記 号 処 理 65－1
（1992. 6. 12）

並列 lisp による X インタフェース

近藤奈々　中西正和
慶應義塾大学

従来のウインドウ環境におけるインタフェースにおいて、1つの処理を始めるとそれが終わるまで、他のイベントを受け付けないので、ユーザを待たせる原因となっている。これはインタフェースを記述するプログラムが並列性を持っていないためである。本稿では、X ウインドウシステムインタフェースを記述する言語に並列性を導入し、X サーバと通信機能をもつ並列 lisp 処理系 Momolisp について報告する。Momolisp は Unix 上の1つのプロセスの中で動作し、その中で複数のスレッドを実行させる。

Implementation of X Interface with Parallel Lisp

Nana KONDO and Masakazu NAKANISHI
Department of Computer Science
Graduate School of Science and Technology
Keio University
3-14-1, Hiyoshi, Kouhoku, Yokohama 223, Japan

This paper describes a parallel X window system interface and a parallel Lisp. Momolisp is a parallel Lisp system implemented on Unix, which can communicate with X-server. Some threads can be created in Momolisp to be used in window environment.

1 はじめに

従来のウインドウ環境におけるインタフェースにおいては、ある処理を始めるとそれが終わるまで、他のイベントを受け付けないので、ユーザは待たされる。これはインタフェースを記述するプログラムが並列性を持っていないために起こる問題点である。

この「逐次性」を解決するために、インタフェース記述言語に並列性を導入することを提案する。並列 lisp によってXインタフェースを構築している処理系に PiXeL [8] や CLIDE [10] などがあるが、ここで提示する並列 lisp 処理系 Momolisp は、PiXeL に似た実現方法で Unix の X ウインドウシステム上に開発したものである。

並列性は、単一の Unix プロセス内で複数のスレッドを生成して、それぞれのスレッドに独立してS式を評価させることで実現している。以降、「プロセス」とは Unix におけるプログラムの実行単位を表し、「スレッド」とは1つのプロセス内で資源を共有しながら並列にプログラムを実行する単位の各々を表す。

Momolisp では、親をメインの lisp 処理系として走らせ、子には親にとってあまり重要でない軽い仕事をさせることにより、親の処理速度を向上させることを目標としている。従って、子としてプロセスを発生させることはプロセススイッチが重いため、かえって処理速度を低下させる。1つのプロセス内であれば、データ空間を共有するのでメモリ領域を変更する必要がなく、スレッド制御の切替は高速に行なわれる。

親の処理速度を高めるため、親の環境は shallow binding を用いている。親スレッドは、シンボル表を持っていて、定数、関数定義と同様、変数もここに保持されている。しかし、子スレッドにも shallow binding を適用する（mUtilisp [4] における環境の保持の方法）と、プログラムが複雑になるなどの欠点があるため、子スレッドに対しては deep binding を採用している。

子スレッドにおいて、定数、関数定義は親のものを共有する。しかし、変数に関しては、スレッド独自の環境を参照させる。

子スレッドの環境は、親の環境を継承させる方法と継承させない方法がある。

1．子スレッド生成時にそのときの親の環境を継承させる方法

子スレッドの環境は、そのスレッドが生成された瞬間の親の環境を継承する。しかし、binding の方法が異なるので、親が shallow binding では継承しにくい。

そこで、親スレッドで変数へのバインドが行われた場合、当然シンボル表へ書き込むのであるが、そのとき同時に、その変数と値を組にした環境リストを作成しておく（Fig.1.1)。

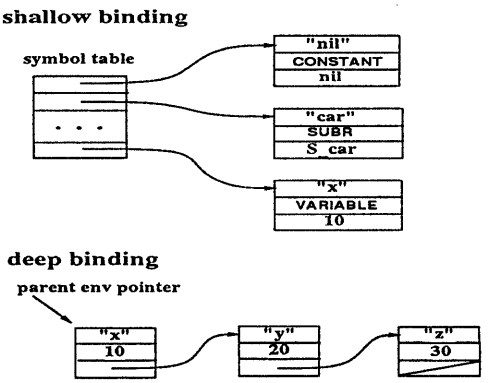

Fig.1.1 Parent Environment

親スレッドはその環境リストを参照しない（但し、明示的に参照させる関数は用意している）が、子スレッドを生成したとき、その環境リストの先頭のポインタを子スレッドへ渡してやると、子スレッドは実行の際、その環境リストを参照できる（Fig.1.2(a)）。それぞれのスレッドの環境は、S式の実行に従って成長するので、それぞれの異なる環境を形成する（Fig.1.2(b)）。さらに新しいスレッドが生成されたときは、そのときの親が指している環境リストの先頭を指す（Fig.1.2(c)）から、子スレッド同士も影響を及ぼすことはない。

変数への値のバインドが起こると、その変数名と値の組が環境リストにつなげられるのだが、すでに存在している変数名でも新しく作られてしまうので、同じ変数に何回ものバインドがなされた場合は、その数だけ環境リストが伸びていくことになる。
親の環境リストは子スレッドに受け継がれるものであるから、常にその時点の変数の束縛の様子を忠実に保持していたいので、既存の変数に対するバインドが行なわれた場合には、新しい組を作らずもとを書き換える方法が考えられる。

この環境の更新に関しても同様のことが考えられる。即ち、バインドが起こるたびにすべて新しく付け加える方法と、書き換えができるならば書き換えてしまう方法である。

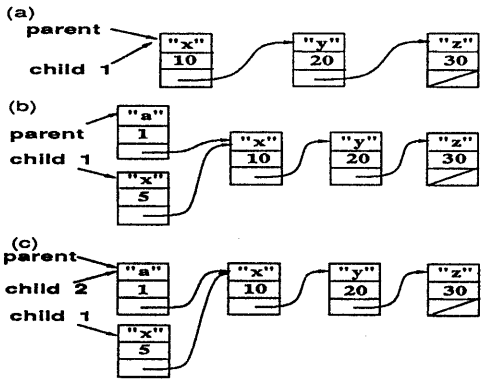

Fig.1.2 Parent and Child Environment

2．親の環境を継承しない方法

子スレッド生成時には、その子スレッド独自の環境リストは存在しないが、子スレッドで変数のバインドが行われたとき、その変数と値の組を環境リストにする。さらにバインドが行われたときは、子の環境リストを探して、その変数名がなければ、新しく組を作って環境リストに付加し、あれば値を書き換える。以下に例を示す。親でxに10がバインドされているとき、子スレッド1で(setq x 2)が評価されると子は独自の環境リストを作る（Fig.1.3(1)）。次に(setq y 1)が評価されると環境リストに加えられ(Fig.1.3(2))、さらに(setq x 5)が評価されると環境リストにおけるxを5に書き換える（Fig.1.3(3)）。子が変数を参照するときは、最初に自分の環境リストを参照し、なければ親のシンボル表を探す。この方法では、親スレッドでのsetqはすべての子に影響するが、子スレッドでのsetqは親や他の子スレッドに影響しない。

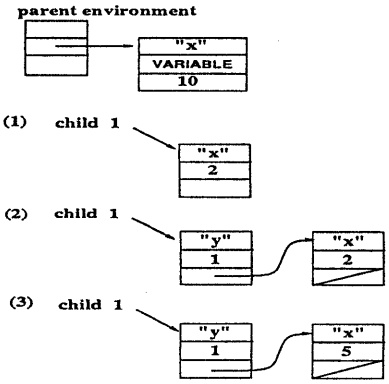

Fig.1.3 Environment

親の処理速度を向上させ、またメモリを多く必要としないという見地からMomolispでは後者を採用している。

2 PiXeLの実現方法

[8]によるPiXeLは、1つのUnixプロセスとして動作し、PiXeLの中でS式の評価は複数のスレッドによって並列に行なわれる。スレッドは他のスレッドやXサーバと通信しながら処理を行なう。PiXeLで書かれたLWP(Lisp Window Manager)はXサーバに対して特別なスレッドでウインドウマネージャとして動作する（Fig.2.1）。

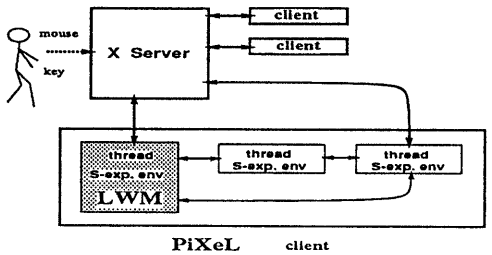

Fig.2.1 PiXeL and X Window System

環境は親、子とも deep binding を用いている。親スレッドが子スレッドを生成する時には親スレッドの環境をそのまま子スレッドのポインタが指すようにし、それぞれの環境は別々に成長して異なる環境が形成される。古い環境の変数の値が変更された場合は、その部分を共有しているスレッドすべてに影響が及ぶが、子スレッドを生成した時点を越えて親スレッドの環境が刈り込まれても子スレッドに影響しない (Fig.2.2)。

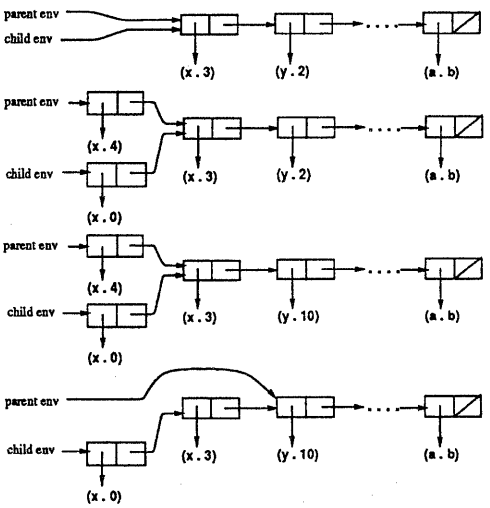

Fig.2.2 Environments

PiXeL ではスレッド間通信の方法として、1 . 変数の共有と 2 . メッセージの 2 種類を用意している。前者は親スレッドと子スレッドで共有している環境を用いて行うので、同じ祖先を持ち、同じ変数にアクセスできるスレッド間でしか通信できない。

後者はメッセージを送る相手を特定して通信するので変数を共有する必要がない。
(thread-send *thread message*) によりメッセージを送信し、(thread-recv *thread*) で受信する。メッセージは送信側スレッド・受信側スレッド・S 式を組にしたベクタであり、送られてきたメッセージを受信側スレッドが FIFO のキューにして保持している (Fig.2.3)。送信は受信側スレッドのキューにメッセージをつなぐことで、受信はキューからメッセージを読み出すことである。

いずれの方法でも S 式を通信することができるので、データ構造をそのままの形で高速に受け渡すことが実現できる。

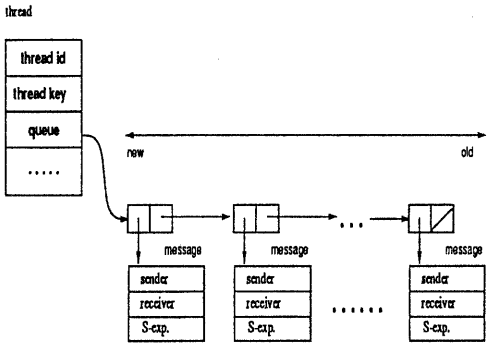

Fig.2.3 Message Queue

スレッドは 1 本の実行待ちキューにつないで管理する。キューの先頭にあるスレッドが現在実行中のスレッドで、S 式の評価を一定回数以上行ったり、入出力でブロックされたりすると他のスレッドに制御が移る。(yield *thread*) によって明示的に他のスレッドに制御を渡すこともできる (Fig.2.4)。

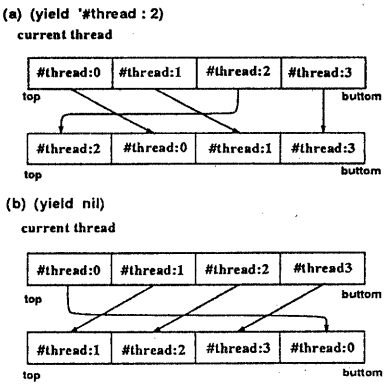

Fig.2.4 Thread Scheduling

3 並列 lisp 処理系 Momolisp の概要

Momolisp は、C 言語で記述された lisp インタプリタであり、次のような 2 つの大きな特徴を持って

いる（Fig.3.1)。

- Xウインドウアプリケーションを記述できるように、Xサーバと通信する関数を用意している。
- 1つのプロセスの中で、複数のスレッドを実行させることで、並列性を実現している。

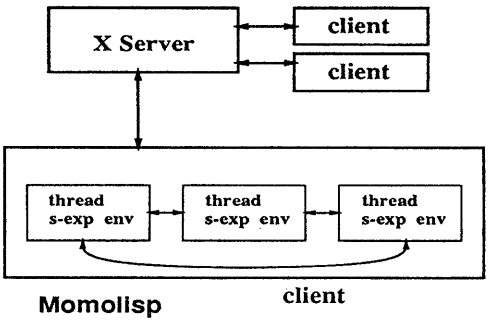

Fig.3.1 Momolisp and X Window System

本節では、Momolispの実現方法について述べる。

3.1 並列性導入の方針及び実現

導入方針として以下を挙げる。

- Momolispは1つのUnixプロセスとして実行されるが、Momolispの中では複数のスレッドが並列に動作できるようにする
- スレッドの生成は、ユーザが明示的に行う

1つのプロセス内でスレッドを実行するために、各スレッドのプログラム・カウンタとスタックを変更する。
Momolispにおける並列性は、SunOS4.0に付属しているLight Weight Processライブラリ[11]を用いることで実現する。

3.2 スレッドの生成

スレッドは、その実行を行うために

- 評価されるS式
- 環境

を独自に保持する。

スレッドの生成は、ユーザが関数create-threadを明示的に呼び出すことで行われる。create-threadは、以下の形式で使用する。
(**create-thread** *s-expression*)
この式が評価されると、*s-expression*を評価するためのスレッドを生成し、親スレッドには式を評価した値として、子スレッドを表すlisp objectを返す。従って、親スレッドは子スレッドの評価が終らなくても、即座に次の評価を行うことができる。

3.3 環境

Momolispでは、各スレッドがそれぞれの環境を独立に保持できるようにするため、以下のような方法を採る（1節)。

- 親スレッド（トップレベル）の環境はshallow binding
- 子スレッドの環境はdeep binding

3.4 スレッド間の通信及び同期

Momolispではスレッド間通信の方法として、メッセージを用いている。
送信側のスレッドは、
(**send** *thread message*)
を呼び出し、メッセージを送信した後は、即座に次のS式の実行に移る。即ち、sendは呼び出したスレッドをブロックしない。
受信側のスレッドは
(**recv** *thread*)
を呼び出してメッセージを受信する。recvが呼び出されると、メッセージを受け取るまで、呼び出したスレッドをブロックする。

各スレッドはFIFOのメッセージキューを持っていて、メッセージは送信側スレッドとメッセージ（S式）の組として保持されている（Fig.3.2)。

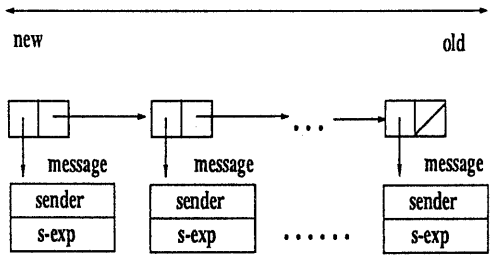

Fig.3.2 Message Queue

3.5 スレッドのスケジューリング

スレッドは1つの実行待ちキューにつないで管理する。生成させるスレッドの優先度はすべて同じにする。このキューの先頭にあるスレッドが現在実行中のスレッドである。

Momolispでは、ある1つの特定のスレッドをスケジューラとして走らせ、一定時間がたつとキューの先頭にあるスレッドをキューの最後に持ってくることによってスケジューリングを行なう。

また、あるスレッドが入力待ちなどでブロックされたときにも、スケジューリングが行なわれるようにする。

あるスレッドがUnixシステムコールを呼び出すと、そのスレッドだけでなく、Momolisp自体がブロックされてしまう。それでは並列実行を有効に実現することができないので、Momolispでは非ブロック型I/Oライブラリを用いる。入力待ちに入ったスレッドに対して、データが準備されていないと、そのスレッドの実行を中断して他のスレッドに制御を渡し、データが準備された時点で再スケジューリングにより、実行を再開する。これにより、Momolisp自体がブロックされるのを回避している。

3.6 Xサーバとの通信

Momolispは、Xウインドウアプリケーションを記述するため、Xサーバと通信する関数を用意している。これらの関数は、サーバとクライアントの通信プロトコルとほぼ1対1に対応しているXlibライブラリの関数を呼び出すことで実現する。

Momolispでの Xの関数は、Xlib関数の仕様にかなり忠実に作っている。Xlibは下位レベルであるので、今後、XToolkitのような上位レベルのインタフェースを、Momolisp上に開発する余地を残している。

4 Momolispの実現方法の検討

本節では、現在のMomolispにおける環境の保持の仕方、及びスケジューリングについて検討を行う。

4.1 環境の保持と通信について

現在の方法では、各スレッドの環境は互いに影響し合うことなく保持されている。また、親ではshallow bindingを採用したため、変数の格納、及び参照は速く行なえる。

子では処理の簡潔さのため、deep bindingを用いている。処理を軽くするため、子での環境の書き換えは行わない。

この方法が実用上、優れたものであるかどうか、また、軽い処理と実行時間の予想が当初の思惑通りになるかどうかは、今後、実験で検討していく予定である。

4.2 スケジューリングについて

現在は、スケジューラにある一定の時間を管理させた時間刻みのスケジューリングを行なっている。この他の手段として、時間ではなく、論理的な評価単位での切断、例えば一定回数のS式の評価を終えると他のスレッドへ制御を渡すことなどが考えられる。また、ユーザが明示的にスレッド制御の切替を行なえる関数を用意することなども考えている。。

5 おわりに

本稿では、「逐次性」を解消したXウインドウシステムインタフェースを構築するために、ウインド

ウアプリケーションを記述するための言語に「並列性」を導入することを考え、実際に Unix 上に並列 lisp 処理系 Momolisp を開発し、その実現方法を述べた。

本研究の最終的な目標は、X ウインドウアプリケーション記述言語に並列性を導入することにより、X インタフェースの向上を図ることである。そのために X サーバと通信機能をもつ並列 lisp インタプリタを作成し、その上に様々なアプリケーションを開発する予定である。

Momolisp は、まだ試作段階であるが、ある程度の実用性（ユーザが耐え得る処理速度）を目標として開発している。今回検討を行なった項目について、実験、比較検討を行なうことを今後の研究課題とする。

参考文献

[1] Ray Chen and S. Partha Dasgupta. Implementing Consistency Control Mechanisms in the CLOUDS Distributed Operating System. In *Proceedings of the 11th International Conference on Distributed Computing Systems*,1991.

[2] R. P. Gabriel and J. McCarthy. Queue-based Multi-processing Lisp. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, 25-44, August 1984.

[3] R. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Trans. on Prog. Languages and Systems*, 501-538, October 1985.

[4] 岩崎英哉他. UNIX 上で作動する mUtilisp システム. 記号処理研究会 48-3, 1988.

[5] 中西正和. Lisp 入門. 近代科学社, 1985.

[6] D. Kranz, R. Halstead, and E. Mohr. Mul-T: A high-performance parallel Lisp. In *SigPlan Conf. on Prog. Language Design and Implementation*, 81-90, 1989.

[7] Albert J. Musciano, Thomas L. Sterling. Efficient Dynamic Scheduling of Medium-Grained Tasks for General Purposing Parallel Processing. In *Proceedings of the 1986 International Conference on Parallel Processing*, 166-175, 1988.

[8] 新田善久. 並列 lisp による X ウインドウ・システム・インタフェイスの実現. 情報処理学会論文誌 Vol.31 No.3., 397-403, 1990.

[9] 大森健児. 並列プログラミングの基礎--マルチプロセッサを指向した応用構成法. 丸善, 1990.

[10] Mark P. Pearson, Partha Dasgupta. CLIDE: A Distributed, Symbolic Programming System based on Large-Grained Persistent Objects. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, 166-175, 1991.

[11] Sun Microsystem Inc. Light Weight Process, *SunOS System Services Overview*, Part No. 800-1753-10, Revision A of 9 May, 1988.

[12] Michael Weiss, C. Robert Morgan, and Peter Belmont. Dynamic Scheduling and Memory Management for Parallel Programs. In *Proceedings of the 1986 International Conference on Parallel Processing*, 161-165, 1988.