している方を**サスペンドレコード**、多重サスペンドしたゴールレコードへのポインタを保持している方を**サスペンドフラグ**と呼ぶ。

サスペンド要因となった複数の未定義変数は、load_wait, read_wait 命令でサスペンドスタックに積まれている。サスペンドスタックは、システム固定領域 (プロセッサ局所固定領域) に用意したスタックで、ユニフィケーション中断の要因となった変数へのポインタを積んでおくために用いられる。

suspend 命令もしくは otherwise 命令において、サスペンドスタックをチェックし、(重複を除いた) 変数が 2つ以上ある時、多重サスペンド構造を作る。

7.3.4 サスペンドに関する注意事項

- EHOOK, EMHOK は、輸出された可能性のある HOOK 系のタイプであるが、これらは未定義変

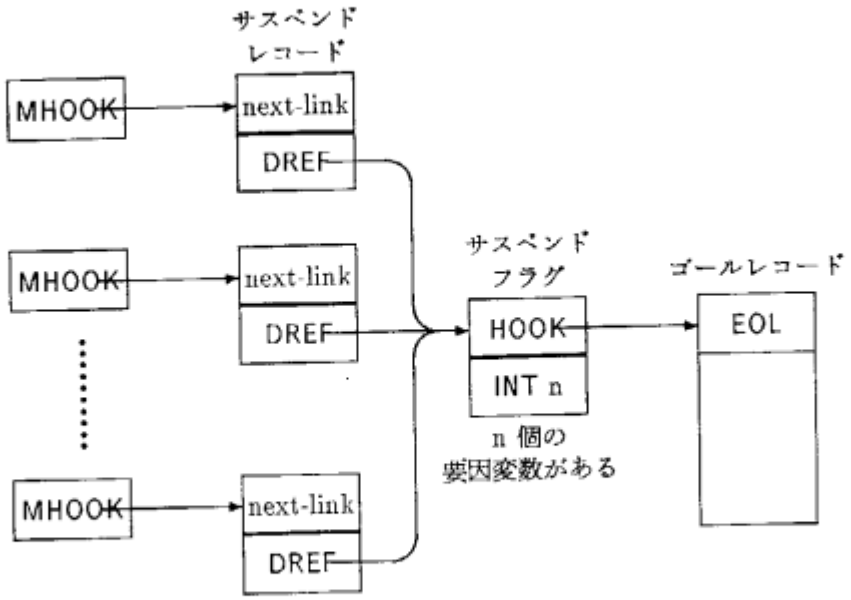


図 7-3: 多重サスペンドの構造

数としてのみ出現し、フックリンクの中には現れない[2]。

- 多重サスペンドの構造で、サスペンドフラグから指されるゴールレコードの次リンクは、必ずEOLである。
- MRB を利用した永久中断ゴールの検出のため、HOOK 系のタイプに付随した MRB は次のような意味を持つ。
  - ○ 白いパスでフックした
  - ● 黒いパスでフックした

7.4 アクティブユニフィケーション

7.4.1 概要

アクティブユニフィケーション とは、KL1のボディ部実行中に行なわれるユニフィケーションである。ボディ部においては、未定義変数に対し、それを具体化することもできる。未定義変数の具体化処理は、他のPE が書き換えるかもしれない値を書き換えるという処理であるため、排他制御が必要となる。この排他アクセスによるユニフィケーションはVPIMでは全て、Compare & Swap によって実現されている。

また、具体化を待つゴールがある未定義変数に値を書き込んだ場合、そのゴールを再開させるための処理(リジューム処理)も必要となる。

[2]1ワードセルの中にのみ出現する。

7.4.2 アクティブユニフィケーションの命令

(1) 一方がアトミックデータのアクティブユニフィケーション

一方がアトミックデータであることがコンパイル時に明らかであったアクティブユニフィケーションは、unify_atomあるいは、unify_integer命令によって行なわれる。

(2) 一方が構造体データのアクティブユニフィケーション

一方が構造体データであることがコンパイル時に明らかであったアクティブユニフィケーション、すなわち、

$$p(X,Y) :- true \mid \underline{X = [a|Y]}.$$

のようなユニフィケーションは、 Read Mode と Write Modeの2通りがある。

モード	X の状態	処理内容
Write	未定義	X を $[a\|Y]$ で具体化する
Read	具体化済み	X と $[a\|Y]$ の比較を行う

そこで、ユニフィケーションの方法としては、

- **Write Mode を仮定する方式**

  ユニフィケーションに先だって、CAR がa, CDR が Yのリストセルを割り付け、 これをXとユニファイする。

- **Read Mode を仮定する方式**

  X と リストの枠をまずユニファイし、その後に、 CAR をa、 CDRをYとユニファイする。

が考えられる。

前者はX が既に具体化済みの場合に不要なリストセルを割り付けることになるが、 後者は、Write Modeの場合に Xを具体化する PE が、CAR, CDRをユニファイし終わるまでリストセルをロックし続ける必要があり、Compare & Swap を用いることができない。

殆どのKL1 プログラムで、アクティブユニフィケーションは Write Modeであるため、ロック期間が短く単純な前者を選択する。

前述の例では、具体的には、

リストの作成	1	alloc_list	リスト割り付け
	2	put_atom	レジスタ上にアトムを用意
	3	write	CAR の書き込み
	4	write	CDR の書き込み
ユニファイ	5	unify_bound_value	アクティブユニファイ

のような命令シーケンスによって実現される。

また、コンパイル時点で、すべて要素が具体化済みの構造体の場合(ネストしている場合も含む)、具体化済み構造体をコンパイル時に作成し、この構造体とのユニファイを行なう[3]。このようなコンパイル時に作成された構造体を構造体定数とよぶ。ただし、この構造体は、(何度も繰り返し呼ばれる可能性があるので) MRB が黒として用意される。

(3) それ以外のアクティブユニフィケーション

unify 命令で行なわれる。

7.4.3 データ種別によるユニフィケーション操作

(1) アトミック・データ同士のユニフィケーション

2つの引数のタイプと値を比較しいずれも等しいならば何もしない(ユニファイ成功)。いずれかが等しくないならば Unification Failure 例外。

(2) 構造体同士のユニフィケーション

2つの引数のタイプと値(構造体本体を指すポインタ)を比較し等しい時、即ち同じ構造体を指しているならば何もしない(ユニファイ成功)。タイプが等しく値が異なる時は構造体本体の各要素毎のユニフィケーションを行う。このような処理では、スタックを用いることが一般的であるが、処理が複雑になる。そこで、リストやベクタ同士のユニフィケーションが次のような KL1 ゴールで表現できることを利用して、処理を単純化する。すなわち、

$$
\begin{aligned}
&list\_unifier([X1|X2],[Y1|Y2]) : -true \mid \\
&\quad X1 = Y1, X2 = Y2. \\
&vect\_unifier(0, V1, V2) : -true \mid true. \\
&vect\_unifier(N, V1, V2) : -N > 0 \mid \\
&\quad N1 := N - 1, \\
&\quad set\_vector\_element(V1, N1, Elm1, \_, NV1), \\
&\quad set\_vector\_element(V2, N1, Elm2, \_, NV2), \\
&\quad Elm1 = Elm2, \\
&\quad vect\_unifier(N1, NV1, NV2).
\end{aligned}
$$

というプログラム(Dコードと呼ぶ, 5.2節参照)を予め組み込んでおき、ユニフィケーション対象の2引数を載せて、通常のユーザゴールと同じレディゴールスタック に入れ、スケジュールする。

専用のスタックを用いる場合に比べ、このKL1ゴールを用いた処理方式では

○ ユーザゴール同様にスケジュールすれば良く、特別なスケジューリングを考える必要がない

[3] この構造体は、他のクラスタに実体が置かれていることがあるので、後述の unify命令が出る。

○ スタックの物理的サイズでユニファイできる構造の深さが決まってしまうことがない

○ クラスタ内の全 PE で同期をとる必要がある時に、ある PE が大きな構造体同士のユニフィケーションを行なっていても、一要素をユニファイする毎に、スリットチェックが入るので、レスポンスが良い。

× 処理が遅い。

× ユニフィケーションが失敗した場合、どの構造体の中の、どの要素が原因となったのかが分からない。

という特徴がある。ただしKL1では、そもそも構造体同士をアクティブユニファイすることが稀であるため、処理の遅さは特に問題にならないと考えられる。

ストリングは要素に未定義変数を含まず、整数データからなる構造体なのでDコードは使用せず要素毎の整数データの比較を行う。

ユニファイに失敗した場合は、 Unification Failure 例外となる。

(3) 未定義変数と具体値とのユニフィケーション

① 未定義変数がボイド変数の場合

ボイド変数とは、変数の割り当て時に参照数が1しかなかった変数である(タイプは、VOID)。ボイド変数とのユニフィケーションはパスMRBの値によって処理が異なる。白パス VOID (REF∘ → VOID)は単一参照の未定義変数(真のボイド変数)を表し、黒パスVOID (REF• → VOID)は黒パスUNDF (REF• → UNDF)同様に複数参照のある可能性がある。白パスVOID は具体化しても参照されないので、具体化すること無く回収して処理を終わる。白パスVOID は 黒パス UNDFと処理上等価である。また、一方の具体化した引数については参照数は減少するので、これを反映するためにcollect-valueと同じ操作を行う。

② 未定義変数が UNDF の場合

Compare & Swapによる未定義変数の具体化を行う。

Compare & Swapの比較操作が失敗した時、即ち未定義変数が他のPEによって書き換えられていた場合、通常はユニフィケーション処理の先頭に戻ってCompare & Swap をやり直す必要がある。この処理を単純化するため[4]に、アクティブユニフィケーションが、ゴールの実行と同様に実行順序の概念がないことを利用した次のような方法で実現する。

すなわち、

$$unify\_retry(X,Y) : -true \mid X = Y.$$

というDコードゴールの、XにCompare & Swap に失敗した未定義変数へのポインタ、YにCompare & Swap 成功時に書き換える値を載せて、このunify_retryゴールを通常のユーザゴールと同じようにスケジュールすることで、処理を単純化する。

[4] というか、「PSLで書いたVPIMの可読性を向上させるため」といった方が適切かも。

③ 未定義変数がHOOK系の場合

HOOK系変数はUNDFにゴールがフックしたものであるから具体化に関してはUNDFの場合と同様の処理を行う。Compare & Swapの比較操作が成功しHOOK変数が具体化した場合、そこにフックしていたゴールをリジュームする。

(4) 未定義変数同士のユニフィケーション

① (白パス)VOID変数とその他の未定義変数との場合

白パスで指されているVOIDセルともう一方の引数とのユニフィケーションではVOIDセルへの参照パスが唯一であるため、別の参照パスからVOIDセルがアクセスされることはない。従ってVOIDセルは無条件に回収してよい。また、一方の未定義変数については参照数は減少することになるので、これを反映するためにcollect-valueと同じ操作を行う。即ちもう一方の変数が(白パス)VOIDであれば回収、(白パス)UNDFまたはHOOKであればCompare & SwapによりVOIDに書き換える。これが、(REF$_o$ → HOOK$_o$)の場合、HOOKが指しているゴールの永久中断の検出になる。

② UNDF同士の場合

並列環境では、2つの未定義変数を同時にユニファイしようとするPEが他にもいる可能性があるため、無作為にリンクを張るとループができてしまうことが考えられる。そこで、未定義変数のアドレスを比較して、

『アドレスの大きい方のセルから小さい方のセルへ向けてリンクを張る』

という約束によってこれを解決する。

従ってUNDF同士のユニフィケーションはCompare & Swapによって値の大きい方のセルを小さい方のセルに向けたポインタ(REF)に書き換えることを行う。Compare & Swapの比較操作が失敗した時、即ちUNDFセルが他のPEによって書き換えられていた場合は、Dコードゴールを生成してユニファイ再実行を試みる。2つの変数セルを指すポインタ(REF)が等しい場合は何もしない(ユニファイ成功)。

③ UNDFとHOOK系変数の場合

Compare & SwapによってUNDFセルをHOOKセルに向けたポインタ(REF)に書き換える。Compare & Swapの比較操作が失敗した時、即ちUNDFセルが他のPEによって書き換えられていた場合はDコードゴールを生成してユニファイ再実行を試みる。

④ HOOK系変数同士の場合

(REF$_o$ → HOOK$_o$)同士の場合、永久中断状態の検出になる。

これ以外の場合は、UNDF同士の場合と同様に、リンクを張ることによってループができてしまうのを防ぐため上述の約束を守り、Compare & SwapによってHOOKセルを小さい方の

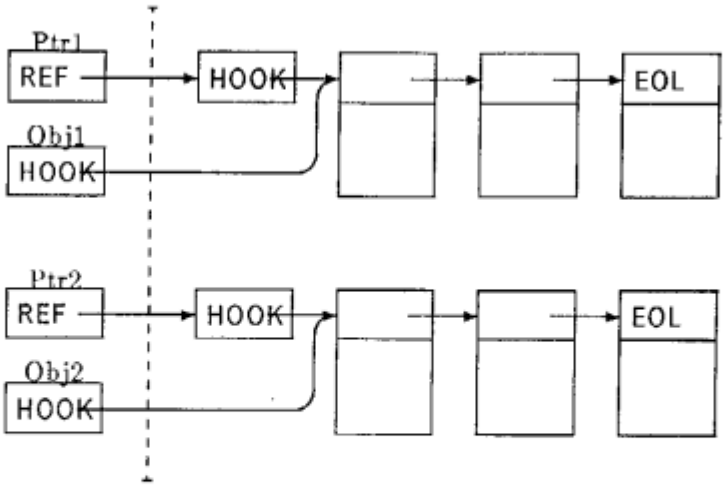


図 7-4: 初期状態

HOOK セルに向けたポインタ(REF)に書き換えることを行う。具体的な手順を以下に述べる。

1. 初期状態を(図7-4)に示す。アドレスは Ptr1 > Ptr2 とする。
2. Ptr1 の指すHOOK セルを Compare & Swap により Ptr2 の指すHOOK セルに向けたポインタ(REF)に書き換える。Compare & Swapの比較操作が失敗した時、即ちHOOK セルが他のPE によって書き換えられていた場合はDコードゴールを生成してユニファイ再実行を試みる。
3. 成功すると(図7-5)の様になる。ここでObj1から指されているリンクはこのPE だけがたどれることに注意する。
4. Obj1のリンクをたどり、最後のセル(ゴールレコード または サスペンドレコード)の『次リンク部』にObj2の値を書き込む(図7-6)。
5. Ptr2 の指すHOOK セルを Compare & Swap により Obj1 に書き換える。これに成功すれば(図7-7)の様になり、処理を終了する。このCompare & Swapに失敗した時、即ちHOOK セルが他のPE によって書き換えられていた場合は、4. で次リンク部に書き込んだ値をEOL に書き換え、Ptr2を基にデレフを行い、その結果具体化されていたならば Obj1 から指されているサスペンドゴールをレディゴールスタックにエンキュー(リジューム)する。(Obj2から指されているサスペンドゴールはHOOK セルを具体化したPE がリジュームする。) 具体化されていなかったならば、4. に戻る。この時、Obj2の値はデレフによって既に新しい値に書き換わっていることに注意する。

7.5 リジューム

7.5.1 概要

リジュームとは、アクティブユニフィケーションによってある変数を具体化した時、その具体化を待つゴールがいた場合に、そのゴールを再スケジュールすることを呼ぶ。

リジュームによる再スケジュールは、そのゴールを作成したPEに対してゴールを送信することで行な

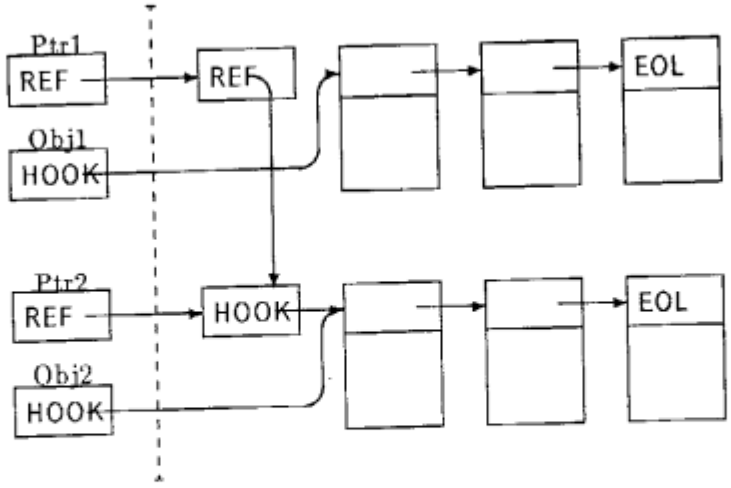


図 7-5: Ptr1 の先を Compare& Swap した状態

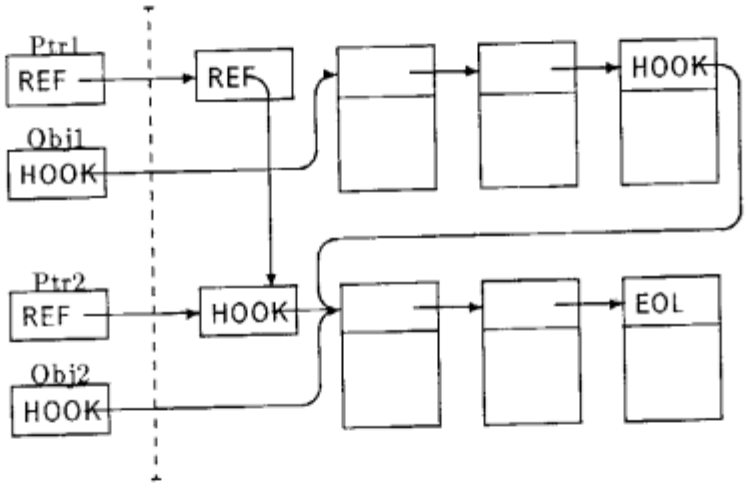


図 7-6: リンクの最後に Obj2 を書き込んだ状態

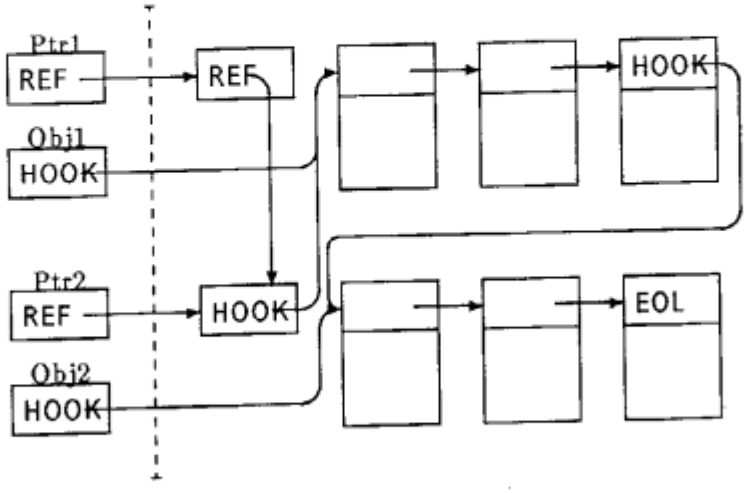


図 7-7: 最終状態

う。これは、

- ゴール引数、コードなどの実行環境が、そのゴールを作成したPEのキャッシュメモリに載っていることが多いであろうと期待されること
- チャイルドカウントをPE ごとにキャッシュしているため

などの理由によるが、今後評価/検討の余地がある。

前述のように、具体化を待つゴールは1つであるとは限らず、複数存在し得る。複数の場合は、ゴールレコード、サスペンドレコード、リプライフックレコード[5]の次リンクスロットを用いてリンクしている。この次リンクスロットがHOOK の場合、単一サスペンド構造のリジューム操作を行ない、MHOOK の場合、多重サスペンド構造のリジューム操作を行なう。

7.5.2 単一サスペンド構造のリジューム操作

ゴールレコードの、次リンクスロットを EOLにし、そのゴールを再スケジュールする。

7.5.3 多重サスペンド構造のリジューム操作

多重サスペンドしているゴールは、そのサスペンド要因となった変数のうち一つでも変数が具体化されると、他のサスペンド要因の変数が具体化されていなくてもリジュームされる。7.3.3節の例では、A が具体化されれば、B, C, D が未定義でも、第1,2クローズの実行が可能な場合があるからである。

多重サスペンド構造のリジュームは、図7-3のような構造を図7-8のような構造にする。

- サスペンドレコードを回収する。
- サスペンドフラグのゴールレコードへのポインタスロットを見る。これが、
  - HOOK の場合、Compare & Swap で、このスロットを EOL にし、このゴールを再スケジュールする。
  - EOL の場合、すでに他のサスペンド要因が具体化され

  て再スケジュールされたことを意味するので、リジューム操作は行なわない。
- サスペンドフラグの参照カウントを減じる。この時結果が 0 になれば、サスペンドフラグを回収する。

7.6 マージ組込述語

7.6.1 マージ組込述語とは

KL1 のような Commited-Choice 型言語では、プロセス間のストリーム通信によって並列処理が記述されるプログラミングスタイルが一般的であるが、VPIM においては、このストリームはリストセルを用いて実現している(Multi-PSIでも同様である)。

[5] クラスタ間処理の章を参照のこと。

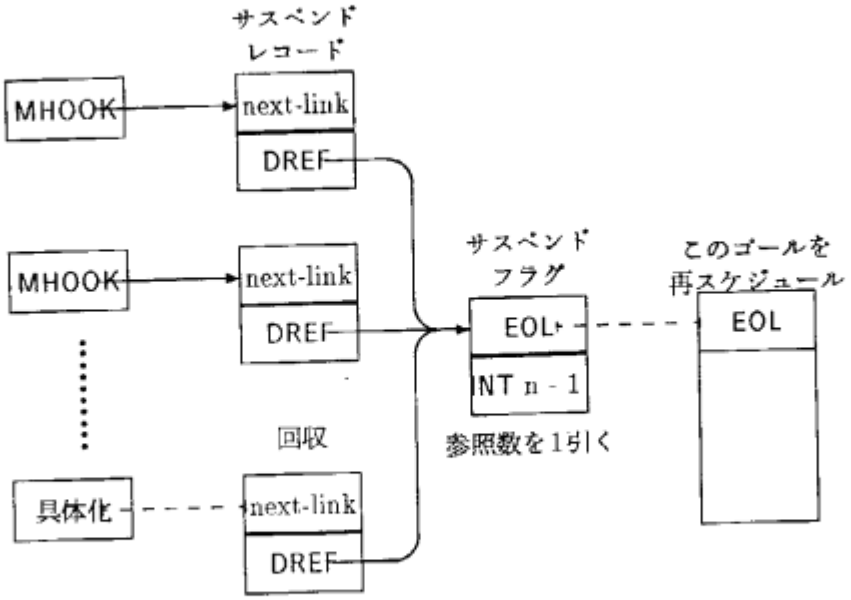


図 7-8: 多重サスペンドの構造のリジューム

このようなストリームに対して頻繁に行われるマージという操作(複数のストリームを1本のストリームにまとめる操作)を、KL1では次のように記述することができる。

$$merge([X|In1], In2, Out) : -true \mid$$
$$\quad Out = [X|NewOut], merge(In1, In2, NewOut).$$
$$merge(In1, [X|In2], Out) : -true \mid$$
$$\quad Out = [X|NewOut], merge(In1, In2, NewOut).$$
$$merge([], In2, Out) : -true \mid Out = In2.$$
$$merge(In1, [], Out) : -true \mid Out = In1.$$

但し、このように記述すると、2本の入力ストリームのいずれかへ1要素 が入力される毎に、ゴールのリジューム、サスペンドが繰り返されることになるため効率が悪い。また、

- 多数の入力ストリームをマージする場合には、入力数に応じて処理の手間が増大する。
- 入力ストリームの本数を増やすためには、別のマージャプロセスを間に一段挿入しなければならない。

などの問題がある。

そこで、 n本の入力ストリームに対して

入力数に依存せず、コンスタントオーダで

実現するストリームマージ方式を実装している。この方法で実現された仮想プロセスをマージャと呼

ぶ。

7.6.2 基本データ構造

以下のようなデータ構造を導入する。

- マージャレコード

  仮想的なマージャプロセスを表現するためのレコードで、

  マージャの入力ストリーム数 (GOAL•)

  マージャレコード自体の参照カウンタになっている。

  マージャの出力先変数へのポインタ (通常 REF)

  出力先の変数セルへのポインタ。

  マージャの所属する里親レコードへのポインタ (FPREC)

  不正な入力があった時の例外処理や、マージャがリダクションを行って良いかどうかの判定をするために用いる。

  モジュールへのポインタ (MOD)

  モジュール内オフセット (INT)

  マージャプロセスが永久中断した場合、あるいはマージャプロセスが異常入力により例外を起こした場合に荘園のレポートストリームを通して報告するために用いられる。merge/3がコンパイルされた結果のKL1-B 命令を指している。

  プロセッサ ID (INT)

  マージャプロセスを作ったPE 番号。

  プライオリティ (INT)

  マージャプロセスを作った時点のプライオリティ例外発生時の代替ゴールや、出力ユニフィケーションが稀な場合のリトライゴールのプライオリティに用いるだけで、通常実行時には守られない。

  属性 (INT)

  マージャプロセスの属性。例外発生時の代替ゴールのプライオリティに用いるだけで、通常実行時には守られない。

  の8ワードによって構成される。ただし、これ以降の説明では、特に必要のない限り、里親へのポインタ以下を省略する。

- MHV (Merger Hooked Variable)セル

  仮想的なマージャプロセスがフックしている変数(入力ストリーム)を表し、上記マージャレコードへのポインタを MGHOK タイプで格納する。

n 本の入力ストリームを持つマージャプロセスの表現を図7-9に示す。

MHV セルは、論理的には未定義変数であるが、仮想プロセスからの参照パスが隠されているので、MHV セルを白パスで指しているのは、たかだか1本である。

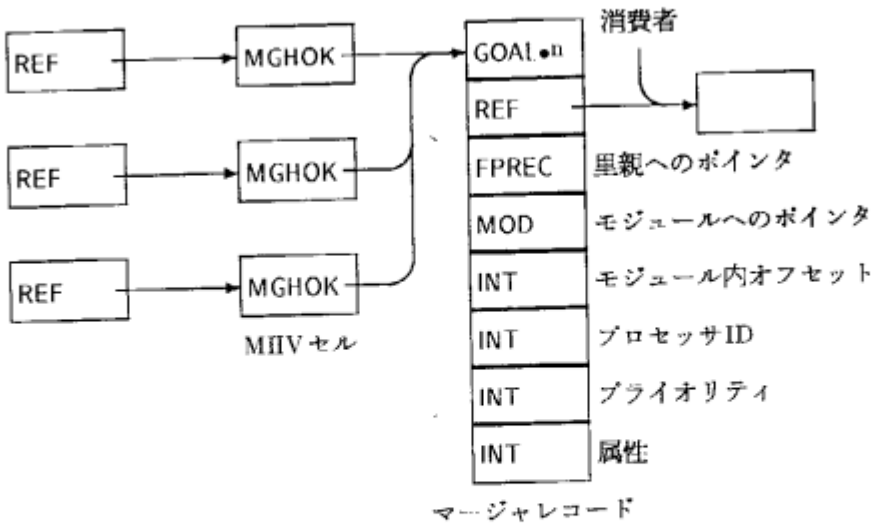


図 7-9: マージャプロセスの表現方法

7.6.3 NIL とのユニフィケーション

ユニフィケーションの意味

入力ストリームが一本消滅したことを意味する。

処理概要

1. MHV セルを NIL で具体化する (失敗したら、D コードによる再試行)
2. 参照カウントを 1 減じる
3. 参照カウントが 0 になったら、出力と NIL をユニファイする

7.6.4 リストとのユニフィケーション

ユニフィケーションの意味

入力ストリームへの 1 入力である。ユニフィケーション前の状態を図 7-10 に示し、ユニフィケーション成功後の状態を図 7-11 に示す。

処理概要

1. 新しいリストセルと変数セルを 1 つずつ割り当てる
2. Car に入力リストの Car をコピー。Cdr には、新しく割り付けた変数セルへのポインタを書く
3. MHV セルを入力リストで具体化する (失敗したら、D コードによる再試行)
4. 新しい MHV セルを割り付け、入力リストの Cdr とユニファイする
5. マージャの出力を、新しく割り付けた変数セルへのポインタとする

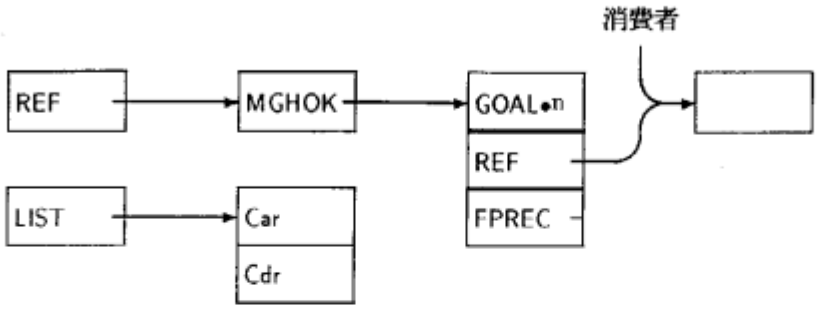


図 7–10: MHV セルとリストのユニフィケーション(前)

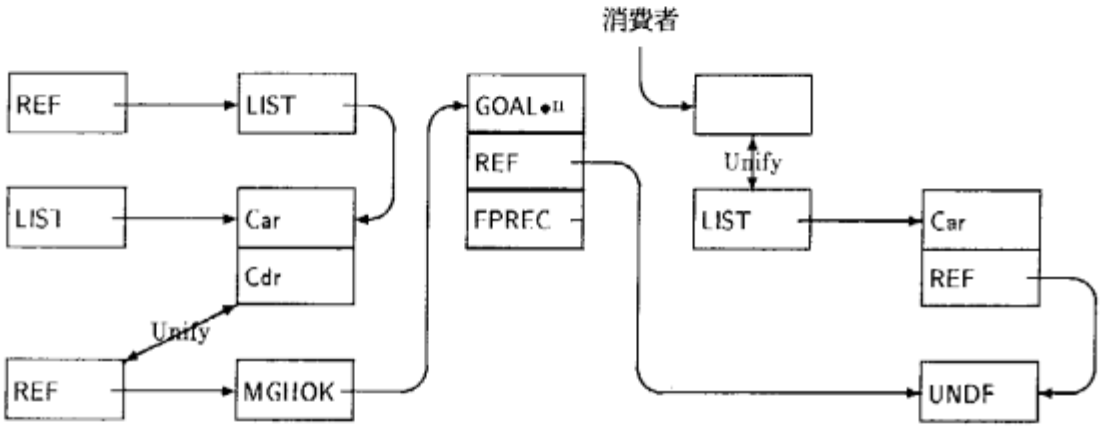


図 7–11: MHV セルとリストのユニフィケーション(後)

6. 書き換え前のマージャの出力と、新しいリストセルをユニファイする(失敗したら、Dコードによる再試行。ただし、これはマージャの所属する里親内で動くゴールとなる)

ただし、MHV セルが白パスの場合、入力リストでの具体化が省略でき、新しいMHV セルとして再利用できる。

更に、入力リストが白パスの場合、出力用に再利用できる。

7.6.5 ベクタとのユニフィケーション

ユニフィケーションの意味

入力ストリームの追加であると解釈する。組込述語 merge/2は、次のように定義されているものとする。

$merge(\{\}, Out)$ $:- true \mid Out = []$.
$merge(\{I1\}, Out)$ $:- true \mid merge(I1, Out)$.
$merge(\{I1, I2\}, Out)$ $:- true \mid merge(I1, I2, Out)$.
$merge(\{I1, I2, I3\}, Out)$ $:- true \mid merge(I1, I2, I3, Out)$.
$merge(\{I1, I2, I3, I4\}, Out)$ $:- true \mid merge(I1, I2, I3, I4, Out)$.
:
:

処理概要

1. ベクタの要素数だけ新しくMHV セルを割り付け、マージャの参照カウントを(要素数-1)増やす
2. 入力 MHV セルをベクタで具体化する
3. 新たに割り付けた MHV セルと、ベクタの各要素をユニファイする

7.6.6 UNDF, VOID とのユニフィケーション

ユニフィケーションの意味

二つの未定義変数を同じものとする。

処理概要

単純に UNDF セルから MHV セルへのパスを張れば良い。VOID へのパスと、MHV セルへのパスが共に白い場合のみ、(それ以降、そのストリームに入力がなくなることが保証されるので)マージャの永久中断の検出となる。

マージャ自体は、リダクションを行なわないので、マージャの里親の状態を見る必要はない。

7.6.7 HOOK 系未定義変数とのユニフィケーション

ユニフィケーションの意味

二つの未定義変数を同じものとする。

処理概要

白パス MGHOK と、白パス HOOK。の場合は、マージャ、ゴールともに永久中断となるので、その報告を行なう。

マージャ自体は、リダクションを行なわないので、マージャの里親の状態を見る必要はない。

図7-12の状態から、ユニフィケーションを行なう方式は以下の通りである。

1. MGHOK から HOOK へのパスを張る(図7-13)。Compare & Swap で行ない、失敗した時は、D コードゴールで再試行する。ただし、MGHOK へのパスが白い場合は、何もしなくて良い。

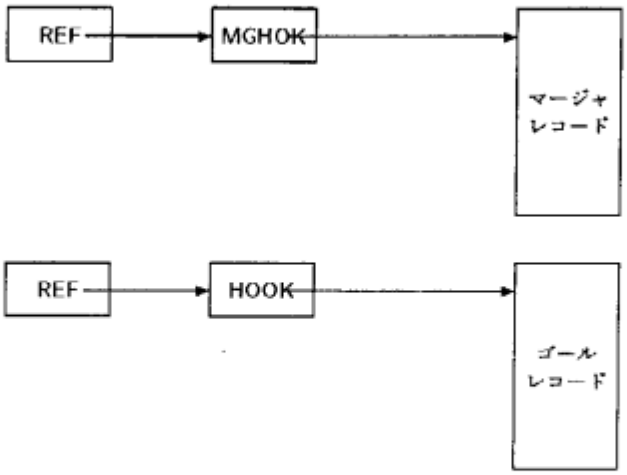


図 7–12: (1) ユニフィケーション前

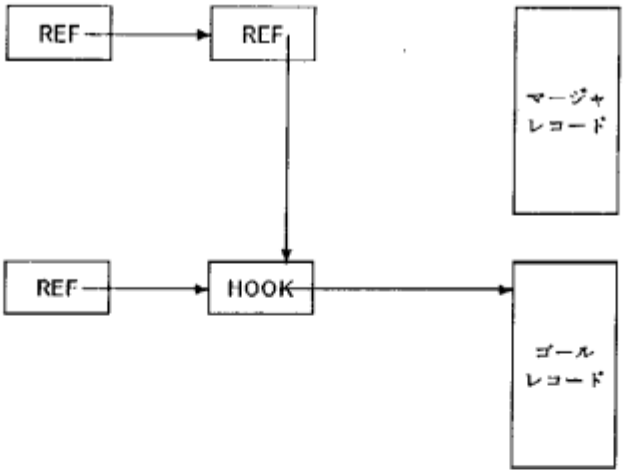


図 7–13: (2) MGHOKからHOOKへのパスを張る

2. 新たにMHV セルを割り付ける(図7–14)。ただし、ただし、MGHOKへのパスが白い場合は、割り付ける代わりに再利用できる。
3. 新たに割り付けたMHV セルと、HOOKをユニファイするゴール(D コードゴール)を作り、HOOKチェーンにプッシュする。

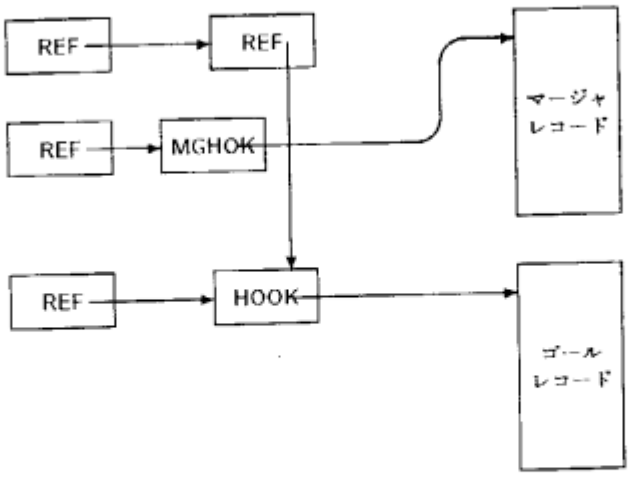


図 7-14: (3) MHV セルを割り付ける

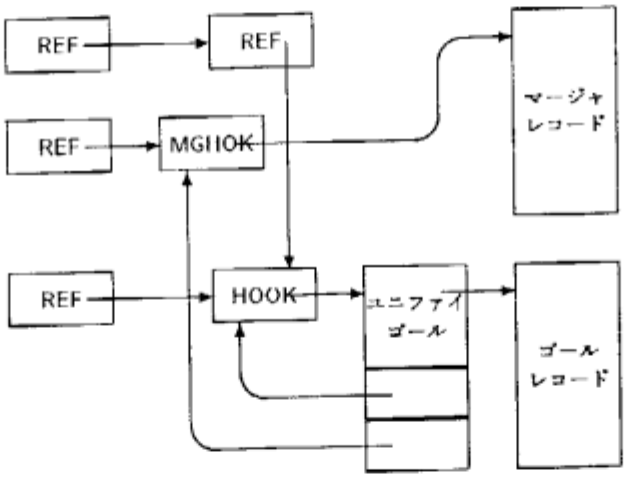


図 7-15: (4) MGHOK と、HOOK をユニファイするゴールをフックする

7.6.8 MGHOK 同士のユニフィケーション

ユニフィケーションの意味

二つの未定義変数同士のユニフィケーションであり、これ以降の入力は両マージャの出力へ出力される。

処理概要

MHV セルへのパスがいずれも白い場合は、両マージャともに永久中断となる。

アドレスを比較し、大きい方から小さい方に向けてパスを張る。ただし、一方でも白パスの場合は、

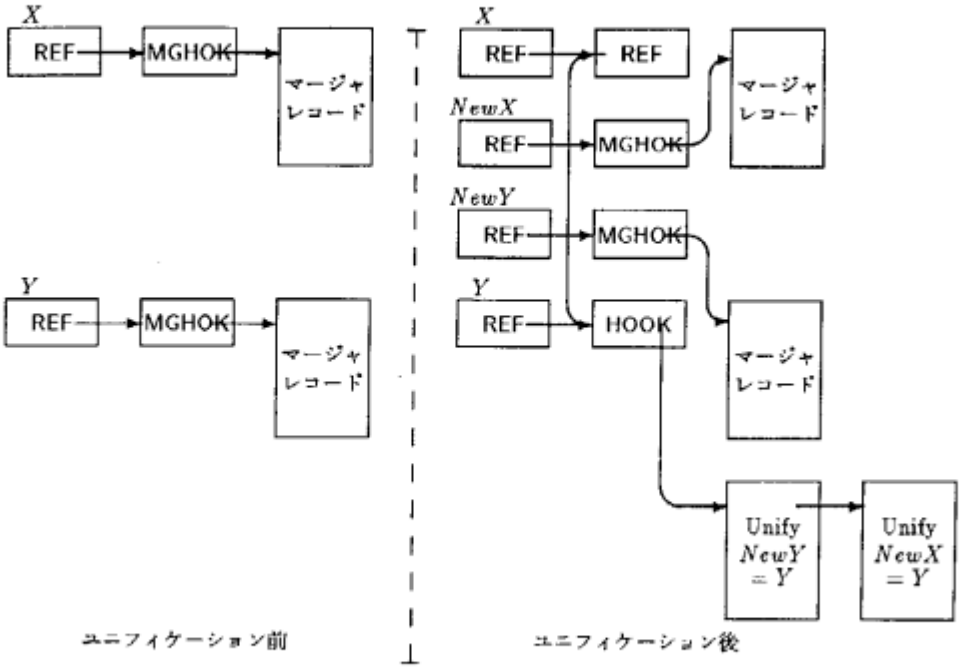


図 7 16: MGHOK同士のユニフィケーション

他からの参照がないのでパスを張らない。

その後、ユニファイを行なうゴールを2つフックさせる(図7-16参照のこと)。

7.6.9 マージャの所属する里親が実行不可能な時

マージャの所属する里親が実行不可能な時(10.2節参照)は、出力(リストとのユニファイ)や入力ストリームの追加/消滅(NILやベクタとのユニファイ)を行なってはならない。

ただし、MHV セルを具体化する操作は(他にこのセルを参照しているパスがあれば)行なわなければならない。

ストップ状態の場合

再開に備え、新たにMHV セルを割り付け(MRBoの場合には具体化する代わりに再利用できる)、このセルとのユニフィケーションを行なうゴールを作る。その後、里親の停止中のゴールをつなぐリンクルート(10.2節参照)にこのゴールをプッシュする。

アボート状態の場合

再開されることがないので、出力を行なわない。

第 8 章

スリットチェック

執筆担当者: 堂前

本章では、KL1の並列実行過程における多種多様なイベント処理を行なう「スリットチェック」という機構について説明する。

8.1 スリットチェックとは

スリットチェックとは、KL1の並列実行過程における多種多様なイベント処理を、(一般のマシンのハード機構としての)割込み機構を用いた場合のコンテクストスイッチによるオーバヘッド、及び要因分析時のキャッシュミスによる共有バスのトラフィック削減を目的として、リダクション間でのみ行なうイベント処理機構である。イベント検出、及びイベント処理の実行順序をソフトで制御するということは、それぞれのイベントにプライオリティを持たせるという意味において重要である。

イベントをリダクション間で一括して処理する環境として、PE毎に**スリットチェックレジスタ** (SCR)を導入し、メモリの参照を削減する。また、イベント検出高速化のためにSCRのデータをORした値を常時保持する**スリットチェックフラグ** (SCF)を導入する。SCRの各ビットにイベントがそれぞれ対応する。現在VPIMでは13個のイベントに対し(表8-1を参照のこと)、SCRの12ビット(第0~11ビット)が対応している。

VPIMで用意しているSCR関連のプリミティブは以下の通りである。

```
p_IfSCFON()
p_GetSCR(DestReg)
p_SetSCRBitImmPosition(ImmSignalPosition,PENumberReg)
p_ResetSCRBitImmPosition(ImmSignalPosition,PENumberReg)
p_SetSCRBitImmPositionAll(ImmSignalPosition)
p_ResetSCRBitImmPositionAll(ImmSignalPosition)
p_PutSCRBit(SignalValueReg,SignalPositionReg,PENumberReg) : 未使用
```

8.2 スリットチェック処理の分類

表 8-1: スリットチェックレジスタのビット位置とイベント ID

ビット位置	:	イベントID
0	:	_NW_MSG_PACKET_ARV_EVENT
1	:	_TIMER_INT_EVENT
2	:	_SPC_INT_EVENT
3	:	_ALL_SYNCHRONIZE_EVENT
4	:	_FP_STATUS_CHECK_EVENT
5	:	_GOAL_REQ_EVENT
6	:	_THROW_GOAL_EVENT
7	:	_SERIAL_EXEC_EVENT
8	:	_HIGH_PRIO_GOAL_EVENT
9	:	_FP_STAT_GATHER_EVENT
10	:	_PAGE_ALLOC_FAIL_EVENT
11	:	_RESOURCE_CACHE_EXHAUST_EVENT
12~31	:	(Reserved Bits)

(1) 外部割込み

外部機器からの割込み。

(2) PE 間シグナル

クラスタ内の他 PE からの割込み。**PE 間シグナル命令** (前述SCR関連プリミティブ参照のこと)によってセットされる。PE間の通信形態には3つのタイプがある。

- 1 to 1 通信：相手先のPEを一つ指定したクラスタ内メッセージ通信
  シグナルは一つのPEのみに送る。自動負荷分散要求への応答、高プライオリティゴール要求への応答、及びリジュームによる3種類のクラスタ内ゴール分散処理に用いる。複数PEが同じPEにスリットチェック処理前に同じ要因で割り込むことを以下では**多重割込み**という。1 to 1 通信における多重割込みは、システム共有固定領域中の通信エリアにスタックを形成し、マージすることによって最適化する。即ち、同一のPEに複数のゴールが(スリットチェック処理前に)送信される場合、通信エリアにある受信PE毎に用意したゴールポストにゴールレコードをプッシュすることによって、一回の処理で複数のゴールを受信する。
  この他に再現性を確保するデバッグモードで、複数PEがオーバーラップ実行しないよう1リダクション実行が終了したら、隣のPEに知らせるために使用している。
- 1 to any 通信：クラスタ内の任意のPEへのクラスタ内メッセージ通信
  シグナルは全PEに送る。自動負荷分散要求や高プライオリティゴール要求に用いる。
  1 to any 通信では通信エリアにPE番号に対応したビットマップを置き、送信するPEが自分の番

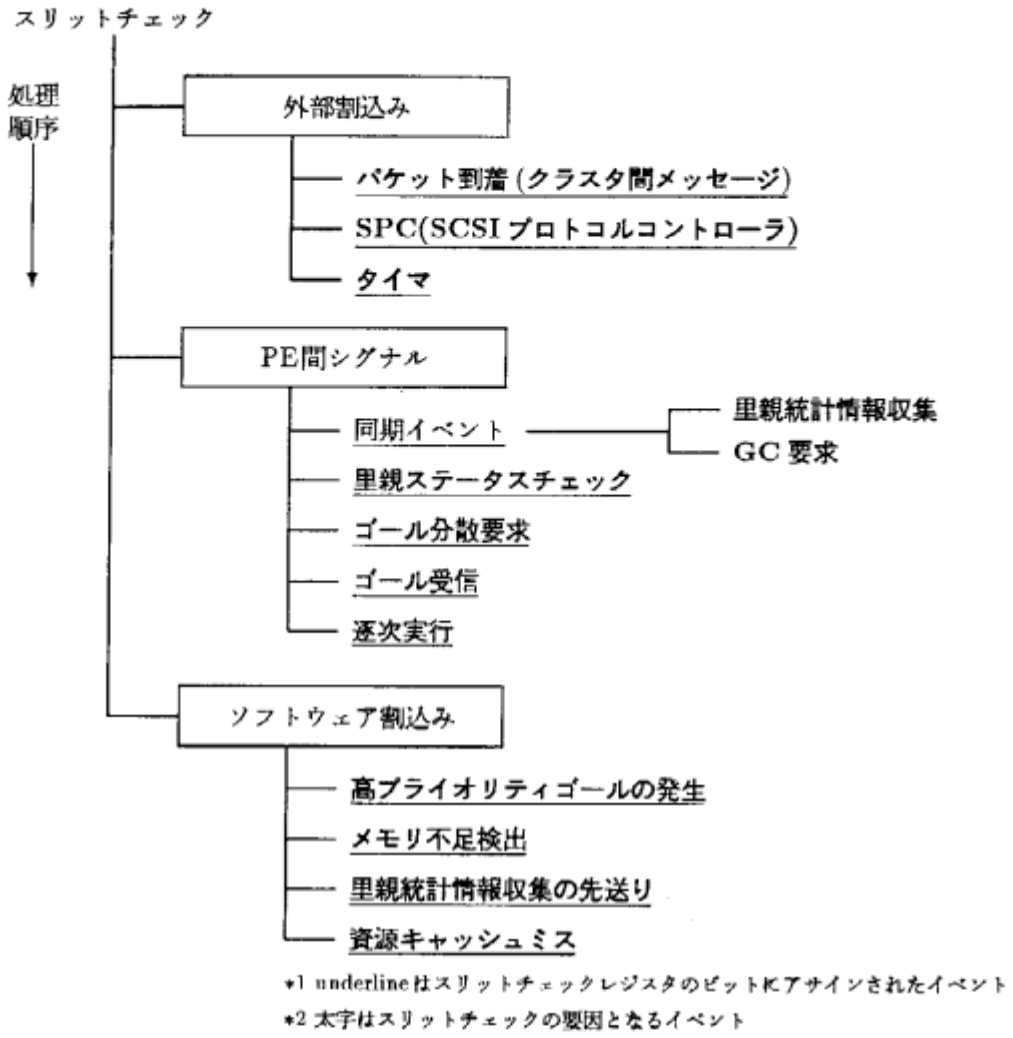


図 8-1: スリットチェック処理の分類

号に対応したビットをONにすることでゴールの分散要求を出す。この通信では要求に対し応答するPEはクラスタ内のいずれか一つのPEである。

- 1 to all 通信：クラスタ内の全PEへのクラスタ内メッセージ通信
  シグナルは全PEに送り、それらが受信されることを期待する。一括GCのように同期が必要な場合、通信エリアに割り当てたバリア を使用する。
  バリアの基本操作：
  (i) シグナルを送信するPEが通信エリアのバリアにクラスタ内PE台数をセットし、全PEにシグナルを送る。
  (ii) シグナルを受信したPEは、バリアのカウンタ値をデクリメントする。デクリメントした結果がゼロであれば、全PEがシグナルに気付いたことを意味する。

(iii) 同期をとるためのシグナルを全PEに送る[1]。

(3) ソフトウェア割込み

自PEからの割り出し。同一の内部イベントが発生し得る状況では、内部イベントスタックを用いる。荘園内の資源問い合わせ等の全PEに知らせ、且つ同期を必要とするイベントが発生すると、PEは通信エリアにあるバリアとメッセージスロットの設定を行ない、全PEにシグナルを送る。この時既にバリアを設定して問い合わせが行なわれていた場合、次のスリットチェックで処理するためにPE毎に用意した内部イベントスタックに先送りするイベントをプッシュし、自PEに割込みをかけておく。

8.3 イベント処理

スリットチェック時に行なう各イベント処理について説明する。

(1) パケット到着割込み

[要因]
ネットワークからクラスタ間メッセージパケットが到着した。
[検出]
SCRの第0ビットが1である。
[処理]
要因をリセットし[2]、パケット受信処理を行なう。

(2) タイマ割込み

[要因]
VPIMでは1リダクション毎にインクリメントするカウンタを設け[3]、そのカウンタが設定値を越えたら割込みを上げる。
[検出]
SCRの第1ビットが1である。
[処理]
要因をリセットし、プライオリティ制御の圧力モデルにしたがって自PEの圧力値を更新する。この時圧力値が上限を越えたら、ゴール分散要求を出して圧力をリセットする。

〈〈 プライオリティ制御の圧力モデル 〉〉

- 要求仕様
  - プライオリティ制御のために頻繁な割込みやポーリングを生じない。

[1] シグナルは最後にバリアをデクリメントしたPEが送る場合と、最初に要求を出してバリアをポーリングして待っていたPEが送る場合の2通りがある。

[2] 要因のリセットとは、自PEのイベントに対応したSCRのビットを下ろすことをいう。ここでは第0ビットが対象である。

[3] 実機では省略できる。

- 最高位プライオリティの陰に中間位プライオリティゴールが沈み込まない。
- クラスタ内で共有のグローバルスタックは考えない。

• 処理方式

次のパラメータを有する制御を行なう。

D_Max_Priority	:	自PEのレディゴールスタック内の最高位プライオリティを示すレジスタ
CL_Max_Priority	:	共有メモリ中にあるクラスタ内の最高位プライオリティと思われる値
D_Pressure	:	圧力値を示すレジスタ
_PRIO_LOW_LIMIT	:	負の整数値。圧力がこれを越えるとCL_Max_Priorityを更新する
_PRIO_HIGH_LIMIT	:	正の整数値。圧力がこれを越えると他のPEにゴール分散要求をだす

タイマ割込みを受けるとPEはD_Pressureを更新する。

D_Pressure ⟵ D_Pressure + (CL_Max_Priority - D_Max_Priority)

D_Pressure < _PRIO_LOW_LIMITの場合：

そのPEはクラスタ内最高位プライオリティよりも高いプライオリティのゴールを実行し続けていたと見做し、CL_Max_PriorityをD_Max_Priorityに更新する。これによりCL_Max_Priorityよりも低いプライオリティのゴールを実行していたPEの圧力が高まり易くなる。

D_Pressure > _PRIO_HIGH_LIMITの場合：

そのPEはクラスタ内最高位プライオリティよりも低いプライオリティのゴールを実行し続けていたと見做し、他のPEに対しゴール分散要求を出す。

ゴール分散要求を受けたPEは投げ得る最高位プライオリティのゴールを投げる。ただし、ゴール分散要求に気付いた最初のPEが投げるので、要求を発したPEの持つ最高位のプライオリティのゴールより高いプライオリティのゴールが投げられるという保証はない。

(3) SPC 割込み

[要因]

SPCから割込み(Selected , Reselected , Disconnected , ResetCondition)が上がった。

[検出]

SCR の第2ビットが1である。割込みがどのような要因で上がったのかはSPC の INTS(Interrupt Sense) レジスタを読み出すことによって解析する。

[処理]

要因のリセットはSPCのINTSレジスタの対応するビットを0にすることで行なう。SCRの第2ビットはINTSレジスタの全てのビットが0になると自動的に0になる。それぞれの割込み要因に対して、それぞれのハンドラを呼び出す。

(4) GC要求

[要因]
ヒープメモリ不足を検出したPEが一括GC要求を出した。

[検出]
SCRの第3ビットが1であり、且つ通信エリアの1 to all通信用メッセージスロットのタイプが_GC_REQUESTである。

[処理]
通信エリアの1 to all通信用のバリアをロックして読み出し、ロック期間中に要因をリセットしてからバリアの値をデクリメントして書き戻しアンロックする。バリアの値が0になった時、即ち割込みに最後に気付いた時GC開始の同期をとるために、もう一度全PEにシグナルを送りGC処理を開始する。最後でなかった時はGC開始のシグナルが送られてくるのをビジーウェイトする。

(5) 里親統計情報収集

[要因]
荘園内の資源問い合わせがあった。要求を出したPEは全PEが要求に気付き、キャッシュされている対象の里親の資源が書き戻されるのを通信エリア内のバリアをポーリングして待っている。

[検出]
SCRの第3ビットが1であり、且つ通信エリアの1 to all通信用メッセージスロットのタイプが_FP_IN_EVT_{ABORTED | ASK_STAT | ANS_STAT}である。

[処理]
メッセージスロットのバリュー部には資源集計対象の里親へのポインタが書かれている。

(i) カレント里親が資源集計対象の場合は、キャッシュしていた資源を里親に書き戻す。実行しようとしていたゴールはリスケジュールする。

(ii) 通信エリアの1 to all通信用のバリアをロックして読み出し、ロック期間中に要因をリセットしてからバリアの値をデクリメントして書き戻しアンロックする。

(iii) メッセージがASK_STAT、ANS_STATの時は、カレント里親が資源集計対象の里親であるか否かにかかわらず、ビジーウェイトして資源集計対象の里親の消費資源量が確定するのを待つ。ABORTの時は他のPEでリダクションを行なっても、既にアボートされた里親の消費資源量に影響を与えないので、ビジーウェイトして待つ必要はない。

(iv) 要求を出したPEが、全PEがこのイベントに気付いたことを知ると、同期をとるためビジーウェイトしている全PEにシグナルを送る。ただし、ABORTの時にはシグナルは送らない。

(6) 里親ステータスチェック

[要因]
ある里親の状態がstopped、またはabortedに変化した。

[検出]
SCRの第4ビットが1である。
[処理]
要因をリセットし、カレントゴールの属する里親が実行不可能状態になっていたら、キャッシュしていた資源を里親に書き戻し、ゴールをリスケジュールする。

(7) ゴール分散要求

[要因]
他PE(複数の可能性がある)が自動負荷分散要求や高プライオリティゴール要求を出した。自動負荷分散要求はPEがアイドル状態に遷移した時、高プライオリティゴール要求はPEの圧力値が上限値を越えた時(前述)に出される。
[検出]
SCRの第5ビットが1である。
[処理]
通信エリアにある要求ビットマップをロックして読み出し、ロック期間中に全PEのSCRのゴール要求ビット(第5ビット)をリセットしてから、ビットマップをクリアしてアンロックする。こうすることにより、他のPEがシグナルを受信することを避ける。ゴール分散要求を受信したPEは、ビットマップに対応した全てのPEの要求に応える。ゴールが足りなくなったら、代わりに要求を出し直す。
尚、要求に応じて投げ得るゴールの条件は次の通りである。

- アイドルゴールの他に3個(可変)以上のゴールがある
- ゴール属性が processor-resident でない

(8) ゴール受信

[要因]
自PEで出した負荷分散要求、高プライオリティゴール要求によって投げられたゴールを受信した。または自PEでサスペンドし、他PEのもとでリジュームして投げられたゴールを受信した。
[検出]
SCRの第6ビットが1である。
[処理]
通信エリアにある自PEのゴールポストからゴールレコードのリンクを読み出す。この時ロックして読み出し、ロック期間中に要因のリセットをしてゴールポストにEOL(End Of Link)を書き込んでアンロックする。これはゴールポストのロック期間中にSCRの操作を入れることで、SCRがOFFなのに処理されていないゴールが残ったり、SCRがONなのに処理すべきゴールがないという事態を回避するためである。受信したゴール(複数の可能性あり)は自PEのそれぞれのプライオリティに対応したレディゴールスタックに入れる。

(9) 逐次実行

[要因]
再現性を確保するデバッグモードで動作している時、隣のPEが1リダクション実行が終って割込みをかけた。
[検出]
SCRの第7ビットが1である。
[処理]
この割込みがかかるまでPEはビジーウェイトしている。割込まれたら要因をリセットし、1リダクション実行して隣のPEに割込み、再びビジーウェイトする。

(10) 高プライオリティゴールの発生

[要因]
前のリダクションで、これまで実行していたゴールのプライオリティよりも高いプライオリティのゴールをフォークした。
[検出]
SCRの第8ビットが1である。
[処理]
要因をリセットし、カレントゴールをレディゴールスタックに戻しカレントスタックエントリポインタ、カレントゴールスタックポインタを更新する。

(11) 里親統計情報収集の先送り

[要因]
前のリダクションで荘園内の消費資源の問い合わせがあったが、既に他PEが通信エリアのバリアを使用していたため、処理を先送りにした。
[検出]
SCRの第9ビットが1である。
[処理]
内部イベントスタックからイベントをポップし、それが最後のイベントだったら要因をリセットする。他PEが問い合わせを行なっていなければ、バリアを設定し全PEにシグナルを送り、里親統計情報収集の処理を行なう。既に他PEが問い合わせを行なっていたならば、イベントを内部イベントスタックにプッシュし、自PEに割込みをかけておく。

(12) メモリ不足検出

[要因]
前のリダクションでヒープメモリ不足を検出した。

[検出]

SCRの第10ビットが1である。

[処理]

通信エリア内の1 to all通信用のメッセージスロットの内容を調べ、他のPEが使用していなかったらメッセージ(_GC_REQUEST)の設定を行なう。メッセージの設定ができたらバリアの設定を行ない、全PEにGC要求のシグナルを送り要因をリセットする。既に他のPEが_GC_REQUESTの設定をしていたならば要因をリセットし、何もしない。_GC_REQUEST以外のメッセージが設定されていた場合は要因はリセットせず、次のスリットチェック時に処理する[4]。

(13) 資源キャッシュミス

[要因]

キャッシュしていた資源が0になり、里親に資源を貰いにいったが里親の資源も0であった。

[検出]

SCRの第11ビットが1である。

[処理]

要因をリセットし、カレントゴールポインタにゴールがあればリスケジュールする。

[4]既に設定されていたメッセージの処理でメモリ消費が伴う場合、GC要求が空振りを続ける可能性がある。今後の課題。

第 9 章

クラスタ間処理

執筆担当者: 仲瀬, 高木

9.1 クラスタ間処理用データ構造

9.1.1 クラスタ間データ参照の生じる状況

KL1処理系ではゴールを他のクラスタに負荷分散すること等により、クラスタ間に渡るデータ参照が生じる。このデータ参照を**外部参照** と呼ぶ。自クラスタ内にあるデータを他のクラスタから参照可能にすることを**データの輸出** と呼び、他のクラスタからのデータを参照可能にすることを**データの輸入** と呼ぶ。

データを輸出する際は、そのデータへのポインタを**輸出表** に登録する。輸出表は**輸出表エントリ** と呼ばれる1ワードセルの集まりで、クラスタ内のシステム固定領域に割り付けられる。

輸出表エリアの先頭からのオフセットとデータを輸出したクラスタ番号、その他データの属性を示す幾つかのフラグを1ワードにまとめたものを**外部参照ID** という。データを輸出する時に送るメッセージには外部参照IDが入れられる。

外部参照IDを受けとったクラスタは、その外部参照IDを**輸入表** に登録する。輸入表を構成する各要素は**輸入レコード** と呼ばれ、輸入したデータの外部参照IDは輸入レコードに入れられる。輸入レコードに外部参照IDを入れた後に、輸入レコードを指す1ワードの**外部参照セル** を割り付ける。

外部参照の様子を図9-1に示す。この例では、クラスタ1にあるデータをクラスタ2に輸出している状態を示している。クラスタ2では、外部参照セル、輸入表、輸出表を通してクラスタ1のデータが参照できる。

9.1.2 外部参照 ID の構成

外部参照 ID の構成を図9-2に示す。タグ部には、WEXREFo, WEXVALo, BEXREFo, BEXVALo, INTo のいずれかが入る。制御フラグとして実際はSafe/Unsafeフラグ、構造体フラグ、 Unifyフラグの 3ビットを使用しており、残りの1ビットは予備としている。これらタグの使い分け、制御ビットについては以下の説明の中で述べる。

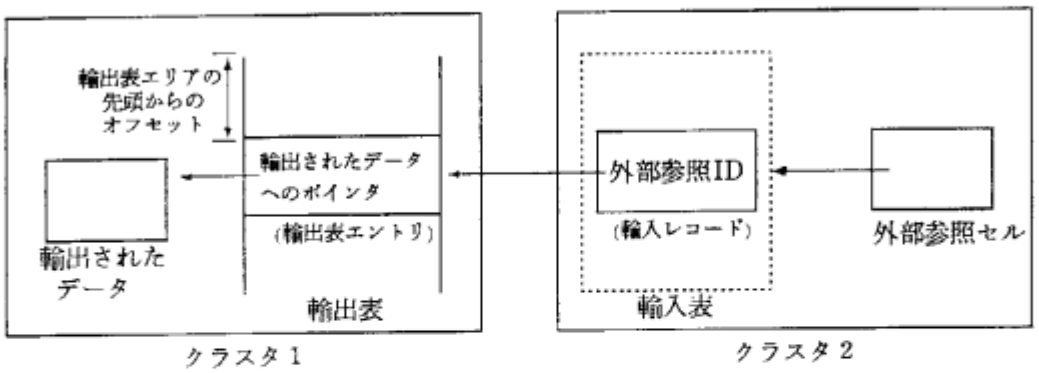


図 9-1: 外部参照方式

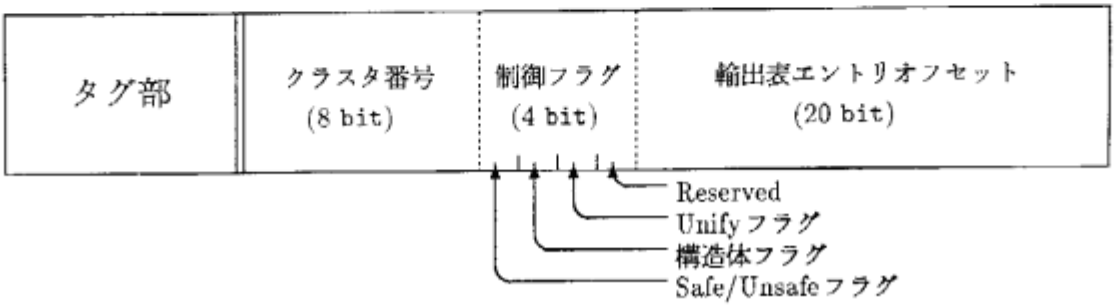


図 9-2: 外部参照 ID の構成

9.1.3 単一参照のデータに対する外部参照

(1) データの輸出

単一参照のデータを輸出する時は、輸出表にそのデータへのポインタを登録する。単一参照のデータへのポインタを持つ輸出表エントリを**白輸出表** と呼ぶ。

白輸出表にデータを登録することによって得られる外部参照IDを、**白外部参照 ID** と呼ぶ。

メッセージ中の白外部参照IDのタグはWEXREF。又はWEXVAL。になる。WEXREF。は値が確定していないデータを輸出した時に付けられ、WEXVAL。は値が具体化しているデータを輸出した時に付けられる。

(2) データの輸入

白外部参照IDを輸入した時は、その内容を**白輸入表** に登録する。**白輸入表は白輸入レコード** より構成され、輸入したデータの外部参照IDが入る。白輸入レコード内の外部参照IDのタグはINT。にする。白輸入の操作を行なうと、白輸入レコードを指す外部参照セルを1ワード割り付ける。外部参照セルのタグは、メッセージ中の外部参照IDのタグと同一のものにする。また一括GCのメンテナンスを容易にするため、白輸入表はシステム固定領域に割り付けられる。

単一参照のデータが白輸出表、白輸入表を介して他のクラスタから参照可能になる様子を図9-3に示す。

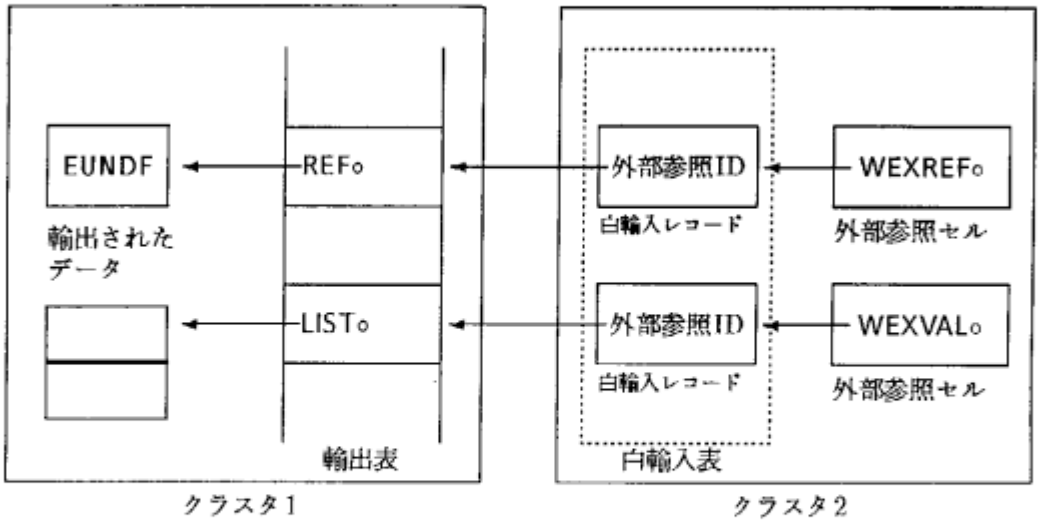


図 9-3: 白輸出入方式

9.1.4 多重参照のデータに対する外部参照

(1) データの輸出

多重参照のデータを輸出する時は、**黒輸出レコード** にデータへのポインタを登録し、黒輸出レコードへのポインタを輸出表に登録する。黒輸出レコードへのポインタを持つ輸出表エントリと黒輸出レコードを合わせて**黒輸出表** と呼ぶ。

多重参照のデータへのパスは複数存在する可能性があるので、同じアドレスの多重参照データは同じ黒輸出レコードに登録される。(同じデータを2回以上輸出することを**再輸出** と呼ぶ。)また、黒輸出表に登録されたデータの検索を高速にするために、黒輸出表は、そのアドレスをキーとしたハッシュ表にも登録する。

黒輸出レコードには、輸出されたデータへのポインタ、WEC、ハッシュコリジョン用のチェーンが入れられる。WECに関しては9.1.5で述べる。

黒輸出表にデータを登録することによって得られる外部参照IDを**黒外部参照ID** と呼ぶ。

黒外部参照IDのタグはBEXREFo又はBEXVALoである。BEXREFoは値が確定していないデータを輸出した時に付けられ、BEXVALoは値が具体化しているデータを輸出した時に付けられる。

(2) データの輸入

黒外部参照IDを輸入した時は、その内容を**黒輸入表** に登録する。黒輸入表は**黒輸入レコード** から構成される。

黒輸入レコードは、輸入された外部参照ID、WEC、外部参照セルを指すポインタ、ハッシュコリジョン用のチェーンの4ワードより構成される。黒輸入レコード内の外部参照IDのタグはINToとする。また、外部参照セルを指すポインタのMRBを1度目の輸入では白とし、再輸入が行なわれると黒とする。このMRBを**再輸入フラグ** と呼び、黒外部参照IDの再輸入のチェックに使用する。

1つの黒外部参照IDは複数回輸入される可能性があるので、同じ黒外部参照IDは同じ黒輸入レコードに登録する。(同じデータを2回以上輸入することを**再輸入** と言う。)黒輸入表に登録した黒外部参照IDの検索を高速にするために、黒輸入表は、その黒外部参照IDをキーとしたハッシュ表にも登録する。

黒輸入を行なうと、黒輸入レコードを指す外部参照セルが1つ割り付けられる。外部参照セルのタグはメッセージ中の外部参照IDのタグと同じにする。また、再輸入の時には新たに外部参照セルを割り付けず、既存の外部参照セルへのポインタを使用する。

多重参照のデータが黒輸出表、黒輸入表を介して他のクラスタから参照可能になる様子を図9-4に示す。

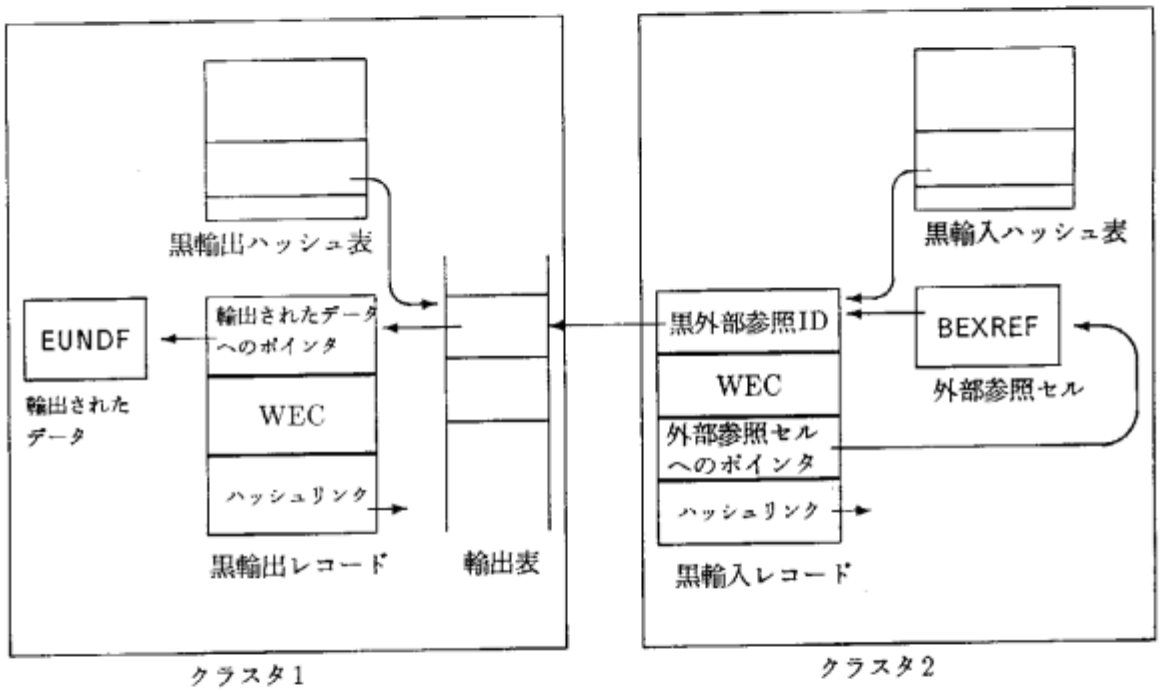


図 9-4: 黒輸出入方式

9.1.5 WEC を用いた即時 GC 方式

クラスタ間で参照されているデータのGC方式として**重みつき参照カウント方式 (WEC 方式[1])** を採用している。WECはデータが輸出表に登録された時にそのデータの外部参照IDに与えられる整数値で、ある外部参照IDに対して、それを持つ輸出入表エントリ(レコード)とネットワークメッセージ中のWECの合計が0となる様に分配される。

WECは、輸出表内では負の値になり、メッセージ内と輸入レコード内では正の値になるものとする。

WECを持つ輸入レコードがGCによって回収されると、輸入レコード内のWECを輸出表に(輸入レコード内の外部参照IDが指す先に)返す。

WECを返された輸出表では、返信されたWECと輸出表内のWECを足し算し、WECが0になると

[1]Weighted Export Count方式

その輸出表エントリと輸出レコードは回収される。(輸出されたデータに対しても回収を試みる。)

図9-5から図9-11にWECを用いてクラスタ間GCを行なう様子を示す。

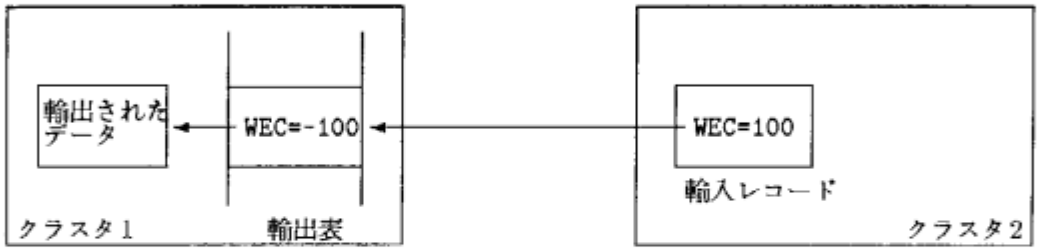


図 9-5: クラスタ 1 からクラスタ 2 にデータが輸出された状態

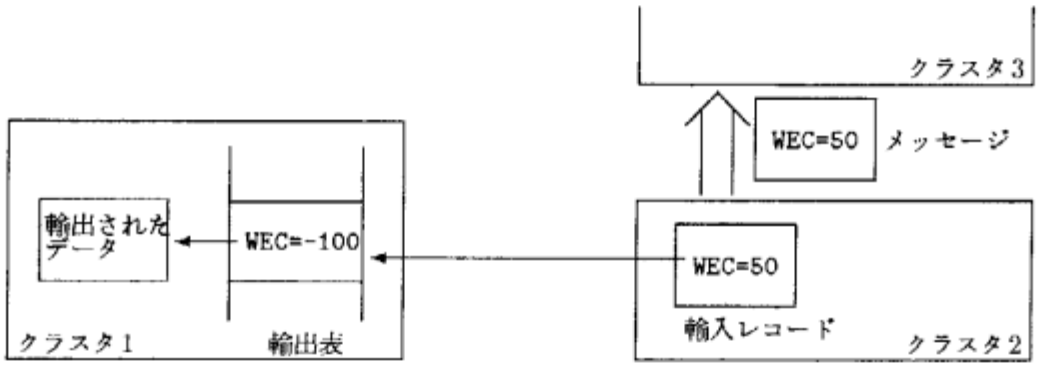


図 9-6: クラスタ 2 からクラスタ 3 にデータが分配される状態

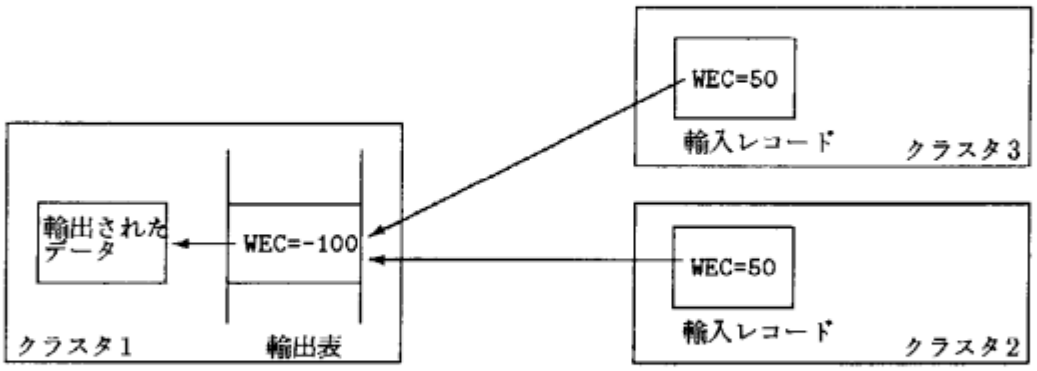


図 9-7: クラスタ 2 からクラスタ 3 にデータが分配された状態

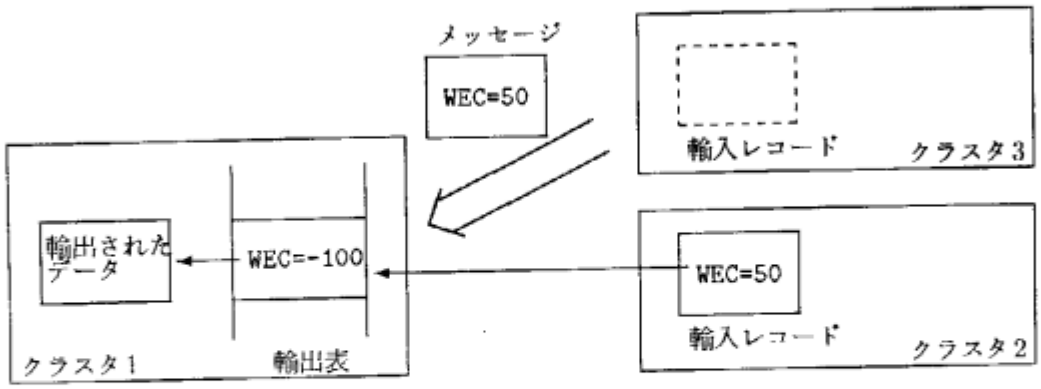


図 9-8: クラスタ 3 のデータ回収されて WEC がクラスタ 1 に返される状態

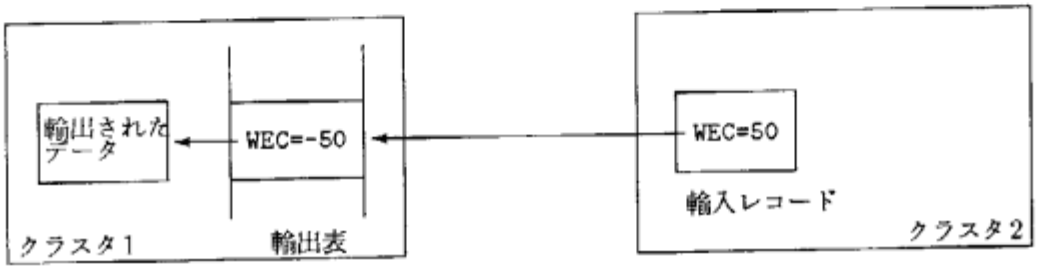


図 9-9: クラスタ 3 のデータの WEC がクラスタ 1 に返された状態

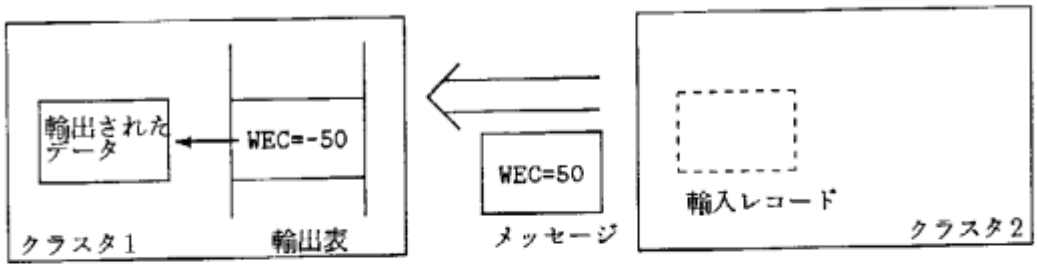


図 9-10: クラスタ 2 のデータが回収されて WEC がクラスタ 1 に返される状態

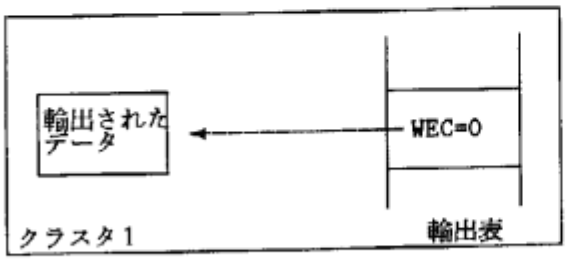


図 9-11: クラスタ 2 のデータの WEC がクラスタ 1 に返された状態

(1) 黒輸出入表における WEC

黒外部参照IDには1回輸出される毎に 2^{24} のWECが付けられる。

黒輸出レコードのWECは2ワード確保されており、0から -2^{64} までWEC値の登録が可能である。黒輸入レコードのWECは1ワードで、0から 2^{32} までの登録が可能である。

黒輸入レコードのWEC値が 2^{32} を越えると、2^{24} のWECが黒輸入レコードに残され、残りのWECは黒輸出レコードに返される。

黒輸出レコードのWECはオーバーフローしないことを前提としているが、クラスタ数やクラスタ内のメモリ容量が大きくなると、2ワード以上確保する必要がある。

(2) 白輸出入表における WEC

白外部参照IDではUnifyフラグをWECの代わりとして用いる。白輸出表エントリではデータへのポインタのMRBをWECの代わりとして用いる。

各々のビットの意味は以下に示す通りである。

表 9–1: 白輸出入表内のフラグと対応する WEC との関係

	フラグの状態	対応するWEC
外部参照ID	unifyフラグoff	WEC=2
	unifyフラグon	WEC=1
白輸出表	MRB白	WEC=-2
	MRB黒	WEC=-1

9.1.6 外部参照セルの Safe/Unsafe 属性

変数セルと外部参照セルをアクティブユニファイする時は、殆んどの場合変数セルから外部参照セルへパスを張るだけで良い。しかし、状況によっては図9–12に示すようにクラスタ間のループ構造が出来てしまうことがある。

そこで外部参照セルに対して、自クラスタ内で変数とアクティブユニファイ可能かを示すフラグをつける。これをSafe/Unsafeフラグと呼び、OFFの時をSafe属性、ONの時をUnsafe属性とする。

ある外部参照セルに対して、外部参照先のクラスタ番号が自クラスタ番号より小さい時に、その外部参照セルは **Safe 属性** であるといい、その逆を **Unsafe 属性** であるという。

輸出された可能性のある変数セルと外部参照セルをアクティブユニファイする時には、外部参照セルがSafe属性ならばそのままパスを張り、Unsafe属性ならば外部参照先のクラスタにアクティブユニファイを依頼する。

図9–12の例では、クラスタ2ではSafe属性の外部参照セルと輸出された変数のユニフィケーションなので、クラスタ内で変数セルから外部参照セルへのパスを張る。クラスタ1ではUnsafe属性の外部参照

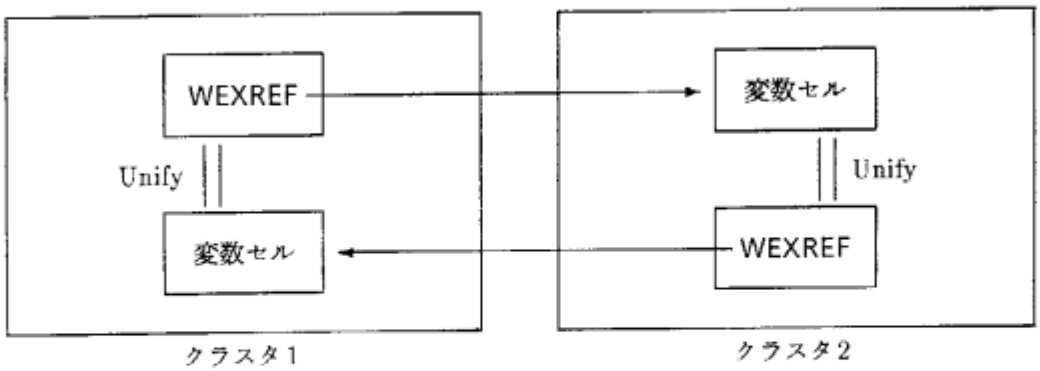


図 9–12: クラスタ間のループ構造

セルと輸出した変数とのユニフィケーションなので、外部参照セルの輸出元(クラスタ1)にユニフィケーションを依頼する。

しかし、Unsafe属性を持つ外部参照セルへのポインタが輸出されると、外部参照セルの属性だけではループ構造を作らないことを保証できない。外部参照セルの輸出は間接輸出と呼ばれ、これについては9.6.1で述べる。

クラスタ番号の大小の属性だけではクラスタ間のループ構造が出来てしまう例を図9–13に示す。

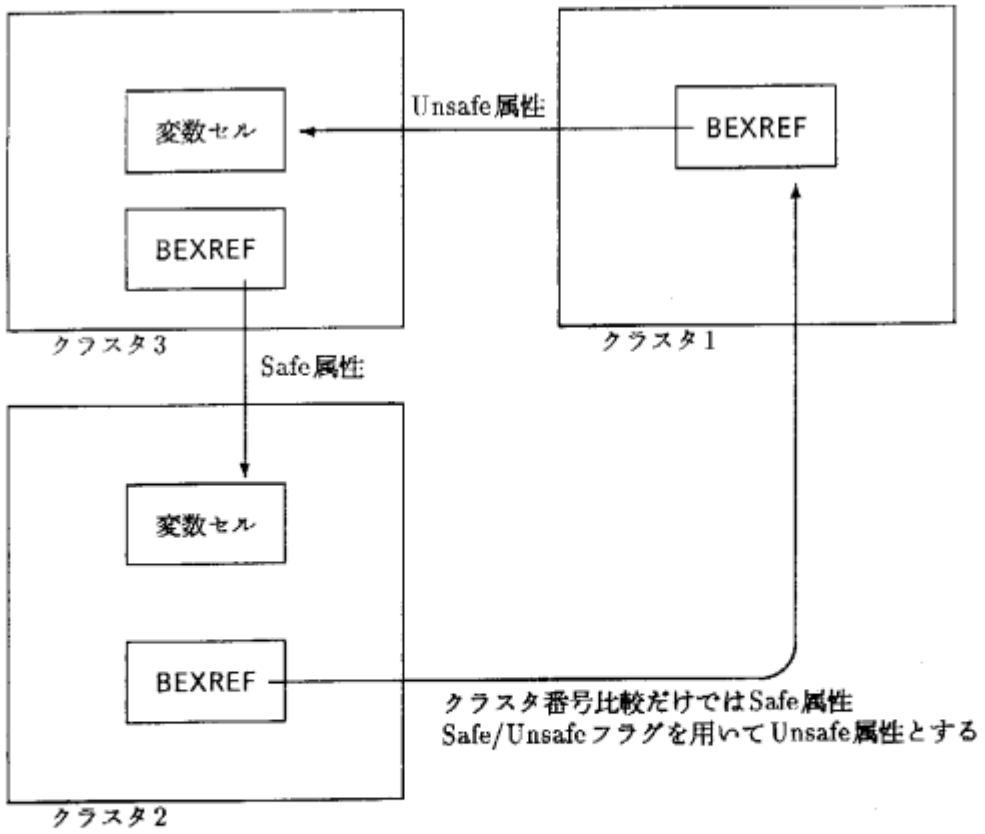


図 9–13: 間接輸出を含むループ構造の防止

クラスタ番号比較のSafe/Unsafe属性だけを用いると、図9–13の例のように、クラスタ2とクラスタ3

の両方で、変数セルから外部参照セルにユニファイパスが張れるので、クラスタ間のループを作ってしまう。

これは、クラスタ1でUnsafe属性の外部参照セルを間接輸出したためである。

そこで、Unsafe属性を持つ外部参照セルへのポインタが輸出される時には、間接輸出された外部参照IDのSafe/Unsafeフラグを ON にする。

ある外部参照セルに対して、クラスタ番号の比較による属性がSafeでも外部参照IDのSafe/Unsafeフラグが立っていたら、その外部参照セルの属性はUnsafeとする。

図9-13の例では、Unsafe属性の外部参照セルをクラスタ1からクラスタ2に間接輸出している外部参照のパスはUnsafe属性となり、クラスタ間のループ構造を防止できる。

9.1.7 構造体管理方式

コードモジュール等の大きなデータ構造は、1つのクラスタ内に複数回実体が輸入されることは望ましくない。コードモジュール等の大きなデータ構造にはシステム内で固有の構造体 ID を与えることにより、これらの構造のコピーを1クラスタ内に複数作られることを防止する。

(1) 構造体管理の方法

構造体表は構造体管理を行なうデータのアドレスと構造体IDの対応表であり、データのアドレスから構造体IDを得る操作、構造体IDからデータのアドレスを得る操作の両方が可能である。

構造体IDは1ワードデータで、タグはINT。、値部の上位8ビットにクラスタ番号、値部の下位24ビットにクラスタ内でのシリアル番号が入る。

構造体管理を行なうデータを輸出する時には、そのデータを構造体表に登録して構造体IDを得る。メッセージには輸出するデータの外部参照IDと構造体IDの両方を入れる。

構造体ID付きのデータを輸入した時の手順としては、

1. 構造体表を調べてその構造体IDをもつデータがクラスタに登録されているか調べる。
2. 構造体IDが構造体表に登録されていれば、構造体表からデータへのポインタを得た後に、メッセージに入れられている外部参照IDを輸出元クラスタに返す。
3. 構造体IDが構造体表に登録されていなければ、メッセージに入っている外部参照IDを輸入し、構造体IDと輸入して割り付けた外部参照セルへのポインタを構造体表に登録する。(この時点で構造体表にはデータへの外部参照ポインタが登録されているので、データ本体を輸入した時にはデータ本体へのポインタを構造体表に登録する必要がある。)また構造体管理を行なう対象となるデータは、輸出時に外部参照IDの構造体フラグを ON にする。

(2) 構造体表の構成

図9-14に構造体表の構成を示す。

構造体表は、構造体IDをキーとするハッシュサーチと、データのアドレスをキーとするハッシュサー

チの両方が可能である。

構造体表は**構造体レコード** と呼ばれる4ワードのセルから構成される。構造体レコードには構造体ID、データへのポインタ、構造体IDをキーとするハッシュリンク、データのアドレスをキーとするハッシュリンクが入れられる。

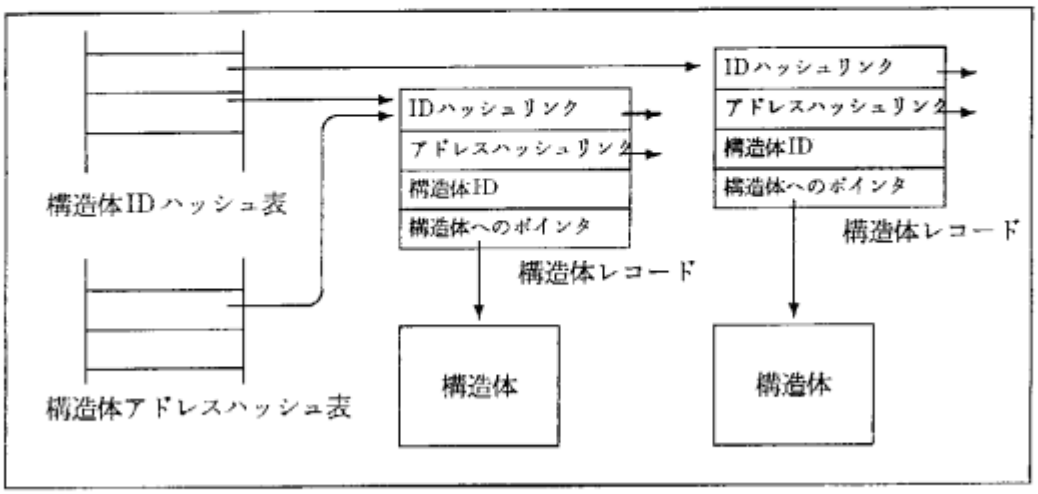


図 9–14: 構造体表の構成

9.2 各 KL1 データの輸出方式

あるKL1データを輸出する際は、そのデータへのポインタのみを輸出する場合と、データの実体を輸出する場合の2種類の形態がある。VPIMでは前者を**レベル0輸出**、後者を**レベル1輸出** と呼んでいる。以下に、各KL1データについての輸出方法を説明する。

(1) アトミックデータ

アトミックデータ(タイプがINT、ATOM、VECT0であるデータ)の輸出は、すべてレベル1輸出である。これらのデータを輸出する時は、そのデータの実体をそのまま送出し、輸出表には登録しない。

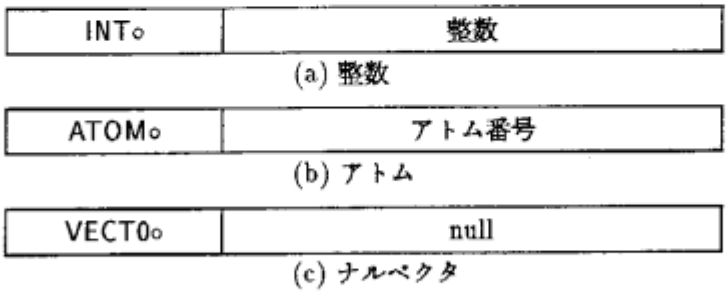


図 9–15: アトミックデータのメッセージ形式

(2) リスト

- レベル0輸出

  リストへのポインタを輸出表に登録して輸出する。

  外部参照IDのタグはWEXVAL。又はBEXVAL。とする。

- レベル1輸出

  タグがLIST。である1ワードをデータの最初につける。

  リストのcar部とcdr部をそれぞれレベル0輸出する。

  リストへのポインタが単一参照の場合consセルを解放する。

WEXVAL。	外部参照ID

(a) 白参照・レベル0輸出

BEXVAL。	外部参照ID
DNTC	WEC

(b) 黒参照・レベル0輸出

LIST。	null
car部のデータ(レベル0輸出)	
cdr部のデータ(レベル0輸出)	

(c) レベル1輸出

図 9–16: リストのメッセージ形式

(3) ショートベクタ

- レベル0輸出

  ベクタへのポインタを輸出表に登録して輸出する。

  外部参照IDのタグはWEXVAL。又はBEXVAL。とする。

- レベル1輸出

  ショートベクタのタグを示す1ワードをデータの最初につける。ベクタの各要素をそれぞれレベル0輸出する。

  ベクタへのポインタが単一参照の場合ベクタ本体を解放する。

(4) ロングベクタ

- レベル0輸出

  ベクタへのポインタを輸出表に登録して輸出する。

  外部参照IDのタグはWEXVAL。又はBEXVAL。とする。

WEXVAL∘	外部参照ID

(a) 白参照・レベル0輸出

BEXVAL∘	外部参照ID
DNTC	WEC

(b) 黒参照・レベル0輸出

VECT1∘ ～ VECT8∘	null
第0要素のデータ(レベル0輸出)	
・ ・ ・	
最終要素のデータ(レベル0輸出)	

(c) レベル1輸出

図 9-17: ショートベクタのメッセージ形式

- レベル1輸出

  VECT∘ タグの1ワードとベクタ長を示す1ワードをデータの最初につける。

  ベクタの各要素をそれぞれレベル0輸出する。

  ベクタへのポインタが単一参照の場合ベクタ本体を解放する。

(5) ストリング

- レベル0輸出

  ストリングへのポインタを輸出表に登録して輸出する。

- レベル1輸出 STRG∘ タグの1ワードをデータの最初につける。

  ストリングの全データを連続して送る。

(6) モジュール

- レベル0輸出

  モジュールを構造体表に登録して得られるモジュールの構造体ID、モジュールへのポインタを輸出表に登録して得られる外部参照IDとWECを送る。構造体IDのタグはMOD• とする。外部参照IDのタグはBEXVAL∘ とし、構造体フラグをONにする。

- レベル1輸出

  MOD∘ タグの1ワードをデータの最初につける。

  モジュールの各要素をレベル0輸出する。

WEXVAL○	外部参照ID

(a) 白参照・レベル0輸出

BEXVAL○	外部参照ID
DNTC	WEC

(b) 黒参照・レベル0輸出

VECT○	null
INT○	ベクタ長
第0要素のデータ(レベル0輸出)	
・ ・ ・	
最終要素のデータ(レベル0輸出)	

(c) レベル1輸出

図 9–18: ロングベクタのメッセージ形式

WEXVAL○	外部参照ID

(a) 白参照・レベル0輸出

BEXVAL○	外部参照ID
DNTC	WEC

(b) 黒参照・レベル0輸出

STRG○	null
CDESC○	ストリング長
ストリング本体	

(c) レベル1輸出

図 9–19: ストリングのメッセージ形式

MOD●	構造体ID
BEXVAL○	外部参照ID
DNTC	WEC

(a) レベル0輸出

MOD○	null
モジュール本体	

(b) レベル1輸出

図 9–20: モジュールのメッセージ形式

(7) コード

コードへのポインタの輸出にはレベル1輸出・レベル0輸出の区別はない。コードを輸出する際には以下の手順に従う。

1. コードのモジュールの先頭を指すポインタと、コードのモジュールの先頭からのオフセットを得る。
2. モジュールを構造体表に登録して構造体IDを得る。構造体IDのタグはCOD•とする。
3. モジュールを輸出表に登録して外部参照IDを得る。外部参照IDのタグはBEXVAL◦とし、構造体フラグをONにする。
4. 構造体ID、外部参照ID、WEC、コードのモジュールの先頭からのオフセットを送る。

COD•	モジュールの構造体ID
BEXVAL◦	モジュールの外部参照ID
DNTC	モジュールのWEC
DNTC	モジュール内オフセット

図 9-21: コードのメッセージ形式

(8) 変数

変数(タイプがUNDF、HOOK、MHOOKである変数セル)の輸出にはレベル1輸出・レベル0輸出の区別はない。

これらのデータを輸出する時は、変数セルのタイプを輸出済み変数タイプとし(UNDFをEUNDFに、HOOKをEHOOKに、MHOOKをEMHOKに)、変数へのポインタを輸出表に登録して輸出する。

外部参照IDのタグはWEXREF◦又はBEXREF◦とする。

(9) 輸出済み変数

輸出済み変数(タイプがEUNDF、EHOOK、EMHOK、RHOOKである変数セル)の輸出にはレベル1輸出・レベル0輸出の区別はない。

これらのデータを輸出する時は、変数へのポインタを輸出表に登録して輸出する。

外部参照IDのタグはWEXREF◦又はBEXREF◦とする。

(10) 外部参照セル

外部参照セルを輸出する時は、特殊な場合を除いて、外部参照セルが指す外部参照IDを輸出する。外部参照セルの輸出では通常、解放輸出方式と分割輸出方式の2つの輸出方式が使われる。

解放輸出方式は外部参照セルが明らかに単一参照である時に用いらる。この方式では、輸入レコードから外部参照IDとWECを獲得し、輸入レコードと外部参照セルは回収する。

WEXREFo	外部参照ID

(a) 白参照

BEXREFo	外部参照ID
DNTC	WEC

(b) 黒参照

図 9-22: 変数のメッセージ形式

分割輸出方式は外部参照セルが多重参照である可能性のある時に用いられる。この方式では、輸入レコードから外部参照IDと輸入レコード内のWECの半分を獲得し、輸入レコードと外部参照セルは回収しない。

通常、外部参照セルへの参照パスの数は表9-2で推定できる。

表 9-2: 外部参照セルへの参照パスの数

外部参照セルの種類	外部参照セルへのMRB	再輸入フラグ	外部参照セルへの参照パス
白外部参照セル	off	not exist	単一参照
白外部参照セル	on	not exist	多重参照
黒外部参照セル	off	off	単一参照
黒外部参照セル	off	on	多重参照
黒外部参照セル	on	don't care	多重参照
ReadHookセル	don't care	don't care	多重参照

しかし外部参照セルに対する輸出操作には例外的な操作が多く、上記の2つの輸出操作以外にも状況によって多くの輸出方式が採用されている。この輸出方式に関しては後で述べる。

9.3 各KL1データの輸入方式

前節の方式で輸出された各々のデータの輸入方式について述べる。

(1) アトミックデータの輸入

アトミックデータ(タイプがINT、ATOM、VECT0であるデータ)を受信した時は、輸入表には登録せずに、データをそのままクラスタ内に書き込む。

(2) 外部参照IDの輸入

白外部参照IDは白輸入表に、黒外部参照IDは黒輸入表に登録し、輸入レコードを指す外部参照セルを割り付ける。

受信した黒外部参照IDが既に黒輸入表に登録されていた時は、外部参照セルへのポインタを得る。

(3) リストのレベル 1 輸入

cons セルを 1 つ割り付け、輸入した car 部と cdr 部のデータを書き込む。

(4) ベクタのレベル 1 輸入

データの先頭のタグからベクタ長を得る。
必要な大きさのベクタを割り付け、輸入したベクタの各要素を書き込む。

(5) ストリングのレベル 1 輸入

データの先頭のディスクリプタからストリングの大きさを得る。
必要な大きさのストリングを割り付け、輸入したストリングの各要素を書き込む。

(6) モジュールのレベル 0 輸入

輸入した構造体 ID が構造体表に登録されていれば、モジュールへのポインタを得て、外部参照 ID を輸出元に返す。

輸入した構造体 ID が構造体表に登録されていなければ、メッセージ内の外部参照 ID を輸入表に登録し、外部参照セルを割り付ける。割り付けた外部参照セルのアドレスと構造体 ID を構造体表に登録する。

(7) モジュールのレベル 1 輸入

必要な大きさのモジュールを割り付け、輸入したモジュールの各要素を書き込む。

(8) コードの輸入

- 輸入した構造体 ID が構造体表に登録されていた時
  構造体 ID からモジュールへのポインタを得る。モジュールへのポインタとコードのモジュール内オフセットからコードへのポインタを得る。すでに、モジュールが自クラスタ内に存在するので、輸入表には登録しないで、メッセージ内の外部参照 ID と WEC をそのまま輸出元に返す。
- 輸入した構造体 ID が構造体表に登録されていない時
  メッセージ内の外部参照 ID を輸入表に登録し、外部参照セルを割り付ける。割り付けた外部参照セルのアドレスと構造体 ID を構造体表に登録する。
  コードとなる変数セルを 1 つ割り付ける。
  モジュールへのポインタ (この場合は外部参照セルへのポインタ) とコードのモジュール内オフセットから、コードへのポインタを得る D コードゴール

$$dcode\_read\_module(Mod, Ofst, Code) \ :- \ wait(Mod) \ | \\ module\_offset\_to\_code(Mod, Ofst, Code).$$

  をゴールスタックに入れる。

9.4 クラスタ間メッセージ送受信を伴う基本処理

ここではクラスタ間メッセージの送受信を発生させる原因となる基本処理について述べる。

9.4.1 ゴールを他クラスタに投げる方式

(1) ゴールの送信方式

ゴールを他クラスタに送信する時は%throw_goalを用いる。

%throw_goalには、ゴールの指すコードへのポインタ、ゴールの各引数をレベル0輸出情報が入れられる。%throw_goalの形式については(1)を参照のこと。またゴールを受けとったクラスタで新たに里親を生成する可能性があるので、荘園ID、WTC、資源、里親の最高最低プライオリティ、ゴールのプライオリティが%throw_goalに入れられる。

(2) ゴールの受信方式

%throw_goalを受信した時には、先ず荘園IDから里親へのポインタを得る。クラスタに里親がなければ各種プライオリティ情報と荘園IDを元に新たに里親を生成し、荘園に対して里親を生成した旨を示す%readyを出す。

続いて、コードへのポインタ、ゴールの各引数を輸入し、ゴールレコードを生成してゴールスタックに入れる。

但しこの時点ではコードへのポインタが直ちに得られるとは限らないので、実際にはDコードゴール

$$dcode\_apply(Code, Argv) \ :- \ true \quad | \quad apply(Code, Argv).$$

をゴールスタックに入れることになる。

(3) ゴールの送受信方式の例

次のKL1プログラムの例を考える。

$$test(X) \ :- \ true \ \mid \ dist(X)@cluster(2).$$

ゴール $test(X)$ はクラスタ1で実行され、呼び出された時点では引数 X は単一参照のUNDFセルとする。

図9-23はゴール *dist* がクラスタ1からクラスタ2に投げられる前の状態を示したものであり、図9-24はゴール *dist* がクラスタ1からクラスタ2に投げられた後の状態を示したものである。

図9-24では、ゴール *dist* のコードモジュールがクラスタ2内に既に存在している時の状態である。

ゴール *dist* のコードモジュールがクラスタ2内に存在していない状態では、9.3の方式にしたがってDコード *read_module* が実行された後に、Dコードの *apply* ゴールが実行される。

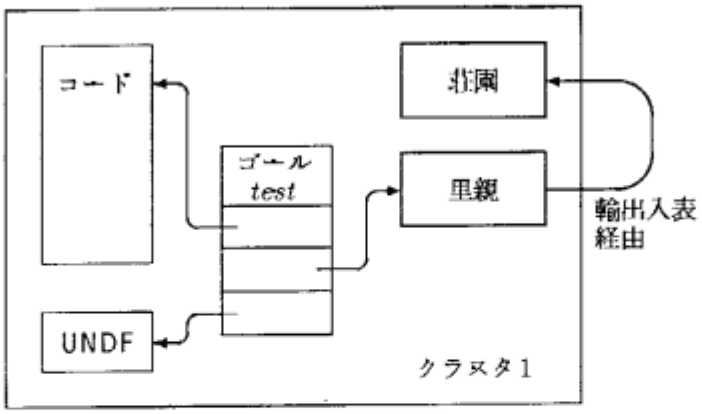


図 9-23: ゴールを他クラスタに分散する例 (1)

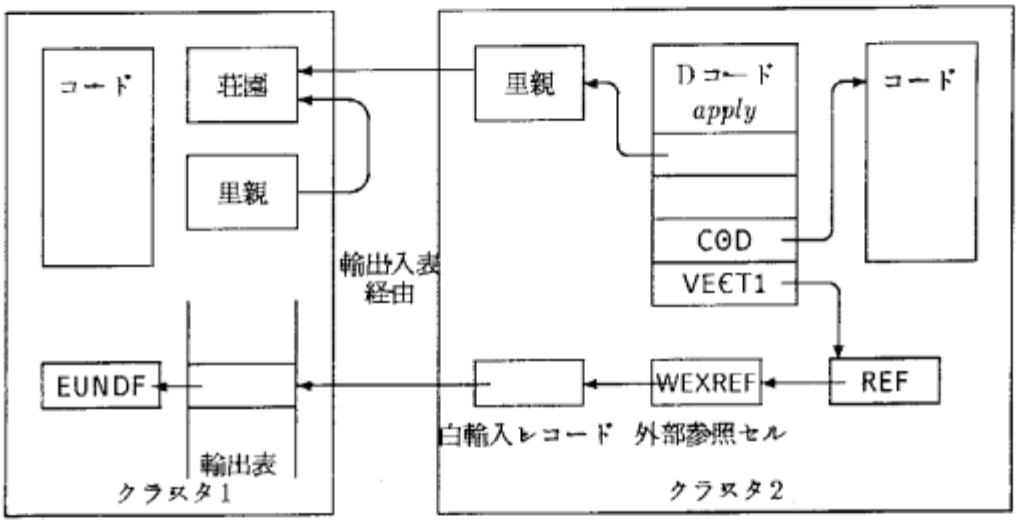


図 9-24: ゴールを他クラスタに分散する例 (2)

9.4.2 外部参照セルに対するパッシブユニフィケーション方式

ある節がサスペンドしてガード部で外部参照セルの値を待つ時は、実行中のゴールを外部参照セルにサスペンドさせて、外部参照セルの輸出元に対して輸出されているデータの読みだし要求メッセージ (%read) を送る。

%read を受けたクラスタでは、読みだし要求対象となっているデータが返信可能ならば、そのデータを返信メッセージ (%answer_value) の中に入れて送る。

%answer_value を受けたクラスタでは返信されたデータを輸入し、外部参照セルにサスペンドしているゴールをリジュームする。

外部参照セルに対するパッシブユニフィケーションの例として以下のプログラムを考える。

$$test : -true \mid generate(X), consume(X)@cluster(2).$$
$$consume(foo) : -true \mid true.$$

ゴール *test* はクラスタ1で実行され *consume*(X) がクラスタ2に投げられる。この時点でクラスタ1内の変数 X はUNDFセルであるとする。

consume(X) がクラスタ2で実行される時には X は外部参照となっているので X に対して%readを送り、ゴール *consume*(X) をサスペンドさせる。

クラスタ1では%readを受け、%answer_valueによって X の値をクラスタ2に返す。

クラスタ2では%answer_value によって X の値を受けとると、サスペンドしていたゴール *consume*(X) をリジュームする。

以下に、%readの送信、%readの受信、%answer_valueの送信 、%answer_valueの受信について各方式を述べる。

(1) %read の送信方式

① 外部参照セルが単一参照で単一待ちの場合

以下の手順に従う。(図9-25参照)また、形式については(3)を参照のこと。

1. 読みだし要求の返信先として白輸出表に外部参照セルへのポインタを登録し、返信先外部参照 ID を得る。
2. 輸入レコードから読み出す対象の外部参照IDとWECを獲得し、輸入レコードを回収する。
3. ゴールを外部参照セルにフックさせる。(外部参照セルのタイプはHOOKとする)
4. %readを送出する。

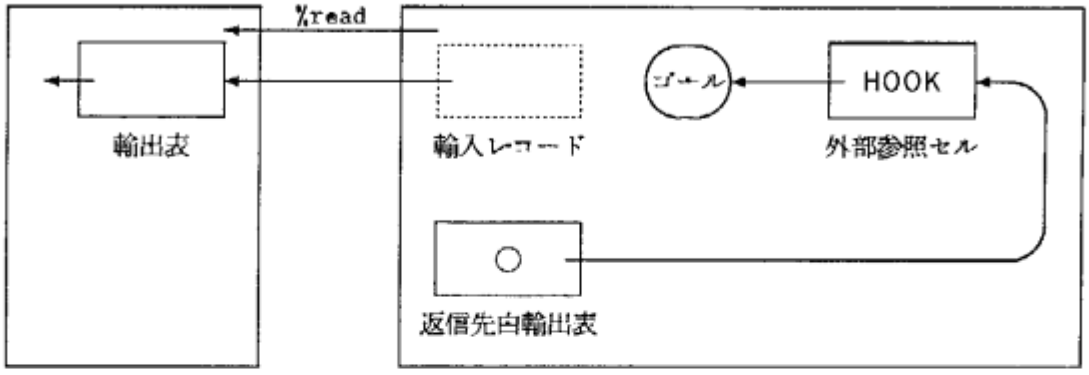


図 9-25: 単一参照の外部参照セルの %read の送出

② 外部参照セルが多重参照又は多重待ちの場合

外部参照セルに2回以上アクセスされる可能性があるので、輸入レコードの回収ができない。このために2ワードの ReadHook レコード を設けて、そこにゴールをフックさせる。

ReadHook レコードは通常外部参照セルと輸入表の間に挿入され、輸入レコードへのポインタ(タイプはReadHook レコードを指す外部参照セルのものと同じにする)とフックしたゴールへのポインタを保持

する。

ReadHook レコードを指す外部参照セルのタイプはRDHOKとする。

外部参照セルが多重参照の場合は、外部参照セルに対する排他制御を行なわねばならない。このため、外部参照セルをソフトロックする際にはタイプEXLOCKを使用する。

%readの送信方式は以下の手順に従う。(図9-26参照)

1. 外部参照セルにソフトロックを掛ける。(タイプをEXLOCKにする)
2. 読みだし要求の返信先として白輸出表に外部参照セルへのポインタを登録し、返信先外部参照 ID を得る。
3. ReadHook レコードを割り付け、輸入レコードへのポインタとフックするゴールへのポインタを入れる。
4. 輸入レコードから読み出す対象の外部参照ID と輸入表内のWECの半分を得る。輸入レコードは回収しない。
5. 外部参照セルがReadHook レコードを指すようにして、外部参照セルのソフトロックを外す。
6. %read を送出する。

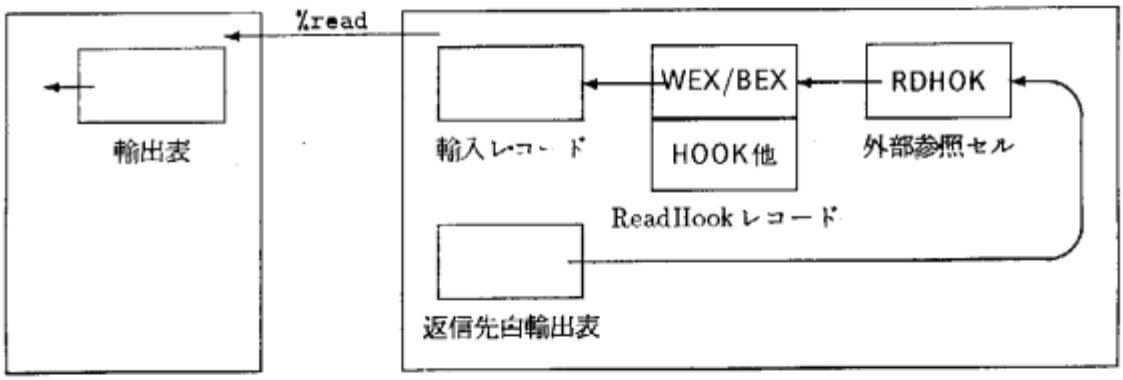


図 9-26: 多重参照又は多重待ちの外部参照セルの %read の送出

③ 外部参照セルが既に %read を送出している場合

この場合は、既存のReadHook レコードにゴールをサスペンドする。手順は以下に示す通り。

1. 外部参照セルにソフトロックを掛ける。
2. ReadHook レコードにゴールをフックさせる。
3. 外部参照セルのソフトロックを外す。

(2) %read の受信方式

%readを受信すると、読みだし対象の外部参照IDから返信するデータへのポインタを得る。

その時点での返信するデータの種類によって処理方式が異なる。

① 返信するデータの値が具体化している場合

返信するデータの値が具体化しているもの(アトム、整数、リスト、ベクタ、モジュール、コード)の時は、データを%answer_valueによって返信する。

%answer_valueの形式については(4)を参照のこと。

② 返信するデータが変数の場合

返信するデータが変数の時は **ReplyHook レコード** を用いてデータの返信操作をサスペンドする。ReplyHookレコードは3ワードのセルからなるデータ構造で、フックしている変数へのポインタ、ゴールリンク、データの返信先を指すポインタが入れられる。(図9-27参照)

データの返信操作のサスペンドは以下の手順で行なう。

1. 返信先外部参照IDから、その輸入表を指す外部参照セルへのポインタを得る。
2. ReplyHookレコードを割り付け、フックする変数へのポインタ、返信先を指すを指している外部参照セルへのポインタを入れる。
3. フックする変数に既にゴールがフックしていたら、そのゴールへのポインタをゴールリンクに入れる。
4. フックする変数に、ReplyHookレコードへのポインタを書き込む。(タイプはRHOOKとして排他的に書き込む)

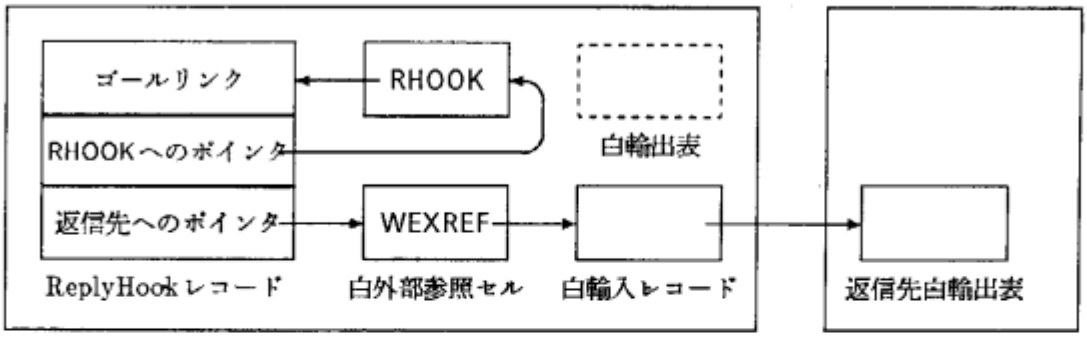


図 9-27: %answer_value の送信をサスペンドする処理

③ 返信するデータが外部参照セルの場合

返信するデータが外部参照セルの場合は、②と同じ方法でReplyHookレコードを作り、外部参照セルにサスペンドする。

外部参照セルが%readを送出していない場合は、9.4.2(1)の方式に従って%readが出される。

(3) %answer_value の受信方式

データ読みだし要求の返信として%answer_valueを受信した時は、以下の手順の処理を行なう。(図9-28参照)

1. 返信先の外部参照IDから%read要求を出している外部参照セルへのポインタを得る。
2. 返信されたデータを輸入する。
3. 返信されたデータを外部参照セルに書き込み、外部参照セルにフックしているゴール(又はReadHookセルにフックしているゴール)をリジュームする。
4. (もし存在すれば) ReadHookセルの指す輸入レコードを回収し、外部参照IDとWECを%releaseに入れて輸出元に返す。

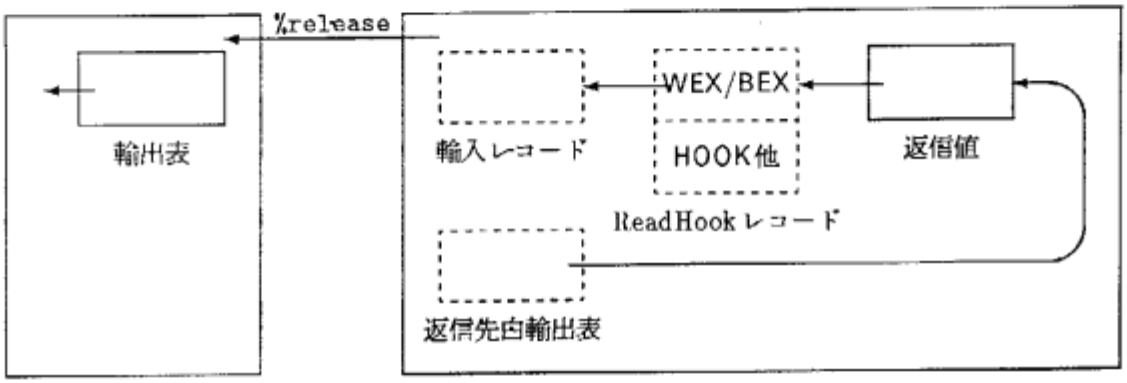


図 9-28: %answer_value の受信処理

9.4.3 外部参照セルに対するアクティブユニフィケーション方式

ボディ部で外部参照セルに対するユニフィケーションが生じた時は、殆んどの場合外部参照セルを輸出しているクラスタにユニフィケーションを依頼する%unifyを送出する。形式については(2)を参照のこと。

%unifyを送る要因となった外部参照セルを%unifyの引数1と呼ぶ。%unifyの引数1の外部参照セルが単一参照の場合は、外部参照セルと輸入レコードを回収して、輸入レコード内の外部参照IDとWECを%unifyに入れる。%unifyの引数1の外部参照セルが多重参照の場合は、輸入レコード内の外部参照IDとWECの半分を%unifyに入れる。

上記の外部参照セルとユニファイされる対象のデータを%unifyの引数2と呼ぶ。%unifyの引数2はレベル1輸出される。

%unifyを受信したクラスタではそのユニフィケーションを実行する上で、新しく里親が必要となる可能性があるので、ユニファイ送信時の荘園ID、WTC、プライオリティを%unifyに入れる。

外部参照セルのアクティブユニフィケーションに関して次の例を考える。

$test : -true \mid generate(X)@cluster(2), consume(X).$

$generate(X) : -true \mid X = foo.$

ゴール $test$ はクラスタ1で実行され $generate(X)$ がクラスタ2に投げられる。この時点でクラスタ1内の変数 X はUNDFセルであるとする。

$generate(X)$ がクラスタ2で実行される時は、外部参照となっている X とアトムの foo をアクティブユニファイするので、X を引数1とし、foo を引数2としてクラスタ1に%unifyを送る。

%unifyを受けとったクラスタ1では、X と foo のアクティブユニファイを行なう。

%unifyは引数1のタイプによって細かな処理が異なるので、以下に各々の%unifyの引数1のタイプによる%unifyの送出処理を述べる。

(1) 値の具体化したデータと外部参照セルのユニフィケーション

値が具体化しているデータ(アトム、整数、リスト、ベクタ、モジュール、コード)と外部参照セルのユニフィケーション時は、値の具体化しているデータを%unifyの引数と2してレベル1輸出する 。

(2) 輸出されていない変数と外部参照セルのユニフィケーション

変数セルと外部参照セルのユニフィケーション時には、単純に変数セルから外部参照セルにパスを張る。変数セルにゴールがフックしていたらそのゴールはゴールスタックに入れる。

(3) 輸出済み変数と外部参照セルのユニフィケーション

この場合はクラスタ間ループを作る可能性があるので、以下の細かいチェックに従う。

① 外部参照セルの輸出元の値が具体化している時

外部参照セルのタイプがWEXVAL、BEXVAL、RDHOK→WEXVAL、RDHOK→BEXVALの時は、外部参照セルの輸出元は値が具体化していてクラスタ間ループを作る可能性がないので、単純に変数セルから外部参照セルにパスを張る。変数セルにゴールがフックしていたらそのゴールはゴールスタックに入れる。

② 外部参照セルの輸出元の値が具体化していない時

外部参照セルのタイプがWEXREF、BEXREF、RDHOK→WEXREF、RDHOK→BEXREFの時の処理を以下に述べる。

外部参照セルの属性がUnsafeの時は、変数セルを%unifyの引数2として外部参照セルの輸出元に%unifyを送る。

外部参照セルの属性がSafeの時は、変数セルから外部参照セルにパスを張る。変数セルにゴールがフックしていたらそのゴールはゴールスタックに入れる。

(4) 外部参照セルと外部参照セルのユニフィケーション

一方の外部参照セルの輸出元に%unifyを送ってユニフィケーションを依頼する。%unifyの引数2は、もう一方の外部参照セルとする。

いずれかの外部参照セルのWECが不足している場合は、WEC補給方式を用いてWECを補給する。ユニフィケーション処理はWECが補給されるまでサスペンドする。

(5) ユニフィケーションメッセージ受信方式

ユニファイメッセージを受信したクラスタでは、先ず%unifyの2つの引数を輸入する。

%unifyの引数1と引数2が簡単にユニフィケーションできる組合せ(片方が変数でもう片方が値の具体化しているデータ)の時は、その場でユニフィケーションを行なう。

%unifyに付いてきた荘園IDの里親が存在する時は、%unify内のWTCをそのクラスタで保持し、里親が存在しない時は、WTCを荘園IDで示される荘園に戻す。

%unifyの引数1と引数2が簡単にユニフィケーション出来る組合せでない時は、ユニフィケーションを行なうDコードゴール

$$dcode\_unify(Arg1, Arg2) \ :- \ true \ \mid \ Arg1 \ = \ Arg2.$$

をゴールスタックに入れる。

Dコードゴールの里親は、%unifyに付いてきた荘園IDの里親とする。里親が存在しない時は新たに里親を生成し、この旨を%readyによって荘園に報告する。

9.4.4 外部参照セルに対する即時GC方式

外部参照セルに対して回収要求があった時は、外部参照セルが単一参照であれば、輸入表から外部参照IDとWECを取り出して、外部参照セルの輸出元に%releaseを送る。外部参照セルが多重参照であれば何もしない。

一括GC後に白輸入表と黒輸入表をスキャンし、使用されなくなった輸入表があれば、外部参照IDとWECを取り出して、外部参照セルの輸出元に%releaseを送る。

9.5 輸出入関連データ構造の排他制御方式

輸出入関連のデータ構造は、クラスタ内の各PEに共有されるため排他制御が必要である。以下にその方式について述べる。

9.5.1 白輸出表の排他方式

白輸出表のMRBが白から黒に変わる(WECが-2から-1に変わる)操作は排他制御が必要なので、白輸出表エントリにソフトロックを掛けてから行なう。

白輸出表エントリのソフトロック中は、エントリのタイプがSLOCKになる。

9.5.2 黒輸出表の排他方式

黒輸出表では、黒輸出ハッシュ表の各ハッシュエントリと黒輸出レコードの2箇所にソフトロックを掛ける。両者ともソフトロック中はセルのタイプがSLOCKになる。

(1) 輸出時の排他方式

1. 輸出されるデータのアドレスをキーとして黒輸出ハッシュ表エントリのアドレスを求め、そのハッシュエントリにソフトロックを掛ける。
2. 輸出するデータが黒輸出表に登録されているか、ハッシュチェーンを検索する。
3. 輸出するデータが既に黒輸出表に登録されていた場合は、黒輸出エントリにソフトロックを掛け、黒輸出表のWECを変化させた後、黒輸出エントリのソフトロックを外す。
4. 輸出するデータが未だ黒輸出表に登録されていなかった場合は、黒輸出表を新たに割り付け、WEC、輸出されるデータへのポインタをセットした後、新たに割り付けた黒輸出レコードを、ハッシュチェーンに挿入する。
5. 黒輸出ハッシュ表に掛けたソフトロックを外す。

(2) 輸出表参照時の排他方式

1. 参照対象の黒輸出エントリにソフトロックを掛ける。
2. 黒輸出レコードのWECを変更して0になった場合。
   - ◇ 黒輸出レコードの解放予約 をする。(輸出されたデータへのポインタのタイプをEOL等の一般には現れない値にする)
   - ◇ 参照対象の黒輸出エントリのソフトロックを外す 。
   - ◇ 黒輸出されたデータのアドレスをキーとして黒輸出ハッシュエントリのアドレスを求め、そのハッシュエントリにソフトロックを掛ける。
   - ◇ 削除する黒輸出レコードを検索する。(削除する黒輸出レコードへのポインタを持つレコードを検索する)
   - ◇ 黒輸出エントリと黒輸出レコードを回収する。
   - ◇ 黒輸出ハッシュエントリのソフトロックを外す。
3. 黒輸出レコードのWECを変更して0にならない場合。
   - ◇ 参照対象の黒輸出エントリのソフトロックを外す 。

黒輸出レコードの解放時に一旦解放予約することにより、黒輸出表への登録と解放が競合することがなくなる。

9.5.3 外部参照セルの排他方式

外部参照セルには%answer_valueにより値が書き込まれ、外部参照セルの指すReadHookレコードや輸入表が回収されることがあるので、外部参照セルへのアクセスには排他処理が必要である。

外部参照セルにはソフトロックを掛けることができ、ソフトロック中の外部参照セルのタイプはEXLOCKとなる。

9.5.4 白輸入表の排他方式

外部参照セルにロックを掛けていれば白輸入表にアクセスするPEは1つなので、白輸入レコード自体の排他制御は必要無い。

9.5.5 黒輸入表の排他方式

黒輸入表では、黒輸入ハッシュ表の各ハッシュエントリと黒輸入レコードにソフトロックを掛ける。両者ともソフトロック中はセルのタイプがSLOCKになる。

(1) 黒輸入時の排他方式

1. 外部参照IDをキーとして黒輸入ハッシュ表のハッシュアドレスを求め、ハッシュエントリにソフトロックを掛ける。
2. ハッシュチェーンを検索して、輸入した外部参照IDが登録されている黒輸入表を探す。
3. 外部参照IDが既に輸入されていた場合。
   - ◇ 黒輸入表にソフトロックを掛ける。
   - ◇ 黒輸入表内のWECと再輸入フラグを更新し、外部参照セルへのポインタを得る。
   - ◇ 黒輸入表のソフトロックを外す。
4. 外部参照IDが未だ輸入されていなかった場合。
   - ◇ 黒輸入表を一つ割り当て、WECを設定する。
   - ◇ 外部参照セルを一つ割り当て、黒輸入表のアドレスを入れる。
   - ◇ 黒輸入表に外部参照セルへのアドレスをセットし、黒輸入表をハッシュチェーンにつなぐ。
5. 黒輸入ハッシュ表に掛けたソフトロックを外す。

(2) 黒輸入表参照時の排他方式

1. 外部参照セルから指されている黒輸入表にソフトロックを掛ける。
2. 黒輸入表を回収する必要があるか調べる。(出すメッセージの種類、外部参照セルへの単一参照性等で決定する。)
3. 黒輸入表を回収する場合
   - ◇ 黒輸入表から外部参照IDとWECを取り出す。

◇ 黒輸入表を解放予約する。(外部参照IDをNULL値等の一般には現れない値にする) をする。

◇ 黒輸入表のソフトロックを外す。

◇ 外部参照IDをキーとして黒輸出ハッシュエントリのアドレスを求め、そのハッシュエントリにソフトロックを掛ける。

◇ 黒輸入表を回収する。

◇ 黒輸入ハッシュエントリのソフトロックを外す。

◇ 外部参照セルのソフトロックを外して、外部参照セルを回収する。

4. 黒輸入表を回収しない場合

◇ 黒輸入表から外部参照IDと必要なWECを取り出す。

◇ 黒輸入表のソフトロックを外す。

◇ 外部参照セルのソフトロックを外す。

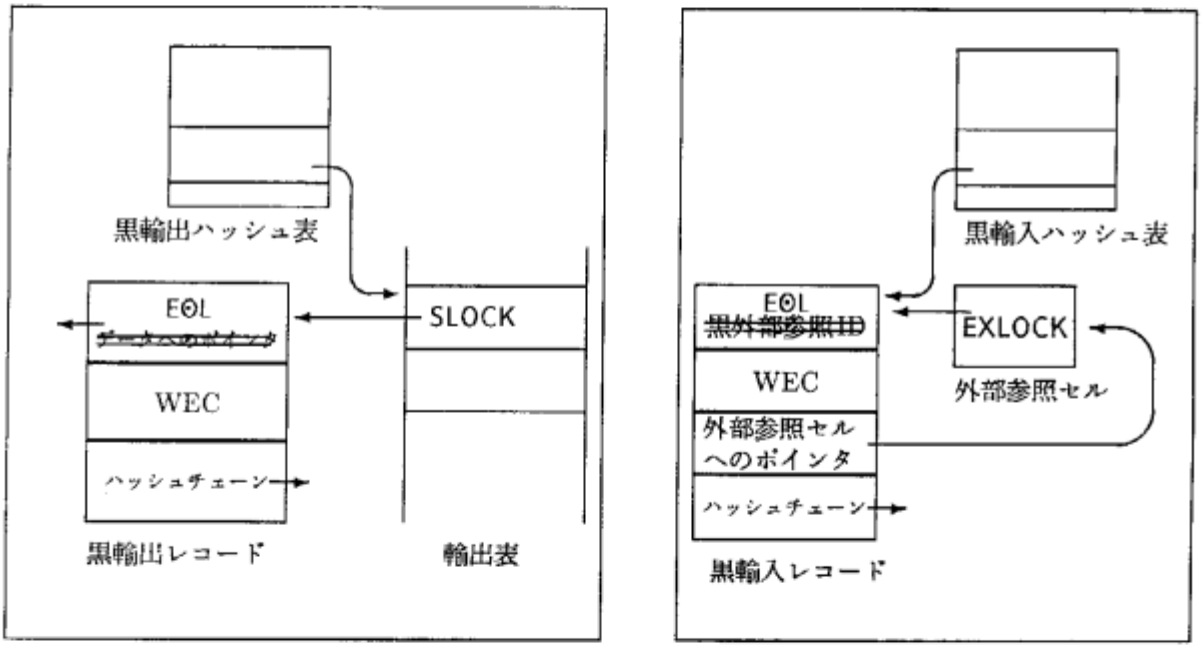


図 9-29: 黒輸出レコード黒輸入レコードの解放予約

9.5.6 構造体表の排他方式

構造体表は、構造体表に1つ付いているロックフラグをソフトロックすることにより排他を実現する。

9.6 輸入表の WEC が不足した時の方式

輸入レコード内のWECが1になりそれ以上分割できなくなった状態(図9-30)で、分割輸出の必要性が生じた時は、**間接輸出方式** 又は **WEC 補給方式** を用いる。

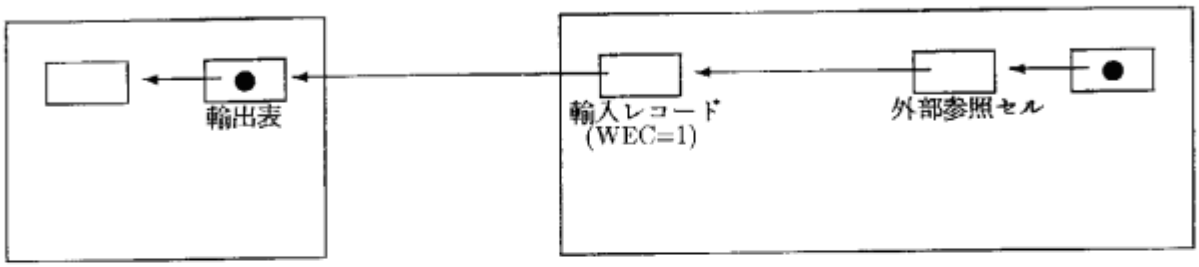


図 9-30: WEC=1 となった状態

9.6.1 間接輸出方式

輸入表レコード内のWECは変化させず、外部参照セルへのポインタを輸出する方式を間接輸出方式と呼ぶ。(図9-31この方式は、データを%throw_goalの引数や%unifyの引数2などでリファレンスを伸ばして輸出する時に用いる。

白輸入レコードに対しては、WEC=1とWEC=2を区別せずに、%throw_goalの引数や%unifyの引数2として分割輸出の必要性が生じた時(黒パスからアクセスのあった時)は、間接輸出することとしている。

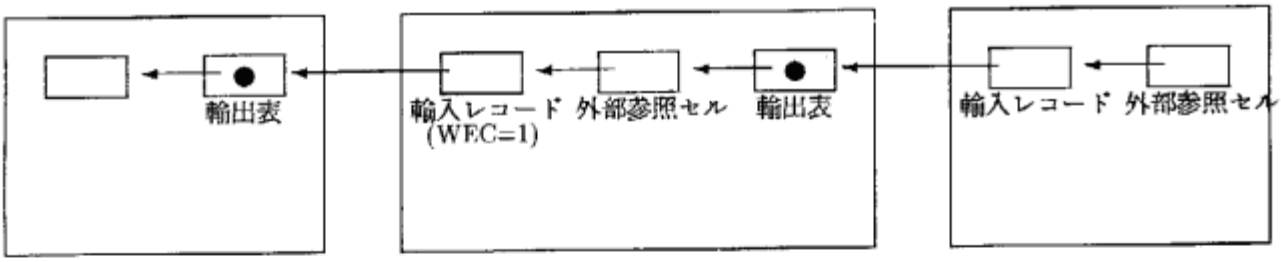


図 9-31: 間接輸出方式

9.6.2 WEC 補給方式

輸入レコード内の外部参照IDの輸出元にWECの補給要求を出す方式をWEC補給方式と呼ぶ。この方式は、データを%readの読みだし対象や%unifyの引数1などで外部参照のリファレンスをたどる方向に輸出する時に用いる。

WECの補給要求を出す%request_BEXIDと、WECの補給を行なう%supply_BEXIDを導入した。

%request_BEXIDを出してから、%supply_BEXIDが到着するまでの間は、その外部参照セルに対するメッセージ送出はサスペンドする。

(1) %request_BEXID の送信方式

%request_BEXIDは以下の手順で送信する。(図9-32)形式については(6)を参照のこと。

1. 外部参照セルの先にReadHookレコードを割り付け、ゴール(%readの送信をサスペンドする時)又はユニフィケーションのDコード(%unifyの送信をサスペンドする時)をフックさせる。また、外部参照セルの先にReadHookレコードがつながっていた時はそれを再利用する。
2. 白輸出レコードを1つ割り付け、%request_BEXIDの返信先とする。
3. 輸入レコード内から外部参照IDとWECを取り出し、輸入レコードを回収する。
4. ReadHookレコードの輸入レコードへのポインタはNULL値とし、外部参照IDのSafe/Unsafeフラグのビットを MRB としてセットする。
5. 外部参照ID、WEC、返信先外部参照IDの入った%request_BEXIDを送出する。

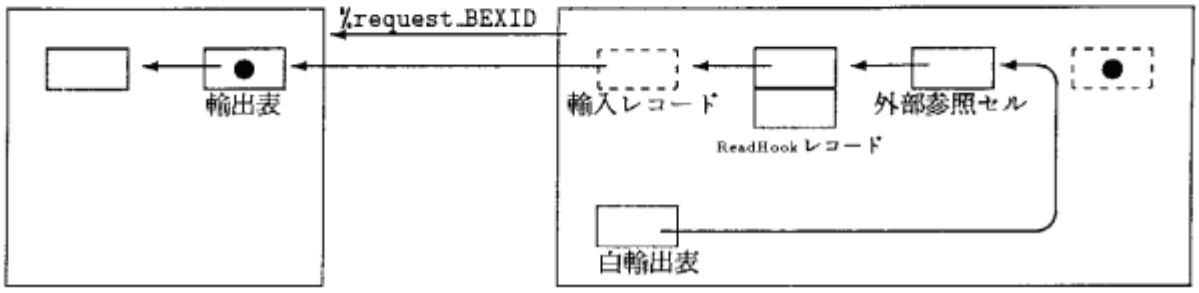


図 9-32: %request_BEXID の送信方式

(2) %request_BEXID の受信方式、%supply_BEXID の送信方式

%request_BEXIDは以下の手順で受信し、%supply_BEXIDを送出する。(図9-33)
%supply_BEXIDの形式については(7)を参照のこと。

1. %request_BEXID内の外部参照IDから輸出されたデータへのポインタを得る。
2. このデータを多重参照で再輸出して黒外部参照IDとWECを得る。
3. 返信先外部参照ID、黒外部参照ID、WECの入った%supply_BEXIDを送る。

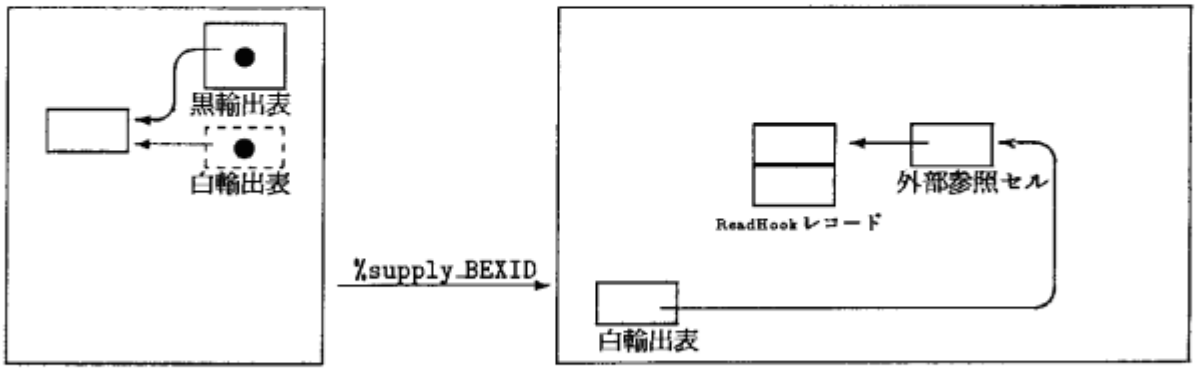


図 9-33: %supply_BEXID の送信方式

(3) %supply_BEXID の受信方式

%supply_BEXIDは以下の手順で受信する。(図9-34)

1. 返信先外部参照IDからWEC要求を出している外部参照セルへのポインタを得る。また、黒外部参照IDとWECを輸入して、WECの補給された外部参照セルへのポインタを得る。
2. WEC要求を出している外部参照セルが具体化した値の場合(%request_BEXIDの他に%readも送っていて、%answer_valueを先に受信している時)は、何もせずに処理を終る。
3. WEC要求を出している外部参照セルのタイプがRDHOKの時は、そこにWECの補給された外部参照セルへのポインタを書き込む。
4. ReadHookレコードにつながっているゴール群をリジュームする。

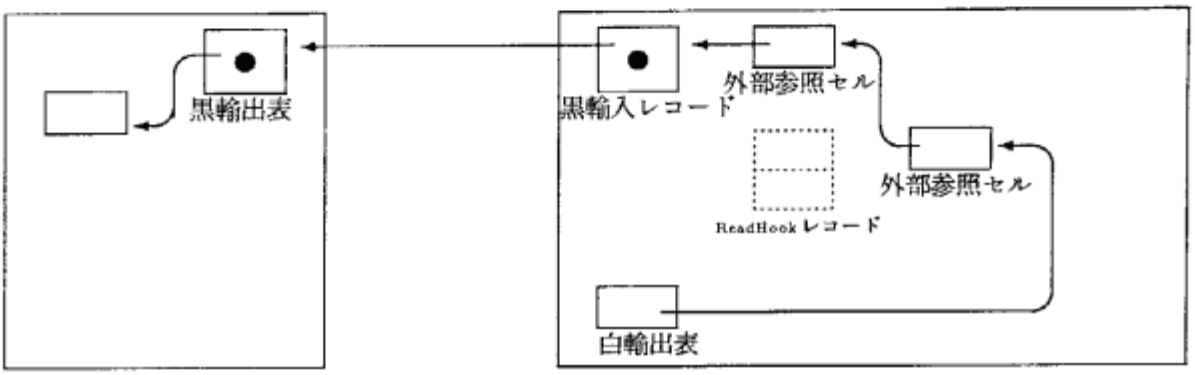


図 9-34: %supply_BEXID の受信方式

9.7 メッセージ送受信処理方式

VPIMにおけるクラスタ間メッセージ通信は、パケットと呼ばれる1続きのデータを単位として行う。1つのメッセージは、通常1つのパケットで通信することができるが、パケットの長さがネットワーク用のバッファ容量を越える場合、メッセージを分割して送受信しなければならない。これに対応するため、マルチパケット方式をサポートしている。

メッセージは、いったんメモリ上に論理パケット と呼ばれる形式で構成する。論理パケットはメッセージ種別記号、メッセージの内容を表すデータ、終了を表すデータから成る。(VPIMで用意された各メッセージの形式については9.8で述べる。)そして論理パケットに、ネットワークを通すのに必要な情報を付けて、物理パケット と呼ばれる形にして通信を行なう。図9-35は、メッセージの送受信の様子を簡単に表したものである。

(1) メッセージを論理パケットとして組み立てる

KL1の処理中にクラスタ間で通信が必要になった場合、その場でメッセージを送信する。メッセージ送信処理に入ったら、まず、そのメッセージによって構成される論理パケットの長さを概算する。この

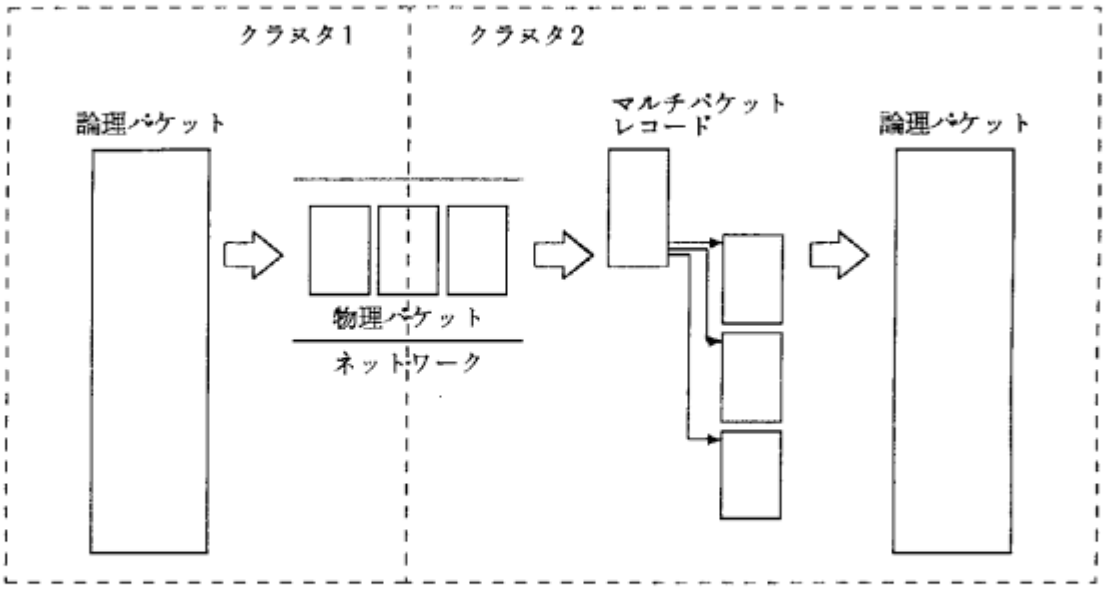


図 9-35: メッセージ送受信方式

概算値は論理パケットが一番長くなった時のことを想定して計算される。メッセージによって送りたいKL1データが何であるかを逐一調べて計算すれば、正確なパケット長が求まるが、概算値を使用するのは、計算にかかるコストが無視できないためである。このパケット長の概算値をもとにメモリを確保し[2]、そこにメッセージの本体部分を構成する。このメモリ上に構成したメッセージの本体部分が論理パケットである。

(2) 論理パケットを物理パケットにしてネットワークに送り出す

論理パケットに制御情報を付けて、ネットワークのバッファに書き込む。ネットワーク中に存在するパケットを物理パケットと呼ぶ。この時には、実際の論理パケットの長さが分かるので、ここでマルチパケットになるかどうかの判断をする。マルチパケットになる場合は、各物理パケット毎に制御情報をつけて送り出す。

1つのメッセージを構成するパケットの形式には、

- 単独の物理パケットで1メッセージが完結するもの (**シングルパケット**)
- 複数の物理パケットで1メッセージを構成するもの (**マルチパケット**)

の2種類がある。図9-36は、それぞれの物理パケットの形式を示したものである。ただし、VPIMでは、基本的に1データ1ワードである。

- 送信クラスタ番号

  メッセージを送り出す側のクラスタ番号。

- 受信クラスタ番号

[2] もしヒープメモリの残量が不足していてメモリが確保できなかった場合は、メッセージ送信処理の内容をシステム固定領域に保存してGC要求を出す必要がある。現在この部分は検討中である。

メッセージを受けとる側のクラスタ番号。

- パケット長

  物理パケットのワード数。(送信クラスタ番号からEnd Of Messageまで)

  タグ部は、パケットがマルチパケットかどうかの判断に使用する。

  シングルパケット(INT○)、マルチパケット(INT●)

- パケットID

  物理パケットの識別番号。他メッセージとの区別は、メッセージ本体の中に書かれたメッセージ種別記号を見て判断するが、マルチパケットの場合は、同じ種別のメッセージが混ざって到着する可能性があるので、このパケットIDを見て区別する。

- 総パケット数

  メッセージを構成する物理パケットの総数。

- パケット番号

  マルチパケットにおいて、この物理パケットが何番目のものであるかを示す番号。

- End Of Message

  パケットの終了を示すデータ。

DNTC	送信クラスタ番号
DNTC	受信クラスタ番号
INT○	パケット長
メッセージ本体	
End Of Message	

(a) シングルパケット

DNTC	送信クラスタ番号
DNTC	受信クラスタ番号
INT●	パケット長
DNTC	パケットID
DNTC	総パケット数
DNTC	パケット番号
メッセージ本体	
End Of Message	

(b) マルチパケット

図 9–36: 物理パケットの形式

(3) 物理パケットをネットワークから読み込む

メッセージの受信は、スリットチェックで、『パケット到着』というイベントを受け取った時点で行なう。したがって、ネットワークのバッファに溜っているパケット全てを1回のメッセージ受信処理で行なう。

メッセージ受信処理に入ると、まず、到着した物理パケットの制御情報を読みとりマルチパケットかどうかのチェックを行なう。シングルパケットであれば、メッセージ本体の内容をメモリ上に論理パケットとして構成し[3]、そのままメッセージのディスパッチャに渡す。

マルチパケットの場合は、マルチパケットレコードの検索を行なう。マルチパケットレコードは、分割して送信された物理パケットがメッセージとして完成するまでの間、各物理パケットをつないでおくレコードである。その構成を図9-37に示す。到着したパケットに対応するレコードが見つかれば、メッセージ本体の内容をメモリ上に展開し、レコードにつなぐ。レコードが見つからなければ、新たにマルチパケットレコードを作って、レコードのリンクにつなぎ、同様の操作を行なう。

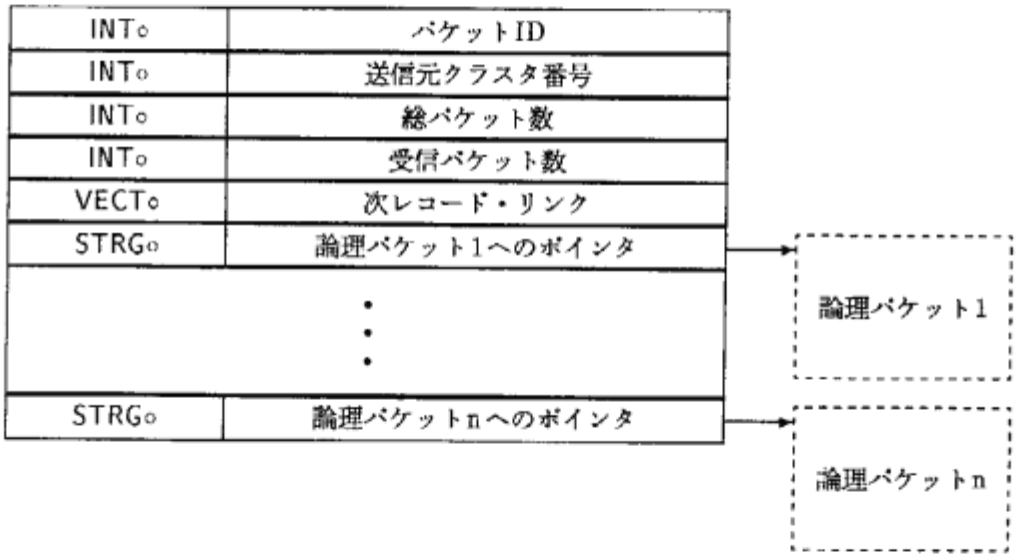


図 9-37: マルチパケットレコードの構成

- パケットID

  パケットの識別番号。

- 送信元クラスタ番号

  このパケットを送信したクラスタの番号。パケットIDと送信元クラスタ番号で他のパケットとの区別をつける。

- 総パケット数

  全部でいくつパケットがあるかを示す値。

[3] もしヒープメモリの残量が不足していてメモリが確保できなかった場合は、メッセージ受信処理の内容をシステム固定領域に保存してGC要求を出す必要がある。現在この部分は検討中である。

- 受信パケット数
  これまでにいくつパケットを受け取ったかをカウントしておく。このカウンタと総パケット数が一致した時に、全てのパケットが到着したと判断する。
- 次レコード・リンク
  他のパケットと混ざって到着する可能性があるので、マルチパケットレコードは複数つなげられるようにしておく。
- 論理パケットポインタ
  到着したパケットのメッセージ本体部分をメモリ上に展開したものを論理パケットと呼ぶ。この論理パケットをつないでおくスロットである。論理パケットは単なるデータの集まりなのでストリング(タイプがSTRG)として扱う。
  また、このスロットはパケットの個数分確保される。

(4) **マルチパケットからメッセージを構成する**

受信したパケット数のカウンタとトータル・パケット数が一致した時に、全パケットが到着したと判断する。全パケットが到着したと分かったら、マルチパケットレコードにつながれたパケットを一まとめにしてメモリ上に構成し[4]、そのポインタをメッセージのディスパッチャに渡し、マルチパケットレコードを回収する。

9.8 各種メッセージの形式

現在 VPIM で用意したメッセージは全部で24種類である。以下では、これらのメッセージの形式を紹介する。

(1) %throw_goal

%throw_goal はクラスタ間の負荷を分散させるために、指定された位置でゴールを他のクラスタに投げる時に送信される。図9-38は%throw_goal の形式を示しており、各スロットの内容は次の通りである。

- THROW_GOAL
  メッセージが%throw_goal であることを示す種別記号
- 荘園ID
  ゴールが所属する荘園の外部参照ID
- WTC
  メッセージに付けられるWTC

[4] もしヒープメモリの残量が不足していてメモリが確保できなかった場合は、メッセージ受信処理の内容をシステム固定領域に保存してGC要求を出す必要がある。現在この部分は検討中である。

- 荘園のプライオリティの下限

  ゴールが所属する荘園のプライオリティの下限値

- 荘園のプライオリティの上限

  ゴールが所属する荘園のプライオリティの上限値

- 資源

  ゴールに対して与えられる資源

- ゴールのプライオリティ

  ゴールが持っているプライオリティ

- モジュールの構造体ID

  ゴールを実行するコードが入っているモジュールの構造体ID

- モジュールの外部参照ID

  ゴールを実行するコードが入っているモジュールの外部参照ID

- WEC

  モジュールの外部参照IDに付けられるWEC

- モジュール内オフセット

  コードのモジュール先頭からのオフセット

- ゴールの引数個数

  ゴールの持っている引数の数

- ゴールの引数

  ゴールの各引数で第1から第n引数までのレベル0輸出

(2) %unify

ユニファイの引数において片方もしくは両方が外部参照であって、自分のクラスタ内でユニファイができない場合は、それを他のクラスタに依頼する。この時送信されるのが%unifyである。図9-39は%unifyの形式を示しており、各スロットの内容は次の通りである。

- UNIFY

  メッセージが%unifyであることを示す種別記号

- 荘園ID

  ユニファイを実行したゴールが所属する荘園の外部参照ID

- WTC

  メッセージに付けられるWTC

- カレントプライオリティ

  ユニファイを実行したゴールのプライオリティ

DNTC	THROW_GOAL
INTo/•	荘園ID
DNTC	WTC
DNTC	荘園のプライオリティの下限
DNTC	荘園のプライオリティの上限
DNTC	資源
DNTC	ゴールのプライオリティ
COD•	モジュールの構造体ID
BEXVALo	モジュールの外部参照ID
DNTC	WEC
DNTC	モジュール内オフセット
DNTC	ゴールの引数個数
ゴールの第1引数	
ゴールの第2引数	
• • •	
End of Message	

図 9-38: %throw_goal の形式

DNTC	UNIFY
DNTC	荘園ID
DNTC	WTC
DNTC	カレントプライオリティ
ユニファイの第1引数	
ユニファイの第2引数	
End of Message	

図 9-39: %unify の形式

- ユニファイの第1引数

  ユニファイの第1引数であり、外部参照データである。

- ユニファイの第2引数

  ユニファイの第2引数で、レベル1輸出

DNTC	READ
WEXREF/VALo	読み出すセルの外部参照ID
WEXREFo	返信先の外部参照ID
End of Message	

図 9–40: 白外部参照セルに対する %read の形式

(3) %read

%read は外部参照セルに対して、その値を他のクラスタに要求する時に送信される。白外部参照と黒外部参照の時では、そのメッセージの内容は異なる。

① 白外部参照セルに対する %read

白外部参照セルに対して、その値を要求する時に送信される。図9–40は、白外部参照セルに対する %readの形式を示しており、各スロットの内容は次の通りである。

- READ

  メッセージが%readであることを示す種別記号

- 読み出すセルの外部参照ID

  読み出し要求をするセルの外部参照 ID で、この外部参照 ID のタイプが WEXREF もしくは WEXVALであることから白外部参照セルの読み出しと判断する。

- 返信先の外部参照ID

  読み出されたデータを返信してもらう先の外部参照ID

② 黒外部参照セルに対する %read

黒外部参照セルに対して、その値を要求する時に送信される。図9–41は、黒外部参照セルに対する %readの形式を示しており、各スロットの内容は次の通りである。

- READ

  メッセージが%readであることを示す種別記号

- 読み出すセルの外部参照ID

  読み出し要求をするセルの外部参照IDで、この外部参照IDのタイプがBEXREFもしくはBEXVALであることから黒外部参照セルの読み出しと判断する。

- WEC

  外部参照セルに付けられるWEC

- 返信先の外部参照ID

  読み出されたデータを返信してもらう先の外部参照ID

DNTC	READ
BEXREF/VALo	読み出すセルの外部参照ID
DNTC	WEC
WEXREFo	返信先の外部参照ID
End of Message	

図 9-41: 黒外部参照セルに対する %read の論理パケット形式

DNTC	ANSWER_VALUE
WEXREF/VALo	返信先の外部参照ID
返信データ	
End of Message	

図 9-42: %answer_value の論理パケット形式

(4) %answer_value

読み出し要求のあったデータを返信する時に送信される。図9 42は、多重参照データに対する %answer_value の形式を示しており、各スロットの内容は次の通りである。

- ANSWER_VALUE
  メッセージが%answer_valueであることを示す種別記号
- 返信先の外部参照ID
  返信データを返すセルの外部参照ID
- 返信データ
  読み出し要求のあったデータ(レベル1輸出)

(5) %release

外部参照セルへの参照がなくなり、クラスタ間の輸出入関係が不要になった時、データを輸出しているクラスタに対して輸出表の解放を依頼する。この時送信されるのが%releaseである。

① 白輸出データの解放

白輸出データの解放を依頼する時に送信される。図9-43は、白輸出データに対する%releaseの形式を示しており、各スロットの内容は次の通りである。

- RELEASE
  メッセージが%releaseであることを示す種別記号
- 解放データの外部参照ID
  解放要求を出すデータの外部参照IDで、タイプがWEXREFもしくはWEXVALであることから白輸出データであると判断する。

DNTC	RELEASE
WEXREF/VALo	解放データの外部参照ID
End of Message	

図 9-43: 白輸出データ解放の %release の論理パケット形式

DNTC	RELEASE
WEXREF/VALo	解放データの外部参照ID
DNTC	WEC
End of Message	

図 9-44: 黒輸出データ解放の %release の論理パケット形式

② 黒輸出データの解放

黒輸出データの解放を依頼する時に送信される。図9-44は、黒輸出データに対する %releaseの形式を示しており、各スロットの内容は次の通りである。

- RELEASE

  メッセージが%releaseであることを示す種別記号

- 解放データの外部参照ID

  解放要求を出すデータの外部参照IDで、タイプがBEXREFもしくはBEXVALであることから黒輸出データであると判断する。

- WEC

  輸出元に返却するWEC

(6) %request_BEXID

白外部参照セルに対する参照が複数になった場合や、黒外部参照セルに対する輸入表のWECが不足した場合に、新しく黒外部参照関係を作る時に%request_BEXIDは送信される。

① 白外部参照IDから黒外部参照IDへの変換要求

白外部参照セルに対する参照が複数になった場合、黒外部参照に切替を要求する時に送信される。図9-45は、この時の%request_BEXIDの形式を示しており、各スロットの内容は次の通りである。

- REQUEST_BEXID

  メッセージが%request_BEXIDであることを示す種別記号

- データの外部参照ID

  黒外部参照に切替を要求するデータの外部参照ID

- 返信先の外部参照ID

  新しい黒外部参照IDを受けるために用意した外部参照ID

DNTC	REQUEST_BEXID
WEXREF/VALo	データの外部参照ID
WEXREFo	返信先の外部参照ID
End of Message	

図 9 45: 白外部参照 ID に対する %request_BEXID の論理パケット形式

DNTC	REQUEST_BEXID
BEXREF/VALo	データの外部参照ID
DNTC	WEC
WEXREFo	返信先の外部参照ID
End of Message	

図 9 46: 黒外部参照 ID に対する %request_BEXID の論理パケット形式

② 黒外部参照 ID の WEC 要求

黒外部参照セルに対する輸入表の WEC が不足した場合、新しい黒外部参照に切替を要求する時に送信される。図9–46は、この時の%request_BEXIDの形式を示しており、各スロットの内容は次の通りである。

- REQUEST_BEXID
  メッセージが%request_BEXIDであることを示す種別記号
- データの外部参照ID
  WECが不足してしまったデータの外部参照ID
- WEC
  残っていたWEC
- 返信先の外部参照ID
  新しい黒外部参照IDを受けるために用意した外部参照ID

(7) %supply_BEXID

%supply_BEXIDは%request_BEXIDに対して、黒外部参照IDを準備して要求のあったクラスタにそれを渡す時に送信される。図9–47は、%supply_BEXIDの形式を示しており、各スロットの内容は次の通りである。

- SUPPLY_BEXID
  このメッセージが%supply_BEXIDであることを示す種別記号
- 返信先の外部参照ID
  準備した黒外部参照IDを返す先のセルに対する外部参照ID
- 補給される黒外部参照ID

DNTC	SUPPLY_BEXID
BEXREF/VALo	返信先の外部参照ID
BEXREFo	補給される黒外部参照ID
DNTC	WEC
End of Message	

図 9-47: %supply_BEXID の形式

DNTC	START
DNTC	荘園ID
DNTC	WTC
End of Message	

図 9-48: %start の形式

準備した黒外部参照ID

- WEC

  準備した黒外部参照IDに付けられるWEC

(8) %start

%startは荘園から里親に対して、(再)起動を指示する時に送信される。図9-48は、%startの形式を示しており、各スロットの内容は次の通りである。

- START

  メッセージが%startであることを示す種別記号

- 荘園ID

  荘園の外部参照ID

- WTC

  メッセージに付けられるWTC

(9) %stop

%stopは荘園から里親に対して、動作停止を指示する時に送信される。図9-49は、%stopの形式を示しており、各スロットの内容は次の通りである。

- STOP

  メッセージが%stopであることを示す種別記号

- 荘園ID

  荘園の外部参照ID

DNTC	STOP
DNTC	荘園ID
DNTC	WTC
End of Message	

図 9–49: %stop の形式

DNTC	ABORT
DNTC	荘園ID
DNTC	WTC
End of Message	

図 9–50: %abort の形式

- WTC
  メッセージに付けられる WTC

(10) %abort

%abort は荘園から里親に対して、実行放棄を指示する時に送信される。図9–50は、%abort の形式を示しており、各スロットの内容は次の通りである。

- ABORT
  メッセージが%abort であることを示す種別記号
- 荘園ID
  荘園の外部参照ID
- WTC
  メッセージに付けられる WTC

(11) %supply_resource

%supply_resource は荘園から里親に対して、要求された資源を供給する時に送信される。図9–51は、%supply_resource の形式を示しており、各スロットの内容は次の通りである。

- SUPPLY_RESOURCE
  メッセージが%supply_resource であることを示す種別記号
- 荘園ID
  荘園の外部参照ID
- WTC
  メッセージに付けられる WTC

DNTC	SUPPLY_RESOURCE
DNTC	荘園ID
DNTC	WTC
End of Message	

図 9-51: %supply_resource の形式

DNTC	ASK_STATISTICS
DNTC	荘園ID
DNTC	WTC
End of Message	

図 9-52: %ask_statistics の形式

(12) %ask_statistics

%ask_statistics は荘園から里親に対して、統計情報を要求する時に送信される。図9-52は、%ask_statisticsの形式を示しており、各スロットの内容は次の通りである。

- ASK_STATISTICS

  メッセージが%ask_statisticsであることを示す種別記号

- 荘園ID

  荘園の外部参照ID

- WTC

  メッセージに付けられるWTC

(13) %supply_wtc

%supply_wtc は荘園から里親に対して、要求された資源を供給する時に送信される。図9-53は、%supply_wtcの形式を示しており、各スロットの内容は次の通りである。

- SUPPLY_WTC

  このメッセージが%supply_wtcであることを示す種別記号

- 荘園ID

  荘園の外部参照ID

- WTC

  要求のあった里親に対して供給されるWTC

(14) %terminated

%terminatedは里親から荘園に対して、里親の処理が全て終了したことを知らせる時に送信される。

DNTC	SUPPLY_WTC
DNTC	荘園ID
DNTC	WTC
End of Message	

図 9-53: %supply_wtc の形式

DNTC	TERMINATED
DNTC	荘園ID
DNTC	送信側クラスタ番号
DNTC	WTC
DNTC	返却される資源(上位)
DNTC	返却される資源(下位)
DNTC	消費された資源(上位)
DNTC	消費された資源(下位)
End of Message	

図 9-54: %terminate の形式

図9-54は、%terminatedの形式を示しており、各スロットの内容は次の通りである。

- TERMINATED

  このメッセージが%terminatedであることを示す種別記号

- 荘園ID

  里親が所属する荘園の外部参照ID

- 送信側クラスタ番号

  里親の存在するクラスタの番号

- WTC

  要求のあった里親に対して供給されるWTC

- 返却される資源(上位)

  里親に残っている資源の量

- 返却される資源(下位)

  里親では資源を2ワードのデータとして扱う

- 消費された資源(上位)

  里親が消費した資源の量

- 消費された資源(下位)

  里親では資源を2ワードのデータとして扱う

DNTC	READY
DNTC	荘園ID
DNTC	WTC
DNTC	送信側クラスタ番号
End of Message	

図 9–55: %ready の形式

(15) %ready

新しく里親が生成された時、その里親が処理開始の準備ができたことを荘園に対して報告する。その時送信されるのが%readyである。図9–55は、%readyの形式を示しており、各スロットの内容は次の通りである。

- READY

  メッセージが%readyであることを示す種別記号

- 荘園ID

  里親が所属する荘園の外部参照ID

- WTC

  メッセージに付けられるWTC

- 送信側クラスタ番号

  里親の存在するクラスタの番号

(16) %request_wtc

%request_wtcは里親が荘園に対してWTCの補給を要求する時に送信される。
図9–56は、%request_wtcの形式を示しており、各スロットの内容は次の通りである。

- REQUEST_WTC

  メッセージが%request_wtcであることを示す種別記号

- 荘園ID

  里親が所属する荘園の外部参照ID

- WTC

  メッセージに付けられるWTC

- 送信側クラスタ番号

  里親の存在するクラスタの番号

DNTC	REQUEST_WTC
DNTC	荘園ID
DNTC	WTC
DNTC	送信側クラスタ番号
End of Message	

図 9-56: %request_wtc の形式

DNTC	RETURN_WTC
DNTC	荘園ID
DNTC	WTC
DNTC	送信側クラスタ番号
End of Message	

図 9-57: %return_wtc の形式

(17) %return_wtc

%return_wtcは里親が荘園に対してWTCを返却する時に送信される。図9-57は、%return_wtcの形式を示しており、各スロットの内容は次の通りである。

- RETURN_WTC
  メッセージが%return_wtcであることを示す種別記号
- 荘園ID
  里親が所属する荘園の外部参照ID
- WTC
  メッセージに付けられるWTC
- 送信側クラスタ番号
  里親の存在するクラスタの番号

(18) %request_resource

%request_resourceは里親が荘園に対して資源の補給を要求する時に送信される。図9-58は、%request_resourceの形式を示しており、各スロットの内容は次の通りである。

- REQUEST_RESOURCE
  メッセージが%request_resourceであることを示す種別記号
- 荘園ID
  里親が所属する荘園の外部参照ID
- WTC

DNTC	REQUEST_RESOURCE
DNTC	荘園ID
DNTC	WTC
DNTC	送信側クラスタ番号
End of Message	

図 9–58: %request_resource の形式

メッセージに付けられるWTC

- 送信側クラスタ番号

  里親の存在するクラスタの番号

(19) %answer_statistics

%answer_statisticsは里親が荘園に対して統計情報を答える時に送信される。

図9–59は、%answer_statisticsの形式を示しており、各スロットの内容は次の通りである。

- ANSWER_STATISTICS

  メッセージが%answer_statisticsであることを示す種別記号

- 荘園ID

  里親が所属する荘園の外部参照ID

- WTC

  メッセージに付けられるWTC

- 資源(上位)

  現在、統計情報は消費資源量になっている

- 資源(下位)

  資源は2ワードのデータとして扱う

- 送信側クラスタ番号

  里親の存在するクラスタの番号

(20) %return_resource

%return_resourceは里親が荘園に対して資源を返却する時に送信される。

図9–60は、%return_resourceの形式を示しており、各スロットの内容は次の通りである。

- RETURN_RESOURCE

  メッセージが%return_resourceであることを示す種別記号

- 荘園ID

  里親が所属する荘園の外部参照ID

DNTC	ANSWER_STATISTICS
DNTC	荘園ID
DNTC	WTC
DNTC	資源(上位)
DNTC	資源(下位)
DNTC	送信側クラスタ番号
End of Message	

図 9-59: %answer_statistics の形式

DNTC	RETURN_RESOURCE
DNTC	荘園ID
DNTC	WTC
DNTC	資源(上位)
DNTC	資源(下位)
DNTC	送信側クラスタ番号
End of Message	

図 9-60: %return_resource の形式

- WTC
  メッセージに付けられるWTC
- 資源(上位)
  荘園に返却する資源
- 資源(下位)
  資源は2ワードのデータとして扱う
- 送信側クラスタ番号
  里親の存在するクラスタの番号

(21) %exception

ゴールを実行中に例外事象が起こった場合、その状況を荘園に報告する時に送信されるのが%exceptionである。図9-61は、%exceptionの形式を示しており、各スロットの内容は次の通りである。

- EXCEPTION
  メッセージが%exceptionであることを示す種別記号
- 荘園ID
  報告をする荘園の外部参照ID

DNTC	EXCEPTION
DNTC	荘園ID
DNTC	WTC
DNTC	例外番号
DNTC	例外情報ベクタへのポインタ
DNTC	コードへのポインタ
DNTC	引数ベクタ
End of Message	

図 9-61: %exception の形式

- WTC

  メッセージに付けられるWTC

- 例外番号

  例外の種類を示す番号

- 例外情報ベクタへのポインタ

  例外情報が入れられるベクタへのポインタ

- 新しいコードへのポインタ

  代わりに実行されるコードへのポインタ

- 新しい引数ベクタ

  代わりに実行される引数ベクタ

第 10 章

荘園 / 里親処理

執筆担当者: 川合, 山本

10.1 荘園の概念

KL1は、Flat GHC(Guarded Horn Clauses)を基にしたCommited-Choice型のAND並列論理型プログラミング言語である。

Flat GHCのような全てのゴールが平板な論理積となっているような言語では、一つのゴールの失敗がシステム全体の失敗となってしまうため、大規模で複雑なプログラムを作成するのは困難である。このためKL1では、Flat GHCに失敗の範囲を局所化できるような構造を持ち込み、この構造ごとにゴールの実行を制御できるようした。このような構造を荘園と呼ぶ。この荘園の機能を用いることによって、例外処理やメタなゴールの制御等が可能となり、オペレーティングシステムのような複雑なプログラムを記述することも可能となる。

荘園は、ゴールの失敗などのような例外事象の処理の他、ゴールの実行制御や資源管理の単位となる機構である。KL1のゴールは必ずいずれかの荘園に属しており、そのゴールから発生した全てのゴールもまた同じ荘園に属する。荘園内のゴールが更に荘園を生成することも可能であり、新たに生成された荘園はゴールが属する荘園の子荘園となる。このように、荘園は一般に図10-1に示されるような親子関係を持った階層構造となっている。

10.2 荘園の制御

荘園の制御は、通常荘園モジュールと呼ばれるKL1で記述された荘園制御用プログラムで定義された述語を用いる[1]。荘園モジュールには、荘園の生成のほか、荘園に対して起動/停止/実行放棄/資源追加などを指示するためのコントロールストリームや、荘園から報告される情報を荘園のユーザに通知するレポートストリーム機能のための述語などが定義されている。

荘園モジュールでは、コントロールストリームは図10-2のようなKL1プログラムとして実現されており、コントロールストリームへの入力は対応する組込述語に変換されて実行される。

[1] KL1の言語仕様として定めている訳ではない。

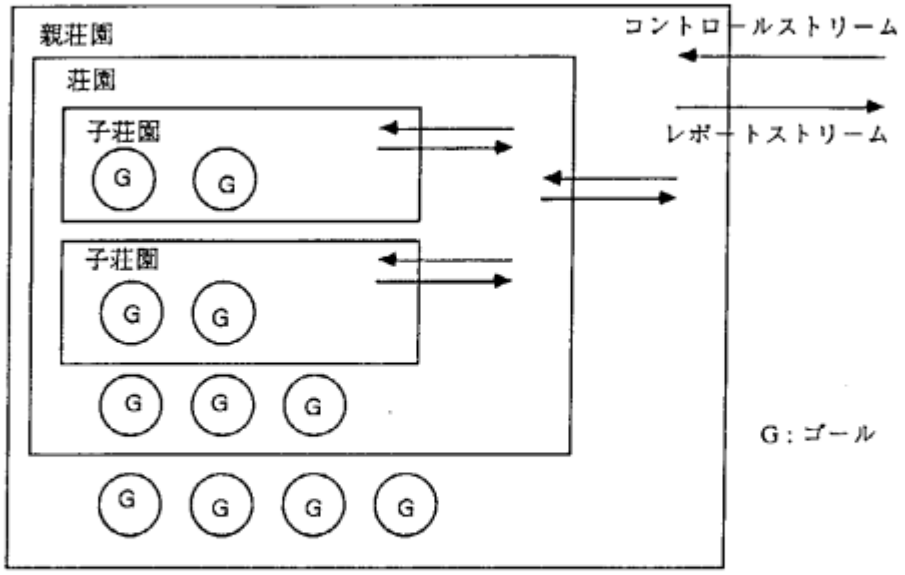


図 10-1: 荘園の親子関係

荘園制御用の組込述語には、一般に制御対象の荘園と制御後の荘園を表す引数がある。制御後の荘園を表す引数は、組込述語の呼び出し時には変数で与えられ、組込述語の実行終了時に荘園へのポインタがバインドされる。このため、荘園に対する制御はコントロールストリームによって逐次性を保っている。

荘園の状態が変化した場合、その状態変化は全ての子孫荘園に反映される。例えば、ある荘園がコントロールストリームからの停止指示で停止した場合、その荘園に属する全てのゴールの実行が停止されると共に、荘園の子孫荘園に属するゴールも全て停止される。荘園の再起動や実行放棄についても同様である。

また、レポートストリームは荘園生成用の組込述語に引数として与えられる変数であり、荘園からの報告時に処理系によって情報がバインドされる。荘園モジュール内にはこのレポートストリームを監視しているゴールがあり、レポートストリームに情報がバインドされると、このゴールによって情報が荘園のユーザに渡される。

10.3 資源の概念

資源とは、プログラムの実行による計算機使用量を、総合的かつ具体的に表現するために導入された概念であり、プログラムが消費した資源量を監視することによってプログラムの挙動を把握したり、プログラムに供給する資源量を調節することによってプログラムの実行を制御したりすることができる。

なお、資源に反映されるべき計算機使用量としては、CPU時間やメモリ消費量などが考えられるが、現在はCPU時間の代用としてリダクション数のみを資源として用いている。

資源を管理する単位は荘園であり、各荘園ではその荘園内で消費できる資源量の上限が定められている。荘園がその資源量の上限近くまで資源を消費すると、荘園のレポートストリームから資源僅少報告がなされる。

$$
\begin{aligned}
&control([start|CNTL], Shoen) : -true| \\
&\quad start\_shoen(Shoen, NewShoen), \\
&\quad control(CNTL, NewShoen). \\
&control([stop|CNTL], Shoen) : -true| \\
&\quad stop\_shoen(Shoen, NewShoen), \\
&\quad control(CNTL, NewShoen). \\
&control([abort|CNTL], Shoen) : -true| \\
&\quad abort\_shoen(Shoen, NewShoen), \\
&\quad control(CNTL, NewShoen). \\
&control([add\_resource(Res)|CNTL], Shoen) : -true| \\
&\quad add\_shoen\_resource(Shoen, Res, NewShoen), \\
&\quad control(CNTL, NewShoen). \\
&\qquad\qquad : \\
&\qquad\qquad :
\end{aligned}
$$

図 10-2: コントロールストリームの例

これに対して、荘園のコントロールストリームから資源を追加することができる。資源僅少報告後荘園に資源を追加しないでおくと、荘園の資源はやがて使い果たされ、資源が使い果たされた荘園ではゴールのリダクションは行われない。

さらに、コントロールストリームからは、荘園が現在までにどれくらいの資源を消費したかを問い合わせることもできるため、問い合わせ結果をもとにして荘園を制御することも可能である。

ただし、荘園への消費資源量問い合わせ時には一般に荘園ではゴールが実行されているため、消費資源量問い合わせに対する応答には一般に誤差が含まれており、次の2点だけが保証されている。

- 資源消費量は単調増加であり、減少はしない。
- 終結した荘園から得られる消費資源量は誤差を含んでいない。

10.4 荘園と里親

ゴールは、負荷分散などにより他のクラスタに投げられることがある。このため、荘園の存在するクラスタ以外のクラスタでも、荘園に代わってゴールを管理する実体が必要であり、これを里親と呼んでいる。里親は荘園によって管理されており、荘園の状態変化は里親にも反映される。しかし、コントロールストリームやレポートストリームは持たず、里親の管理するゴールの実行中に発生した例外の情報などは荘園のレポートストリームを通じて報告される。また、里親の管理するゴールが子荘園を生成した場合、この子荘園は里親に双方向リンクによってつながれ、里親によって管理される。

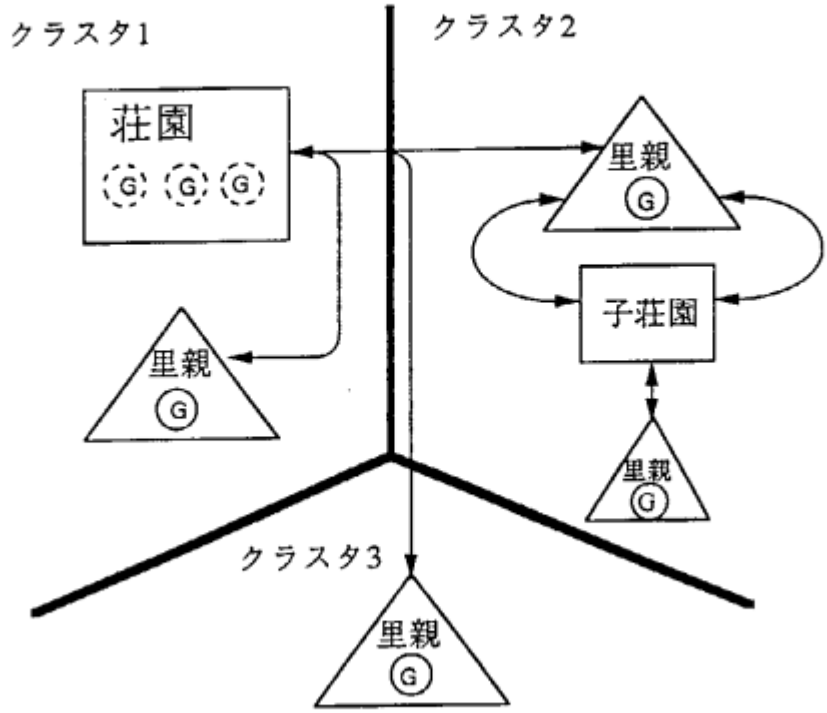


図 10–3: 荘園と里親

また、荘園の存在するクラスタと里親の存在するクラスタを処理系レベルで区別して処理を分けると処理系が複雑になるため、図10–3に示すように荘園の存在するクラスタにも里親を置き、荘園と里親間の通信は全てクラスタ間メッセージを用いて行うものとしている。(実際にネットワークに出すか否かはネットワークハンドラのレベルで決める)

このため、荘園と同一クラスタにある里親も他クラスタにある里親と同様にゴールや子荘園を管理しており、実質的にはゴールや子荘園は里親に属していると考えてよい。

里親はゴールを受け取ったクラスタに、そのゴールが属する里親が存在しない場合に動的に生成され、その里親に属するゴールや子荘園がなくなった時に消滅する。

以下に、荘園/里親の生成/終結/状態遷移についてその概要を述べる。

10.4.1 荘園 / 里親の生成

荘園の生成は、組込述語 create_shoen/5 によって行われる。荘園はこの述語を実行したクラスタにでき、同時に里親も同じクラスタに生成される。この荘園で実行されるゴールは、引数で与えられたコードと引数ベクタから新たに生成され、里親の最初のゴールとしてゴールキューに入れられる。その際のゴールのプライオリティは、引数で与えられた最高プライオリティとなる。

荘園及び里親の実体は、ヒープ領域に確保される構造体であり、それぞれ荘園レコード、里親レコードと呼ばれる。これらのレコードの内容については、10.5節を参照されたい。

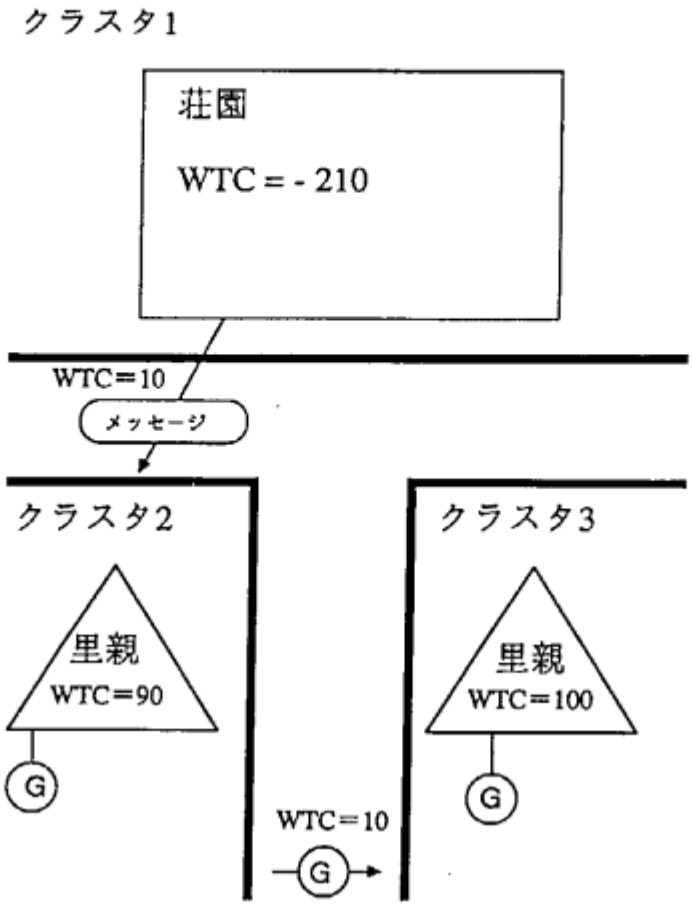


図 10–4: 荘園の WTC 管理

10.4.2 荘園 / 里親の終結

里親はゴール及び子荘園の終了を判定するためにチャイルドカウント と呼ばれる数値を管理している。チャイルドカウントはその里親に属するゴールの総数と子荘園の総数との和であり、里親が他のクラスタから投げられてきたゴールを受け取ったり、ゴールが新たなゴールや子荘園を生成した時にカウントアップされ、ゴールや子荘園が終了した時にはカウントダウンされる。このため、チャイルドカウントが0になったことで、この里親の中で行なう処理がなくなったことが検出できるため、この時点で里親は終結し、%terminatedを荘園に送信する。

また、荘園の実行放棄(アボート)によって里親が終結する時は、里親はすべての子荘園に対して%abortを送り、全ての子荘園が終結した時点で里親も終結する。この場合一般に里親には実行されないまま放棄されるゴールが存在するため、チャイルドカウントは0にはならない。

一方、荘園は図10–4に示されるようにWTC(Weighted Throw Count) と呼ばれる値を用いて、里親及びクラスタ間メッセージ処理の管理を行なう。WTCは、一つの荘園とその里親との間でトータルが0になる値で、里親の生成時に一定量のWTCが里親に与えられ、それと同じ量のWTCが荘園から引かれる。

メッセージの送受信時には、メッセージ送信側ではメッセージに付加した量のWTCを減じ、受信側

ではメッセージに付加されてきたWTCを加算する。また、里親が終結した時には、その時点でその里親の保持している全てのWTCが荘園に返され、里親の存在しないクラスタに到着したメッセージは荘園に送り返される。里親のWTCが不足した場合には里親は荘園にWTC要求を出すが、WTCが供給されるまでの間メッセージの送信処理は中断される。WTCは一つの荘園および、高々クラスタ数分の里親同士の間でやりとりされる値であるため、特定の里親にWTCが偏在した場合でも適宜WTCを荘園に返すことによって、荘園でのWTC不足や里親でのオーバフローを避けることが出来る。なお、VPIMではクラスタ間メッセージ同士で追い越しが生じても荘園のWTCが0にならないことを保証するために、全てのクラスタ間メッセージにWTCが付加されている。

従って、荘園はメッセージが到着するたびにメッセージに付加されてきたWTC値を加算し、WTC値が0になった時点で終結すればよい。

以上をまとめると、

- 荘園の終結

  WTCが全て荘園に戻され、荘園のWTCが0となった時。

- 里親の終結

  里親のチャイルドカウントが0となった時[2]、およびアボートされた時。

となる。

10.4.3 荘園 / 里親の状態遷移

荘園はコントロールストリームからの入力や親荘園の状態変化に伴って状態を変化させる。また、里親も荘園の状態変化に伴ってその状態を変化させる。荘園及び里親は、それぞれ次のような状態を遷移する[3]。

荘園:

started

実行可能状態

stopped

停止状態(停止の要因によってさらに三つに分けられる)

stopped_by_control

コントロールストリームから停止されている状態

stopped_by_parent

祖先の荘園が停止されたことに伴って停止されている状態

stopped_by_both

上記の両方の要因によって停止されている状態。

aborted

[2]厳密にいうとチャイルドカウント=0となった時点でも終結できないタイミングがあるが、これについては後述する。

[3]内部処理の都合上さらに幾つかの状態があるが、ここでは述べない。

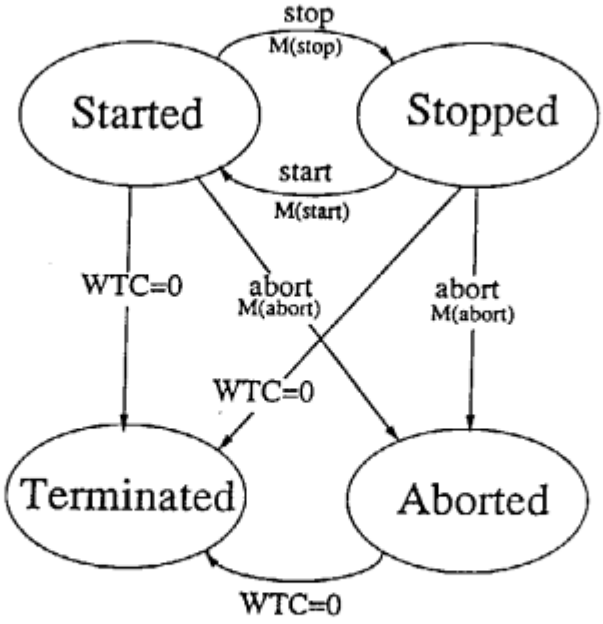


図 10–5: 荘園の状態遷移 (1)

アボートされてから終結するまでの状態。

terminated

終結した状態。

里親:

started

ゴールの実行可能状態。

stopped

ゴールの実行不可能状態

aborted

アボートされた状態。

terminated

終結処理中の状態。

また、それぞれの状態遷移は図10–5と図10–6に示される。

さらに、荘園のstarted状態とstopped状態間の状態遷移については、図10–7に示される。

荘園及び里親はstarted状態の時が所属するゴールを実行してもよい状態であり、それ以外の状態の時にはゴールを実行してはいけない状態である。また、一度terminatedやabortedの状態になった荘園/里親は、二度と実行可能状態には戻らない。なお、実際のゴールの実行の可否は、荘園/里親の持つ資源量とも関係してくる。

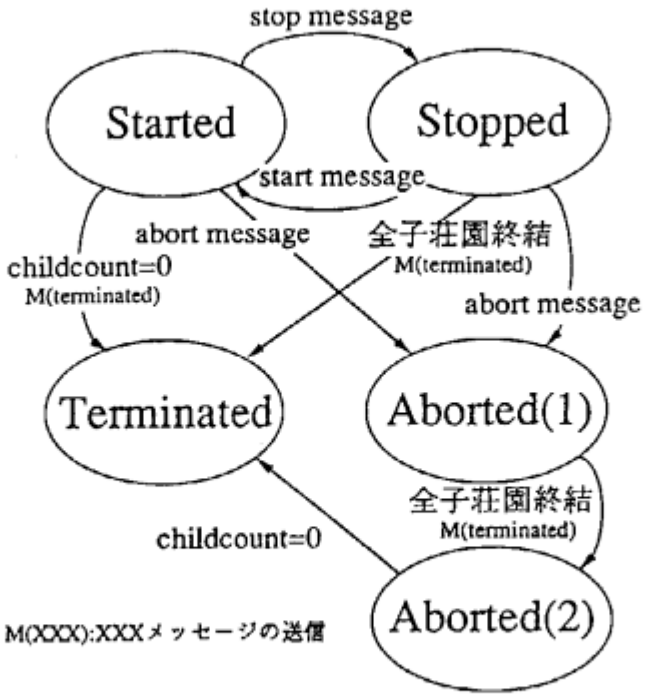


図 10–6: 里親の状態遷移

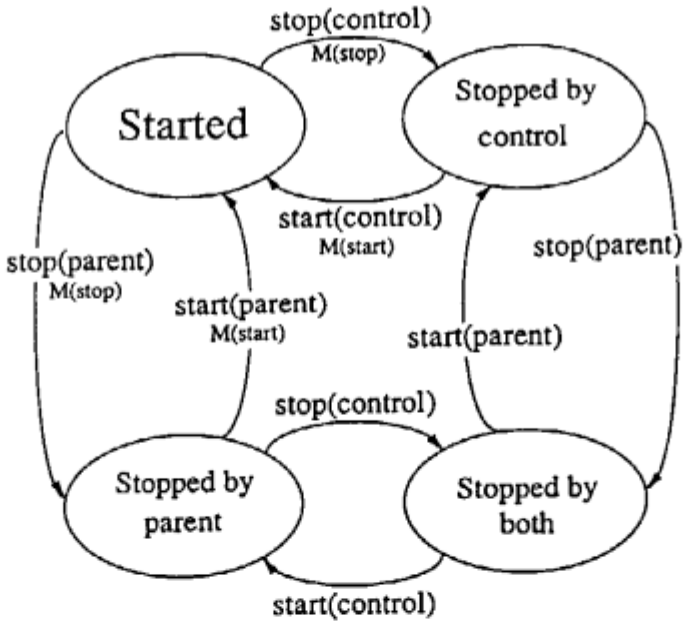


図 10–7: 荘園の状態遷移 (2)

10.5 荘園と里親のデータ構造

荘園及び里親は共有メモリ上の構造体として実現される。これらの構造体はそれぞれ荘園レコード及び里親レコードと呼ばれている。以下に荘園レコードおよび里親レコードの構造とその中に格納されるデータについて述べる。

荘園レコードは 24 ワードからなる構造体で各ワードには図 10–8 に示されるようなデータが格納される。

第0ワード	荘園レコードのソフトウェアロック用フラグ
第1ワード	荘園ID
第2ワード	親里親レコードへのポインタ
第3ワード	兄弟荘園へのポインタ(右方向のリンク)
第4ワード	兄弟荘園へのポインタ(左方向のリンク)
第5ワード	里親テーブル
第6ワード	資源要求中の里親テーブル
第7ワード	レポートストリーム用の変数
第8ワード	Ready/Terminatedメッセージカウンタ
第9ワード	CDESC（以下のワードのタグはDNTC)
第10ワード	荘園のステータス
第11ワード	荘園の最低プライオリティ
第12ワード	荘園の最高プライオリティ
第13ワード	エクセプションマスクパターン
第14ワード	未回答の統計情報調査要求数
第15ワード	まだ統計情報調査に回答していない子里親の数
第16ワード	荘園の消費した資源量(上位)
第17ワード	荘園の消費した資源量(下位)
第18ワード	最も最近荘園が報告した消費資源量(上位)
第19ワード	最も最近荘園が報告した消費資源量(下位)
第20ワード	荘園の手持ち資源量(上位)
第21ワード	荘園の手持ち資源量(下位)
第22ワード	荘園の資源リミットまでの残量(上位)
第23ワード	荘園の資源リミットまでの残量(下位)

図 10-8: 荘園レコードのデータ形式と格納されるデータ

- 第0ワード：荘園レコードのソフトウェアロック用フラグ

  荘園レコード全体をソフトウェアロックするためのフラグ。

- 第1ワード：荘園ID

  荘園IDは、荘園を格納した黒輸出表エントリのオフセット値の上位3ビットにクラスタ番号を結合したものである。

- 第2ワード：親里親レコードへのポインタ

  この荘園の直属の親里親(必ず同じクラスタ内に存在する)へのポインタ。

- 第3ワード：兄弟荘園へのポインタ(右方向のリンク)

  同じ直属の親里親の下に作られた兄弟荘園を双方向リンクでつなぐ際に、右方向の荘園レコードを

差すポインタ。

- 第4ワード：兄弟荘園へのポインタ(左方向のリンク)
  同じ直属の親里親の下に作られた兄弟荘園を双方向リンクでつなぐ際に、左方向の荘園レコードを差すポインタ。
- 第5ワード：里親テーブル
  32ビットのビットマップになっていて、各ビット位置のオン/オフによってそのビット位置に対応するクラスタに里親が存在するか否かを示す。なお、このテーブルはクラスタ数に応じたワード数のベクタとなっている。
- 第6ワード：資源要求中の里親テーブル
  32ビットのビットマップになっていて、各ビット位置のオン/オフによってそのビット位置に対応するクラスタにある里親が資源要求をしているか否かを示す。なお、このテーブルはクラスタ数に応じたワード数のベクタとなっている。
- 第7ワード：レポートストリーム用の変数
  レポートストリームからの応答として返すデータをバインドするための変数。
- 第8ワード：Ready/Terminated メッセージカウンタ
  里親から送られてくる %ready/%terminated の数をカウントしておくためのカウンタ。クラスタ間のメッセージ追い越し現象への対応用カウンタである。
- 第9ワード：CDESC ワード
  以下のワードではタグが意味を持たないことを表すワード。GC ルーチンが利用する。
- 第10ワード：荘園のステータス
  荘園のステータスには次の2種類のものがあり、それぞれ8ビットずつを用いて表現される。

  1. 遷移ステータス
     荘園の論理的な状態遷移を表現するものであり、次の6種類がある。
     - ◇ started :
       実行可能状態
     - ◇ stopped_by_control :
       コントロールストリームから停止されている状態
     - ◇ stopped_by_parent :
       親の荘園から停止されている状態
     - ◇ stopped_by_both :
       コントロールストリームと親荘園の両方から停止されている状態
     - ◇ aborted :
       アボートされた状態
     - ◇ terminated :
       終結された状態

2. 状態フラグ

荘園の状態をフラグで表現するものであり、次の3種類がある。(互いに排他的ではない)

◇ resource_report_disallowed :

荘園の資源残量が少なくなっても、レポートストリームから報告してはいけないことを示す。

◇ no_more_external_event :

コントロールストリームが閉じられて、もう外部からの状態変化が起こらないことを示す。

◇ resource_request_fp_existance_flag :

資源要求を出している里親が存在することを示す。

- 第11ワード：荘園の最低プライオリティ

この荘園内で実行されるゴールの取り得る最低のプライオリティ。

- 第12ワード：荘園の最高プライオリティ

この荘園内で実行されるゴールの取り得る最高のプライオリティ。

- 第13ワード：エクセプションマスクパターン

里親から伝えられたエクセプションをこの荘園で受け付けるか否かを判定するためのマスクパターン。

- 第14ワード：未回答の統計情報調査要求数

実行を待っている統計情報調査の数。

- 第15ワード：まだ統計情報調査に回答していない子里親の数

統計情報調査の要求を送った子里親の内、まだ回答を返していない子里親の数。

- 第16-17ワード：荘園の消費した資源量(上位、下位)

この荘園及びその子孫の荘園が消費した資源量。

- 第18-19ワード：最も最近荘園が報告した消費資源量(上位、下位)

最も最近この荘園が親里親またはレポートストリームに報告した消費資源量。

- 第20-21ワード：荘園の手持ち資源量(上位、下位)

この荘園内で当面消費して良い資源量。

- 第22-23ワード：荘園の資源リミットまでの残量(上位、下位)

この荘園で消費して良い最大資源量までの残量。

里親レコードは22ワードからなる構造体で各ワードには図10-9に示されるようなデータが格納される。

- 第0ワード：黒輸入表エントリへのポインタ

里親レコードが登録されている黒輸入表エントリへのポインタ

- 第1ワード：子荘園へのポインタ(右方向リンク)

第0ワード	黒輸入表エントリへのポインタ
第1ワード	子荘園へのポインタ(右方向リンク)
第2ワード	子荘園へのポインタ(左方向リンク)
第3ワード	停止中のゴールをつなぐリンクルート
第4ワード	WTC不足で送信できないメッセージのリンクルート
第5ワード	里親レコードのソフトウェアロック用フラグ
第6ワード	子荘園リンクのソフトウェアロック用フラグ
第7ワード	CDESCワード (以下のワードのタグはDNTC)
第8ワード	里親のステータス
第9ワード	里親の手持ち資源量(上位ワード)
第10ワード	里親の手持ち資源量(下位ワード)
第11ワード	最も最近荘園に報告した消費資源量(上位ワード)
第12ワード	最も最近荘園に報告した消費資源量(下位ワード)
第13ワード	チャイルドカウント
第14ワード	チャイルドカウント(GC時の永久中断ゴール検出用)
第15ワード	まだ統計情報調査に回答していない子荘園の数
第16ワード	里親の最小優先度
第17ワード	里親の最大優先度
第18ワード	スタート/ストップカウンタ
第19ワード	子荘園消費報告資源量(上位ワード)
第20ワード	子荘園消費報告資源量(下位ワード)
第21ワード	消費資源収集イベントカウント

図 10-9: 里親レコードのデータ形式と格納されるデータ

里親の子荘園双方向リンクにおける右方向の荘園レコードを指すポインタ。

- 第2ワード: 子荘園へのポインタ(左方向リンク)

  里親の子荘園双方向リンクにおける左方向の荘園レコードを指すポインタ。
- 第3ワード: 停止中のゴールをつなぐリンクルート
- 第4ワード: WTC不足で送信できないメッセージのリンクルート
- 第5ワード: 里親レコードのソフトウェアロック用フラグ
- 第6ワード: 子荘園リンクのソフトウェアロック用フラグ
- 第7ワード: CDESCワード
- 第8ワード: 里親のステータス

  里親のステータスには次の2種類のものがある。

(1) 論理ステータス

互いに排他的な、里親の論理的な状態を表現するものであり、次の6種類がある。

・reserved、started、stopped、aborted、terminate、aborted-terminate

(2) 物理ステータス

互いに独立な、里親の物理的な状態を表現するものであり、次の4種類がある。

・resource_exhausted、WTC_requested、

・resource_requested、child_resource_requested

- 第9ワード：里親の手持ち資源量(上位ワード)
- 第10ワード：里親の手持ち資源量(下位ワード)

この里親内で当面消費して良い資源量。

- 第11ワード：最も最近荘園に報告した消費資源量(上位ワード)
- 第12ワード：最も最近荘園に報告した消費資源量(下位ワード)

荘園からの統計情報調査に対して、最も最近報告した里親の消費資源量。

- 第13ワード：チャイルドカウント

里親に属するゴールの数と里親の子荘園の数の合計。里親の終了判定に使う。

- 第14ワード：チャイルドカウント(GC時の永久中断ゴール検出用)
- 第15ワード：まだ統計情報調査に回答していない子荘園の数
- 第16ワード：里親の最小優先度
- 第17ワード：里親の最大優先度
- 第18ワード：スタート／ストップカウント

荘園から来る%start,%stopメッセージをカウントし、メッセージの追い越しが有っても、最終的に辻褄が合うようにする。

- 第19ワード：子荘園消費報告資源量(上位ワード)
- 第20ワード：子荘園消費報告資源量(下位ワード)

統計情報調査の回答の集計値

- 第21ワード：消費資源収集イベントカウント

10.6 リダクションサイクル

クラスタ内のPEにおけるゴールリダクションのサイクルは、一般に次のようなループを回っている。

```
LOOP(){
```

- ゴール実行
- 消費資源カウント
- チャイルドカウントメンテナンス
- スリットチェック

・次ゴールスケジューリング(里親状態チェックを含む)

}

以下、それぞれについて簡単に説明する。

- ゴール実行
  ゴールを実行する。
- 消費資源カウント
  ゴールリダクションにより消費した資源量を里親の資源から減ずる。なお実際には、リダクションごとに里親にアクセスするのはオーバヘッドが大きくなるため、里親の資源の一部をPEごとのレジスタにキャッシュしている。
- チャイルドカウントメンテナンス
  里親のチャイルドカウントをデクリメントする。その結果0になったら里親を終結する。なお実際には、リダクションごとに里親にアクセスするのはオーバヘッドが大きくなるため、里親のチャイルドカウントの一部をPEごとのレジスタにキャッシュしている。
- スリットチェック
  ゴールの実行中に割り込みやメッセージ到着などのイベントが発生していないかどうかをチェックする。イベントが発生していた場合は、それを処理する。
- 次ゴールスケジューリング(里親状態チェックを含む)
  次に実行すべきゴールをゴールキューから取りだし、その里親の状態(実行可能状態か否か、資源があるか否か)をチェックして次に実行すべきゴールを決定する。もし、実行可能なゴールがなければアイドルゴールが実行される。

10.7 資源管理

上述したように、ゴールのリダクションに伴って資源は里親から減じられていく。つまり、資源を実際に消費するのは里親である。ゴールリダクションが繰り返され里親の資源残量が一定値以下になると、里親は荘園に対して資源要求メッセージ(%request_resource)を出す。

実際の資源要求の流れは、里親から荘園へ、荘園から親里親へ、親里親から親荘園へというように子孫から祖先方向に流れる。これに対して資源供給の流れは、荘園や里親の終結時を除いて祖先方向から子孫方向に流れる。この様子は図10-10に示される。

荘園では、資源は資源リミット残量 と手持ち資源残量 として管理される。資源リミット残量は、子孫を含めてその荘園が最大消費することが許される資源の上限値である。また、この値は荘園の親里親から貰って来ることのできる資源の残量を示しており、実際に荘園が親里親から資源を貰ってくるたびに減ぜられる。そして、この値を越えて資源を貰おうとすると、荘園のレポートストリームから資源僅少報告 (resource_low)が報告される[4]。 一方、手持ち資源残量は荘園が一時的に保管する資源であり、

[4]実際には、資源リミット残量がある程度少なくなった時点で報告される。

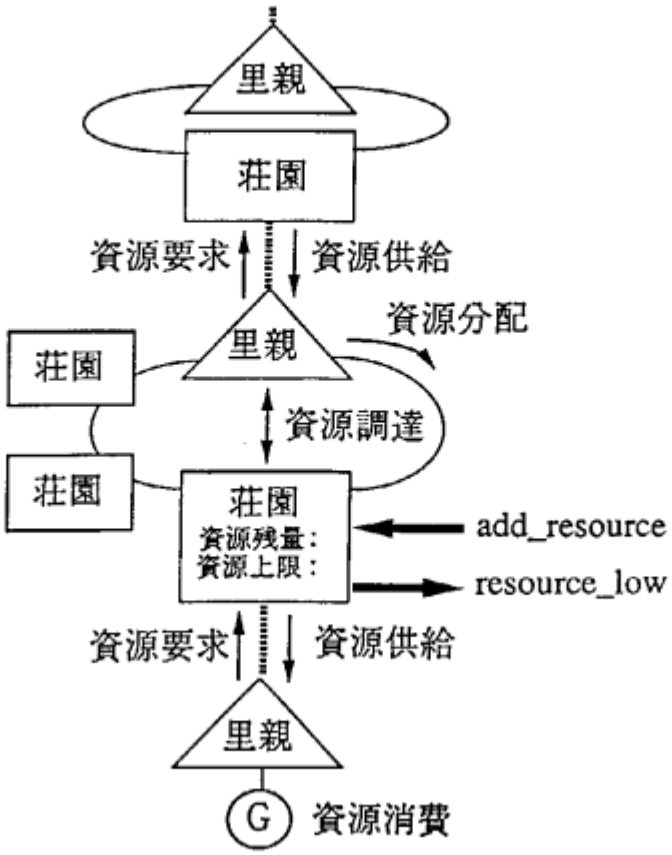


図 10-10: 資源要求と供給の流れ

里親からの資源要求が来た時にここから資源が与えられる。また、里親の終結などによって資源が返却されてきた場合には、その資源は荘園の手持ち資源として保管される。

荘園が里親から%request_resourceを受け取った場合、荘園の手持ち資源残量および資源リミット残量の値によって次のように処理が分かれる。

- 荘園の手持ち資源残量が十分にある時
  里親に資源供給メッセージ (%supply_resource) を送る。里親に供給する資源の単位はResourceSupplyUnitと呼ばれる一定の値である。

- 荘園の資源リミット残量がまだあり、手持ち資源残量がない時
  荘園の親里親から資源を貰ってくる。親里親に資源がない場合には親里親から親荘園に対して%request_resourceを送る。

- 荘園の資源リミット残量がない時
  荘園のレポートストリームから資源僅少報告(resource_low)を流す。

資源要求を出した里親に%supply_resourceが到着したとき、その里親に資源不足以外の実行不可

能要因がなければ、資源不足によって実行できずに里親につながれていたゴールを全てゴールキューに入れる。また、里親の子荘園の中に資源要求が満たされていないものがあれば、この子荘園にも資源を分配する。さらに子荘園では、子荘園の里親のうち資源要求が満たされていない里親に対して%supply_resourceを送信する。

里親や荘園が終結した場合、消費されずに残った資源は荘園及び親里親に返される。また、荘園が終結する時には、荘園に与えられた資源リミットと実際の消費資源量との差分が余剰分として荘園のレポートストリームから報告される。

10.8 荘園制御処理の流れ

コントロールストリームからの入力によって荘園が制御された場合の、荘園 / 里親 / 子荘園での処理内容について、以下に述べる。

- start処理

  コントロールストリームからstart命令を受けた荘園(つまり、組込述語start_shoenを実行する荘園)では、荘園のstopped_by_control状態を解除する。

  この結果、stopped_by_control 状態の荘園は started 状態に、stopped_by_both 状態の荘園は stopped_by_parent状態となる。なお、VPIM では荘園 / 里親間の%start/%stop メッセージの追い越しにカウンタを用いて対応しているため、前述の荘園モジュール内のKL1 プログラムで started 状態の荘園へのstart_shoen、stopped 状態の荘園へのstop_shoen などを禁止している。従って、荘園がこれら以外の状態になっていたら、処理系のエラーである。また、荘園の状態がstarted となる場合には、その時点で荘園が存在を把握している全ての里親に対して、%startを送信する。

  %start を受信した里親では状態を started とする。また、里親がstopped 状態である間にスケジュールされ、実行できずに里親につながれていたゴールをゴールキューに入れる。さらに、里親に子荘園があれば全ての子荘園のstopped_by_parent状態を解除する。

  この結果、stopped_by_parent状態の子荘園はstarted状態に、stopped_by_both状態の子荘園はstopped_by_control状態となる[5]。また、子荘園の状態がstarted となる場合には、その時点で子荘園が存在を把握している全ての子里親に対して、%startを送信する。

- stop処理

  コントロールストリームからstop命令を受けた荘園(つまり、組込述語stop_shoenを実行する荘園)では、荘園のstopped_by_control状態を付加する。

  この結果、started状態の荘園はstopped_by_control 状態に、stopped_by_parent 状態の荘園はstopped_by_both状態となる[5]。また、荘園の状態がstopped_by_control となる場合には、その時点で荘園が存在を把握している全ての里親に対して、%stopを送信する。

[5] これら以外の状態になっていたら、処理系のエラーである。

%stop を受信した里親では状態を stopped とする。また、クラスタ内の各 PE で実行されているゴールの里親が stop メッセージを受信した里親か否かをチェックするために、クラスタ内各 PE に割り込みをかける。この割り込みは各 PE でリダクションの切れ目(スリットチェック時)に受け付けられ、里親の状態が stopped に変化していた場合には、スケジュールされたゴールは里親につながれ実行されない。

さらに、里親に子荘園があれば全ての子荘園に stopped_by_parent 状態を付加する。

この結果、started 状態の子荘園は stopped_by_parent 状態に、stopped_by_control 状態の子荘園は stopped_by_both 状態となる[5]。また、子荘園の状態は stopped_by_parent となる場合には、その時点で子荘園が存在を把握している全ての子里親に対して、%stop を送信する。

- abort処理

荘園の状態を aborted とする。また、その時点で荘園が存在を把握している全ての里親に対して、%abort を送信する。

%abort を受信した里親では状態を aborted とする。また、クラスタ内の各 PE で実行されているゴールの里親が abort メッセージを受信した里親か否かをチェックするために、クラスタ内各 PE に割り込みをかける。この割り込みは各 PE でリダクションの切れ目(スリットチェック時)に受け付けられ、里親の状態が aborted に変化していた場合には、スケジュールされたゴールは実行されずに捨てられる。

また、里親に子荘園がいない場合、里親が abort された時点でその里親での総リダクション数が確定する[6]ため、クラスタ内の各 PE で里親の状態チェックが終わり、PE にキャッシュされていた資源を里親に戻した後に、荘園に対して %terminated に乗せて里親での最終的な消費資源量を報告する。

一方、里親に子荘園があれば全ての子荘園を aborted 状態とし、その時点で子荘園が存在を把握している全ての子里親に対して、%abort を送信する。里親が荘園に対して %terminated を送信するのは、全ての子荘園が終結した時点である。

- add_resource処理

荘園の資源リミット残量を、与えられた資源の分だけ引き上げる。この結果として、荘園は親里親から資源を貰うことが出来るようになる訳であり、荘園の資源残量に直接与えられた資源を加算する訳ではない。親里親から貰ってきた資源は、荘園を経由して資源要求を出している里親に %supply_resource によって分配される。

なお、%supply_resource を受信した里親の処理内容については、資源管理の項に述べた通りである。

[6] abort後の里親のゴールが実行されることはない。

- ask_statistics処理

  コントロールストリームからask_statistics命令を受けた荘園(つまり、組込述語shoen_statisticsを実行する荘園)での全消費資源量(荘園の全里親およびその子孫荘園での消費資源量 の和)を計測する。荘園は、その時点で荘園が存在を把握している全ての里親に対して、%ask_statisticsを送信して各里親での全消費資源量の計測を依頼する。そして、計測を依頼した全ての里親からの回答が揃った時点で、全消費資源量をレポートストリームから報告する。なお、荘園は全消費資源量をレポートするまでは次の消費資源量計測を実行せず、カウントして覚えておく。

  一方、%ask_statisticsを受信した里親では、クラスタ内の各PEで実行されているゴールの里親が消費資源量計測の対象となる里親か否かをチェックするために、クラスタ内各PEに割り込みをかける。この割り込みは各PEでリダクションの切れ目(スリットチェック時)に受け付けられ、里親が消費資源の計測対象であった場合には、子荘園の有無によって以下のように処理が分かれる。

  [里親に子荘園のない場合]

  クラスタ内の各PEにキャッシュされている資源を一時里親に戻して消費資源量を求める。このとき、あるPEが里親に資源を戻しても他のPEがその里親のゴールを実行してしまうと里親の消費資源量が正しく求められないため、クラスタ内の全PEで同期を取って里親にキャッシュ資源を戻す。その後、里親の消費資源量を%answer_statisticsに乗せて荘園に回答する。

  [里親に子荘園がある場合]

  全ての子荘園に対して消費資源量の問い合わせ処理を行う。すべての子荘園の消費資源量が求まった後に、上記の「里親に子荘園のない場合」の処理を行う。

  なお、終結していない荘園に対して消費資源量の問い合わせを行った場合、その荘園がたとえstopped状態であったとしてもその荘園の子孫が全てstopped状態になっている保証はないため、回答される消費資源量には誤差が含まれる可能性が有る。

10.9 処理系実装における留意点

本節では、前節までに述べた機能をVPIMに実装する時にとった方針を説明する。

10.9.1 荘園実装の基本方針

通常のリダクションを遅くしなければ良い。荘園生成オーバヘッドや、荘園を通じたスタートストップなど制御のオーバヘッドは構わない。

クラスタ間に渡る荘園の情報は、KL1データの輸出入の枠組を借り、**荘園レコードを輸出表に、里親レコードを輸入表に登録することにより実装している。** 形式的には、**「荘園レコードを輸出した物が里親レコードである」**という形になっていて、荘園を識別する荘園IDも外部参照IDを用いている。

ただし、実現のしやすさから、同一クラスタ内にも里親を生成している。従って1クラスタでも、VPIMの荘園をそのまま実装するにはクラスタ間機能が必要になる。なお、荘園レコード自身が別のクラスタに移動することはない。

ゴールリダクション処理の中で、比較的頻繁にアクセスされるのは、ゴールの実行環境を握っている里親レコードであり、特にリダクション毎にアクセスされる消費資源量などは、PE毎にキャッシュし、排他制御の必要な里親レコード自身へのアクセスの頻度を低減している。やむを得ず、里親レコードにアクセスする場合も、ロックは出来るだけ少なく、また、短くしている。

10.9.2 荘園里親レコードの排他制御

排他制御の対象は、**荘園レコード**、**里親レコード**、**里親-子荘園リンク**の3つがあり、ビジーウェイトによるソフトロックマクロを用いている。このソフトロックマクロは、ロックの状態を記録するロックフラグを必要とする。上記の3種類の排他制御のためのロックフラグは、それぞれ、荘園レコードのロックフラグは、荘園レコード中に、里親レコードのロックフラグ及び里親-子荘園リンクのロックフラグは里親レコード中に設けている。

ソフトロックマクロの実現には、メモリアクセス命令に付随した、ハードロック命令を用いている。

複数の排他制御対象を重ねてロックする場合に、デッドロックを回避するため、荘園と輸出表、里親と輸入表、里親と子荘園の排他制御の順番を定めている。

(1) 荘園の排他制御

- 荘園と輸出表

  荘園→輸出表の順。

  輸出入されているデータは、一般に指されている側を、指している側より先にロックする。後の、里親と輸入表も同様。

- 割当時、回収時の処理

  荘園レコードを割り当てるタイミングは、唯一 b_create_shoen組込述語が呼ばれた時である。本組込述語が Shoen 出力変数を具体化しない限り、新荘園が他から参照される恐れはない。（正確に言うと、ボディ組込述語の出力のユニフィケーションは、組込述語実行後に行なうことを、コンパイラがコード生成の約束として保証しているので、KL1プログラムレベルで、他から参照が可能になるのは、このユニフィケーションの実行の後である。）

  割当処理は、荘園レコード自身を一通り初期設定をした後、輸出表に登録し、子荘園リンクをロックしてから、これにつなぐ。子荘園リンクのロックを解除した時に、初めて他のプロセッサから見えるようになるので、荘園レコード自身のロックは必要ではない。

  回収処理は、他からの参照が無いことを確認したPEが行なうので、フリーリストに返す前の適当なタイミングでロックを解除すればよい。荘園レコードの回収は、すべてのWTCの回収と b_remove_shoen 組込述語の発行の両方が必要であることに注意すること。

(2) 里親の排他制御

- 里親と輸入表

  輸入表→里親の順。

- 割当時、回収時の処理

  里親レコードを割り当てるタイミングは、荘園の生成時の他に、他クラスタから投げられてきたゴールの受信時にその可能性がある。

  割り当て処理は、黒輸入ハッシュエントリをロックしてから、里親レコードを本当に割り当てる必要を確認して、里親レコード自身を仮に初期設定をした後、黒輸入表に登録する。黒輸入ハッシュエントリのロックを解除する時は里親もロック解除の状態でなければならないので、代わりに里親の遷移状態を _FP_RESERVED とする。黒輸入ハッシュエントリのロックを解除した時に、始めて他のプロセッサから見えるようになる。

  荘園生成時に同時に割り当てられる里親の場合は、他から参照されることがないので、里親レコード自身のロックは必要ではない。

  一方、ゴールの受信で生成される里親に関しては、改めて里親レコードをロックしてから、初期化の続きを行ない、遷移状態は _FP_RESERVED → _FP_STARTED とする。また、遷移状態が _FP_RESERVED の里親を見つけてしまったプロセッサは、_FP_STARTED に成るまでビジーウェイトする。[7]

  回収処理は、他からの参照が無いことを確認した人が行なうので、フリーリストに返す前の適当なタイミングでロックを解除すればよい。里親の回収は子荘園、ゴール、統計情報収集処理、サスペンドメッセージなどが全て無くなった時に行なわれる。アボート時には、ゴールのみ残っていると、輸入表からは削除されるが、里親レコードはすべてのゴールがスケジュールされてくるまで回収できないの注意すること。

- 里親をロックしなくても良い時

  リダクション処理中に里親の情報を必要とする時、処理中のリダクションに対応するゴールが里親レコードに参照カウントを持っているので、まず、里親が無くなってしまう事はない。そこで、里親レコードの特定フィールドを読むだけで事が足りるならば、里親レコードにロックを掛けなくて良い。

- 後から輸入表にアクセスしたくなった時

  里親をロック中に、クラスタ間メッセージを送信するために、輸入表をアクセスする必要が生じた時には、一度里親レコードのロックを解除し、輸入表をロックした後に、必要ならば、里親レコードを再ロックする。

  多くの処理では、最初の里親ロック中に必要な情報は里親レコードから取り出してしまい、その時点で里親レコードのロックを解除してしまうのが、ロック期間を短くするという現在の作戦に合致している。

---

[7] PIMでは、ここらへんの排他制御が大変複雑になっている。荘園里親の実装において、KL1変数の輸出入方式としての枠組を借りるだけでなく、本当に輸出入表まで借りてしまったのが敗因ともいえる。里親の登録は里親表にしてしまえばいろいろ簡単になるはず…

(3) 子荘園リンクの排他制御

- 里親、子荘園リンクと子荘園

  子荘園リンク→里親→子荘園の順。

  里親の下にある子荘園は、双方向リンクでつないだリングの形で管理している。このリンクを子荘園リンクと呼んでいる。

  子荘園の生成消滅、里親側からの子荘園のサーチなど、子荘園リンクをたどり、またレコードの挿入削除を行なう処理では、子荘園リンクのロックが必要である。

  子荘園の仕事、里親の仕事の双方からこのリンクにアクセスしに来る場合があるが、どちらの場合も、それぞれ、荘園レコードあるいは里親レコードのロックを解除してから、まず子荘園リンクをロックし、次に必要に応じて荘園レコードあるいは里親レコードをロックする。

  また、里親と子荘園を同時にロックする場合は里親からロックする。

- 子荘園リンクをロックしないで良い時

  里親の仕事の延長で、子荘園の有無をチェックするためだけに、里親レコード中の子荘園リンクへのポインタをテストする時は、里親レコードのロックだけで良い。

10.9.3 ゴールの実行制御方式

資源（リダクション数）に関するアクセスはリダクション毎にあるので、これを毎回排他制御しながらアクセスするのはオーバヘッドが大である。これらはPE毎にキャッシュする。

具体的にはカレントゴールの里親が切り替わるときにのみ、里親レコード中の資源残量等をメンテする。

ちなみに里親の状態が、メッセージ受信により変化した時には、クラスタ内のイベントとして、各PEに通知されるので、スリットチェックに掛かった時にのみ里親を調べるようにしている。

また、実行環境には、ゴール毎に、あるいはゴール間で共有可能な環境レコードに、持たせる情報もある。（2.4節**ゴールレコードおよび環境レコードの構造**を参照）

以下に、VPIMにおけるゴールの実行制御方式と問題点及びその解決策について詳しく述べる。

クラスタ内の各PEで実行されるゴールはレディゴールスタックから順次取り出されて、そのゴールの属する里親が実行可能状態(荘園が止められていなくて、かつ資源残量 > 0)である場合に実行される。しかし、VPIMではクラスタ内の複数PEで同じ里親に属するゴールが実行されることがあるため、ゴールの取り出し毎にそのゴールが属する里親の状態をチェックすると、里親の排他制御やPEのキャッシュメモリ間のバストラフィックが増大することなどのためにオーバヘッドが大きくなってしまう。また、里親の資源残量やチャイルドカウントの更新についても同様である。そこで、VPIMでは、できるだけ各PEが里親レコードにアクセスせずに済むように、以下のような処理方式を取っている。

① 里親の資源をPE毎にキャッシュする。ゴールを実行する前に、そのゴールが所属する里親の資源から一定量(キャッシュユニット)をPEに取り分けておき、以後連続して同じ里親に属するゴールを実行する時は、このキャッシュした資源からデクリメントする。そして、キャッシュした資源が尽きた

時に再び里親から資源をキャッシュして来る。この時、里親に資源がなければ実行しようとしていたゴールは中断ゴールとして里親レコードにつながれ、レディゴールスタックから次のゴールがスケジューリングされる。なお、里親レコードにつながれた中断ゴールは、里親に資源が補給された時点で里親レコードから外され、再びレディゴールスタックに入れられる。また、違う里親に属するゴールを実行する時は、直前まで実行していたゴールが属する里親にキャッシュしてあった資源を書き戻した後、次に実行するゴールが属する里親から資源をキャッシュして来る。なお、この時同時にこの里親が実行可能状態であるか否かのチェックも行なう。

② 里親のチャイルドカウント値をPE毎にキャッシュする。ゴールを実行する前に、PE毎のチャイルドカウント値を0にしておく。そして、連続して同じ里親に属するゴールを実行する際に、ゴールの中で新たなゴールをゴールスタックにエンキューした場合にはこのチャイルドカウント値をインクリメントし、ゴールの実行が終了した場合にはデクリメントする。また、違う里親に属するゴールを実行する時は、直前まで実行していたゴールが属する里親にチャイルドカウント値を書き戻し、PE毎のチャイルドカウント値を0にクリアした後にゴールを実行する。

③ 里親の状態変化はスリットチェック機構によって各PEに通知する。荘園の状態変化に伴って、里親の状態が実行可能状態から実行不可能状態に変化した場合、レディゴールスタックからのゴールの取り出し毎に里親の状態をチェックしていれば、里親が実行可能状態から実行不可能状態に変化した時にその里親に属するゴールの実行を防ぐことができる。しかし、上記のように、同一の里親に属するゴールが連続して実行される時には原則として里親にアクセスしないような処理方式を取ると、里親の状態をチェックできるタイミングが

- キャッシュ資源が尽きた時
- 今までと違う里親に属するゴールがスケジュールされた時

に限られてしまうため、実行不可能状態になった里親に属するゴールがそのまま実行されてしまう可能性がある。このため、クラスタ内に存在するいずれかの里親が実行不可能状態に変化した時には、スリットチェック機構によってクラスタ内の全PEに通知し、その時点で実行不可能状態に変化した里親に属するゴールを実行していたPEでは、違う里親に属するゴールがスケジュールされた時と同じ処理を行なうことによって、実行不可能状態に変化した里親に属するゴールの実行を防ぐ。また、その時点で実行していたゴールの里親が、実行不可能状態に変化した里親ではなかった場合には、何もなかったように処理を続行すれば良い。

10.9.4 クラスタ内同期処理

PIMのクラスタ内部では、一般に複数のPEが独立に動作している。しかし、次の処理を行う場合にはクラスタ内の全PEが同期して動作することが必要である。(第8章スリットチェック参照)

- GC(Garbage Collection)時
- 里親の統計情報（消費資源量）調査時

また、一つの同期処理中には、他の同期処理は行えないため、クラスタ内同期処理同士の排他制御が必要になってくる。このため、クラスタ内同期処理を行う権利を示すフラグを設け、両者間の排他制御を実現した。

(1) 一括GC時のクラスタ内同期処理の必要性

一括GC(Gargabe Collection)は、メモリ中の不要になったデータ構造を回収し、再利用するための処理である。GC中にクラスタ内のPEが独自に動作すると、新たなデータ構造が追加されたり参照関係が変化してGCが正しく行われないため、GC中はクラスタ内の全PEが同期を取って動作する必要がある。

(2) 里親の消費資源量調査時のクラスタ内同期処理の必要性

クラスタ内のゴールは全ていずれかの里親に属し、同じ里親に属するゴールは里親の状態および資源残量によってその実行が制御される。また、一般にゴールはクラスタ内のどのPEでも実行できるため、各PEから里親へのアクセス頻度を減らすために、里親の資源の一部をPEにキャッシュしている。

このため、里親の消費資源量を調査する時には、キャッシュしている資源を一旦里親に書き戻すと共に、里親の消費資源量調査が終了するまでこの里親に属するゴールの実行を防ぐために、クラスタ内の全PEが同期を取って動作する必要がある。

(3) クラスタ内同期処理同士の排他制御

異なる種類のクラスタ内同期処理を同時に実行しようとすると、クラスタ内のPE間で互いに他PEが同期を取ってくれるのを待ってしまうために、デッドロックが生じてしまう。

このため、他の同期処理を実行中か否かを示すフラグを設け、同期処理を行おうとした時点で既に他の同期処理が実行されていた場合には、次の同期タイミングまで同期処理を先送りすることによって排他制御を実現した。

10.9.5 メッセージ追い越し対策

10.9.5.1 メッセージカウント (ready-terminate,start-stop)

クラスタ間ネットワーク通信においてメッセージの追い越しがあった場合、ゴールの実行制御まわりで問題となるのは次の2点である。

- メッセージがネットワーク中にある状態での荘園の終結
- 荘園が状態変化した時の里親への反映

以下それぞれの場合について述べる。

(1) メッセージがネットワーク中にある状態での荘園の終結

- 問題点

Multi-PSIの処理系では、ネットワークメッセージにWTCと呼ばれる値を付加して通信を行なうことで、ネットワーク中にメッセージが残っているか否かを判断している。しかし、Multi-PSIではネットワークメッセージの追い越しがなく、里親が存在する間は必ず荘園も存在するため、里親から荘園方向のメッセージにはWTCを付加する必要がなかった。一方、PIMではネットワークメッセージの追い越しがあるためこの保証がなく、里親から荘園へのメッセージがネットワーク中にある状態で荘園が終結してしまう可能性がある。

- 解決策

里親から荘園方向へのメッセージにもWTCを付加することとした。また、このために里親のWTC不足によってメッセージの送出ができなくなることがあるので、里親レコードに送出できないメッセージを一時つないでおくためのスロットを設け、WTC補給時にメッセージの再送を行なうこととした。

(2) 荘園が状態変化した時の里親への反映

- 問題点

荘園を制御するために、命令を荘園に与えるための機構をコントロールストリームと呼んでいる。いま荘園と里親(他クラスタ内)があり、荘園を動かしたり、止めたりすることを考える。コントロールストリームから入力された"start"及び"stop"命令は、組み込み述語 start_shoen、stop_shoenの呼び出しに変換されるが、それぞれの組み込み述語では処理の終了時に荘園を識別する引数を具体化するため、荘園に対するコントロールストリームからの入力の順序関係は保存される。しかし、荘園の状態変化を里親に伝えるためのクラスタ間ネットワーク通信でメッセージ追い越しがあると、荘園の状態変化の順序と他クラスタにある里親が受け取るメッセージの順序が異なっている可能性がある。このため、start状態の里親にstart、stop状態の里親にstopといった(その時点の里親にとって)論理的に正しくないメッセージが到着する可能性がある。このとき、Multi-PSIの処理系のように、論理的に正しくない(ように見える)メッセージが里親に到着した時には、そのメッセージを捨ててしまうといった処理方式を取っていると、次の例のような問題が発生する。

[例]いま、荘園とその里親がstop状態にあるとして、荘園のコントロールストリームに

start、stop、start、stop、start、stop、start

をこの順序で流したとする。この時、ある里親に到着したメッセージの順序が追い越しによって

stop、stop、start、start、start、start、stop

に変わってしまったとすると、荘園と里親の状態は次のように変化する。

荘園 / 里親の状態変化(1)

荘園の状態変化	stp(初) ->	stat ->	stp ->	stat ->	stp ->	stat ->	stp -> stat
CSへの入力		stat	stp	stat	stp	stat	stp stat

里親の状態変化	stp(初) ->	stp ->	stp ->	stat ->	stat ->	stat ->	stat ->stp
到着メッセージ		stp	stp	stat	stat	stat	stat	stp
		*	*		*	*	*	

stp : stop、 stat : start 、* : 捨てられてしまうメッセージ

つまり、荘園は最終的にstart状態となるのに対し、里親は最終的にstop状態とな ってしまう。

- 解決策

上記のような問題が発生するのは、論理的に正しくないように見えるメッセージを捨ててしまうことに原因がある。そこで、里親に到着するメッセージの内、start状態でのstart、stop状態でのstopなど論理的に正しくないように見えるメッセージを受け取った時はそれらをカウントしておき、次にそのメッセージに対応するメッセージが到着した時点でカウントを減らし、カウントが0の時に論理的に正しいメッセージが到着した時だけ里親の状態を変化させるようにする。先の例では以下のようになり、最終的に荘園と里親の状態が一致する。

荘園/里親の状態変化(2)

荘園の状態変化	stp(初) ->	stat ->	stp ->	stat ->	stp ->	stat ->	stp ->stat
ＣＳへの入力		stat	stp	stat	stp	stat	stp	stat

里親の状態変化	stp(初) ->	stp ->	stp ->	stp ->	stat ->	stat ->	stat ->stat
到着メッセージ		stp	stp	stat	stat	stat	stat	stp
statカウント		0	0	0	0	0	1	0
stopカウント		1	2	1	0	0	0	0

なお、この解決策を採用した場合、コントロールストリームへの入力がstart/stop 同数であると、どんなにstart/stopの数が多くても里親の状態が一回も変化しない可 能性もあるが、これは並列分散処理系の許容範囲内と考えられる。 また、この解決策が全てではなく、例えば、「荘園から里親へ送られたメッセージに対して里親は荘園に必ず応答を返し、荘園は 全ての里親から応答が返ってきたことを確認してから、次のメッセージの送信を行 なう。」 といった方式を取ると、メッセージの追い越しそのものが防げるため、先のような問 題は生じなくなる。 しかし、この方式には

- ◇ クラスタ間のメッセージ数が増大する。
- ◇ メッセージ処理がより複雑になる。

といった問題点がある。

従って、現段階では前者の解決策を採用するが、問題が生じた場合には後者の解決策も含めて他の解決策を検討するものとする。

10.9.5.2 WTC 不足時のメッセージ保留および再送処理

(1) 概要

WTC とは、ネットワーク上のメッセージをも正しくカウントするための、重みつき参照カウントである。特徴は、参照カウントの元締めによる一括管理を必要としないことである。しかし、カウントの元手が無くなれば元締めに増資を要求することになる。

ここでは、WTC を必要とするメッセージを発送しようとした時に、WTC が足りなかった場合の処理、およびWTC が再び供給された時の処理に関して説明をする。これらの処理は、Multi-PSI処理系からVPIMへと機能拡張されている。

WTCは、Multi-PSI処理系において、荘園終了判定時にネットワーク上にメッセージが残っていないか判定するため導入されたものである。%throw_goal メッセージなど一部のメッセージにWTCを付与することで問題が解決されていた。

しかし、PIMの場合は、メッセージの追い越しが問題となる。このため荘園里親間のすべてのメッセージにWTCをつけることにした。

WTC は、荘園の側はこれを発行する立場であり、カウンタがオーバフローしない限りいくらでも発行できるが、里親はWTC が不足した場合には、荘園に対して追加発行を要求しなくてはならない。このために新たに追加した処理は「(2)メッセージ保留処理」以降で説明する。

本概要では、メッセージの追い越しが荘園の終了判定に影響を与える例を具体的に説明しておく。

Multi-PSIでは、1クラスタ - 1PEであるが、PIMは1クラスタ - 複数PEで、連続したメッセージ送信処理も最大限並列に実行できるように設計している。また、メッセージの載るネットワークも、発受信クラスタを定めても、ルートの一意でない一般の場合を想定している（例えばPIM/pの2重ハイパキューブ）。よって、メッセージの追い越しは容易に発生すると思って設計しなければならない。

実際のゴールの実行環境はクラスタ毎に設けた里親にあるので、荘園の状態が変化する時には、荘園と里親の間にネットワークを経由したメッセージが交換される。メッセージの追い越しが問題になる例を挙げよう。里親は荘園から統計情報を問われると %answer_statistics メッセージによりこれに答える。また、終了時には %terminated メッセージを荘園に送る。この二つのメッセージが追い越しにより、入れ替わって、%terminated メッセージが %answer_statistics メッセージより先に着いてしまうことが有り得る。しかし、いずれ来るであろう %answer_statistics メッセージを待つための機構が必要となる。このために、すべての里親荘園間メッセージにはWTC をつけることにした。

(2) メッセージ保留処理

- 処理

  メッセージの種類に応じたメッセージサスペンドレコードを割り当て、メッセージ本体をコピーして、里親レコードのWTC 不足メッセージキューに、リンクし、さらに %request_WTC メッセージを発行する。

  WTC は、%terminated メッセージの分と %request_WTC メッセージの分を確保したあとの、残

りの分に関して、残量をチェックして、WTC 不足を検出する。すなわち、%request_WTC メッセージおよび%terminatedメッセージは送信保留にはならない。

以下には、クラスタ間のメッセージの内、里親から取り出したWTC を付けるメッセージの種類を上げる。

- Multi-PSIでもWTC が付いていたメッセージ（里親間）
  %throw_goal, %unify :メッセージを受信したクラスタでゴールが作られる（可能性がある）メッセージ。
- PIM用に新規にWTC を付けることにしたメッセージ（里親→荘園）
  %ready, %request_resource, %return_resource, %answer_statistics, %request_WTC,
  %terminated :メッセージ追い越しの結果、荘園を終了させる可能性があるメッセージ。

(3) メッセージ再送処理

%supply_WTC メッセージに限らず、WTC のついたメッセージを受信した時、送信の保留されたメッセージがあれば、再送を試みる。

ただし、%unify メッセージに限り、ユニファイの対象となる変数が、メッセージのサスペンド中に外部参照から、具体値に書き変わっている可能性もあるので、ユニファイのDコードを実行する。

(4) アボート関連処理

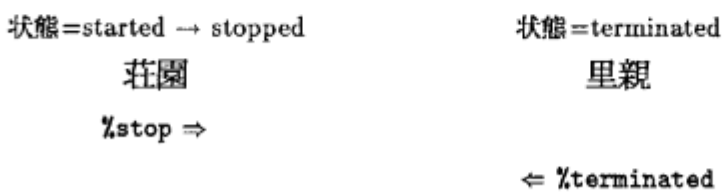
里親から荘園に送るメッセージの中には、%return_resource メッセージ、%answer_statistics メッセージの様に、そのメッセージをなくしてしまうとシステムの統計情報の収集に誤りを生じるものがある。よって、里親が%abort メッセージを受信して、終了処理を行なう場合にも、送信保留中のメッセージがあれば、%supply_WTC メッセージが来るはずなので、終了せずに待つ。

10.9.5.3 次世代里親によるメッセージ不正受信

荘園/里親間を行き来するコントロールメッセージについても、メッセージの追い越し現象を考慮しておく必要があり、VPIM においても、コントロールメッセージをその種別に応じてカウントしておく方式によって、追い越し現象に対応している。しかし、このカウント方式だけでは不十分な場合があり、新たなる方式を実装を行なった。

10.9.5.3.1 問題となる状況

あるクラスタの里親が終結後、すぐにゴールが投げられてきて新たな里親が生成されるような場合、以下のような状況となることがある。



荘園の状態が started から stopped に変わったため、%stop メッセージが送信される。同時に里親が

終結したため、%terminatedメッセージが送信される。

状態=stopped

荘園 (里親) 消滅

②

%stop ⇒

⇐ %terminated

里親は消滅し、里親からの%terminatedメッセージにより、荘園も里親が消滅したことを認識する。%stopメッセージはまだネットワーク中を飛んでいる。

状態=stopped 状態=started

荘園 里親 ⇐ ゴール

③

%stop ⇒

⇐ %ready

クラスタに新たにゴールが投げられてきて、新たに里親が生成される。これに伴って、%readyメッセージが送信される。なお、新たに生成される里親の状態は started である。%stopメッセージはまだネットワーク中を飛んでいる。

状態=stopped 状態=stopped

荘園 里親

④ %stop ⇒

⇐ %ready

%stop ⇒

%stopメッセージが新たに生成された里親に届き、里親は stopped となる。また、荘園が%readyメッセージを受信して里親の生成を認識するが、この時荘園は stopped なので、里親に %stopメッセージを送信する。

状態=stopped 状態=stopped + 1

⑤ 荘園 里親

%stop ⇒

荘園からの %stopメッセージが里親に到着し、里親の状態は stopped + 1 となる。

このような状態になると、荘園が再起動して stopped から started となっても、里親は stopped + 1 から stopped となるだけで再起動しなくなり、荘園から里親をコントロール出来なくなってしまう。

10.9.5.3.2 対策

この状況への対策として以下のような方式を検討した。

1. 荘園がコントロールの対象とする里親を識別するために、里親に一連の識別番号を付加し、コントロール対象の里親か否かを区別する方式。
2. 荘園がクラスタから送られてくるメッセージをもとに、そのクラスタでの里親の状態を判断し、必要に応じてコントロールメッセージを再送することにより、有限時間内に里親の状態を正しく保つ方式。

このうち、1.の方式については里親識別番号のオーバーフロー等が問題となるため、2.の方式を採用した。この方式について以下に述べる。

この方式は、荘園が %stop メッセージを送信したクラスタを覚えておき、そのクラスタから届く %ready 及び %terminated メッセージと、里親が存在しないクラスタに到着した %start および %stop メッセージの返信 (それぞれ%NAKstart/%NAKstop と呼ぶ) に基づいてクラスタごとに里親の状態を判断し、必要に応じて %start/%stop メッセージを再送することにより、有限時間内に里親の状態を正しく保とうとするものである。

このため、この方式では新たに次のデータ構造およびメッセージを導入する。

(1) stop/start ビットテーブル

クラスタ数個のビットを持ったテーブル。各ビットの ON/OFF は、次のことを意味する。

ビットが ON:

ビットに対応するクラスタに里親が存在すれば、その里親は有限時間内に stop する。

ビットが OFF

ビットに対応するクラスタに里親が存在すれば、その里親は有限時間内に start する。

(2) %NAKstop/%NAKstart メッセージ

%stop/%start メッセージが里親の存在しないクラスタに到着した時に、荘園に送り返されるメッセージ。従来は start/stop を区別せず、%return_WTC メッセージとしていた。

また、実際のオペレーションは次のようになる。

(a) 荘園の状態が変化した時

(i) 荘園の状態が started から stopped に変化した時

荘園が里親の存在を認識しているクラスタの内、stop/start ビットが OFF のクラスタに対して %stop メッセージ を送信する。同時に、%stop メッセージを送信したクラスタに対応する stop/start ビットを ON にする。

(ii) 荘園の状態が stopped から started に変化した時

荘園が里親の存在を認識しているクラスタの内、stop/start ビットが ON のクラスタに対して %start メッセージ を送信する。同時に、%start メッセージを送信したクラスタに対応する stop/start ビットを OFF にする。

(b) 新たに里親が生成されたことを認識した時

(i) 荘園の状態が stopped の時

stop/start ビット = OFF ならば %stop メッセージを送信し、ビット = ON とする。

stop/start ビット = ON ならば 何もしない

(ii) 荘園の状態が started の時

stop/start ビット = OFF ならば 何もしない。

stop/start ビット = ON ならば %start メッセージを送信し、ビット = OFF とする

(c) %NAKstop/%NAKstart を受信した時

(i) %NAKstopを受信した時

stop/startビット = ON ならば ビット = OFF とする。また、荘園の状態が stopped で、そのクラスタに里親が存在することを認識していた場合には %stop メッセージを送信し、ビット = ON とする。

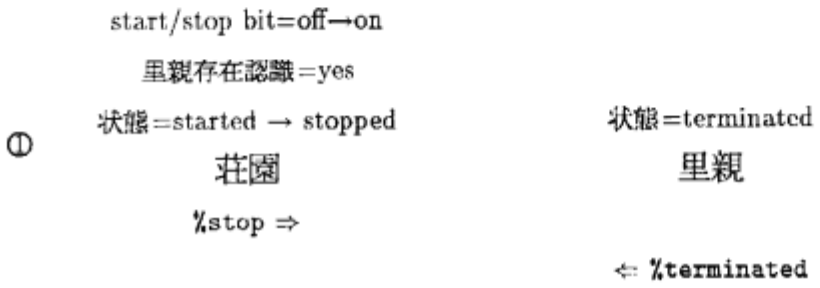
stop/startビット = OFF ならば そのクラスタに %stopメッセージを送信する。

(ii) %NAKstartを受信した時

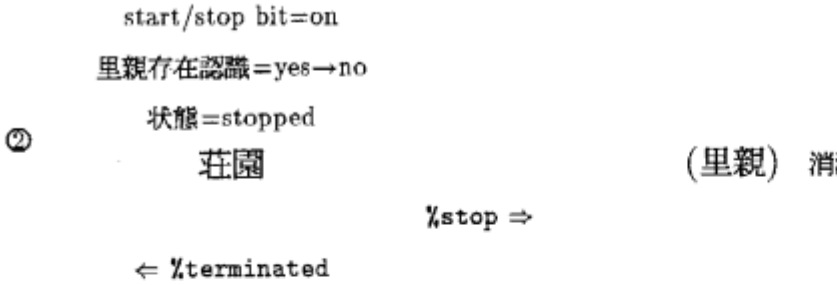
stop/start ビット = ON ならば そのクラスタに %startメッセージを送信する。

stop/start ビット = OFF ならば ビット = ON とする。また、荘園の状態が started で、そのクラスタに里親が存在することを認識していた場合には %start メッセージを送信し、ビット = OFF とする。

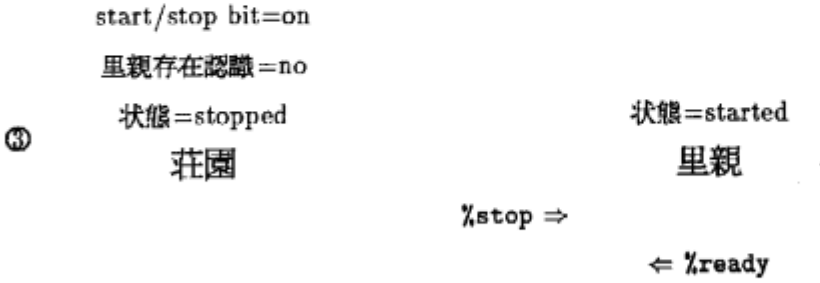
これを、先の例で示すと、



荘園の状態が started から stopped に変わったため、%stop メッセージが送信される(stop/start ビットはonになる)。同時に里親が終結したため、%terminated メッセージが送信される。



里親は消滅し、里親からの%terminated メッセージにより、荘園も里親が消滅したことを認識する。%stop メッセージはまだネットワーク中を飛んでいる。



クラスタに新たにゴールが投げられてきて、新たに里親が生成される。これに伴って、%ready メッセージが送信される。なお、新たに生成される里親の状態は started である。%stop メッセージはまだネットワーク中を飛んでいる。

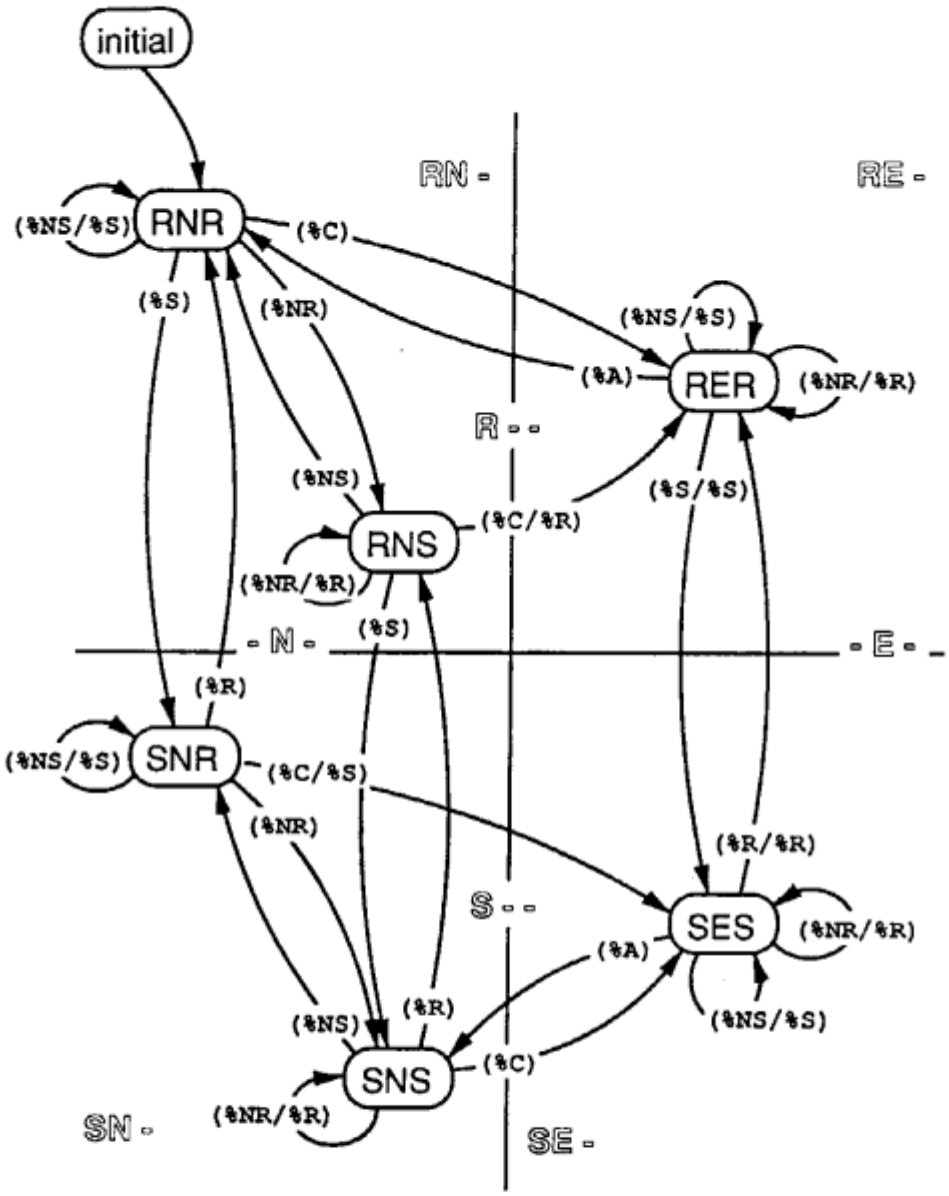


図 10-11: ストップスタートビットによる荘園の状態遷移図

start/stop bit=on

里親存在認識=no→yes

状態=stopped

④ 荘園

状態=stopped

里親

%stop ⇒

⇐ %ready

%stop メッセージが新たに生成された里親に届き、里親は stopped となる。また、荘園が %ready メッセージを受信して里親の生成を認識するが、この時上記[新たに里親が生成されたことを認識した

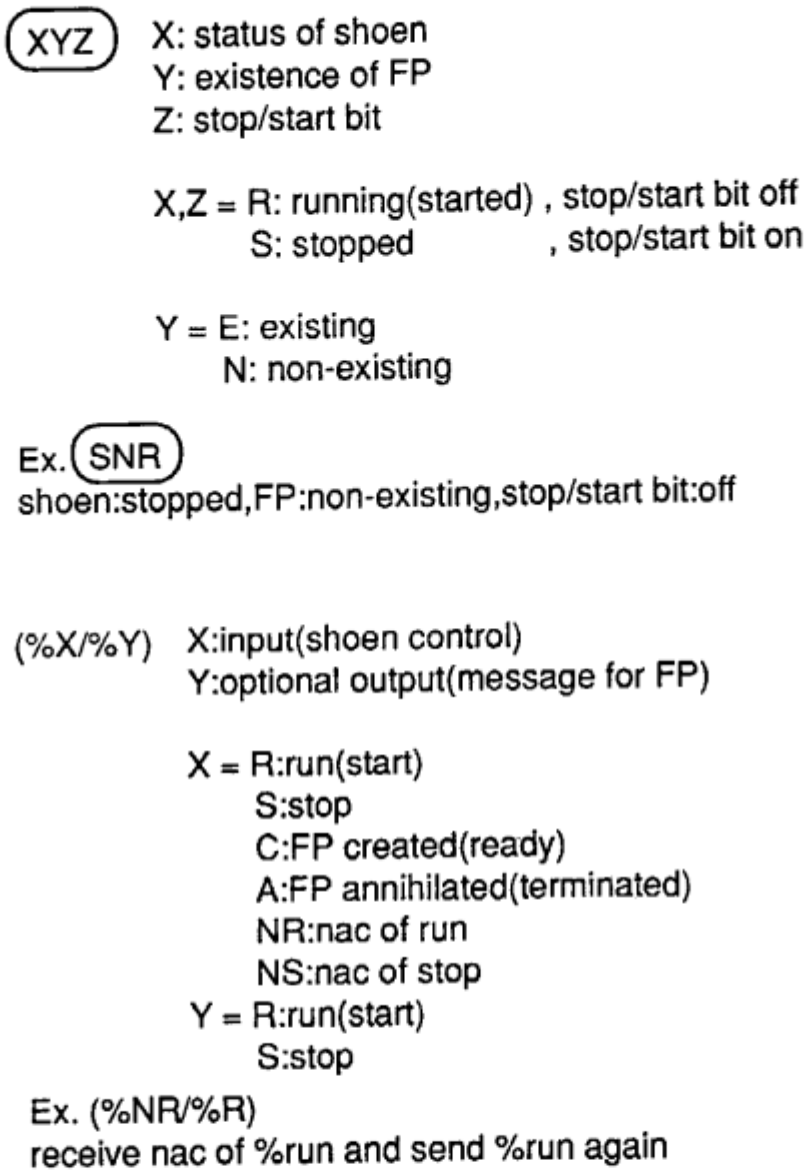


図 10–12: 状態遷移図で用いている記号

時]の処理により荘園は %stop メッセージを送信しないため、荘園と里親の状態が一致する。

荘園レコードにおける、stop/start ビット、荘園の論理状態、里親の有無、に関する状態遷移を図 10–11 に示した。図 10–12 は記号の説明である。

図中の記号は、スペースの都合本文中と異なるものを使っているので、以下の通り読み換えて欲しい。

run → start

running → started

created → ready

anihilated → terminated

図中で状態は、荘園の論理状態、里親の有無、stop/start ビットの順に 3 文字の組で表現している。具体的には記号の説明図を参照のこと。

状態遷移を引き起こす入力および、いくつかの遷移で生ずる出力はメッセージのペアの形で（%Input/%Output）の様に表し、それぞれの遷移を表す矢印の上においている。出力がない場合は（%Input）としている。

状態遷移を引き起こす入力に関して注意しておく。状態遷移図中の 6 種類の入力の内%R,%S,%C,%A の 4 種類はメッセージ風の表記をしているが、メッセージカウントなどの手法ですでにシーケンシャライズされた後の、本当にその操作が必要になった場合の操作を示している。一方、%NR,%NS は、まさにそのメッセージを受信した時の処理を表している。

10.9.6 メッセージ送受信処理における排他制御と里親の参照数管理

10.9.6.1 里親の参照数管理

原則は、ゴールの数をチャイルドカウントの名の元に管理し、これがゼロになったら里親の仕事は終了である。

また、里親処理のメインパスは、「チャイルドカウントをテストしたが、ゼロでないので仕事を続行」である。

ここで、ゴール以外の「仕事」もチャイルドカウントに反映させるとメインパスの負担にならなくなる。

リダクションの切れ目において終了の判断に出てくるゴール以外の仕事：

- 子荘園
- 統計情報収集で先送りされた同期処理
- 荘園からのメッセージ受信処理
- 統計情報収集の依頼受付から結果報告までの処理

チャイルドカウントのインクリメント／デクリメントのタイミングは処理によっていろいろ。

アボートされた時は、残っている仕事がゴールだけなら、輸入表からは削除して、残りのゴールがスケジュールされるのだけを待つ。このため、ゴール以外の仕事は 専用のカウンタも持つ必要がある。子荘園は子荘園リンクポインタがEOLか否かで分かる。その他の仕事は、まとめて一つのカウンタを用いる。

10.9.6.2 メッセージ受信処理の延長にあるメッセージ送信処理

荘園里親の処理は、荘園組み込み処理の延長と、メッセージ受信処理の延長が主である。

これらの処理の中でまたメッセージを送信することが多いのだが、メッセージ受信処理の中で、メッセージを送信する場合、同一の外部参照IDに関する送受信の場合里親の排他制御や参照数管理に注意が必要。

注意すべき処理：

- 里親の終了判定
  荘園からのメッセージの受信処理では、処理の間チャイルドカウントを1だけ上げているので、処理の最後にチャイルドカウントを下げる必要があるが、この時里親の終了を検出すると、%terminated を送信する必要がある。
- サスペンデッドメッセージのリジューム
  WTCが不足して送信を中断していたメッセージは、%supply_WTC を受信した時に、その処理の延長で再送が試みられる。

10.10 例外処理

例外は一般にゴールの実行に伴って発生するため、例外発生時にはゴールが所属している里親が分かる[8]。このため、例外発生時の情報(コード、引数など)は適当な形のベクタにまとめられて里親から荘園へのメッセージとして送られる。また、それぞれの例外には固有のタグが決められており、このタグもメッセージに含まれている。

例外メッセージを受信した荘園では、まず例外メッセージ中のタグを見てそのタグが荘園でマスクされているか否か(つまり、その荘園で受け付けるか否か)を判断する。

そして、その例外が荘園でマスクされていなかったら、その荘園のレポートストリームから例外として報告される。また、その例外がマスクされていた場合には、例外情報は荘園の親里親を通じてさらに祖先の荘園に送られる。

この例外のマスクパターンはビットマップの形式で荘園生成時に引数で与えられる。

例外メッセージ中には、例外を発生させたゴールに代えて実行する代替ゴール を指定するために、コード及び引数ベクタを与えるための変数を処理系が付加する。例外報告を受けとった(荘園の)ユーザは、これらの変数に新たにコードと引数ベクタをバインドすることによって代替ゴールを実行することが出来る。なお、例外を発生させたゴールは実行が終了したものとして捨てられる。

10.10.1 例外処理一覧

例外タグは32ビット長であり、これを図 10–13 に示すように言語で定義するもの, PIMOS が定義するもの, ユーザが定義できるものの 3 つの領域に分けて用いる。

また、KL1言語で定義している例外の種類とビット位置は、表10–1に示される。

なお、組み込み述語 raise を用いて例外を起こす場合には、引数で例外タグを指定するので, 例外タグ

[8] GC中に発見された永久中断ゴールなどゴールの実行に伴わない例外でもゴールレコード中の里親ポインタから里親が分かる。

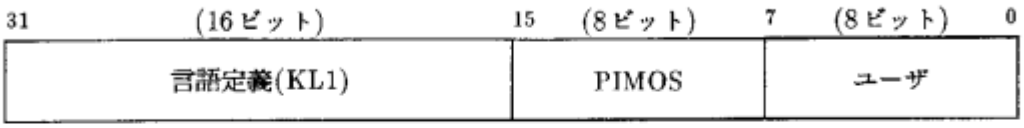


図 10–13: 例外タグの形式

表 10–1: KL1 言語定義の例外

例外の種類	ビット位置	意味
illegal_input	16	入力引数型不正
range_overflow	17	入力引数範囲不正
out_of_bounds	18	インデクス範囲不正
integer_overflow	19	整数演算桁あふれ
integer_zero_division	20	ゼロによる除算
invalid_floating_point_operation	21	浮動小数点数無効演算
arity_mismatch	22	引数個数不整合
undefined_module	23	存在しないモジュールの呼びだし
undefined_predicate	23	存在しない述語の呼びだし
illegal_merger_input	24	マージャ入力不正
reduction_failure	25	全候補節の失敗
unification_failure	26	ユニフィケーション失敗
perpetual_suspension	27	永久中断ゴール検出
merger_perpetual_suspension	28	永久中断マージャ検出
trace	29	トレース
spy	29	スパイ
(reserved)	30	
etc	31	その他

のビット位置は決まっていない.

10.10.2 例外メッセージ一覧

例外メッセージは荘園のレポートストリームに流すメッセージのひとつであり、全て以下のような形式の5要素ベクタで表現する。

{shoen#exception, ExpNumber, ExpInfo, NewCode, NewArgv}

ここで、**shoen#exception**はこれが例外メッセージであることを示す番号、**ExpNumber**は例外の種類を表す番号、**ExpInfo**は例外に関する情報を保持するベクタ、**NewCode**と**NewArgv**は例外を

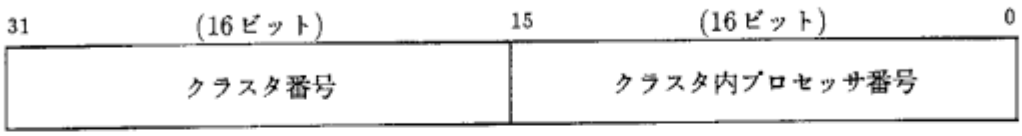


図 10-14: 例外発生プロセッサ情報の形式

起こしたゴールに代わって実行すべき述語のコードと引数ベクタを指示する(PIMOS側で具体化する)ための変数である。

このうち、例外情報ExpInfoの仕様は例外の種類毎に異なるので、それぞれを順に説明する。なお、このベクタ内における各種の情報の順番は、分かりやすいように、また、処理し易いように、全ての例外情報でなるべく共通の順番、即ち、以下の順に並べることとする。

- **ClsAndProc**
  これは例外が発生したクラスタ番号とプロセッサ番号の情報で、図10-14に示されるような形式をしている。
- **Predicate**
  これは例外を起こした述語を識別するための情報で、具体的には以下の4種類で表現される。

  Code KL1定義の述語で発生した
  {Code, CallerMod, CallerOffs} KL1定義の組込述語で発生した
  {KlbID, OriginMod, OriginOffs} ..... ファーム定義の組込述語で発生した
  {KlbID, OriginMod, OriginOffs, CallerMod, CallerOffs}
  ..... KL1定義の組込述語から呼ばれたファーム定義の組込述語で発生した

  ここで、**Code**はKL1定義の述語を指定するためのコードデータである。**KlbID**はファーム定義の組込述語を識別するための番号で、組込述語に対応するKL1-B命令に付与されている番号KL1B-instruction-IDを使用する。この番号はマルチPSIにおけるオペコードに相当するものである。**OriginMod**と**OriginOffs**はファーム定義の組込述語で例外が発生した時に生成されるもので、この組込述語がどのKL1定義の述語のどの部分から呼び出されたものであるかを示すものである。具体的には、KL1のコードモジュール内における組込述語の位置、即ち、その組込述語に対応する機械語命令列の先頭を指定する情報で、モジュールポインタとモジュール先頭からの相対位置(オフセット)の2つから成る。**CallerMod**と**CallerOffs**はKL1定義の組込述語を実行中に例外が発生した時に生成されるもので、この組込述語がどのユーザー定義の述語のどの部分から呼び出されたものであるかを示すものである。具体的には、ユーザー定義のコードモジュール内からKL1定義の組込述語を呼び出すためのKL1-B命令(enqueue系命令)の位置、即ち、そのKL1-B命令に対応する機械語命令列の先頭を指定する情報であり、モジュールポインタとモジュール先頭からの相対位置(オフセット)の2つから成る。
- **ArgVect**

これは例外を起こした述語の引数情報で、具体的には全引数を要素とするベクタである。

- **ArgPos**
  これは例外を起こした述語の引数のうち原因となった引数の位置を示す情報で、具体的には、0オリジンの番号で指定することにする。

(1) Illegal Input 例外

これはボディ組込述語の入力引数に正しくない型のデータを入力したという例外であり、以下のように4要素ベクタの例外情報を伴う。

{ClsAndProc, Predicate, ArgVect, ArgPos}

(2) Range Overflow 例外

これはボディ組込述語の入力引数(ベクタ等の要素位置を指定するものを除く)に正しくない範囲の値を持つデータを入力したという例外であり、以下のように4要素ベクタの例外情報を伴う。

{ClsAndProc, Predicate, ArgVect, ArgPos}

(3) Out of Bounds 例外

これはボディ組込述語の入力引数のうちベクタ等の要素位置を指定するものに正しくない範囲の値を持つデータを入力したという例外であり、以下のように4要素ベクタの例外情報を伴う。

{ClsAndProc, Predicate, ArgVect, ArgPos}

(4) Integer Overflow 例外

これはボディの整数演算の組込述語において計算結果が表現できる範囲を越えたという例外であり、以下のように3要素ベクタの例外情報を伴う。

{ClsAndProc, Predicate, ArgVect}

(5) Integer Zero Division 例外

これはボディの整数除算/剰余算の組込述語においてゼロで割ろうとしたという例外であり、以下のように3要素ベクタの例外情報を伴う。

{ClsAndProc, Predicate, ArgVect}

(6) Arity Mismatch 例外

これはapply系組込述語、create_shoen系組込述語において、指定された述語の引数個数と引数ベクタの要素数が合わないという例外であり、以下のように3要素ベクタの例外情報を伴う。

{ClsAndProc, Predicate, ArgVect}

(7) Illegal Merger Input 例外

これはマージャの入力変数に不正なデータを具体化したという例外であり、以下のように4要素ベクタの例外情報を伴う。

{ClsAndProc, Predicate, Input, Merger}

ここで、PredicateはKlbIDを含む形式のものであるが、この例外ではKlbIDの値は常にmerge組込述語を意味する番号とする。また、Inputは検出された不正データ、Mergerは不正データが入力されたマージャ変数である。

(8) Reduction Failure 例外

これはKL1で定義された述語においてゴールリダクションが失敗したという例外であり、以下のように3要素ベクタの例外情報を伴う。

{ClsAndProc, Predicate, ArgVect}

(9) Unification Failure 例外

これはボディのユニフィケーションに失敗したという例外であり、以下のように3要素ベクタの例外情報を伴う。

{ClsAndProc, DataX, DataY}

ここで、DataXとDataYはユニフィケーションしようとした2つのデータである。

(10) Raised 例外

これはraise組込述語が実行されたことにより発生する例外であり、以下のように3要素ベクタの例外情報を伴う。

{ClsAndProc, Info, Data}

ここで、InfoとDataはraise組込述語の引数に指定されたものである。

(11) Perpetual Suspension 例外

これは永久中断ゴールを発見した時に発生する例外であり、発見のタイミングにより以下の4要素または5要素の例外情報を伴う。

{ClsAndProc, Type, Predicate, ArgVect}

{ClsAndProc, Type, Predicate, ArgVect, Collector}

ここで、Typeは永久中断検出の種類を示す番号である。また、Collectorは永久中断を実時間GCで検出した場合に付加される情報で、永久中断ゴールが具体化されるのを待っていたHOOK変数を回収しようとした述語の情報である。この形式はPredicateと同じである。

(12) Merger Perpetual Suspension 例外

これは永久中断マージャ、即ち入力変数を捨てられてしまったマージャを発見した時に発生する例外であり、発見のタイミングにより以下の4要素または5要素ベクタの例外情報を伴う。

{ClsAndProc, Type, Predicate, Inputs}
{ClsAndProc, Type, Predicate, Inputs, Collector}

ここで、Typeは永久中断検出の種類を示す番号である。また、PredicateはKlbIDを含む形式のものであるが、この例外ではKlbIDの値は常にmerge組込述語を意味する番号とする。Inputsは捨てられたマージャ入力変数を格納したベクタである。Collectorは永久中断を実時間GCで検出した場合に付加される情報で、永久中断ゴールが具体化されるのを待っていたHOOK変数を回収しようとした述語の情報である。この形式はPredicateと同じである。

(13) Trace 例外

これはKL1レベルのデバッガのために用意された例外であり、トレースモードのゴールのリダクションを行なった時に発生する。この例外は以下のように4要素ベクタの例外情報を伴う。

{ClsAndProc, ListOfTraceID, SubGoalInfoList, ClauseInfo}

ここで、ListOfTraceIDは例外要因となった全てのトレース指定に付けられたIDのリストで、現在は要因がapply_tracingの1つしかないので、その第3引数に指定されたTraceIDを要素とする長さ1のリストである。また、SubGoalInfoListはサブゴール情報SubGoalInfoのリストである。ClauseInfoはコンパイル時にクローズに付加しておいたID番号で、コミットしたクローズを識別するために使用する。

サブゴール情報SubGoalInfoは個々のサブゴールに関する情報で、その種類により以下のようなベクタで表現される。

{KlbID, DataX, DataY} …… ユニフィケーション
{KlbID, ArgVect} …… ファーム定義の組込述語
{KlbID, Code, ArgVect} …… KL1定義の述語の呼び出し
{KlbID, Code, ArgVect, Priority} …… KL1定義の述語の呼び出し(優先度指定)
{KlbID, Code, ArgVect, Cluster} …… KL1定義の述語の呼び出し(クラスタ指定)
{KlbID, Code, ArgVect, Processor} …… KL1定義の述語の呼び出し(プロセッサ指定)

ここで、各ベクタの最初の要素KlbIDはサブゴールの種類を識別するための番号で、ファーム定義の組込述語を含む全KL1-B命令に付与されている番号KL1B-instruction-IDを使用する。この番号はマルチPSIにおけるオペコードに相当するものである。Codeはサブゴールが KL1定義の述語を実行するものの場合に付加されるもので、その述語を指すコードデータである。ArgVectはサブゴールの引数を格納したベクタ、DataXとDataYはユニフィケーション対象の2つのデータである。Priorityはenqueue_with_priorityによりサブゴールが生成された時に付加される情報で、その時指定された論理実

行プライオリティである。**Cluster** は enqueue_to_cluster または enqueue_resident_to_cluster によりサブゴールが生成された時に付加される情報で、その時指定されたクラスタ番号である。**Processor** は enqueue_to_processor または enqueue_resident_to_processor によりサブゴールが生成された時に付加される情報で、その時指定されたプロセッサ番号である。

(14) Spy 例外

これは KL1 レベルのデバッガのために用意された例外であり、スパイモードのゴールがスパイ対象の述語を実行するサブゴールを生成した時に発生する。この例外は以下のように5要素ベクタの例外情報を伴う。

{ClsAndProc, SpyID, SubGoalInfo, Module, Offset}

ここで、**SpyID** は例外要因となったスパイ指定に付けられた ID で、apply_spying の第3引数に指定されたものである。また、**SubGoalInfo** は Trace 例外の場合と同じ仕様のサブゴール情報である。**Module** と **Offset** は報告されるサブゴールがどの述語のどの部分から呼び出されたものであるかを示すものである。具体的には、コードモジュール内からサブゴールを呼び出すための KL1-B 命令 (enqueue 系命令) の位置、即ち、その KL1-B 命令に対応する機械語命令列の先頭を指定する情報であり、モジュールポインタとモジュール先頭からの相対位置 (オフセット) の2つから成る。

第 11 章

クラスタ内一括 GC

執筆担当者: 今井

本章では、通常実行を一時停止し、不要なメモリ領域を回収する一括ガーベジコレクション(GC)について解説する。

11.1 概要

KL1のような副作用を許さない言語の実装においては、不要となったメモリ領域を回収するガーベジコレクション(GC) の処理効率が特に重要である。KL1処理系では、Prolog処理系のように、スタックを仮定した効率の良いGC方式が採用できないこともあり、その効率向上のための手法は、かなり複雑な最適化手法を採用している。

KL1プログラムでは、ほとんどの未定義変数に対する参照パスが、

- その変数に値を書込むパス
- 値を読み出すパス

の2本であるという性質[1]を持つ。これにより、多くのセルが、低コストでかつゴミになった時点(値を読み出した時点)で回収できる MRB (Multiple Reference Bit) 方式が有効である(4.4節参照)。MRB方式による即時GC方式は、Multi-PSIのKL1処理系に実装され、その効果が確認されている。ただし、MRB 方式では、

- その変数に値を書込むパスが一本
- 値を読み出すパスが複数

というような状況では回収できないため、メモリを使い切った時に、一度通常実行を中断し、メモリ中で生きているセルと、ゴミになったセルを仕分けする操作(一括 GC) との併用が必須となる。

なお、他のクラスタにゴールを投げることで、あるデータは他のクラスタから参照されることになるが、ある瞬間に生きているデータであるかそうでないかを判定するためには、全てのクラスタで同期をとる必要がある。しかし、そのためのオーバヘッドが大き過ぎることが予想されるため、クラスタ間に

[1]即ち、1ゴール対1ゴールの通信が多い。

跨るデータは、輸出表(9.1節参照)という間接テーブルを経由し、クラスタが独立して GC できるような仕組みを設けている。

本章で呼ぶ一括GCとは、

- クラスタ毎に独立し
- クラスタ内の全てのPEが同期して通常実行を停止し
- クラスタ内の全てのPEが独立にGC処理を行い
- クラスタ内の全てのPEが同期してGC処理を完了して、通常実行に復帰する

のような処理である。

また、あるゴールが決して再開されない状態(永久中断状態)に陥っていることの検出も、一括 GC 時に行なわれる。

以下、この一括GCの詳細について述べる。

11.2 クラスタ内並列実行メカニズム

11.2.1 設計方針

一括GCは本質的に並列実行しても高速化しにくい処理であるが、次のような点に着目してその高速化を計った。

(1) GC 操作におけるメモリアクセスを削減すること

(a) コピー方式

メモリ空間を2分割し、片方の半空間がなくなった時点で、生きているセルをもう片方の半空間にコピーする方式を採用する。これは、メモリ空間全体をアクセスするのではなく、生きているセルにのみにアクセスすれば良いためである。

(b) MRBを利用したマーク削減

コピー方式では、同一のセルを多重にコピーしないようにするために、旧領域のコピー済みセルにはコピー先の新領域のアドレスを書き込むマーク処理が必要がある。このマーク処理は、コピーしようとするセルが単一参照であることが分かれば削減できる処理である。そこで、前述のMRB情報などを用いて、単一参照であることが保証されている場合は、マーク処理を省略するという最適化を導入した。

なお、Multi-PSI で採用されている MRB メンテナンスは、メモリアクセスを増大させることになるので、実装しないことにした。

(2) 一括 GC を複数の PE で並列に実行すること

一括GCを高速化するために、GC処理を複数のPEで並列に実行する。GCルートを PE 毎に分配し、各 PE はそれを自分のルートとしてコピー操作を始めることにより、並列実行が可能となる。

(3) 共有データアクセスを削減すること

GCを並列実行する際に共有データとなるものの代表的な例として、新領域をどこまで使ったかというポインタ (GlobalB) がある。従来の逐次処理における移動法を単純に並列化すると、新領域に1つのデータ構造をコピーする毎に、このポインタを排他制御しながら更新しなければならず、この更新がネックになることが予想される。

そのため、このポインタをある単位 (HEU,後述) で更新することで、更新頻度を抑える。

(4) 各PE間の負荷を分散すること

クラスタで一つの未スキャン領域プールを設け、未スキャン領域が増えたPEがこのプールに領域を入れ、アイドルになったPEはこのプールから取り出すことで、負荷を分散する。これにより、初期分配だけに依存しないで、動的に負荷を均等化することができる。

11.2.2 コピー方式による GC

ここでは、ベースとなったコピー方式 による GC について簡単に述べる。コピー方式では、ヒープ領域を二等分し、通常実行時にはそのうちの片方のみを使う。片方のヒープを使い切った時点で、その中にある全ての生きているセルをもう一方のヒープにコピーする。

この方式の特長としては、GC時に生きているセルのみをアクセスすれば良いことがあげられる。この特長は、PEが個々にキャッシュメモリを持ち、バス結合される共有メモリマルチプロセッサでは、バストラフィックの点で特に有利である。

図11-1に、この基本的なアルゴリズムを記す。ここで、S は現在のスキャン(scan)ポイントを意味し、Bは、新領域をどこまで使ったかを示す底(bottom)を意味する[2]。

ルートをコピーした後、Sをインクリメントしながらその指している先をチェックする。ここで、Sが指している先が、アトミックデータであれば何もしない。旧領域へのポインタであれば、その旧領域に置かれたセルを新領域にコピーする。このための領域は、Bを進めることで確保する。この際、旧領域には、多重にコピーされることを防ぐためにマークと移動先アドレスを書き込んでおく。ただし、旧領域が既にマーク済みであった場合、その移動先アドレスを S の先に書き込む。

GC処理は、S と B が一致した時点で終了する。

11.2.3 並列化方式

コピー方式のアルゴリズムには、それぞれのセルのコピー、スキャンは基本的に並列に行なうことができる。ただし、これをナイーブに行なうと、S および B の更新に、毎回ロックを掛ける必要があり、充分な並列動作が見込めない。

これを防ぐ方法として、新領域を PE 台数で均等割し、それぞれのPE にローカルな領域として分配し、各 PE は、ローカルに S および B を管理するという方法がある。このような単純な方法でも、旧

[2]従来、ヒープトップという言葉を用いていたが、Baker のオリジナル文献に合わせて Bottom とした。

```
copy(P) {
    — P は旧領域へのポインタである —
    Temp := oldheap(P);
    if (Temp が新領域の移動先であれば)
        newheap[S] := Temp;
    else {
        newheap[S] := B;
        Arity := arity(Temp);
        — 旧領域に移動先を書き込む (マークする) —
        oldheap[P] := B;
        for (i = 1; i ≤ Arity ; i++)
            — 旧領域の内容を、新領域にコピーする —
            newheap[B++] := oldheap[P++];
    }
}

main() {
    S := B := 新領域の先頭アドレス;
    for (i := 1; i ≤ # roots; i++ )
        copy( root(i) );
    while (S < B) do {
        P := newheap[S];
        if (P が旧領域へのポインタ)
            copy(P);
        S++;                                          }
}
```

図 11–1: コピー方式のアルゴリズム

領域のマークづけさえ排他的に行なえば、多重コピーなどの問題は起こらない。しかしながら、次のような問題がある。

- 許される最大の大きさの構造体が、ヒープ片面の PE 台数分の 1 に制限されてしまう。
- PE毎のコピー速度が異なることが予想され、ある PE が自分のローカルヒープを使い切った時に、他の PE から領域をもらう方法の実装が複雑になる。
- PE間でのコピー自体の負荷分散が難しい。

そこで、GC中に必要になった新領域を、

- PEローカルな領域として何らかの方法で動的に割り付ける仕組み

- GC中にもPE間で負荷を分散できるような仕組み

うな方法が必須であると考え、次のような方式を設計し、実装した。

アルゴリズムを、図11-2, 図11-3 および 図11-4に示す。なお、このアルゴリズムの中で使われているマクロの定義は次の通りである。

is-at-top-of-LDU(x) ≡ ((Top-of-LDU(x) mod LDU) = 0)

is-at-top-of-HEU(x) ≡ ((Top-of-HEU(x) mod HEU) = 0)

Bottom-of-HEU(x) ≡ (Top-of-HEU(x) - HEU)

Bottom-of-prev-LDU(x) ≡ (Top-of-LDU(x) - 1)

新領域は、必要になった時点で GlobalB (新領域をどこまで使ったかを示すポインタ) を「ヒープ獲得ユニット」 (Heap Extension Unit : **HEU**) 単位で更新することで、PE ローカルな領域として確保される (「ページ」とか HEU とか呼ぶ[3])。

また、ページを使い切るために、サイズ毎にページを割り付け、各構造体は、割り当て時に 2 の冪乗に丸めて割付けられている (4.3節参照)。各PEは、各サイズ毎に B と S のポインタを持つ。

ここで、PE(j) のサイズ k 用のページの底を示すポインタを、B_k^j で表現する[4]。

あるページが全てコピーで埋まった時、すなわち、B が次のページの先頭に到達した時[5]、新たに GlobalB を更新してページを確保する。ここで、後に連続してスキャンするために、今使い切ったページと確保したページとの間を、何らかの方法で繋げてやる必要がある。実際には、物理的には一切繋がらず、図11-3の checkB にて、未スキャンページを共有プールに入れるかどうかの判定を行い、S と B が 2ページ以上離れる場合は、今までSが指していたページを共有プールに入れる。

もし、PE(j) がアイドルになったら、すなわち $\forall k(S_k^j = B_k^j)$ のような条件になったら、その PE は、共有プールから未スキャンページを取り出し、スキャンを行なう。このようにして、PE間での負荷分散が行なわれる。GCは、$\forall j,k(S_k^j = B_k^j)$ かつ 共有プールが空になった時点で終了する。

ここまでで述べた方式は、それだけで正しく動作するものであるが[6]、負荷分散効率をあげるために、ヒープ獲得ユニットと独立した単位である「負荷分散ユニット」 (Load Distribution Unit: **LDU**) [7]を導入している。これは、ヒープ獲得ユニットを大きくすると GlobalB の更新頻度が押えられる反面、負荷分散の機会が減ってしまうためで、この独立した単位の導入により、GlobalB の更新頻度を抑えたまま、負荷分散の機会を増やすことができるようになった。図11-3のcheckSにおいて、SがLDU境界に達した時に、LDUを単位として共有プールに未スキャン領域を入れる (図11-5)。

---

[3] VPIM ver1.0では 256語である。

[4] ただし、図中のアルゴリズムは、ある特定のPEの実行方式を記したものであるから、PE 番号を示す j は、特に明記しない。

[5] これは他の PE のページであるかもしれない。というよりその確率の方が高い。

[6] 事実 VPIM 0.5 より前のバージョンでは、ここまでの処理で動いていた。

[7] VPIM ver1.0 では、32語である。

```
scan-all() {
    repeat {
        — ある PE のページを全てスキャンする —
        repeat {
            Scanning := false;
            for (i := HEU; i ≥ 1 ; i := i/2;)
                if (S_i < B_i) {
                    scan(i);
                    Scanning := true;
                }
        } until not(Scanning);

        — 共有プールからのページ取り出しを試行する —
        if (get-from-pool(S_HEU, B_HEU) {
            — ページ取り出しに成功 —
            — S_HEU と B_HEU が、このページスキャンに使われる —
        } else
            アイドル状態であると宣言する;

    } until (全てのPEがアイドルである);
    — GC はここで終了する —
}

scan(n) {
    — サイズ n の構造が入ったページをスキャンする —
    while (S_n < B_n) do {
        P := newheap[S_n];
        if (P が旧領域へのポインタならば) {
            — m は更新されたページのサイズ —
            m = copy(P, S_n);
            if( (m ≠ 0)and(m ≠ HEU) )
                checkB( Arity );
        }
        S_n := S_n + 1 ;   checkS( n );
    }
}
```

図 11–2: 並列 GC アルゴリズム: 全ヒープのスキャン

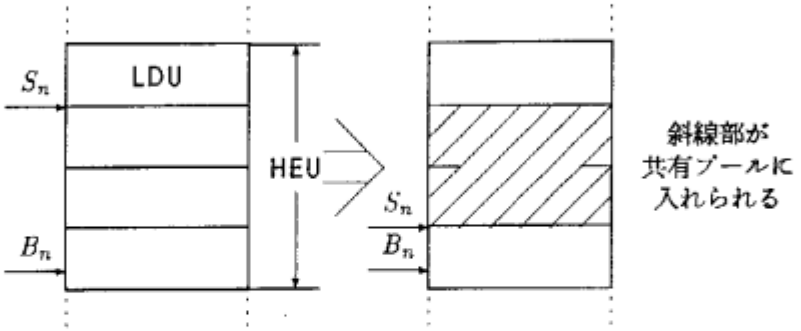
```
checkB(m) {
    — B_m がページを使い切ったかどうかをチェック —
    if (is-at-top-of-HEU(B_m)) {
        if (m ≠ HEU) {
            if (Top-of-HEU(S_m) ≠ Top-of-HEU(B_m-1))
                add-to-pool(Top-of-prev-HEU(B_m),
                            Bottom-of-prev-HEU(B_m));
            — 共有の B を更新して、ページを獲得する (アトミック操作) —
            B_m := GlobalB;   GlobalB := GlobalB + HEU;
        }
        — もしも (m = HEU) ならば、単に S_HEU = B_HEU まで繰り返す —
    }
}

checkS(n) {
    — S_n が負荷分散ユニットを越えたかどうかのチェック —
    if (is-at-top-of-LDU(S_n)) {
        if (n ≤ LDU) {
            switch (distance(S_n,B_n)) {
                case (同じ LDU 内): 何もしない;
                case (同じ HEU 内): {
                    add-to-pool(S_n,Bottom-of-prev-LDU(B_n));
                    S_n := Top-of-LDU(B_n);
                }
                case (異なる HEU): {
                    add-to-pool(S_n,Bottom-of-HEU(S_n));
                    add-to-pool(Top-of-HEU(B_n), Bottom-of-prev-LDU(B_n));
                    S_n := Top-of-LDU(B_n);
                }
            }
        } else if (LDU < n < HEU)
            if (is-at-top-of-HEU(S_n))
                S_n := Top-of-HEU(B_n);
        — else (HEU = n) 何もしない —
    }
}
```

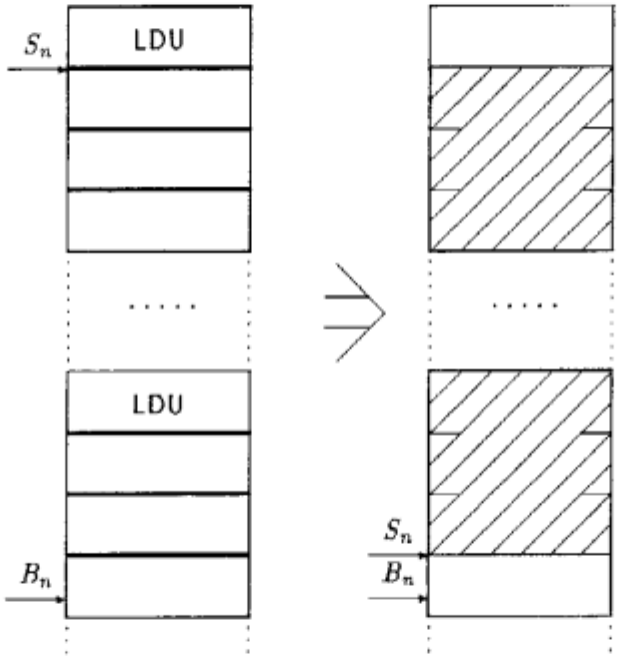
図 11-3: 並列 GC アルゴリズム: 領域オーバフローのチェック

```
copy(P,S_c) {
    — P は旧領域へのポインタ, S_c は今のスキャンポイント —
    Temp := oldheap[P];
    if (Temp が新領域の移動先アドレス) {
        newheap[S_c] := Temp;
        — どのサイズの B も更新されなかった -
        return(0);
    } else {
        Arity := arity(Temp);
        if ( Arity < HEU )
            m := Arity;
        else {
            m := HEU;
            — 共有の B を更新して、ページを獲得する (アトミック操作) —
            from := B_HEU := B_global;
            B_global := B_global + Arity;
        }
        newheap[S_c] := B_m;
        — 旧領域のマークと移動先アドレスの書き込み —
        oldheap[P] := B_m;
        for (i := 1; i ≤ Arity; i++)
            — 旧領域の内容を全て新領域にコピー —
            newheap[B_m++] := oldheap[P++];
        if ( Arity ≥ HEU ) {
            — 直ちに共有プールに入れる —
            add-to-pool(from, B_HEU);
        }
        — B_m が更新された -
        return(m);
    }
}
```

図 11–4: 並列 GC アルゴリズム: セルのコピー



Case 1: $\{S_n, B_n\}$ が同じページ(HEU)の時。S_n と Bottom-of-prev-LDU(B_n) の間の連続した LDU (この場合は2ユニット)が共有プールに入れられる。



Case 2: $\{S_n, B_n\}$ が異なるページ(HEU)の時。2つの連続領域、S_n と Bottom-of-HEU(S_n) の間、および Bottom-of-HEU(B_n) と Top-of-prev-LDU(B_n) の間が共有プールに入れられる。

図 11–5: S が LDU 境界に達した時の負荷分散

11.2.4 注意事項

並列 GC アルゴリズムで注意すべき点を列挙する。

- HEU より小さい構造体用のページは、足りなくなった時点ですぐに割り付けられるが、HEU より大きい構造体用のページは必要になった時点で割り付けられる。予めどれだけ確保して良いかが分からないからである。

1. メモリ不足によるGC要求の検出 (内部イベント処理として行う)
2. GC フェーズへの切り替えのための同期 †1
3. 前処理 (ヒープ切替えなど, 1 PE)
4. GC開始のための同期 †2
5. コピー処理 (全 PE)
   A. ルートのコピー (全 PE)
      a. レディゴールスタックから指されるゴールのコピー
      b. クラスタ内ゴール通信ポストから指されるゴールのコピー
      c. 輸出表から指されるゴールのコピー (PE毎に領域分割する)
      d. 輸入表からたどれる全ての里親のコピー
   B. コピー GC のメイン部分 (全 PE)
6. コピー終了の同期 †3
7. 後処理 (輸入表メンテナンス、永久中断ゴール検出など)
   A. 黒輸入表のスキャンと永久中断状態の検出、Sweep-and-Copy (1 PE)
   B. ページの残りをフリーリストにする (全 PE)
   C. 構造体表の再ハッシュ (1 PE)
   D. 同期 †4
   E. Mark-and-Sift (1 PE)
   F. 白輸入表のメンテナンス (1 PE)
   G. 黒輸出表の再ハッシュ (1 PE)
8. GC終了の同期 †5

図 11-6: GC処理フロー

- 共有プールから取り出した領域は、S_{HEU} と B_{HEU} の間に設定される。こうなると、S_{HEU} は、必ずしも HEU サイズ以上の構造体の中を指さなくなるが、スキャン中にはそれがどのような大きさの構造体の中かは考慮する必要がないので、問題にならない。

11.3 一括 GC の処理フローと PE 間の同期

一括GCは、図11-6のような手順で進む。このうち、(†1) の同期では、メモリ中のPE数カウンタ (バリア) を用いてGCフェーズへの切り替えの同期を実現する。(†2)では、クラスタ内の全PEがGCフェーズに入っており、最後にGCフェーズに入ったPEがヒープの切り替えを行うのを待っているだけなので、単純にシグナルをブロードキャストすれば良い。(†3) はスリットチェックでなく、ビットマップ(アイドルかそうでないかをシステム固定領域の特定番地に書き込んでいる)のポーリングで同期を取っている。(†5)は、(†1) と同様のバリアを用いている。後処理中の同期(†4)タイミングは、旧領域を破壊してよくなる時点 (構造体表のメンテナンスが終った時点)である。

なお、後処理中の (a), (c) および (e), (f), (g) は、別の PEで同時に実行できる。

11.4 データ種別による GC 処理

本節では、スキャンした語のタイプ毎に、それらをどう処理しているかをまとめて述べる。

11.4.1 アトミックデータ

Value 部が意味を持たないもの、あるいはポインタでないものは、スキャン時には何もメンテナンスを行なわない。

このタイプを以下に列挙する。

EOL, ATOM, INT, VOID, UNDF, EUNDF, GOAL, VECT0, HLINK

11.4.2 単一参照が保証されている構造体データ

値部がポインタであり、それが指している先の構造体が単一参照であることが保証されている場合、その構造体データは、コピー先アドレス(新領域のポインタ)を書き込むことなく、そのサイズに応じたページにコピーする。単一参照の保証は、タイプを見れば定まる場合と、(タイプ + MRB) で定まる場合がある。なお、MGHOKや、DREF のように、そのポインタの先に参照カウント[8]がついており、それが 1 の場合に単一参照の保証ができるものもあるが、判定が複雑になるので、この最適化を行なっていない。

このタイプを以下に列挙する。

タイプだけで判定できるもの

MHOOK, EMHOK, RHOOK, RDHOK

タイプ + MRB。で判定できるもの

LIST, VECT1, VECT2, VECT3, VECT4, VECT5, VECT6, VECT7, VECT8, VECT, STRG

なお、単一参照であることが保証されているが、コピーされたかどうかをGC後に判定しなければならないためにマークをつける[9]タイプとして、

BEXREF, BEXVAL, HOOK, EHOOK

がある(これらの処理の詳細については後述)。

11.4.3 多重参照かもしれない構造体データ

値部がポインタであり、それが指している先の構造体が多重参照であるかもしれない場合、その構造体データを二重にコピーしないためにコピー先アドレス(新領域のポインタ)を旧領域に書き込んでから、そのサイズに応じたページにコピーする。

このような構造体のうち、ベクタやリストは更にその中にポインタを含んでいると、単純にコピーすると上述の単一参照の最適化ができなくなる。すなわちMRB は、ゴールから、目的のセルまでに経由

[8] MGHOK の場合はマージャ入力数、DREF の場合は、多重サスペンド要因のうち、まだ具体化されていない変数の数

[9] 多重コピー防止のためではない。

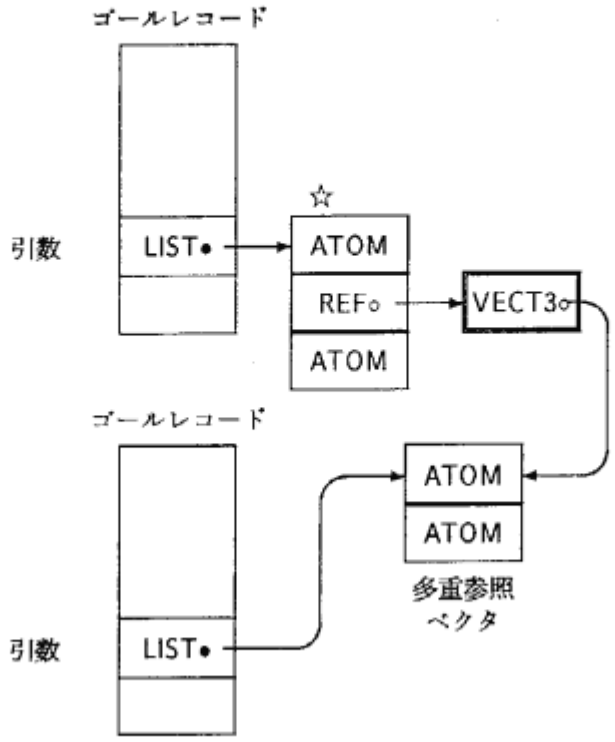


図 11–7: MRB ∘で指しても多重参照である例

したポインタの MRB のうちひとつでも • の場合、そのセルが多重参照であることを意味するのに対して、コピー方式の GC では、パスをたどりながら順にコピーするのではないことが原因である[10] (図11–7参照)。

このような構造体データは、その要素をマークして(MRB を • にして)書き込むことでこれを回避する[11]。すなわち、図11–7の例では、☆のセルに対しては、その要素をのMRBを • にしてコピーする。

この多重参照かもしれないタイプを以下に列挙する。

タイプだけで判定できるもの (MRBは OR しない)

DREF, SHREC (後述), FPREC (後述), MGHOK (後述)

タイプ + MRB • で判定できるもの (MRBをORする)

LIST, VECT1, VECT2, VECT3, VECT4, VECT5, VECT6, VECT7, VECT8, VECT

タイプ + MRB • で判定できるもの (MRBをORしない)

STRG

[10]幅優先(breadth-first)で生きているセルをたどる(traverse)ため。

[11]PIM/p のように、p_Read と s_ReadWithMRBor のコストが変わらないマシンであれば、この判定はオーバヘッドがない。

11.4.4 間接参照セル

スキャン中にREF を発見した場合、デレファレンスを行なう。これは、不要な間接参照ポインタをコピーする(時間と領域の)オーバヘッドを減らすためである。

デレファレンス後、それが指す先が具体値か未定義かによって処理が異なる。

具体値の場合 メモリ割り付けは行なわず、その値を直接スキャンポインタの先に書き込み、もう一度スキャンをリトライする[12]。

未定義の場合 1ワードのメモリを割り付け、未定義変数の内容を書き込む(後にスキャンされる)。なお、(REF→○未定義変数) の場合であっても、単一参照であるとは限らないので、旧領域には移動先アドレスを書き込む。

11.4.5 ゴールレコード

HOOK もしくは EHOOK で指される構造であるゴールレコード(2.4節参照)は、本来単一参照であることが明らかなデータであるが、永久中断の検出のために、コピー後にそのゴールレコードはコピーされたか否かを判定できなければならない。このため、多重参照同様に移動先を書き込んでコピーする[13]。他の構造体と異なる点は、サイズの計算方法である。

コピーサイズ 引数個数 (アリティ) + 制御用語数 (6語)

割り付けサイズ $2^{n-1} <$ (コピーサイズ) $\leq 2^n$ の時、2^{n+1}。ただし、最低 16 ワード。

これは、GC後の再利用に備えるためである。

永久中断状態の検出(12.2節参照)のため、里親毎にコピーしたゴールレコード数をカウントしておく。すなわち、ゴールレコードをコピーする毎に、その所属する里親(旧領域)の GC 中のチャイルドカウントスロットを1 増やす。なお、GOALタイプの置かれている場所に移動先アドレスを書き込むので、わざわざGOALタイプを無効化する必要はない。

また、ゴールの所属する里親が Aborted あるいは、Aborted Terminated の状態の時は、このゴールは実行されないので、ゴール引数、およびコードは、既にゴミになっていると考えられる。そこで、これらはコピーしない。アリティを 0 とし、コードへのポインタを EOL にする。

11.4.6 マージャレコード

MGHOK で指される構造であるマージャレコード(7.6節参照)は、多重参照であるかもしれないので、二重にコピーしないためにコピー先アドレス(新領域のポインタ)を旧領域に書き込んでから、そのサイズに応じたページにコピーする。

マージャも、ゴールの一種であると考えられ、またそれが永久中断状態に陥ることもあるので、その検出のため、里親毎にコピーしたマージャレコード数をカウントしておく。すなわち、マージャレコー

[12] アトミックデータであるかどうかを判定して、リトライしない最適化もあり得るが、実装していない。

[13] この移動先書き込みには本来排他制御は不要であるが、他と処理を共通化したいので、現在は排他制御を行なっている。性能に響くようであれば、別のルーチンとした方が良いかも

ドをコピーする毎に、その所属する里親(旧領域)の GC 中のチャイルドカウントを 1 増やす。なお、GOALタイプの置かれている場所に移動先アドレスを書き込むので、わざわざGOALタイプを無効化する必要はない。

また、マージャの所属する里親が Aborted あるいは、Aborted Terminated の状態の時は、マージャレコードをコピーする代わりにスキャンしている語をEUNDFに書き変える[14]。MGHOK は、マージャがフックした未定義変数であるが、そのマージャが動けないことが明らかであるためである。

11.4.7 白輸入表エントリ

WEXREF, WEXVAL のポインタが指すのは、システム固定領域(非GC領域) にある白輸入表エントリ(9.1節参照)である。白輸入表エントリは、GC後にスキャンされるので、値部を保存してマーク付けだけを行ない、GC時に生きていたことの印とする。

11.4.8 黒輸入レコード

BEXREF, BEXVAL のポインタが指すのは、ヒープ中にある黒輸入レコード(9.1節参照)である。黒輸入レコードは単一参照が保証されているが、マーク付けを行っておかないとその黒輸入レコードはコピーされたか否かを判定できない。このため、多重参照同様に移動先を書き込んでコピーする[15]。

11.4.9 コードモジュール

COD, MOD は、コードモジュール(3.1節参照)を指すポインタである。KL1 データの中で、コードモジュールが唯一例外であるのは、一つの構造体の中で、他から指される場所がいくらでも[16]存在し得ることである。このため、コピーに当たっては、常に同じ位置に移動先アドレスを書き込めるようになっていないと、多重にコピーしてしまう可能性がある。

実際に移動先を書き込むのは、コードモジュールの先頭アドレスであり、MODタイプの場合はここを指している。CODタイプの場合は、述語定義の先頭を指しているが、この直前にモジュール先頭ワードからのオフセットが書かれているので、それを元にコードモジュールの先頭アドレスを求め、そこに移動先アドレスを書き込む。

11.4.10 荘園レコード

SHREC は、荘園レコード(10.5節参照)を指すポインタである。荘園タイプ変数は、通常KL1プログラム上では単一参照が保たれ、誤って多重参照にした場合は、その正常な動作は保証されないが、メモリ中では兄弟荘園リンクなどに SHRECタイプを用いているので、荘園レコード自体は多重参照である。このため、新領域の移動先を書き込んでコピーする。

---

[14] EMGHOK というタイプを用意していないので、念のためにEUNDFにする。

[15] この移動先書き込みには本来排他制御は不要であるが、ゴールレコードの時と同様に、他と処理を共通化したいので、現在は排他制御を行なっている。性能に響くようであれば、別のルーチンとした方が良いかも

[16] 実際には述語数だけのバリエーション

ゴールレコード同様に、荘園レコードも、その所属する(その荘園にとって親の荘園の)里親レコードのチャイルドカウントに(参照カウントとして)カウントされているので、GC中の参照カウントを1増やす。

ただし、子荘園リンクからは外されていて、それをコントロールするゴールが引数として指している荘園レコードがあり得る[17]ので、この場合のみ、チャイルドカウントを操作しない。

11.4.11 里親レコード

FPREC は、里親レコード(10.5節参照)を指すポインタである。里親レコード自体は多重参照であるので、新領域の移動先を書き込んでコピーする。里親レコード自体は、GCに先だって全てコピーする。コピーに当たっては、新領域の GC 中のチャイルドカウントのスロットだけは単純にコピーしないで、0 にする。

11.4.12 スキッパ

CDESC (2.3節参照)はオブジェクトタグであり、それ以降の何ワードかはアトミックデータであることが保証されている。これは、ストリング、コードモジュールの機械語コード領域の先頭のように通常処理で入れられるものと、GCのコピー中に割り付けサイズよりもコピーサイズが小さい時に(ゴミを後でスキャンしないようにするために)入れられる場合とがある。

スキャン中にCDESCを発見すると、その値部の下位 24 ビットを、符号なし整数と解釈し、スキャンポインタをその値(実際には、CDESC の書かれた語が含められていないので、更に1を加えて)進める。これにより、その間のスキャンを省略する。

11.4.13 スキャン時に出現しないタイプ

以下のタイプは、スキャン時には現れ得ない。

FLC

コピー方式のGCでは、生きているセルだけをアクセスするため。フリーリスト中のセルは決してアクセスされない。

SLOCK, EXLOCK

GC処理に入る前に、ソフトウェアロックは外しておかなければならないため。

MARKED

旧領域にのみ現れるタイプであるため。スキャンは、新領域を行なう。

VISIT

GC後に、永久中断状態を発見した後に付けられるタイプであるが、そのフェーズでのみ用い、それ以外 (通常実行時を含めて) 使わないタイプであるため。

[17] terminated をレポートストリームに報告し、それを監視するゴールが remove_shoen を発行するまでの間。

DNTC

これが置かれる前には、必ず CDESC が入っているため。スキップされなければならない。

11.5 クラスタ間データの処理

本節では、一括GCにおいて、クラスタ間データ構造に関して考慮の必要な点をまとめて述べる。

11.5.1 輸出表エントリのマーキングルート化

輸出表エントリ(9.1節参照)から指されている他のクラスタに輸出されたデータは、一括GC時点では他のクラスタからの参照があるかもしれないので、レディゴールスタック(6.2節参照)同様にマーキングルートとしなければならない。

一括GCでは、輸出表エントリをスキャンし、未使用のエントリ(タイプがFLC)以外の場合は、そこから指されたデータ(黒輸出レコードを含む)を新領域にコピーする。黒輸出レコードは、他のセルと同様にスキャンされるが、ハッシュリンクに関しては、タイプをHLINK (Hash LINK) とし、スキャン時の扱いをアトミックデータと同様にすることで、不要な黒輸出レコードのコピーを防ぐ。

11.5.2 コピー後の %release 送信

全てのセルのコピー後に、参照の消えた輸入表(9.1節参照)に対しては、それを輸出したクラスタに対して、%releaseを送信し、参照の消滅を通知する必要がある。これは、白輸入表と黒輸入表では、処理が異なる。

(a) 白輸入表

GC中にスキャンしたデータが白外部参照セル(タイプが WEXREF または WEXVAL)であった場合は、それが指す白輸入表のタイプをINTからMARKEDに変更する(値部は保存)。全てのセルのコピー終了後、固定領域に連続して確保された白輸入表を順にスキャンし、タイプによって次のような処理を行なう。

MARKED

参照が消滅していないので、使用中のタイプであるINTに戻す。

INT

参照が消滅した(マークされなかった)ことを意味するので、%releaseを送信し、そのエントリをフリーリストに返す。

FLC

GC以前から未使用であることが明かであったので、何もしない。

(b) 黒輸入表

GC中にスキャンしたデータが黒外部参照セル (タイプがBEXREF または BEXVAL)であった場合は、それが指す黒輸入レコードを新領域にコピーする。一括GC終了後に、システム固定領域にある黒

輸入ハッシュ表から、すべての黒輸入レコードをたどり、マークの付けられなかった(参照が消滅し、コピーされなかった)エントリに対して、%releaseを送信すると同時に、ハッシュチェーンを新領域のチェーンにする。

11.5.3 黒輸出表の再ハッシュ

黒輸出表は、輸出されたデータの置かれたアドレスをキーとした黒輸出ハッシュ表からリンクされている。このデータの置かれたアドレスは、一括GCのコピーによって変化するので、再ハッシュ処理が必要となる。この処理は、全てのセルのコピー後に、以下のように行う。

1. 黒輸出ハッシュテーブルをクリアするとともに、エントリチェーンを一本化する。
2. 一本化したエントリをたどり、輸出されたデータの置かれたアドレスをキーに、再ハッシュ操作を行なう。

11.5.4 構造体表の再ハッシュ

構造体表は、構造体の置かれたアドレスをキーとしたアドレスハッシュ表と、構造体IDをキーとしたIDハッシュ表の二つのハッシュリンクを持っている。全てのセルのコピー後に、構造体表に対して次のような操作が必要となる。

- 構造体レコードはヒープ領域に置かれているので、参照の残っている構造体を指す構造体レコードを新領域にコピーする。参照の消滅した構造体レコードはコピーしない。
- 構造体の置かれているアドレスが変化するので、アドレスハッシュ表を再ハッシュする。

具体的には、次のような処理を行なう。

1. IDハッシュ表をオールクリアするとともに、すべての構造体レコードをID ハッシュリンクスロットを用いて一本のリンクにする
2. アドレスハッシュ表をオールクリアする
3. 一本化されたチェインをたどり、構造体がコピーされていた場合はID/アドレスともに、新しいハッシュ値のところへプッシュする[18]。構造体がコピーされていなかった場合は何もしない。

[18] ただし、構造体IDはGCによって変化しないので、IDをキーにしたリハッシュは、本来不要なはずである。ただし、コリジョンチェーンの中に消滅したものがあるかもしれないので、このような方法で単純化している。もっと良いアルゴリズムがあるに違いないと思うが、せいぜい3桁個程度のオーダなので、あまり真面目にインプリメントしても高速化が期待できないので、単純な方法を選んでいる。

第 12 章

デバッグサポート機能

執筆担当者: 今井[1]

本章では、KL1プログラマのプログラム開発/デバッグをサポートするためにVPIM に備えた「永久中断ゴール報告」と「トレース / スパイ」機能について説明する。

12.1 概要

永久中断 (Perpetual Suspension)とはある変数の具体化を待って実行を中断したゴールが、(プログラムのバグなどで) 当該変数への具体化パスが捨てられたことにより、実行が再開されることがあり得なくなった状態のことである。このような状態が発生するのは、KL1における実行制御方式、すなわち、共有変数に対するパッシブ / アクティブユニフィケーション機構を用いたデータ駆動計算手法が原因となっている。この共有変数によるデータ駆動方式のために、KL1では、外世界との相互作用を含む並列処理の記述は非常に容易となっているが、反面、上に述べたように、必要なアクティブユニフィケーション処理を記述し忘れて永久中断状態に落ち込む可能性があり、KL1によるアプリケーション作成時の初期デバグにおける厄介な問題となっている。このような永久中断ゴールが問題となるのは

- ゴールのサスペンド処理をNon Busy-waiting方式で行っているため、そもそもサスペンドしたゴールの所在が分からない。
- 一つの永久中断ゴールはそのゴールが生成するデータを待っているゴール(群)を永久中断させる。そのため一般に、永久中断ゴールの連鎖が起こり、その数は非常に多くなり易い。

といった特徴があるためである。たとえ、前者を解決するためにBusy-waitingによるサスペンド方式を採用したとしても、後者を考慮すると効率的な解決策とはなり得ないことが明らかであろう。そこで、VPIMでは、Multi-PSI同様に以下のような二種類の永久中断ゴール発見・報告方法を実装している。

- 一括GC時に永久中断ゴールを発見し、永久中断ゴール群の内、因果関係において極大(後述)であるゴールのみを報告する

[1] 12.2節は猿村、12.4節および12.5節は平野が執筆したものを元に、今井が本資料向けに再編集した。

• MRB による即時GCで永久中断ゴールを発見し、報告する

トレースは、KL1プログラムのデバッグのために開発された機能の一つで、トレース対象となったゴールの実行時のサブゴールへの展開について知ることができる。この情報を元に、KL1のゴールの実行状況を得ることができ、KL1プログラムの開発・デバッグを効率良く進めることができる。

トレースは、ゴール単位で指定でき、そのゴールがコミットした時に生成したサブゴール(組込述語、ユニフィケーションも含めて)を、実行したりエンキューする代わりにトレース例外としてレポートストリームに報告する。この入り口となる組込述語が、apply_tracing/3 である。

PIMOS 上のデバッガ (リスナと呼ぶ) は、この機能を用いてサブゴール情報を表示し、それらのサブゴールのうち、それ以下の実行状況をトレースするか、あるいはトレースをやめるかをユーザに問う。トレースを継続する枝にはそれらのサブゴール情報を元に、apply_tracing/3 を再び行ない、トレースを継続しない枝については、apply/2を行なう。

スパイも、KL1プログラムのデバッグのために導入された機能の一つであり、特定の述語の実行状況等を調べることを目的としている。

KL1は、並列論理型言語であり、実行可能なゴールはどれから実行しても良い、という特徴があるため、デバッグには逐次言語の場合のような実行順序に依存した手法は使用できない。スパイ例外は、ある述語を実行するゴールが生成される時点で、そのゴールに関する情報を報告するもので、その述語の実行の様子やサブゴールについての情報を得るきっかけが得られる。そのゴールからサブゴールへの展開等に関する情報は前述のトレース例外を用いて得ることができる。実際のデバッグでは、挙動の知りたい述語にスパイをかけておき、プログラムをスパイモードで実行することにより、その述語が呼び出される時点でスパイ例外が上がり、そのサブゴール群の実行状況をまたスパイ例外やトレース例外を用いて調べるといった手順をとることになる。

スパイの指定は、あるゴールの子孫ゴールに対して有効となる。スパイの対象となるゴールは、モジュール名、述語名、引数の数の組で指定でき、ワイルドカードを使用することも可能である。スパイ中にスパイ対象ゴールが生成された場合には、そのゴールをエンキューする代わりにゴールに関する情報をレポートストリームに報告する。この入り口となる組込述語が、apply_spying/4 である。

12.2 一括 GC における永久中断ゴール検出

12.2.1 因果関係グラフと極大ゴール

この節では永久中断ゴール間の因果関係、およびその中の極大ゴールという概念について説明する。

(1) 因果関係

ゴールAとゴールBの間に、

• ゴールAが行うアクティブユニフィケーションによって中断中のゴールBがリジュームされる

$$?-a(X,Y,Z),b(Y),c(Z).$$

$$a([msg1|X],Y,Z):-true \mid Y=[msg2|Y1],\ Z=[msg3|Z1],\ a(X,Y1,Z1).$$
$$b([msg2|Y]):-true \mid b(Y).$$
$$c([msg3|Z]):-true \mid c(Z).$$

図 12-1: 永久中断ゴール例

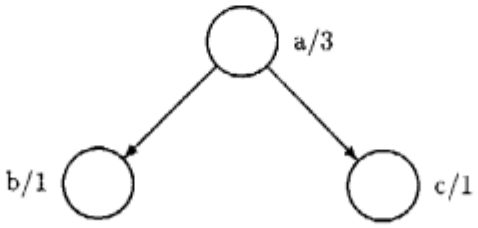


図 12-2: 因果関係グラフ

という関係がある場合に、**ゴールBの実行はゴールAに依存する**と言い、両ゴール間にはA⇒Bの向きの因果関係が存在すると言う。一つ注意しなければならないのは、ここで言う因果関係とはデータフローによる制御の流れのみを考慮している、という点であり、

$$a(X):-true \mid b(X).$$

のような呼び出し関係にある二つのゴールa/1とb/1との間には、この意味での因果関係は存在しない。一方、

$$?-a(X),b(X).$$
$$a(msg):-true \mid ...$$
$$b(X):-true \mid X=msg.$$

におけるa/1とb/1との間にはb/1⇒a/1という向きの因果関係が存在するのである。以降ではこのような因果関係に沿った有向線分で各ゴールを結んだグラフのことを**因果関係グラフ**と呼称する。

図12-1、図12-2に永久中断するゴールのプログラム例と、その因果関係グラフを挙げる。

この例におけるゴール呼び出しを実際に行うと、3つのゴールはVPIMのNon Busy-waiting方式のサスペンド機構により、図12-3のようなデータ構造を形成する。

図12-2、図12-3を比較することによって、KL1のサスペンション・メカニズムが実際の因果関係グラフを構成し、また、因果関係において下流のゴールには上流のゴールの引数から到達可能であることが明らかであろう。

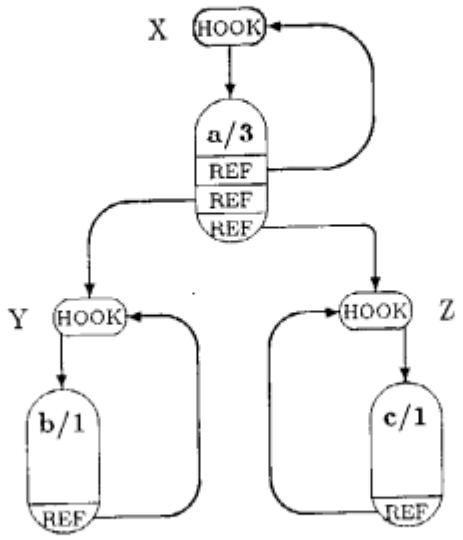


図 12-3: 永久中断ゴールの状態

(2) 極大ゴール

ここで『極大ゴール』という概念を定義するためにゴール間の関係を表す記法

$$A \succ B$$

もしくは

$$B \prec A$$

を導入し、これを『ゴールBがゴールAの引数から到達可能である』と読む。さて、ゴールAとゴールBの間に

$$(A \succ B) \wedge (A \not\prec B)$$

なる関係がある場合、因果関係においてゴールBはゴールAの下流にある。また、

$$(A \succ B) \wedge (A \prec B)$$

なる関係がある場合、因果関係においてゴールAとゴールBは同値であると言える。この記法を使用することにより永久中断ゴール G_p は

$$\forall G_e, G_p \not\prec G_e)$$

(ただし、G_e は、実行可能ゴール)のようなゴールであると定義出来る。

以上の関係を考慮すると、永久中断ゴール群中の極大ゴール(群)とは永久中断ゴール群を同値類 G_i に分割した後、それらの中で

$$(\forall G_i, G_j \not\prec G_i)$$

という条件を満たす同値類G_jのことであると言える。

このような極大ゴールは、永久中断ゴール群の因果関係の中で最も“因”の位置にあるため、永久中断ゴール群の中から極大ゴールのみを抽出・報告することは、効率的なデバグ環境の実現という点で、非常に意義が高いと言える。真に永久中断の要因であるゴールが、他の膨大なゴール群に紛れてしまい、特定し難くなるということがなくなるためである。

12.2.2 極大ゴール発見方式

ここでは前節で定義した永久中断ゴール中の極大ゴール発見方式について説明する。アルゴリズムは

1. 永久中断状態の検出
2. 永久中断ゴール(群)の発見
3. 極大ゴール抽出

の3フェーズに分かれる。

(1) 永久中断状態の検出

前節で説明したように、VPIMではNon Busy-waiting方式でゴールの実行中断を行っているため、通常実行中に永久中断状態を検出することはMRBを用いて限られた場合にのみ行なうことができる(12.3節参照)が、一般には困難である。しかし、同時にこの機構のために永久中断ゴールは通常の実行可能ゴールの引数からは到達不可能となっている。この事実を利用して、コピー方式による一括GC時に永久中断状態の検出を行うこととした。一括GCでは実行可能なゴールのみをルートとするために、永久中断ゴールは発見不可能であり、故に一括GC前後でゴール数を比較することで永久中断の発生が検出出来るのである。このために通常実行時と一括GC時にゴール数を数えるための、二つのカウンタ、すなわち

- チャイルドカウント (*goal_counter*)
- GC中のチャイルドカウント *copied_goal_counter*

が必要となる。但し、通常実行時用のgoal_counterに関しては、計算の終了を検出するために既に導入済みであるため、通常実行時の処理には全くオーバーヘッドなしとすることが出来た。

(2) 永久中断ゴール(群)の発見

一括GCでルートから到達可能なデータオブジェクトを旧領域から新領域へコピーした結果、永久中断ゴールはコピーされずに旧領域に残ることになる。次のフェーズでは、この旧領域をスイープして永久中断ゴールを探さなければならない。このようなスイープを可能にするために、ゴールレコードおよびマージャレコードにしか現れないような特殊なタイプGOALが導入され、他のデータとゴールレコードとの識別に利用されている。ゴールレコードの場合、先頭スロットにGOAL$_\circ$が、マージャレコードの場合、先頭スロットにGOAL$_\bullet$が書かれる。

永久中断したゴールはその引数情報も含めて荘園に報告されるため、それらのデータは通常のデータ

と同様、GCにおける新領域にコピーされなければならない。そのため、スイープで永久中断ゴールが発見された場合にはそのゴールをルートとして、旧領域から新領域へのコピーを行う必要がある。実は、以下に述べるように、このコピー操作で永久中断ゴール間の依存関係の一部が判明するために、このアルゴリズムは非常に効率的なものとなっている。

なお、VPIMにおいてもこの処理は1PEで行なう。これは、

- Multi-PSIで実装された方式が、逐次実行であることを利用した効率の良い枝刈りを行なっていること
- 永久中断ゴールが発生するのは「プログラムにバグがある時」であるから、それほど高速処理が必要でないこと
- その間に、構造体表の再ハッシュ、白輸入表のメンテナンス、黒輸出表の再ハッシュなどの独立した処理があり、これらを他のPEで担当させることができること

などの理由による。

(3) 極大ゴール抽出

最後に、極大ゴール抽出のための方式について述べる。このアルゴリズムは*sweep-and-copy*フェーズと*mark-and-sift*フェーズとに分けることが出来る。以下にそれぞれのフェーズにおける処理について説明する。

(a) Sweep-and-copy フェーズ

このフェーズでは前節で説明したように、旧領域のスイープ&コピーを行うことにより、全ての永久中断ゴールが新領域に移される。処理は図12-4の通りである。

極大ゴール候補テーブルは、Multi-PSIのインプリメントでは旧領域中のゴールレコードのリストとして実現されているが、実現の容易さからVPIMではユニファイスタック領域を用いている。

このフェーズ終了時点での新旧領域の状態は図12-5のようになっている。

この図中でG_iはメモリ・スイープで発見された極大候補ゴールであり、$CP-G_i$はG_iをルートとしたコピーによって新領域に移されたデータオブジェクトの集合である。また、$i<j$であればG_iはG_jの前に発見されていることになる。

さて、ここで極大候補ゴールG_iの間に

- $G_i \not\succ G_j$, if $i<j$

なる関係があるのは明らかであろう。このことは以下のようにして簡単に証明することが出来る。

もしG_jがG_iに依存し、かつ、$i<j$ならばゴールレコードG_jはG_iの引数から到達可能であり、G_jはG_iをルートとしたコピーで発見されていなければならない。しかし、$i<j$であるから、G_jはそのコピーで発見されてはいない。故にG_jの実行はG_iに依存しないのである。

特に、図12-5中のG_nはこのフェーズ終了時点で極大ゴールであることが保証されている。G_nの実行は$G_1,\ldots,G_{n-1}$に依存しないからである。

```
procedure Sweep-and-copy
begin
    while (goal_counter > copied_goal_counter) do
        begin
            旧領域をスイープ;
            if  ゴールレコードGを発見 then
                begin
                    Gを新領域へコピー;
                    Gを極大ゴール候補テーブルに登録;
                    copied_goal_counterをインクリメント;
                    Gから到達可能なデータオブジェクトをコピー;
                end
        end
end
```

図 12-4: Sweep-and-copy フェーズ処理

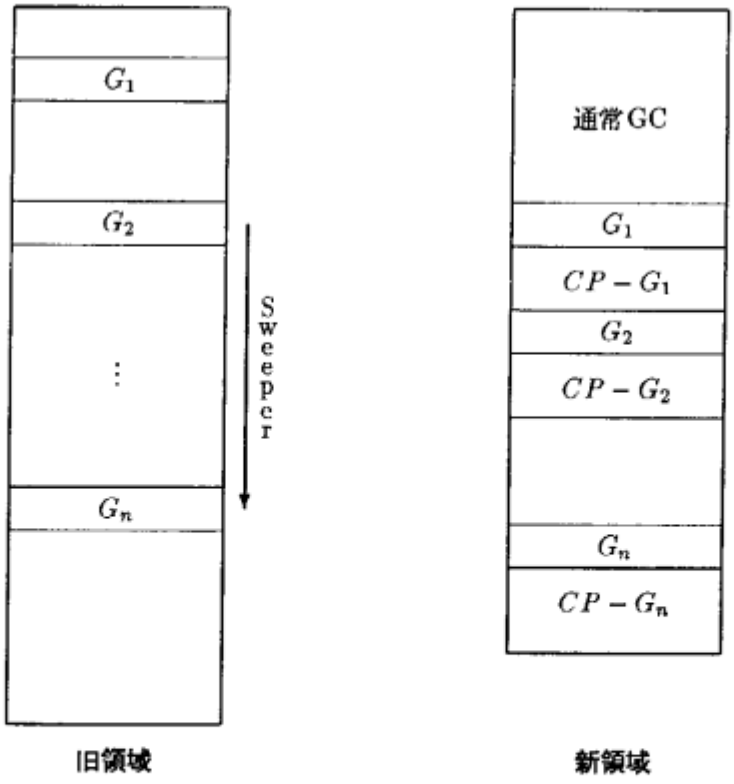


図 12-5: Sweep-and-Copy フェーズ後の新・旧領域

```
procedure Mark-and-sift
begin
    while (∃ 極大候補ゴール G_j) ∧ (G_j は未マーク) ∧ (∀ i ≤ j, 未マークゴール G_i) do
        begin
            G_j にマーク;
            while G_j の引数からデータをトレース&マーク;
                begin
                    if  (極大候補ゴール G_i を発見) ∧ (i ≠ j) then
                        begin
                            G_i を極大候補テーブルから削除;
                        end
                end
        end
    メモリセルのマーク外し;
end
```

図 12–6: Mark-and-sift フェーズ処理

(b) Mark-and-sift フェーズ

Mark-and-sift フェーズでは前フェーズで確かめられた因果関係と逆向きの

- $G_i \prec G_j, i < j$

なる関係があるか否かを確かめる。

真の極大ゴールは『ゴール G_j の引数からデータをトレースし、到達可能なゴール G_i $(i < j)$ を探す』という方法で、極大候補ゴールテーブルを篩にかけることによって発見される。同じデータを複数回トレースすることを避けるためにゴールレコードにVISITを用いて、一度トレースされたデータとしてのマークを付けることとした[2]アルゴリズムは図12-6の通りである。

データのトレースは一つのゴールの全引数をルートとして再帰的に行われる。このフェーズでは旧領域が自由に使えるため、旧領域をスタックとして用いた処理を採用している。[3]

この処理が終了した時点で極大候補ゴールテーブルには真の極大ゴールのみが登録されていることが保証される。この処理によって、

- $(G_i \prec G_j \wedge i < j)$

[2]ただし、これだけではX=[X] というような循環構造を作られた後に永久中断を起こすと、絞り込みフェーズで無限ループに陥り、スタックオーバーフローとなる。ベクタや、リストの場合に、再上位ビットを立てることでマークの代わりとする予定である。なお、Multi-PSI では、専用のGCビットを用いている。

[3]このためVPIM では、他のPEが構造体表のメンテナンスを行なっている最中に、Sweep-and-Copy を行ない、両者が終了した時点で同期をとって、Mark-and-sift に移る。

の関係にあるゴール G_i は候補テーブルから削除されるからである。

最後に、ここで発見された極大ゴールのみが永久中断ゴールとして報告されることになる。なお、報告の形式については、10.10節を参照のこと。

12.3 即時 GC における永久中断ゴール検出

MRB を用いると、パスがなくなることが保証されることにより、決して具体化されることのないパスを待つゴールが検出できる。例えば、

$$p(a) : -true \mid true.$$

のような定義に対して、

$$? - p(\_).$$

のような呼び出しを行なうと、前述のような一括GCによる検出に頼らなくとも、参照パスが1本しかないものにフックしようとしたということで、即時に検出/報告を行なうことができる。

この検出を行なうため、HOOK 系の未定義変数のオブジェクト側のMRBに意味を持たせた。すなわち、ある引数が未定義であったためにサスペンド処理を行なう時、その未定義変数へのパスが白かった場合 HOOK$_\circ$, 黒かった場合HOOK$_\bullet$ と書き込むことにした。

これにより、全部で6通りのパターンで即時検出が行なえるようになった。なお、マージャ(REF$_\circ$ → MGHOK)の場合 2,4 が起こり得ないので全部で4通りである。

1. REF$_\circ$ → HOOK$_\circ$ に対する `collect_value` (図12–7)
2. REF$_\circ$ → HOOK$_\circ$ に対する 白パスでのサスペンド操作 (図12–8)
3. REF$_\circ$ → HOOK$_\circ$ と REF$_\circ$ → HOOK$_\circ$ のアクティブユニファイ (図12–9)
4. REF$_\circ$ → VOID に対する 白パスでのサスペンド操作 (図12–10)
5. REF$_\circ$ → HOOK$_\circ$ と REF$_\circ$ → VOID のアクティブユニファイ (図12–11)
6. REF$_\circ$ → EHOOK$_\circ$ に対する `%release`受信 (図12–7)

なお、2, 3 では、同時に二つの永久中断ゴールを発見したことになるので、それぞれの所属する荘園に対して(同時に二つの)永久中断例外を上げる。なお、報告の形式については、10.10節を参照のこと。

12.4 トレース

12.4.1 トレースの指示

トレースモードのゴールを生成するために、VPIM ではMulti-PSI用処理系と全く同じ仕様の組込述語

apply_tracing(Code, Argv, TraceID)

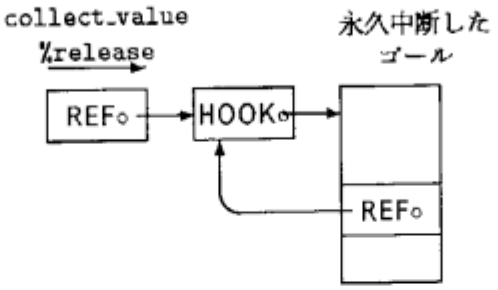


図 12-7: REF∘ → HOOK∘ に対する collect_value, %release

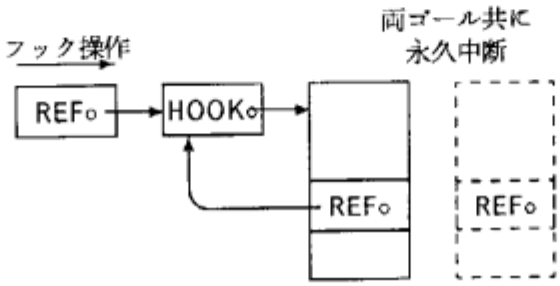


図 12-8: REF∘ → HOOK∘ に対する 白パスでのサスペンド操作

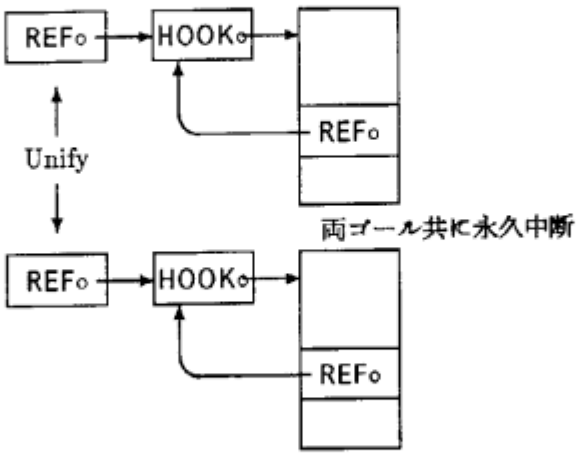


図 12-9: REF∘ → HOOK∘ と REF∘ → HOOK∘ のアクティブユニファイ

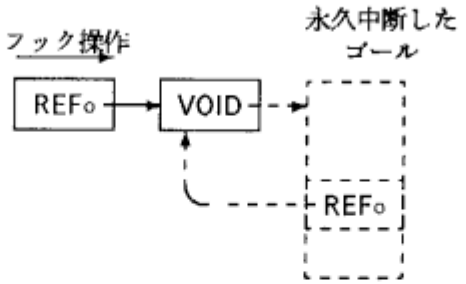


図 12–10: REF∘ → VOID に対する 白パスでのサスペンド操作

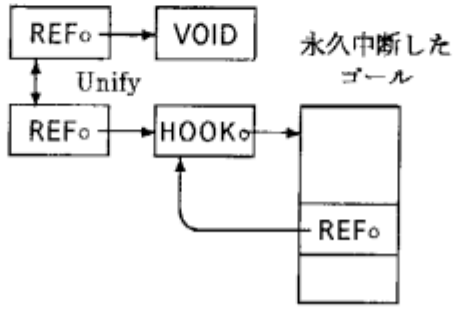


図 12–11: REF∘ → HOOK∘ と REF∘ → VOID のアクティブユニファイ

を用意した。これはCodeで指定される述語をArgvを引数としてトレースモードで実行するサブゴールを生成するものである。このサブゴールにはTraceIDが付加されるが、これはトレースの指示を識別するための情報であり、後でトレース例外が発生した時に例外メッセージを作成するのに使われる。これには未定義変数を含む任意の型のデータを使用できる。

この組込述語における処理は、

- この組込述語を実行しようとした時に既にトレースモードになっていれば、この組込述語自身をトレース例外として報告する。
- この組込述語を実行しようとした時にスパイモード(次項を参照)になっていれば、無条件でこの組込述語自身をスパイ例外として報告する。
- 第1引数Codeが未定義変数ならばサスペンドさせる。また、そのデータ型がコード以外であればIllegal Input 例外を報告する。
- 第2引数Argvが未定義変数ならばサスペンドさせる。また、そのデータ型がベクタ以外であればIllegal Input 例外を報告し、その要素数がCodeで指定される述語の引数個数と異なる場合にはArity Mismatch例外を報告する。

というような検査を行なった後、全てをパスした場合には、

- Codeで指定される述語をArgvを引数としてトレースモードで実行するようなサブゴールを生成しエンキューする。この時、TraceIDをゴール(正確には環境レコード, 2.4節参照)に付加する。

というものである。

12.4.2 トレース例外の処理方式

トレース例外のための処理は、トレースモードのゴールに対して行なう。このトレースモードのゴールの実行は特別な方式で行なう。即ち、このモードではゴールのリダクションだけを行ない、その結果のサブゴールをリストにまとめてトレース例外として報告する。サブゴールのエンキューは行なわない。なお、ユニフィケーションや組込述語の場合も実行は行なわずに、サブゴールの1種として扱う。

このための具体的な処理は以下のようになる。

① 専用レジスタとしてサブゴールリストを保持するレジスタ **TSGL** (Top of Sub Goal List) [4] とクローズID(整数)を記憶するレジスタ **CID** (Clause ID) [5] の2つが必要になる。また、トレースモードか否かを示すフラグも必要である。

② ゴールをデキューしてきた時にそれがトレースモードになっていた場合すなわち、環境レコードへのポインタがVECT3であった場合には、トレースモードを示すフラグをセットする。実際には、D_CurrentTraceId に トレースID を書き込む。トレースモードを示すフラグは、EOL タイプでリセットされている状態を示す。これは、EOL が KL1 ユーザにセットできないタイプであることを利用している。

③ トレースモードでcommit命令を実行した場合には、**TSGL** レジスタを □ で初期化する。また、commit命令の引数にあるクローズIDを**CID**レジスタに覚えておく。

④ ボディ部の処理を行なおうとした時に、トレースモードになっていた場合には通常の処理の代わりに、以下のように、サブゴール情報を生成しサブゴールリストに追加する処理を行なう。なお、この処理は、

TSGL := [サブゴール情報 | **TSGL**]

という操作により行なう。

- アクティブユニフィケーション命令を実行しようとした時には、

  {ユニフィケーションを表す番号, {DataX, DataY}}

  というサブゴール情報をサブゴールリストに追加する。ここで、**ユニフィケーションを表す番号**の値としてはKL1B-instruction-IDを用いる。

- 組込述語(enqueue系組込述語を除く)を実行しようとした時には、

  {組込述語の種類を表す番号, 引数を格納したベクタ}

  というサブゴール情報をサブゴールリストに追加する。ここで、**組込述語の種類を表す番号**の値としてはKL1B-instruction-IDを用いる。

- enqueue系命令、enqueue系組込述語を実行しようとした時には、その種類に応じて、

[4] VPIM のソースでは D_TraceSubGoalListTop という名前になっている。

[5] VPIM のソースでは D_CurrentTraceId という名前になっている。

{enqueue の種類を表す番号, コード, 引数ベクタ }

{enqueue の種類を表す番号, コード, 引数ベクタ, 論理実行優先度 }

{enqueue の種類を表す番号, コード, 引数ベクタ, クラスタ番号 }

{enqueue の種類を表す番号, コード, 引数ベクタ, プロセッサ番号 }

というサブゴール情報をサブゴールリストに追加する。ここで、enqueue の種類を表す番号の値としてはKL1B-instruction-IDを用いる。

- execute系命令を実行しようとした時には、

  {execute の種類を表す番号, コード, 引数ベクタ }

  というサブゴール情報をサブゴールリストに追加し、その後、次で説明する proceed 命令と同じ処理を行なう。ここで、execute の種類を表す番号の値としては KL1B-instruction-IDを用いる。

⑤ トレースモードでproceed命令を実行した場合には、通常のproceed処理の前に、サブゴールリストを引数に持つトレース例外のメッセージ

{ 例外を表すメッセージ番号, トレース例外を表す例外番号,
{ クラスタ番号とプロセッサ番号,
トレース ID のリスト, サブゴールリスト, クローズ ID},
代わりに実行する述語コード, 代わりに実行する述語の引数ベクタ }

を荘園のレポートストリームに報告する。

以上のように、トレースモードでの実行は通常の場合とは大きく異なり、ボディ部におけるユニフィケーション、組込述語、サブゴールのエンキューの処理を全く別のものにする必要がある。従って、これらの操作の先頭ではトレースモードか否か[6]の検査を毎回行なう必要がありオーバーヘッドが大きい。従って、Multi-PSIにあるKL1-B命令単位の割り込み機能のようなハードウェアサポートがない場合には、KL1コンパイラが

- トレース可能だが検査のオーバーヘッドがあって遅くなってしまうモード
- トレース不可能だが検査のオーバーヘッドが無くて速いモード

という2種類のコードを生成できるようにすべきである。

12.5 スパイ

12.5.1 スパイの指示

スパイモードのゴールを生成するために、VPIMではMulti-PSI用処理系と全く同じ仕様の組込述語

apply_spying(Code, Argv, SpiedPredVector, SpyID)

[6] 具体的には、D_CurrentTraceId が EOL か否かの検査。

を用意した。これはCodeで指定される述語をArgvを引数として、述語群SpiedPredVectorを対象とするスパイモードで実行するサブゴールを生成するものである。このサブゴールにはSpyID が付加されるが、これはスパイの指示を識別するための情報であり、後でスパイ例外が発生した時に例外メッセージを作成するのに使われる。これには未定義変数を含む任意の型のデータを使用できる。

この組込述語における処理は、

- この組込述語を実行しようとした時に既にトレースモードになっていれば、この組込述語自身をトレース例外として報告する。
- この組込述語を実行しようとした時にスパイモードになっていれば、無条件でこの組込述語自身をスパイ例外として報告する。
- 第1引数Codeが未定義変数ならばサスペンドさせる。また、そのデータ型がコード以外であればIllegal Input 例外を報告する。
- 第2引数Argvが未定義変数ならばサスペンドさせる。また、そのデータ型がベクタ以外であればIllegal Input 例外を報告し、その要素数がCodeで指定される述語の引数個数と異なる場合にはArity Mismatch例外を報告する。
- 第3引数SpiedPredVectorが未定義変数ならばサスペンドさせる。また、そのデータ型がベクタ以外であるか、要素数が3の倍数でないか、各要素がアトムまたは整数でない場合にはIllegal Input 例外を報告する。要素の具体化はユーザの責任で行ない、要素が未定義であるからといって中断する仕様ではなく、Illegal Input 例外となる。

というような検査を行なった後、全てをパスした場合には、

- Codeで指定される述語をArgvを引数としてスパイモードで実行するようなサブゴールを生成しエンキューする。この時、SpyIDをゴールに付加する。

というものである。

この組込述語では、スパイ対象の述語群はSpiedPredVectorで指定する。これは長さが3の倍数で、要素にスパイ対象の述語のモジュール名、述語名、引数個数が繰り返し格納されているベクタ、

{ModName1,PredName1,PredArity1, ModName2,PredName2,PredArity2, … }

というようなものである。なお、この要素は通常、モジュール名と述語名はアトムで、引数個数は整数であるが、モジュール名と述語名では整数を、引数個数ではアトムを指定することにより全てのデータにマッチするワイルドカードを表現することができる。

12.5.2 スパイ例外の処理方式

スパイ例外のための処理は、スパイモードのゴールに対して行なう。このスパイモードのゴールの実行では、サブゴールをエンキューしようとした時に、それがスパイ対象であるかを検査し、対象ならばそのサブゴールをスパイ例外として報告する。この場合サブゴールのエンキューは行なわない。なお、スパイ対象でない場合には、サブゴールをスパイモードでエンキューする。

このための具体的な処理は以下のようになる。

① スパイモードか否かを示すフラグを用意する必要がある[7]。

② ゴールをデキューしてきた時にそれがスパイモードになっていた場合、すなわち環境レコードへのポインタがVECT8になっていた場合には、スパイモードを示すフラグをセットする[8]。

③ ボディ部の処理を行なおうとした時に、スパイモードになっていた場合には通常の処理の代わりに、以下のように、サブゴール情報を生成し、それを引数に持つ例外メッセージ

{ 例外を表すメッセージ番号, スパイ例外を表す例外番号,
{ クラスタ番号とプロセッサ番号, スパイ ID, サブゴール情報 },
代わりに実行する述語コード, 代わりに実行する述語の引数ベクタ }

を荘園のレポートストリームに報告する。

- enqueue系命令、enqueue系組込述語を実行しようとした時には、それにより生成するサブゴールがスパイ対象か否かを検査し、もしスパイ対象ならば、enqueueの種類に応じて、

{enqueue の種類を表す番号, コード, 引数ベクタ }
{enqueue の種類を表す番号, コード, 引数ベクタ, 論理実行優先度 }
{enqueue の種類を表す番号, コード, 引数ベクタ, クラスタ番号 }
{enqueue の種類を表す番号, コード, 引数ベクタ, プロセッサ番号 }

というサブゴール報情を持つスパイ例外を報告する。ここで、**enqueue の種類を表す番号**の値としてはKL1B-instruction-ID を用いる。

- execute系命令を実行しようとした時には、それにより生成するサブゴールがスパイ対象か否かを検査し、もしスパイ対象ならば、

{execute の種類を表す番号, コード, 引数ベクタ }

というサブゴール情報を持つスパイ例外を報告し、その後、proceed命令と同じ処理を行なう。ここで、**execute の種類を表す番号**の値としてはKL1B-instruction-IDを用いる。

- apply組込述語を実行しようとした時には、それにより生成するサブゴールがスパイ対象か否かを検査し、もしスパイ対象ならば、

{ 組込述語 apply を表す番号, apply の引数のベクタ }

というサブゴール情報を持つスパイ例外を報告する。ここで、**組込述語 apply を表す番号**の値としてはKL1B-instruction-IDを用いる。

- apply_tracing, apply_spying, create_shoen, create_profiling_shoen の各組込述語を実行しようとした時には常に、

[7] VPIM のソースではD_CurrentSpyIdを用い、このタイプがEOLの時スパイモードでない状態を表し、それ以外の時はそのスパイ ID のスパイモードであることを示している。EOLがKL1 ユーザから設定できないタイプであることを利用している。

[8] 同時にスパイIDがセットされる。

{ 組込述語の種類を表す番号, 組込述語の引数のベクタ }

というサブゴール情報を持つスパイ例外を報告する。ここで、組込述語の種類を表す番号の値としてはKL1B-instruction-IDを用いる。

④ スパイモードの時にenqueue, apply等によりサブゴールを生成する場合には、そのサブゴールも親と同じ述語群をスパイ対象とするスパイモードで生成する必要がある。これは、クラスタ間にまたがる場合も同様で、ゴール送信の処理においてスパイモードのための情報/属性を正しく伝達する必要がある。このため、

$$dcode\_enqueue\_cluster\_with\_spy\_mode(Cluster, Code, Argv, SpiedCodeV, SpyId) :- integer(Cluster), wait(Code) \mid builtin : apply\_spying(Code, Argv, SpiedCodeV, SpyId)@cluster(Cluster).$$

というDコードゴール(5.2節参照)を用いる。

第 13 章

SCSI

執筆担当者: 武井, 堂前

本章では、ディスクやFEP(フロントエンドプロセッサ)を操作するためのSCSI (Small Computer System Interface)のバス制御やそれを操作するKL1組込述語の実装方式について説明する。

13.1 概要

KL1プログラムからのSCSI操作は、すべて組込述語によって実現される。SCSI組込述語は、SCSI操作に対する予約をするというインタフェースとなっている。

予約を受け付けたひとつのSCSI操作(組込述語)は、操作レコードの形で表現し、このレコードをキュー管理することで、実行順序の制御を行う。

SCSIバスの制御は、イニシエータ動作・ターゲット動作のそれぞれについて状態遷移を行うオートマトンとして記述している。SCSI上での実際の信号制御などは、SPC (SCSIプロトコルコントローラ)を用いる。

以下に、VPIMにおけるSCSIバス制御、及びSCSI組込述語について記す。

13.2 SCSIバス制御

13.2.1 処理方式概要

SCSIバス制御は、イニシエータ時・ターゲット時それぞれについて、状態(フェーズ)遷移モデルを想定しそれにしたがって制御を進めるという方法をとっている。状態遷移モデルでは、SCSIバス・フェーズの実行を制御する上で、正常と思われるフェーズの遷移をメインパスとして考え、それから外れる場合をエラーとして処理する。

状態遷移モデルでの、各状態での処理はPSLのサブルーチンとして定義している。状態遷移に基づく一連のバス・シーケンスの制御は、各サブルーチンの返す「次に進むべき状態」をもとに、その状態に対応するサブルーチンを呼び出していくことで実現している。これによって、ディスコネクト・リコネクト処理を簡便にでき、またエラー検出時の詳細な情報を把握できる。

一連のバス・シーケンスの実行の正常終了後は、KL1への結果報告処理[1]を行う。個々の状態で何らかの異常を検出した場合には、エラーリカバリをせずにその時点でSCSIバス上でのシーケンスを終結させ、エラー終了とする。エラー終了時には、「異常」についての情報をKL1へ返す。

本節では、状態遷移に基づく個々の状態での処理について述べる。

13.2.2 イニシエータ状態遷移

(1) 概要

イニシエータ時の動作は、SCSI上でのターゲットの要求フェーズに応答しながら処理を進めていく。

イニシエータ時の状態遷移としては、

アービトレーション → セレクション → コマンド → データ →
ステータス → Command Complete メッセージ受領

というフェーズの移行を期待している(図13-1, 13-2参照)。この間、ターゲットがディスコネクトを要求してくることがある。その場合には、リコネクト後に再開すべき状態をコマンドレコード内に保存しておき、リコネクト時にその状態から制御する。また、ターゲットがこれ以外のフェーズの移行をする場合には、ディスコネクト要求以外はエラー終了となる。

(2) セレクション (図 13-1 **Selection & Send Identify** 参照)

アービトレーション・フェーズ、セレクション・フェーズを行い、成功後にIdentifyメッセージ(及び拡張Identifyメッセージ)を送る。

① アービトレーション・セレクションフェーズの実行

アービトレーション・セレクションフェーズの実行は、SPCのSelectコマンドによって行う。実行結果と処理内容は以下の通り。

- セレクションに成功した場合。
  Identifyメッセージの送出に移る。
- アービトレーションに失敗した場合(図中 *1)。
  コマンドレコードをバス権キューにエンキューして、予約操作に関する当面の処理を終え、ゴールリダクションに戻る。この際、バスフリー・フェーズ検出割り込みの制限を外す。
  (後述 (8) エラーハンドリング)
- 他バスデバイスからセレクトされた場合(図中 *2)。
  コマンドレコードをバス権キューにエンキューし、セレクト時の処理に移る。
- 他バスデバイスからリセレクトされた場合(図中 *3)。
  コマンドレコードをバス権キューにエンキューし、リセレクト時の処理に移る。
- タイムアウトを検出した場合(図中 *4)。
  セレクションタイムアウト検出によるエラー終了とする。

[1]実行結果のKL1への通知を行う。KL1変数の具体化による。

- バスフリーフェーズを検出した場合(図中 *1)。

  コマンドレコードをバス権キューにエンキューして、次の予約操作の処理に移る。

- ハードウエアエラーを検出した場合(後述 (8) エラーハンドリング)。

  Selectコマンドのアボート処理に移る。

- リセットコンディションを検出した場合(後述 (8) エラーハンドリング)。

  リセットコンディション検出によるエラー終了とし、リセットコンディション処理に移る。

② Identifyメッセージの送出(図 13-1 **Send Identify**参照)

セレクション成功後、メッセージアウト・フェーズになったところでIdentifyメッセージ(及び拡張 Identify メッセージを送出する。メッセージアウト・フェーズになっていない場合は、オプションメッセージを使わないコマンド実行とする。(現在はサポートしていない。)

Identify メッセージの送出は、SPCのTransferコマンドを用いて行う。Identify メッセージ送出の実行後の状態として、以下の場合がある。

- 送出が正常に終了した場合。

  コマンド転送に移る。

- エラーを検出した場合。

  以下の場合がある。

  ◇ パリティエラーを検出した場合。

  ◇ ハードウエアエラーを検出した場合。

  ◇ 転送途中でフェーズが変更された場合。

  この場合は、ディスコネクト要求を考えない。

  以上の場合は、転送アボート処理を行いエラー終了とする(図中 *5)。

  ◇ バスフリーフェーズを検出した場合。

  エラー終了とし、結果報告処理に移る。

  ◇ リセットコンディションを検出した場合。

  リセットコンディション検出によるエラー終了とし、リセットコンディション処理に移る。

(3) コマンド転送 (図 13-1 **Command** 参照)

ここでは、コマンド・フェーズでCDBの送出を行う。正常にCDBを送出できた場合のみ、次のデータ転送に移る。他のフェーズをターゲットが要求してくる場合には、それに応じた処理を行う。以下にターゲットの要求フェーズ毎の処理を記す。

① コマンド・フェーズ

SPCのTransferコマンドによってCDBの送出を行う。実行結果と処理内容は以下の通りとなる。

- 転送が正常終了した場合。

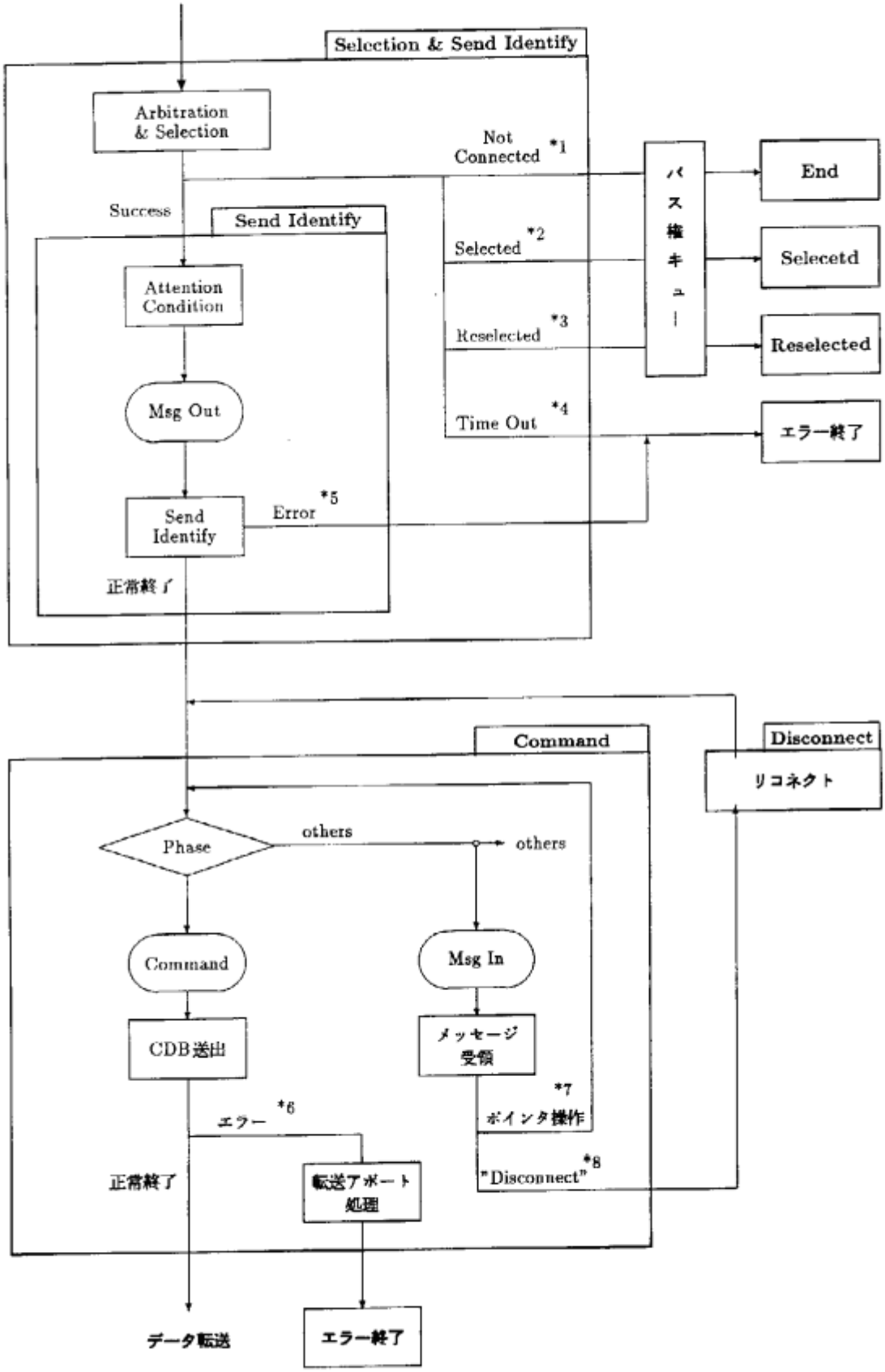


図 13-1: イニシエータ状態遷移(その1)

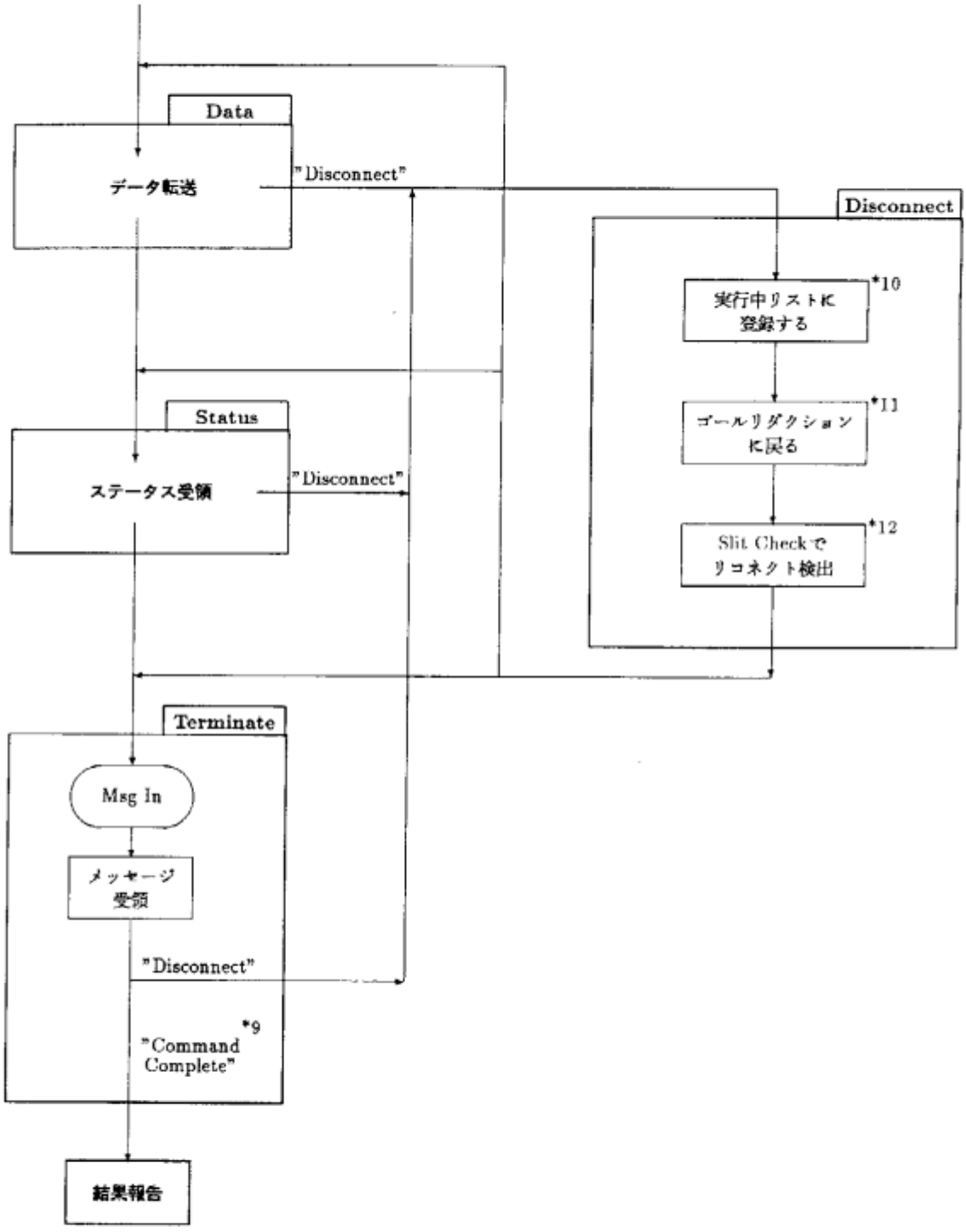


図 13-2: イニシエータ状態遷移(その2)

次の状態(この場合はデータ転送)に移る。

- エラーを検出した場合。

以下の場合がある。

◇ パリティエラーを検出した場合。

◇ ハードウエアエラーを検出した場合。

◇ 転送途中でフェーズが変更された場合。

この場合はディスコネクト要求を考えない。

以上の場合は、転送アボート処理を行いエラー終了とする(図中 *6)。

◇ バスフリーフェーズを検出した場合。

エラー終了とし、結果報告処理に移る。

◇ リセットコンディションを検出した場合。

リセットコンディション検出によるエラー終了とし、リセットコンディション処理に移る。

② メッセージイン・フェーズ

メッセージを受領し、メッセージ毎の処理を行う。

- "Save Data Pointer"メッセージの場合(図中 *7)。

カレントデータポインタをコマンドレコード内の待避領域に格納する。現在の状態(この場合はコマンド転送)のトップレベルに戻る。

- "Restore Pointers"メッセージの場合(図中 *7)。

コマンドレコード内のデータポインタをカレントデータポインタとしてロードする。現在の状態(この場合はコマンド転送)のトップレベルに戻る。

- "Disconnect"メッセージの場合(図中 *8)。

現在の状態(この場合はコマンド転送)をリコネクト後に再開すべき状態としてコマンドレコード内に保存する。バスフリーフェーズ確認後、コマンドレコードを実行中リストに登録する。

- "Command Complete"メッセージの場合。

不適切であるが、完了状態をメッセージ受領にする。エラー終了とする。

上記以外のメッセージであった場合は、全てエラーとして転送アボート処理を行う。

③ メッセージアウト・フェーズ

当面送出すべきメッセージがないので、"NOP"メッセージを送出した後、現在の状態(この場合はコマンド転送)のトップレベルに戻る。

④ 上記以外のフェーズ(データイン・フェーズ、データアウト・フェーズ、ステータス・フェーズ)

状態遷移から制御できないフェーズ要求として、転送アボート処理を行いエラー終了とする。

(4) データ転送 (図 13–2 **Data** 参照)

ここでは、組込述語の引数で指定されるデータイン・フェーズ或いはデータアウト・フェーズでのデータの送受信を行う。正常にデータの送受信が完了した場合に、次のステータス転送に移る。また、データ転送を伴わないコマンドの実行もあるので、ターゲットがステータス・フェーズを要求してきた場合にもステータス転送に移る。他のフェーズをターゲットが要求してくる場合には、それに応じた処理を行う。以下にターゲットの要求フェーズ毎の処理を記す。

① データイン・フェーズ/データアウト・フェーズ

組込述語の引数に指定されたフェーズと一致しない場合には、状態遷移から制御できないフェーズ要求として、転送アボート処理を行いエラー終了とする。

そうでなければ、SPCのTransferコマンドによってデータ転送を行う。実行結果と処理については、コマンド転送の場合に準ずるが、以下の場合のみ異なる。

- 転送途中でフェーズが変更された場合。

  ターゲットがディスコネクト要求のためにフェーズを変更した場合を考えて、現在の状態(この場合はデータ転送)のトップレベルに戻る。

② メッセージイン・フェーズ/メッセージアウト・フェーズ

コマンド転送の場合の処理に準ずる。

③ ステータス・フェーズ

データ転送を伴わないコマンド実行として、ステータス転送に移る。

④ 上記以外のフェーズ(コマンド・フェーズ)の場合

状態遷移から制御できないフェーズ要求として、転送アボート処理を行いエラー終了とする。

(5) ステータス転送 (図 13–2 **Status** 参照)

ここでは、ステータス・フェーズでステータスバイトの受領を行う。正常にステータスバイトの受領が完了した場合にのみ、次の完了報告に移る。他のフェーズをターゲットが要求してくる場合には、それに応じた処理を行う。以下にターゲットの要求フェーズ毎の処理を記す。

① ステータス・フェーズ

SPCのTransferコマンドによってステータスバイトの受領を行う。実行結果と処理については、データ転送の場合に準ずる。

② メッセージイン・フェーズ/メッセージアウト・フェーズ

コマンド転送の場合の処理に準ずる。

③ 上記以外のフェーズ(コマンド・フェーズ、データイン・フェーズデータアウト・フェーズ)の場合

状態遷移から制御できないフェーズ要求として、転送アボート処理を行いエラー終了とする。

(6) 完了報告 (図 13-2 **Terminate** 参照)

ここでは、メッセージイン・フェーズで"Command Complete"メッセージの受領を行う。正常に"Command Complete"メッセージを受領した場合にのみ、このコマンド実行処理の正常終了とし、結果報告処理に移る。他のフェーズをターゲットが要求してくる場合には、それに応じた処理を行う。以下にターゲットの要求フェーズ毎の処理を記す。

① メッセージイン・フェーズ

コマンド転送時の処理に準ずるが、受領したメッセージが以下の場合のみ異なる。

- "Command Complete"メッセージの場合(図中 *9)。

  正常終了として、結果報告処理に移る。

② 上記以外のフェーズ(コマンド・フェーズ、データイン・フェーズ、データアウト・フェーズ、ステータス・フェーズ)

状態遷移から制御できないフェーズ要求として、転送アボート処理を行いエラー終了とする。

(7) ディスコネクト・リセレクト (図 13 2 **Disconnect** 参照)

① ディスコネクト時の処理

ターゲットからの"Disconnect"メッセージ受領時の処理は既に述べた。コマンドレコードを実行中リストに登録(図中 *10)後は、次の予約操作のレコードをバス権キューからデキューして実行に移る。バス権キューが空なら当面の処理を終え、ゴールリダクションに戻る(図中 *11)。

② リコネクト時の処理

ターゲットからのリコネクト要求は、SPC の割り込み(Reseleted 割り込み)によって通知される。

リセレクトの割り込みは、以下のタイミングで検出される。

- ゴールリダクション時のスリットチェック(図中 *12)
- GC 時の割り込み
- セレクション/リセレクション起動時(図中 *3 参照)

割り込み検出後の処理は以下のようになる。

1. リセレクション後のフェーズとしてメッセージイン・フェーズを待ち、ターゲットから Identify メッセージ(及び拡張 Identify メッセージ)を受領する。

   それ以外のフェーズ要求があった場合は、転送アボート処理を行う。

2. リセレクション時のターゲットの Device Number と Identify メッセージ(及び拡張 Identify メッセージ)より、実行中リストを検索する。

   検索結果により以下の処理にわかれる。

   - 該当するコマンドレコードがなかった場合は転送アボート処理を行う。

- 該当するコマンドレコードがキャンセルされていた場合は、転送アボート処理を行い結果報告処理に移る。
- 上記以外の場合は、リコネクト後の処理に移る。

3. リコネクト後は、ディスコネクト時にコマンドレコード内に保存した再開状態から実行を始める。

(8) エラーハンドリング

転送アボート処理、Selectコマンドのアボート処理、リセットコンディション検出時の処理、アービトレーション失敗後の処理、及び非同期に起こり得る割込みの検出について記す。アービトレーションの失敗は、エラーではないがここにおいてその処理を記す。

① 転送アボート処理

エラー終了とする場合に、ターゲットとの接続をアボートによって終了させる。イニシエータから接続を切るために、アテンションコンディションを生成し、メッセージアウト・フェーズで"Abort"メッセージを送出する。バスフリー・フェーズの検出をもって処理の終了とする。

② Selectコマンドのアボート

SPCのSelectコマンド発行後に、SPCがハードウエアエラーを検出した場合、正常の動作シーケンスの保証がされない。SPCによるタイムアウト検出が正常に機能しない可能性があるため、制御プログラムで監視する必要がある。

ここでは、Selectコマンドの終了を確認後、以下の処理を行う。

- セレクションが成功していた場合には、転送アボート処理を行い、ターゲットとの接続を切る。
- それ以外の場合の処理については、前述のSelectコマンド実行後の処理に準ずる。

③ リセットコンディション検出時の処理

SCSIバス上でリセットコンディションを検出した場合、全ての予約操作をリセットコンディション検出によるエラー終了とする。

実行リスト、バス権キュー、予約キュー内のコマンドレコード・転送レコードをエラー終了として結果報告処理に移す。

その後、SPCを初期化してリセット状態を解除する。

④ アービトレーション失敗後の処理

アービトレーション失敗時のコマンドレコードをバス権キューにエンキューした後、セレクション実行のリトライのために、以下の処理を行う。

1. SPCのバスフリー・フェーズ検出割り込みの抑制を外し[2]、バスフリー・フェーズ検出をスリットチェックにおいて検出可能にしておく。

[2]通常は、SPCのバスフリー・フェーズ検出割り込みの条件を、イニシエータとしてSCSIに接続中の場合に限っている。この条件を動作モードに関係なくバスフリー・フェーズの検出時とする。

2. 当面の処理を終えて、ゴールリダクションに戻る。
3. スリットチェックでバスフリーフェーズ検出割り込みであったら、バス権キューから予約操作のレコードをデキューして実行に移る。

⑤ 非同期に起こり得る割り込みの検出について

SCSIとイニシエータとして接続中、非同期に起こり得る割り込みとしては、リセットコンディション、及びバスフリー・フェーズの検出がある。

これらの検出については、状態遷移の各状態での最初の処理として行うものとする。検出後の処理については、これまでに述べた各割り込みの処理と同様である。

13.2.3 ターゲット状態遷移

(1) 概要

ターゲットとしての動作には、主に2つの処理がある。第一はSCSIからの選択に対する処理で、第二は組込述語scsi_transfer/6に対する転送実行の処理である(図13-3参照)。

セレクト時の動作は、

セレクト → メッセージ受領 → KL1 プログラムへの報告

となる(図中 *1)。

組込述語の仕様は1回の呼び出し・1回の転送となっている(図中 *4)。このため、ターゲット動作の状態遷移は、

アービトレーション → リセレクション → 転送実行(組込述語の引数で指定)
→ ディスコネクト・バス解放(組込述語の引数で指定)

となる。この間、エラーおよびアテンションコンディションの検出をした場合には、処理を止めエラー終了処理に移る。

(2) セレクト (図 13-3 Select 参照)

SCSIからのセレクトはSPCの割り込みによって通知される。SCSI上でのセレクションフェーズへの応答はSPCが自動的に行う。処理内容は以下の通り。

① セレクションレコードを1つとってくる。

セレクションレコードは、本来的にはGC等の理由により、KL1への通知を後回しにする場合に用いられる。ここでは、セレクション時の情報の格納領域として利用する。

② イニシエータのDevice Numberをとり出し、整数値としてセレクションレコードに格納する(図中 *2)。

③ アテンションコンディションになっているかどうかにより、以下の2通りにわかれる。

- アテンションコンディションになっている時(図中 *3)。
  - アテンションコンディションの間、バッファにあふれない限りメッセージを読み込んで、セレクションレコードに格納する。

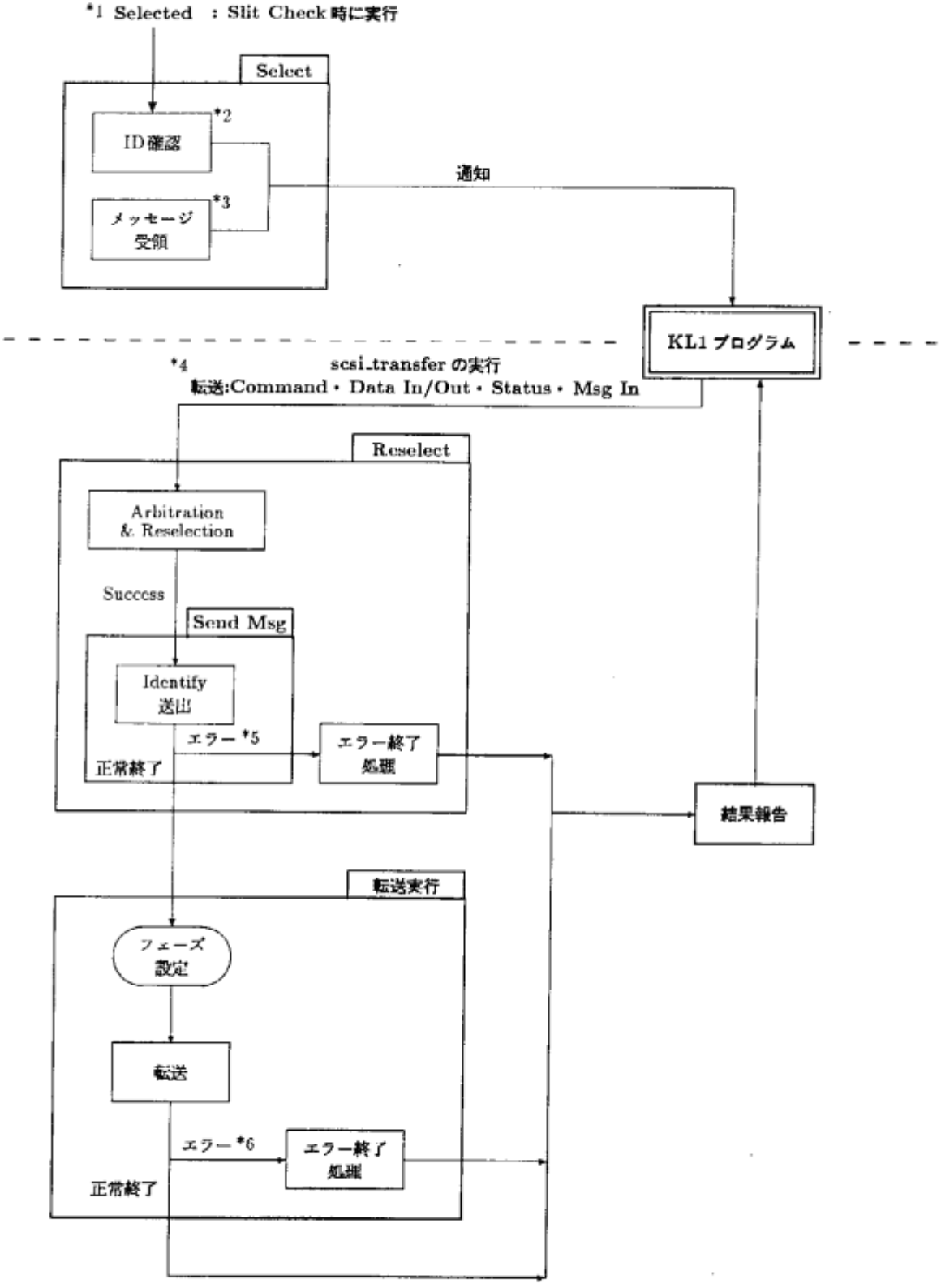


図 13-3: ターゲット状態遷移

- アテンションコンディションになっていない時。
  - ◇ オプションメッセージなしの実行が行われる。
    (現在はサポートしていない。)

④ KL1プログラムへの結果通知処理に移る。

(3) リセレクト (図 13-3 **Reselect** 参照)

アービトレーション・フェーズ、リセレクション・フェーズを行い、成功後に Identify メッセージ (及び、拡張 Identify メッセージ) を送る。

① アービトレーション・リセレクションフェーズ

イニシエータ状態遷移・セレクトに準ずる。

② Identify メッセージの送出 (図 13-3 **Send Msg**参照)

リセレクション成功後、メッセージイン・フェーズとして Identify メッセージ(及び拡張 Identify メッセージ)を送出する。Identify メッセージの送出は、SPC の Transfer コマンドを用いて行う。Identify メッセージ送出の実行後の状態として、以下の場合がある。

- 送出が正常に成功した場合。
  組込み述語の引数で指定されたフェーズでの転送に移る。
- エラー(パリティエラー、ハードウエアエラー)及びアテンションコンディションを検出した場合(図中 *5)。
  エラー終了処理を行い KL1 プログラムへの結果報告処理に移る。
- リセットコンディションを検出した場合。
  リセットコンディション検出によるエラー終了とし、リセットコンディション処理に移る。

(4) 転送実行 (図 13-3 転送実行参照)

転送実行は、組込み述語の引数で指定されたフェーズでの転送を行う。転送は、コマンド転送の第1バイト目を除いて SPC の Transfer コマンドを用いて行う。入力動作と出力動作の二つの処理に分けられるが、基本的に同様の処理となる。

転送中に SPC の割り込み(アテンションコンディションを含む)が検出された場合は、直ちに転送を止め、実行結果の解析を行い終了処理に移る。

以下に転送実行後の処理内容を述べる。

- 転送が正常に終了した場合。
  KL1 プログラムへの結果報告処理へ移る。
- エラー(パリティエラー、ハードウエアエラー)及びアテンションコンディションを検出した場合(図中 *6)。
  エラー終了処理を行い KL1 プログラムへの結果報告処理に移る。
- リセットコンディションを検出した場合。

リセットコンディション検出によるエラー終了とし、リセットコンディション処理に移る。

(5) エラーハンドリング

ターゲット状態遷移でのエラーハンドリングとしては、エラー終了処理、Selectコマンドのアボート処理、リセットコンディション検出時の処理、アービトレーション失敗後の処理、及び非同期に起こり得る割り込みの検出についての処理がある。

Selectコマンドのアボート処理、リセットコンディション検出時の処理、アービトレーション失敗時の処理については、イニシエータ状態遷移・エラーハンドリングに準ずる。

① エラー終了処理

転送中のエラー、アテンションコンディション検出の際の処理を行う。処理内容の概要は以下の通りである。

- アテンションコンディションになっていれば、アテンションコンディションが続く間、バッファがあふれない限り、イニシエータからのメッセージを読み込んで、バッファに格納する。
- KL1プログラムへの結果報告処理に移る。

② 非同期に起こり得る割り込みの検出について

SCSIとターゲットとして接続中、非同期に起こり得る割り込みとしては、リセットコンディション、及びアテンションコンディションの検出がある。

これらの検出については、状態遷移の各状態の最初の処理として行うものとする。検出後の処理については、これまでに述べてきた割り込みの処理と同様である。

13.2.4 SPCを用いたバス・シーケンスの実行

SCSIプロトコルコントローラ MB87033Bを用いたバス・シーケンスの実行について、VPIMでの使用に基づいて簡単に述べる。

(1) SPCコマンド

SPCはバス・シーケンス実行のために、いくつかのコマンドを備えている。このコマンド発行によって、任意のバス・シーケンスを実行できる。タイミング規約などのプロトコルはSPCがサポートする。コマンドの実行結果はSPCの割り込みによって通知される。VPIMからの操作は、コマンドをSPCに対して発行し、実行結果を割り込み解析によって判断することになる。以下にSPCコマンドのいくつかをあげる。

- Selectコマンド

  アービトレーション、セレクション(リセレクション)を実行する。

- Transferコマンド

指定したフェーズでの転送を行う。コマンド発行後、VPIMからはSPC内部のデータバッファレジスタへのアクセスすることで、SCSIとの情報のやりとりができる。

(2) SPC の割り込み

SPCの生成する割り込みは、前述のSPCコマンド実行終了の通知の他、SCSIからセレクト(リセレクト)を受けた場合や、リセットコンディション、アテンションコンディションの検出も割り込みの対象となる。

SPCの持つ割り込みフラグには以下のものがある。

Selected	:: セレクトされた
Reselected	:: リセレクトされた
Disconnected	:: バスフリー・フェーズを検出した
Command Complete	:: SPCコマンドが終了した
Service Required	:: イニシエータ時：転送動作中にターゲットがフェーズを変更した
	:: ターゲット時：アテンションコンディションを検出した
Time Out	:: セレクション(リセレクション)タイムアウトを検出した
SPC Hardware Error	:: ハードウエアエラーを検出した
Reset Condition	:: リセットコンディションを検出した

SPCからの割り込みは、CPUへの割り込み要因になるが、通常は割り込み禁止モードとしてスリットチェックの対象とする。但し、GC中には割り込み許可モードとして、SCSIからセレクト(リセレクト)に応じる。

13.3 SCSI組込述語

まずPIMOSからのSCSI組込述語の呼び出され方を概説する。PIM実機上でPIMOSを立ち上げる際、まずSCSIデバイスドライバ・プロセスが生成され、その中でのみSCSI組込述語が呼ばれる。SCSIデバイスドライバを生成するのは、PIMOSのブートストラップルーチンあるいはPIMOS自身である。SCSIデバイスドライバは、KL1で書いたPIMOSローダやユーザ・プロセスとストリームで結合され、そのストリーム上にopen / close / read / writeといったコマンドが流れる。SCSIデバイスドライバは、あるクラスタのあるPEにレジデントである。従って、SCSI組込述語もSCSIバス・インタフェースを持った特定のCPUからのみ呼ばれ実行される。

CPUへのSPC(SCSIプロトコルコントローラ)からの割込みの受け方は(GCが生じないとして)次のようになっている。ゴール・リダクション時及びスリットチェック時、CPUはSPCからの割込み禁止モードで動作している。イニシエータとしてSCSI組込述語を実行した場合、まずターゲットとコネクションを張った後、ターゲットより一旦ディスコネクトされる。その後ターゲットよりリコネクトされるが、それはCPUがスリットチェック時にSPCの割込みフラグを検査して検出され、リコネクト後の処理に入る。ターゲットとしてセレクトされた場合、やはりスリットチェック時のSPC割込みフラグ検査で検出され、割込み要因の同定を行った後、ターゲット用の“SCSIからの割込みを表す変数”({Unit, Mesg, Next}というベクタ)とのユニフィケーションを行う。“SCSIからの割込みを表す変数”が具体化されたことにより、その変数にフックしていたゴールをリジュームし、適当な処理を経てターゲットからデータ転送を起動する。

13.3.1 各SCSI組込述語の仕様

SCSI組込述語も通常の組込述語と同様に引数チェックを行なう。ただし、処理過程で用いるレコード形式等は各ハードウェアモジュールの都合でそれぞれの処理系毎に定義されるべきものなので、引数の範囲チェックで使用する定数は実機処理系毎に変わり得る。SCSI組込述語によるSCSIインターフェイスは、直接実時間で操作するものではなく、SCSIバスに対する入出力の予約をするというレベルのインターフェイスであるため、予約時点で決定される出力引数もあれば実際の転送が終了した時点で決定される出力引数もある。

以下ではVPIMで実装したSCSI組込述語

```
scsi_command(SCSI,ArgVect,^NewData,^TransferredLength,^ID,^Result_I,^NewSCSI)
                ArgVect :: {Unit,LUN_I,Command,Length,Direction,Data,DataPos}
scsi_transfer(SCSI,ArgVect,^NewData,^TransferredLength,^Result_T,^NewSCSI)
                ArgVect :: {Unit,LUN_T,Kind,Length,Continue,Data,DataPos}
scsi_abort(SCSI,ID,^NewSCSI)
scsi_reset(SCSI,Unit,LUN_T,^NewSCSI)
scsi_init(^SCSI,^ReportInterruptionVect,^Info)
```

における入力引数の例外チェック仕様、及び出力引数のユニフィケーションについて説明する。

(1) 入力引数の例外チェック

- Command : コマンドディスクリプタブロック

  Commandが未定義ならば中断。バイトストリング以外ならばIllegal Inputの例外。現在VPIMデバイスドライバでは {6 | 10 | 12} バイトコマンドをサポートしている。従って、ストリング長はコントロールバイトを除く {5 | 9 | 11} のいずれかで、それ以外ならばRange Overflowの例外。

- Continue : 転送後にディスコネクトして続行に備えるかの否かの指定

  Continueが未定義ならば中断。整数以外ならばIllegal Inputの例外。

  0または1以外の整数値ならばRange Overflowの例外。

- Data : 転送データバッファ

  Dataが未定義ならば中断。バイトストリング以外ならばIllegal Inputの例外。

  ストリング長がLengthより小さいならばRange Overflowの例外。

- DataPos : 転送開始要素番号

  DataPosが未定義ならば中断。整数以外ならばIllegal Inputの例外。

  0~[Dataの指すストリングの長さ]以外の整数値ならばRange Overflowの例外。

  データ転送がある場合、Dataの指すストリングの長さからDataPosを差し引いた値がLengthよりも小さいならばRange Overflowの例外。

- Direction : データ転送方向指定

  Directionが未定義ならば中断。整数以外ならばIllegal Inputの例外。

  0~2以外の整数値ならばRange Overflowの例外。

  双方向の転送を伴うコマンド(Direction : 3)のサポートはしない。

- ID : 予約受け付け番号

  IDが未定義ならば中断。整数以外ならばIllegal Inputの例外。

- Kind : 転送種類指定

  Kindが未定義ならば中断。整数以外ならばIllegal Inputの例外。

  {0 | 2 | 4 | 6 | 7} 以外の整数値ならばRange Overflowの例外。

- Length : データ転送長

  Lengthが未定義ならば中断。整数以外ならばIllegal Inputの例外。

  scsi_commandでは0~16K(VPIMで定めたコマンドレコードのバッファサイズ)以外の整数値ならばRange Overflowの例外。scsi_transferでは0~4K(VPIMで定めた転送レコードのバッファサイズ)以外の整数値ならばRange Overflowの例外。また、Kindが6(status)の場合1以外の整数値ならばRange Overflowの例外。

- LUN_I : 操作対象論理ユニット番号

  LUN_Iが未定義なら中断。整数以外ならばIllegal Inputの例外。0~2047以外の整数値の場合はRange Overflowの例外。

- LUN_T : 操作対象ターゲット側論理ユニット番号

  LUN_Tが未定義ならば中断。整数またはアトム□以外ならばIllegal Inputの例外。

  0~2047以外の整数値の場合はRange Overflowの例外。

- SCSI : 操作予約インターフェイス

  SCSIが未定義ならば中断。整数以外ならばIllegal Inputの例外。

- Unit : 操作対象ユニット番号

  Unitが未定義ならば中断。整数以外ならばIllegal Inputの例外。

  0~7以外の整数値の場合はRange Overflowの例外。

(2) 出力引数のユニフィケーション

- ^ID : 予約受け付け番号

  コマンド予約受け付け時点で予約受け付け番号(整数値)をユニファイする。

- ^Info : 制約情報

  実機処理系に依存する制約情報を4要素ベクタで返す。 ベクタの構成は次の通り。

  {

  コマンドレコードのデータバッファサイズ,

  転送レコードのデータバッファサイズ,

  拡張メッセージが使えるか否かのフラグ(0 or 1),

  未使用

  }

- ^NewData : 転送後のデータバッファ

  コマンド終了時に転送後のデータバッファ(バイトストリング)をユニファイする。転送の途中でエラー終了した場合であっても、その時点までに転送されたデータは反映されている。

- ^NewSCSI : 操作予約完了後の操作予約インターフェイス

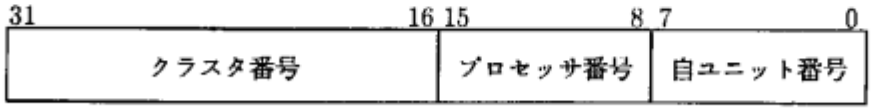
  操作予約受け付け時点で次の操作予約のために入力引数SCSIとユニファイする。

- ^ReportInterruptionVect : SCSIからの割込みを表す変数

  ターゲットとしてセレクトされた場合に、そのSCSIからの割込みをKL1プログラムに知らせるための変数—{Unit,Msg,Next} の3要素ベクタを割り当てる。

- ^SCSI : 操作予約インターフェイス

  VPIMでは、クラスタ番号,プロセッサ番号,自ユニット番号をエンコードした形式の整数値



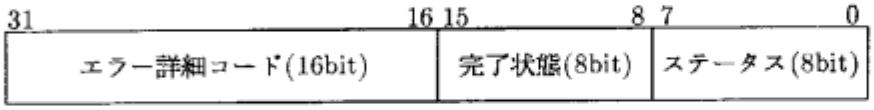
  をユニファイする。

- ^TransferredLength : 実転送バイト数

コマンド(転送)終了時に実際に転送が行なわれたデータのバイト数をユニファイする。エラー終了した場合であっても、その時点までに正常に転送されたデータの正確なバイト数を示す。

- ^Result_I : コマンド実行結果

コマンド終了時に以下のフィールドを持つ整数値をユニファイする。



各フィールドの意味は次の通り。

① エラー詳細コード

Bit 25~16の各ビットに実行開始後に検出されたエラーが割り付けられている。エラーは同時に検出され得るものや、検出後のエラーハンドリング中に検出され得るものがあるため、複数ビットが立つ可能性がある。現在Bit 31~26は使用しておらず、常に0である。

Bit 31~26: (Reserved Bits)

Bit 25 : Unexpected Phase
VPIMデバイスドライバの想定する状態遷移から外れるフェーズ要求を受けた。

Bit 24 : Unexpected Message
ターゲットから受領したメッセージが、サポートしていないものであった。
或は、シークンス上不適当なメッセージであると判断した。

Bit 23 : Bus Released
コマンド実行中にディスコネクト以外にバスフリーフェーズを検出した。

Bit 22 : Transfer Interrupted
転送動作中にディスコネクト以外にターゲットがフェーズを変更した。

Bit 21 : Parity Error In Output Data
アウトプット動作中にSCSIに送出しようとしたデータにパリティエラーを検出した。

Bit 20 : Parity Error In Input Data
インプット動作中に受領データにパリティエラーを検出した。

Bit 19 : Hardware Error
SPC内部レジスタへの書き込み時のパリティエラー、またはREQ,ACK信号の波形割れを検出した。

Bit 18 : Error In Selection
セレクションフェーズ実行時にハードウェアエラーを検出した。

Bit 17 : Time Out
セレクションタイムアウトを検出した。

Bit 16 : Reset Condition Detected

リセットコンディションを検出した。

② 完了状態

コマンドの完了状態を表す。各ビットの全ての組合せがある訳ではない。
現在Bit 15~13は使用しておらず、常に0である。

Bit 15~13: (Reserved Bits)
Bit 12 : Error End
コマンド実行中に異常を検出した。
Bit 11 : Status Not Received
ステータスバイトを受け取っていない。
Bit 10 : Command Complete Not Received
"Command Complete"メッセージを受け取っていない。
Bit 9 : Canceled After Execution
コマンド完了前にキャンセルされた。
Bit 8 : Canceled Before Execution
コマンド実行前にキャンセルされた。

③ ステータス

コマンドのステータスバイト。
ステータスバイトが送られてこなかった場合は0である。

- ^Result_T : 完了状況

転送終了時に完了状況を以下の形式で返す。

① 正常な完了の場合

整数値0をユニファイする。

② アテンションによる完了の場合

受領したメッセージ(バイトストリング)をユニファイする。

③ キャンセルされた場合

アトム□をユニファイする。

④ 異常終了した場合

エラー詳細コード(整数値)をユニファイする。Bit 6~0の各ビットに検出されたエラーが割り付けられている。エラーは検出後のエラーハンドリング中に検出され得るものがあるため、複数ビットが立つ可能性がある。現在Bit 7は使用しておらず、常に0である。

Bit 7 : (Reserved Bit)

Bit 6 : Invalid CDB Length
コマンドフェーズでイニシエータから受領したCDBの第1バイト目の
グループコードが、サポートされていないものであった。

Bit 5 : Parity Error In Output Data
アウトプット動作中にSCSIに送出しようとしたデータにパリティエラーを
検出した。

Bit 4 : Parity Error In Input Data
インプット動作中に受領データにパリティエラーを検出した。

Bit 3 : Hardware Error
転送途中でハードウェアエラーを検出した。

Bit 2 : Error In Reselection
リセレクションフェーズ実行時にハードウェアエラーを検出した。

Bit 1 : Time Out
リセレクションタイムアウトを検出した。

Bit 0 : Reset Condition Detected
リセットコンディションを検出した。

13.3.2 各SCSI組込述語の機能

scsi_init/3 を除き、どの組込述語もSCSI変数を用いて実行の制御を行う。次のようなプログラムを考える。

```
p(SCSIa) :- true | .., scsi_a(SCSIa, .., SCSIb), ... ,
                   scsi_b(SCSIb, .., SCSIc), ... .
```

SCSI型変数 SCSIaへの値の束縛によって scsi_aが起動される。scsi_aの実行が終了するとSCSIbに値が束縛されて、scsi_bの実行が開始される。SCSI型変数は実行順序を逐次化するためのみに設けられているので、VPIMではその具体化される値として実際には整数を用いる。ただし、その整数値には、クラスタ番号、PE番号、自身のdevice unit numberがエンコードされている。

PIMがイニシエータとして一つのSCSIのセッションを行う時、scsi_commandを1回実行する。ここでセッションと言うのは、相手ターゲットとの間で一つのコマンドが完了するまでのことを指す。従って一つのセッションの中では、多くのフェーズの推移が生じており、セレクション・フェーズや情報伝送フェーズだけでなく、バスフリーフェーズもそれに含まれる。

PIMがターゲットとして、scsi_transferを実行すると、バスフリーフェーズを挟まない1回分のデータ転送が起動される。従って、イニシエータからの1回分のコマンド要求を満たすために、scsi_transferを複数回実行して、データを転送することも生じる。これはKL1プログラムによって制御される。

scsi_command、scsi_transferはいずれもSCSIバスに対する入出力の予約をするというレベルのインターフェイスであるため、KL1-Bの実行シーケンスにおける実行終了とは、予約が完了したことを意味する。

同じユニット、同じLUN (Logical Unit Number)に対するコマンド、及び転送の実行順序は保証されるが、いつ実行するかは保証されない。つまりコマンド、転送がバッファリングされる可能性がある。scsi_abort は、以前に投入し、バッファリングされているコマンドを取り消す組込述語である。入力引数 ID は、scsi_commandを実行して得られた ID の値を用いる。ただし、scsi_abort の実行自体も任意の時間遅れが許されている。scsi_abort がスケジューリングされた時にコマンドは、予約、実行中、実行終了のいずれかの状態である。既に実行が終了していた場合は何もせずに終了する。

ターゲットとして scsi_transfer を実行中に、転送エラーやアテンション等が生じ正常終了しなかった時、同じユニット, LUNに対する転送予約はキャンセルされ続ける。
scsi_resetによってこのキャンセル状態を解除することができる。尚、転送予約がキャンセルによって終了したことは、出力の完了状態を見れば分かる。

PIMOSが立ち上がりSCSIデバイスドライバ・プロセスを生成する時、イニシエータ・ターゲット用順序制御のための変数やSCSI関係のシステムパラメータが必要となる。
scsi_initは、そのような変数やパラメータをKL1処理系からPIMOS(KL1プログラム)の世界へ受け渡す。従って、通常はPIMOS立ち上げ時に1回だけ呼ばれる。

13.3.3 キュー管理による実行制御

処理系内におけるコマンドや転送の予約は、それぞれに対応するレコード構造で表現し、キュー管理を行なう。以下にSCSIで使用するデータ構造とキュー管理について述べる。

(1) SCSIで使用するデータ構造

以下に述べるコマンドレコード、転送レコード、セレクションレコードの形式は、PIM実機処理系の抽象仕様であるVPIMとして定めたものである。レコード数、レコード内要素の形式(データバッファ等)は各PIMの特性に合わせて最適化することが望ましい。

① コマンドレコードコマンドレコードはコマンド予約に用いるデータ構造で、SCSIバスに接続された各PE毎にフリーリスト管理し、GC中であってもターゲットからの割込み(Reselect)に対応できるようシステム固定領域に置く。一度に予約できるコマンド数の上限はVPIMでは16個とする。コマンドレコードであることを示すフラグはポインタ側のタイプ(ATOM)を使用する。また、予約受け付け番号IDはレコード中の要素としては持たず、コマンドレコードを指すポインタの値をIDとして使用する。コマンドレコードの構造を図13-4に示す。

各要素について以下に説明する。

- 次のレコードへのポインタ、前のレコードへのポインタ
  フリーリスト管理、キュー管理に使用する。

- 里親へのポインタ、ゴールの実行環境

  ゴールの実行環境は環境レコードへのポインタを保持する。出力引数のunifyゴール生成に使用する。

- キャンセルフラグ

  コマンドがキャンセルされたかどうかのフラグ。初期値は偽(0)。一旦ディスコネクトされ、再びリコネクトされた時は必ずこのフラグをチェックする必要がある。

- 操作対象ユニット番号、操作対象論理ユニット番号、転送データ長、転送方向、転送データストリング内の転送開始要素番号

  入力引数で渡される簡単な整数データ。予約時点で各要素にコピーする。

- コマンド長

  現在VPIMデバイスドライバでは、{6 | 10 | 12}バイトコマンドをサポートしている。従って、コマンド長として設定され得る値は、{6 | 10 | 12}のいずれかである。入力引数で渡されるコマンド(CDB：バイトストリング)は、コントロールバイト(1バイト)を除いたものなので、ストリング長に1を加算した値を設定する。

- 転送データカウンタ

  実際に入出力されたデータのバイト数。

- KL1出力変数 : NewData

  結果の返し先。コマンド完了時に入力引数Dataに渡された転送データへのポインタをユニファイする。

- KL1出力変数 : TransferredLength

  結果の返し先。コマンド完了時に転送データカウンタの値をユニファイする。

- KL1出力変数 : Result

  結果の返し先。コマンド完了時にステータスバイトバッファ、エラー詳細コード、完了状態の値を32Bitにエンコードした値をユニファイする。

- 転送データバッファ中の要素位置

  save data pointerメッセージを受けた時に更新される。

- 入力引数Dataに渡された転送データへのポインタ

  転送方向がDATA IN、引数DATAのMRBが黒い場合はDATAのコピーを作ってそれを使用する。

- ステータスバイトバッファ

  ターゲットから送られたステータスバイトを格納する。初期値は0。

- エラー詳細コード

  コマンド実行開始後に検出されたエラー情報を保持する。初期値は0。

- 完了状態

  コマンドの完了状態を保持する。初期値は2進"00011000"。("Command Complete"メッセージを受け取っていないことを示すBit 4とステータスを受け取っていないことを示すBit 3を"1"に

する)

- 次に期待するフェーズ

  ディスコネクトされた時に、次にリコネクトされた時に期待するフェーズ(定数値)を保持する。リコネクト時にディスコネクト前の状態から制御を開始するために、ターゲットが要求するフェーズをチェックするために用いる。

- コマンド本体

  現在VPIMデバイスドライバでは12バイトコマンドまでをサポートしているが、コマンドとして許す最大長として16バイト領域を確保しておく。VPIMではバイトアクセス命令をサポートしていないため、ストリング(ストリングデータ本体)の形式で格納する。

- 転送データバッファ

  VPIMではバッファサイズは4Kワード(= 16KByte)とする。データは、ストリング(ストリングデータ本体)の形式で格納する。

② 転送レコード転送レコードは転送予約に用いるデータ構造で、SCSIバスの接続された各PE毎にフリーリスト管理し、GC中にアクセスされることはないがシステム固定領域に置く。一度に予約できる転送数の上限はVPIMでは64個とする。転送レコードであることを示すフラグはポインタ側のタイプ(INT)を使用する。転送レコードの構造を図13-5に示す。

各要素について以下に説明する。

- 転送の種類

  どのような転送を行なうか{0 | 2 | 4 | 6 | 7}のいずれかを指定する。

  (0 : data out , 2 : data in , 4 : command , 6 : status , 7 : message in)

- 転送後のディスコネクトの指定

  転送後にディスコネクトして続行に備えるか否かの指定。入力引数Continueによって渡される0か1の整数値。

- KL1出力変数 : Result

  結果の返し先。転送完了時に設定値がINT : 0、またはATOM : 0(□)の場合はその設定値をユニファイする。設定値のタイプがSTRGであった場合イニシエータから受け取ったメッセージ(メッセージバッファに格納されている)を、ヒープ中に割り付けたバイトストリングにコピーし、ユニファイする。

- 入力引数Dataに渡された転送データへのポインタ

  転送方向がDATA OUT、引数DATAのMRBが黒い場合はDATAのコピーを作ってそれを使用する。

- 完了状況

  転送完了時の完了状況を保持する。初期値は0。

- メッセージ長

  受け取ったメッセージの長さ(バイト数)を格納する。

(word)	
0	次のレコードへのポインタ
1	前のレコードへのポインタ
2	里親へのポインタ*
3	ゴールの実行環境*
4	キャンセルフラグ
5	操作対象ユニット番号
6	操作対象論理ユニット番号
7	コマンド長
8	転送データ長
9	転送方向
10	転送データストリング内の転送開始要素番号
11	転送データカウンタ
12	KL1出力変数 : NewData*
13	KL1出力変数 : TransferredLength*
14	KL1出力変数 : Result*
15	転送データバッファ中の要素位置
16	入力引数Dataに渡された転送データへのポインタ*
17	ステータスバイトバッファ
18	エラー詳細コード
19	完了状態
20	次に期待するフェーズ
21 ⋮ 25	コマンド本体 ディスクリプタ(1word) + データ領域(4word = 16Byte)
26 ⋮ 4122	転送データバッファ ディスクリプタ(1word) + データ領域(4Kword = 16KByte)

*印の要素はGCのマーキングルートにしなければならない

図 13-4: コマンドレコードのデータ構造

- メッセージバッファ

  イニシエータから受け取ったメッセージをストリング(ストリングデータ本体)の形式で格納する。VPIMではメッセージバッファの大きさは16バイト(ディスクリプタを含む5ワード領域)である。

- 転送データバッファ

  VPIMではバッファサイズは1Kワード(= 4KByte)とする。データは、ストリング(ストリングデータ本体)の形式で格納する。

尚、省略した要素についてはコマンドレコードと同じ仕様である。

(word)	
0	次のレコードへのポインタ
1	前のレコードへのポインタ
2	里親へのポインタ*
3	ゴールの実行環境*
4	操作対象イニシエータ側ユニット番号
5	操作対象ターゲット側論理ユニット番号
6	転送の種類
7	転送データ長
8	転送後のディスコネクトの指定
9	転送データカウンタ
10	転送データストリング内の転送開始要素番号
11	KL1出力変数 : NewData*
12	KL1出力変数 : TransferredLength*
13	KL1出力変数 : Result*
14	転送データバッファ中の要素位置
15	入力引数Dataに渡された転送データへのポインタ*
16	メッセージ長
17	完了状況
18 ⋮ 22	メッセージバッファ ディスクリプタ(1word) + データ領域(4word = 16Byte)
23 ⋮ 1047	転送データバッファ ディスクリプタ(1word) + データ領域(1Kword = 4KByte)

*印の要素はGCのマーキングルートにしなければならない

図 13–5: 転送レコードのデータ構造

③ セレクションレコードセレクションレコードはPIMがターゲットとしてセレクトされた時に用いるデータ構造で次の2通りの使用法がある。

- セレクトされた時にイニシエータから受け取るメッセージのバッファ。
- GC中にセレクトされた時、GC後にKL1プログラムにセレクションを通知するための記憶レコード。

セレクションレコードはSCSIバスの接続された各PE毎にフリーリスト管理し、システム固定領域に置く。GC中に対応できるセレクト数の上限はVPIMでは64個とする。セレクションレコードであることを示すフラグはポインタ側のタイプ(FLT)を使用する。セレクションレコードの構造を図13–6に示す。

各要素について以下に説明する。

- 次のレコードへのポインタ

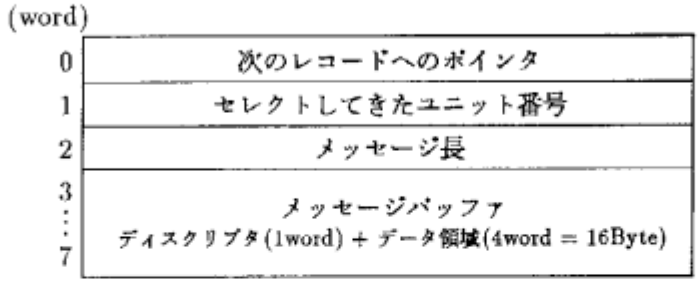


図 13–6: セレクションレコードのデータ構造

フリーリスト管理、キュー管理に使用する。

- セレクトしてきたユニット番号
  セレクトしてきたイニシエータのユニット番号
- メッセージ長
  受け取ったメッセージの長さを格納する。
- メッセージバッファ
  セレクト後にイニシエータから送られたメッセージをストリング(ストリングデータ本体)の形式で格納する。
  VPIM ではメッセージバッファの大きさは 16 バイト(ディスクリプタを含む 5 ワード領域)である。

(2) キュー管理方式

前述した各レコードは以下のキュー、テーブルによって管理する。

① 予約キュー

コマンド予約、転送予約を記録するためのキュー。ここには同じユニット、LUN に関するコマンドまたは転送が現在処理中(バス権キュー、実行中リストに存在する)であるためまだ実行には入れないようなコマンド、転送に関するレコードが入る。

② バス権キュー

予約済みのコマンドおよび転送のうち、バスさえとれれば実行開始可能であるが、バスの獲得を待っているもののキュー。

③ 実行中リスト

現在実行中であるがディスコネクト状態であるコマンドを登録するテーブル。

④ 後処理キュー

割込み処理が GC 中のため即座に実行できない時に、処理を GC 後に延期するために記録しておくためのキュー。以下の 2 通りの事象がある。

- ターゲットにリセレクトされ実行が終了した場合

GC中にリセレクト処理で実行が終了したコマンドについて、終了後の処理(KL1出力変数の具体化)が即座に実行できないコマンドレコードを後処理キューに保持する。

- イニシエータにセレクトされた場合

  PIMをターゲットとして動作させた場合イニシエータからセレクトされるという事象は非同期にGC中にも起こり得る。その場合、バスを早期に解放するためにGC中でも即座に対応し、GC終了後にKL1プログラムにこのセレクションを通知できるようセレクションレコードとして割込みを記憶し、後処理キューに保持する。

⑤ 異常コネクションリスト

PIMがターゲットとして動作している時に転送がアテンションや他の異常で正常終了しなかった場合にその時のユニット、LUNを記録しておくテーブル。記録レコードにはヒープ中の4ワードセルを使用する。登録したレコードは同じユニット、LUNを指定したscsi_resetによるリセットが行なわれるまで保持する。GCのマーキングルートにしなければならない。

前述のキュー、及びテーブルを用いたVPIMデバイスドライバの実行制御フローをscsi_commandによるコマンド予約を例に図13-7, 13-8, 13-9 に示す。

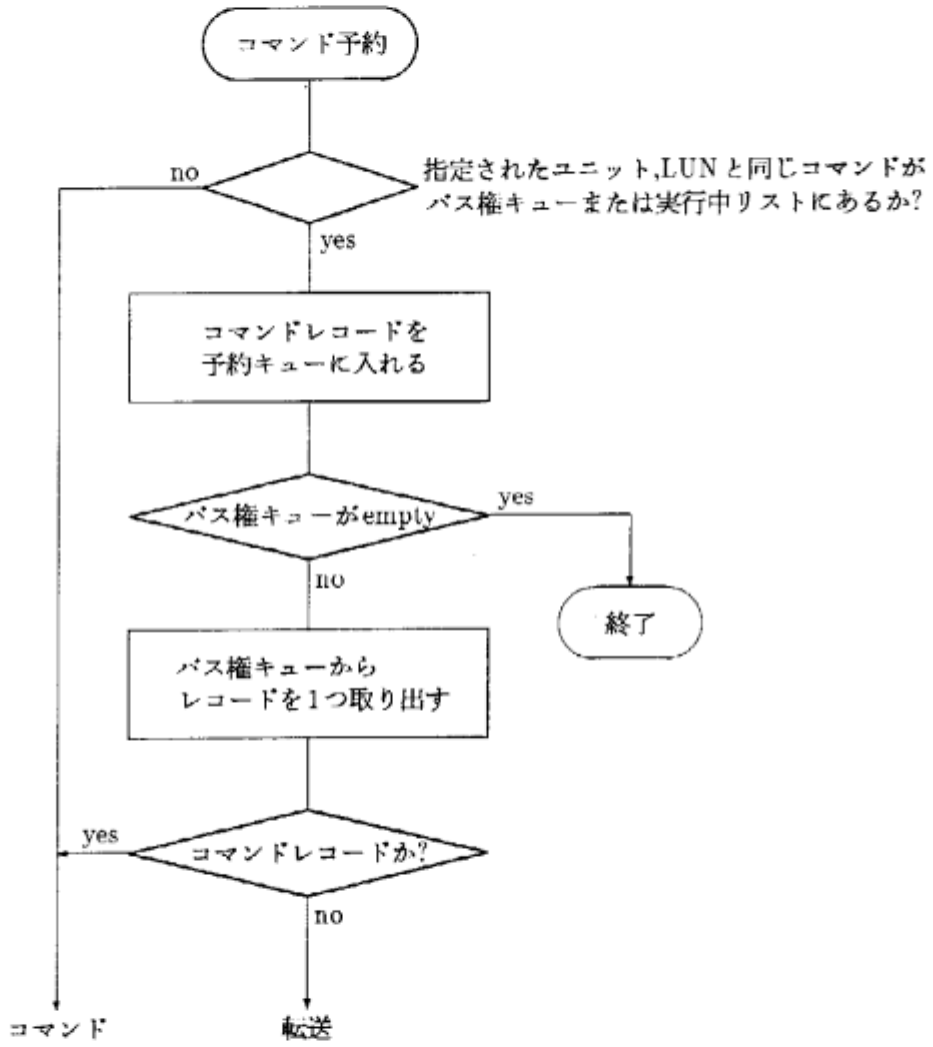


図 13–7: コマンド予約のフローチャート (1)

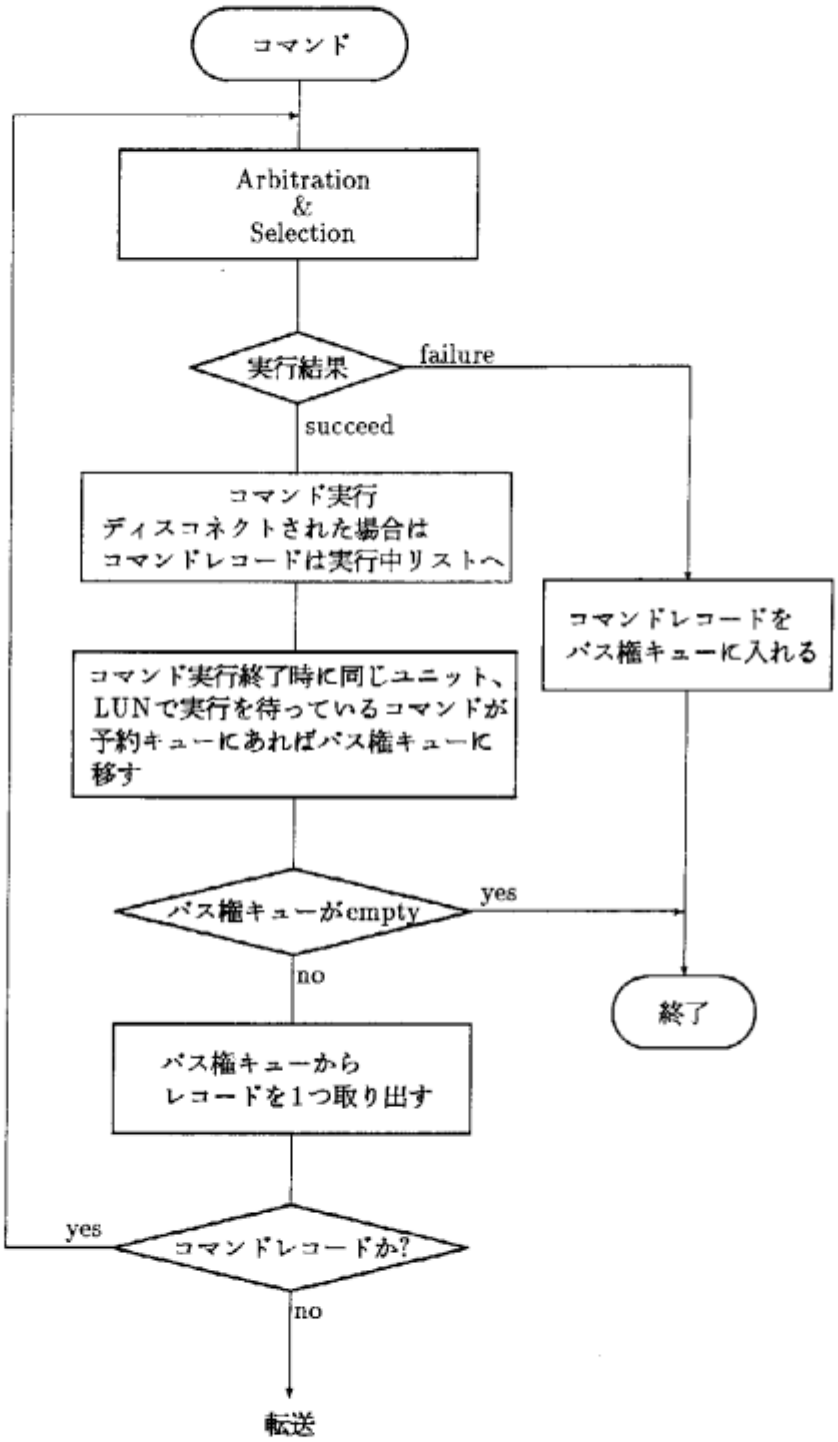


図 13-8: コマンド実行のフローチャート (2)

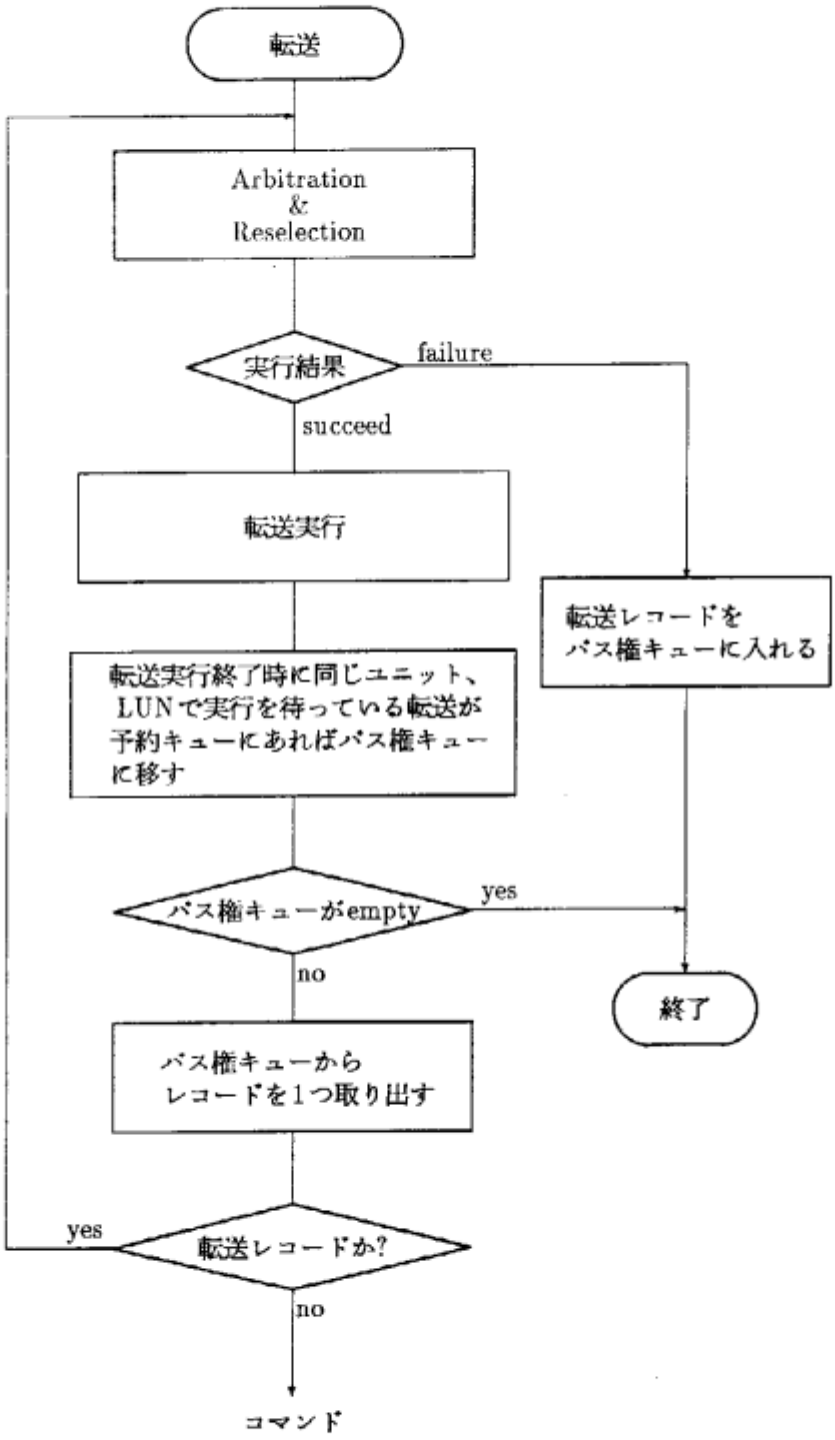


図 13–9: 転送実行のフローチャート (3)

付録 A

SCSI のはなし

執筆担当者: 武井

A.1 SCSIって何？

SCSI(すかじぃと発音される)はSmall Computer System Interface の略で、その名の通り本来的には、小型コンピュータ・システムの汎用入出力インタフェースとして考えられたもの。SCSIは、基本的にはデータのやりとりをするためのハードウエア(ケーブル)と、その際の約束事(プロトコル)で成り立っている。

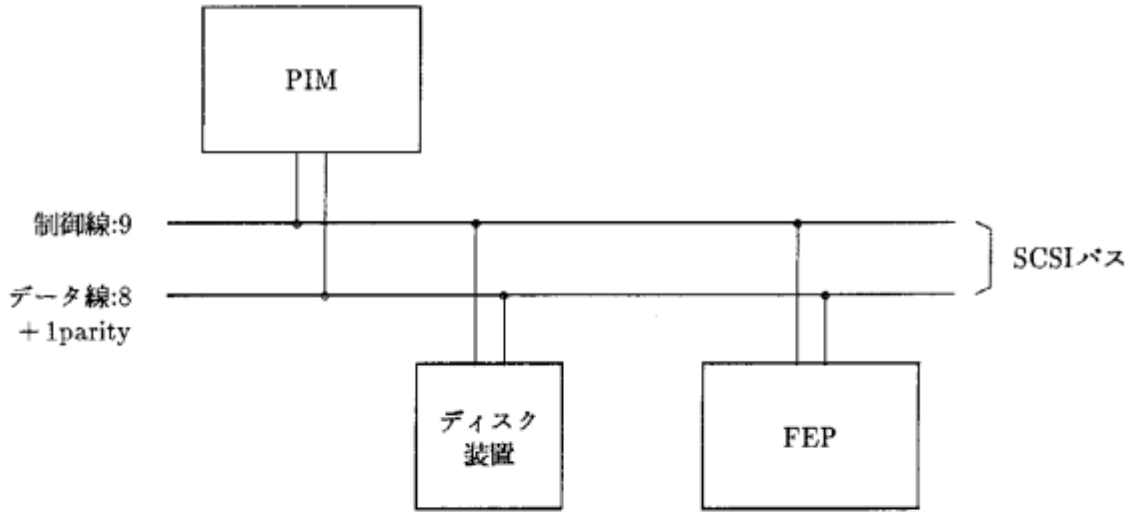


図 A-1: SCSI システム例の簡単な図

- ハードウエア

  ケーブルは18本の信号線からなっている。内訳は、制御線9本、データ線9本(1バイト＋パリティ・ビット)である。これがSCSIバスを構成している。制御線は、9種類の信号の組合せによってSCSI上で何を行うかの設定をする。設定方法はプロトコルで規定されている。データ線は、実際の情報転送が行われるもので、現在の規格では1バイト(8ビット)幅になっている。

- プロトコル

  SCSIプロトコルとしては、**バス・フェーズ**、**バス・コンディション**、**メッセージ・システム**がある。これらに基づいて、SCSI上で行われるすべての動作を制御する。

A.2 SCSIシステムの構成

A.2.1 SCSI装置

SCSIバスには最大8台の装置(ホスト・コンピュータ、ディスクドライブ等)を接続できる。それぞれのSCSI装置は、SCSIバス上での認識番号として0 ～ 7までのSCSI IDを持つ。これはデータバスの8本の信号線のうちの1本への対応を表している。例えば、ID=3の装置を表すにはデータバスのビット番号"3"に相当するビットを真に駆動する。これによって、自分のIDを示したり、通信の相手を表したりする。これは通常システム立ち上げ時に、それぞれの装置毎に設定される。

SCSIバスを用いた操作のやり方は、何らかの操作を他の装置に対して行いたい(例えばディスクからデータを読み込みたい)装置が、その相手との間の経路を設定し、相手に対して操作の実行を指示する。最初に操作をしたいと思った側を**イニシエータ**と呼ぶ。イニシエータから指示された操作を実行する側を**ターゲット**という。ある時点において、SCSIバスを使用できるのは1台のイニシエータと、そのイニシエータから選択されたターゲットの都合2台の装置に限られる。イニシエータ/ターゲットのひと組がSCSIバス上で操作を行っている間、他の装置はバスを使用できない。

A.2.2 イニシエータ・ターゲット

ある装置Aが、SCSIバスにつながっている別の装置Bから、格納されているデータを読み込みたい、とする。この場合、装置Aを**イニシエータ** (Initiator)、装置Bを**ターゲット** (Target)という。SCSI装置はバス上でイニシエータかターゲットの機能を受け持つ。イニシエータとなる装置は、まず相手のターゲットとなる装置に対して通信を行うことを打診する。ターゲットから応答が得られると、「物理的な」接続経路の設定(**コネクト**)が終了する。

> イニシエータ/ターゲットという呼称は、ひとつのセッションにおける役割に対するもので、必ずしも個々のSCSI装置に固定されたものではない。SCSI装置がイニシエータ/ターゲットの両方の機能を備えていれば、SCSI上で双方の役割を果たすことができる。PIMは両方の機能を持つ。

イニシエータはコネクト後、ターゲットを通じて実際の通信の相手となる**論理ユニット**を指定する。**論理ユニット**(Logical Unit)は、ターゲットが管理する仮想的・物理的な子装置である。それぞれの論理ユニットには**論理ユニット番号** (Logical Unit Number, **LUN**)が付けられる。イニシエータ側からは、論理ユニット番号によって実際の操作対象となる「装置」を指定する。つまり、**SCSI ID, LUNの両方を指定**して通信を開始する。

SCSIの基本仕様では、イニシエータは入出力の対象として、ターゲット配下の論理ユニットを、各ターゲットあたり8台まで指定できることになっている。但し、これは各装置の条件に依存する。例えば

LUNは"0"固定という装置もある。

A.3 SCSIプロトコル

前述したように、SCSIバス上での動作を制御するための規約として、バス・フェーズ、バス・コンディション、及びメッセージ・システムが規定される。

SCSIバス上の動作は**フェーズ**の連なりから構成される。フェーズとはバスの使用状態のことで、「今、バスはどういう状態で何をしているか」を表している。

SCSIバス上のシーケンスは、バス・フェーズによって制御されるが、ある特定の信号を送出することによって、非同期的な動作を要求することができる。この非同期的に生成される条件を**バス・コンディション**という。

イニシエータ・ターゲットの間での操作の実行にともなう一連のバス・シーケンスを、メッセージ通信によって制御することができる。**メッセージ・システム**は、各メッセージの「意味」と付随する動作について規定している。

A.3.1 バス・フェーズ

実際のSCSI上の動作は、以下に示すように複数のバス・フェーズの実行からなる。バス・フェーズは、SCSIプロトコルで定められた任意のバスの状態を指し、制御信号(の組合せ)によって各フェーズを設定する。

SCSI上でイニシエータ/ターゲットの接続経路が設定された後は、実行するSCSIバス・フェーズの指定は**すべてターゲット側から行われる**。従って、イニシエータは相手を選びはするが、後はターゲットの要求してくるフェーズに応じるのみである。たとえ、ターゲット側から期待していないフェーズが要求されても、イニシエータはそれに応答しなければならない。イニシエータ側からは、アテンション・コンディションを生成して、ターゲットに「いいたいことがある」ことを示唆することのみ可能である。(A.3.3 **バス・コンディション - アテンション・コンディション**の項を参照)

典型的なバス・フェーズの推移を例にとって、各フェーズの説明をする。

典型的なバス・フェーズの推移例

バスフリー → アービトレーション → セレクション ··· **コネクト**
→ メッセージアウト → コマンド → データ → ステータス → メッセージイン ··· **情報転送**

以下で、括弧内の"I"はイニシエータ、"T"はターゲットを示す。また矢印はデータなどの情報の流れる方向を示す。

1. **バスフリー・フェーズ**
   SCSIバスが使用されていない状態。
2. **アービトレーション・フェーズ**
   バスを使用できるのはひと組のイニシエータ/ターゲットのみである。このため、バスを使用する

にあたって他のSCSI装置との間でバス使用についての調停(Arbitration)を行う。バスを使用したいSCSI装置はアービトレーション・フェーズを実行し、他にバスの使用を希望する装置もこれに参加することになる。アービトレーションに参加したSCSI装置のうち、SCSI IDの値が最大のものがバス使用権を獲得する。これによって、ひとつのSCSI装置がSCSIバスの使用権を獲得し、イニシエータまたはターゲットとして機能する。

3. **セレクション・フェーズ**
   バスの使用権を獲得した後、相手の装置を選ぶことになる。相手からの応答があった場合にのみ次に行くことができる。(通常は、時間制限を設けて相手からの応答を待つことになる。)
4. **メッセージアウト・フェーズ** (I → T)
   ここでイニシエータ側から、接続する論理ユニットの指定を"Identify"メッセージによって行う。このメッセージには、イニシエータがディスコネクト/リコネクト機能(後述)を備えているか否かの情報も含まれる。ここまでで、実行のための経路が確定する。
5. **コマンド・フェーズ** (I → T)
   イニシエータ側から実行する操作内容について指示を出す。具体的には6 ～12 バイト長のデータがコマンドとしてターゲットに渡される。このコマンドについても規定がある。
6. **データイン / アウト・フェーズ** (I ↔ T)
   SCSIコマンド実行にともなうデータの送受を行う。
7. **ステータス・フェーズ** (I ← T)
   SCSIコマンド終了状態を、ターゲットが1バイトのデータ(**ステータス・バイト**)によって報告する。
8. **メッセージイン・フェーズ** (I ← T)
   コマンド実行終了のメッセージ("Command Complete")がターゲットから送出される。これによって、ひとつのコマンドの実行が終了し、バスフリー・フェーズに移行する。

   番外：**リセレクション・フェーズ**
   いったんディスコネクトした後に、ターゲット側からイニシエータにリコネクトする。コネクトと同様の手順で、

   アービトレーション → リセレクション → メッセージ・イン

   を行う。

A.3.2 ディスコネクト・リコネクト

例えば、ターゲットが処理に時間のかかるコマンドを受けとった場合について考える。イニシエータはターゲットが結果を返すのをじっと待つ。SCSIバスはその間実質的な転送がない(遊んでいる)にも関わらず、Busy状態のままである。このような場合にバスを解放できれば、他の装置がバスを使用して仕事をすることができる。

これを実現するのが、**ディスコネクト・リコネクト機能**である。ターゲットがバスを一旦解放し(ディ

スコネクト)、然るべき後に今度はターゲット側からイニシエータを指定して(リコネクト)コマンド実行の続きを行う(次に示す推移例を参照)。こうすることで、バスが効率的に使用できることになる。

但し、イニシエータ・ターゲットの双方でこの機能をサポートしていなければならない。前述の通り、この機能をサポートしているか否かについての情報は"Identify"メッセージ中に含まれる。

ディスコネクト・リコネクトを含むフェーズ推移例

```
… → コマンド → メッセージイン："Disconnect" → バスフリー   … ディスコネクト
                    …
バスフリー → アービトレーション → リセレクション              … リコネクト
  → データ → ステータス → メッセージイン                     … 情報転送
```

A.3.3 バス・コンディション

以下のふたつの非同期操作が用意されている。

- **アテンション・コンディション**
  イニシエータ側から「何かいいたいことがある」場合には、アテンション・コンディションというバス・コンディションを生成する。ターゲットはアテンション・コンディションを検出した場合、任意の時点でメッセージアウト・フェーズを実行して、イニシエータの「いいたいこと」に耳を傾ける。
- **リセット・コンディション**
  これは、SCSIシステム全体の強制的なリセットを意味し、他のすべてのフェーズやバス・コンディションよりも優先度が高い。リセット・コンディションが生成された場合、すべてのSCSI装置はSCSIバスへの信号送出をやめ、バスフリー・フェーズへ移行する。リセット・コンディションによってSCSI装置がどの程度クリアされるかは、装置に依存する。SCSIシステムの初期化、及びどうにもならない位ボロボロになった時に使用されるはず。

A.3.4 メッセージ・システム

コマンド実行中に、何かお互いに言いたい場合の「語彙」がメッセージとして規格化されている。すでに述べた、論理ユニットの通知の他に、ディスコネクト・リコネクト機能もメッセージシステムによって実現されている。

メッセージは、形式的には1バイト長メッセージと拡張メッセージの2種類がある。どちらの場合も、SCSIバス上ではデータとして送受される。メッセージは、メッセージイン・フェーズ、メッセージアウト・フェーズにおいて転送する。

以下にいくつか1バイト長メッセージの例をあげておく。なお、以下で括弧内のIはイニシエータ、Tはターゲット、矢印はメッセージの送られる方向を示す。

- "Command Complete"メッセージ(I ← T)

コマンド実行終了を示すメッセージ。ちなみに、このメッセージ以外はすべてオプションとなる。

- "Identify"メッセージ(I ↔ T)
  すでに述べたように、論理ユニットの指定を通知する。論理ユニット番号は1バイト中の3ビットを用いて指定する。リコネクトの際には、ターゲットから自分の素性を明らかにするために送られる。
- "Disconnect"メッセージ(I ← T)
  ターゲットからのイニシエータに対するディスコネクト通知。メッセージ送出後、ターゲットはバスを解放する。
- "Abort"メッセージ(I → T)
  イニシエータが異常を検出した場合などに、ターゲットに対してコマンドの実行をやめることを通知する。

A.4 異常検出について

A.4.1 コマンドの異常終了

コマンドの実行結果は、ターゲットからのステータス・バイトによって知らされる。コマンドが正常終了しなかった場合には、通常ターゲットは「私の状態をお調べなさい」というステータスを送り、「異常」についてのデータを作成・保存する。イニシエータは、あらためてターゲットにコネクトをかけて「異常についてのデータを下さい」というコマンドを送り、ターゲットは保存していたデータを送る。これによって、コマンド終了状態の詳細を知ることができる。

A.4.2 シーケンス推移上の"異常"

前述の通りイニシエータは、自分からフェーズを設定することはできない。従って、ターゲットがバスシーケンス上"異常"なフェーズ要求をしてくることも覚悟しなければならない。そこで、「これはめちゃくちゃ異常だ」と思った場合には、アテンション・コンディションを生成し、メッセージアウト・フェーズを待って"Abort"メッセージを送る。これができないようでは仕方がない。

A.4.3 ハード的なエラー

- パリティエラー
  転送にともなうパリティエラーについては、リトライを行うことができる。
- 信号の波形割れなど
  「コネクタはしっかり差し込みましょう。」?

索引