システムソフトウェアと
オペレーティング・システム 69－3
（１９９５．６．２２）

実時間記号処理のためのイベントボックス

天海良治 山崎憲一 中村昌志 吉田雅治 竹内郁雄

NTT 基礎研究所, NTT ヒューマン・インタフェース研究所, NTT ソフトウェア研究所

記号処理カーネル TAO/SILENT は, 専用 VLSI を中心としたタグアーキテクチャハードウェア SILENT 上に, マルチパラダイム言語 TAO がのったシステムである. TAO は, Lisp, 論理型, オブジェクト指向の 3 つのパラダイムを機械語レベルで融合した言語である. TAO/SILENT は, 知能ロボットの脳部分, オブジェクト指向コンピュータグラフィクス等, 実時間処理を必要とする実世界への適用をはかることを狙っている.

TAO では, このような実時間記号処理をプログラムするためのプロセス間の通信, 同期, 排他制御, 割り込みといったプリミティブと, これらを組み合わせたり, 1 つの事象を複数のプロセスで待つといった機構を実現するイベントボックス機構でもって実現する.

An Event-Box mechanism for real-time symbolic processing

Yoshiji Amagai Kenichi Yamazaki Masashi Nakamura
Masaharu Yoshida Ikuo Takeuchi

NTT Basic Research Laboratories, NTT Human Interface Laboratories
NTT Software Laboratories

This paper describes primitive functions and an eventbox mechanism for real-time symbolic processing in TAO kernel.

TAO/SILENT system is a dedicated real-time symbolic processing kernel for real world applications, for example, autonomous intelligent robotics, object-oriented computer graphics. For this end, TAO provides a set of powerful primitives for concurrency and real-time processing facilities, such as interprocess communications, data locking and interprocess interrupt. Eventboxs are the most powerful primitives for interrupt handling and real-time processing. By using eventboxes, interrupt disposal, asynchronous communication and broadcasting become possible in a modular way.

1. はじめに

記号処理カーネル TAO/SILENT は, 専用 VLSI を中心としたタグアーキテクチャハードウェア SILENT 上に, マルチパラダイム言語 TAO がのったシステムである. TAO は, Lisp, 論理型, オブジェクト指向の 3 つのパラダイムを機械語レベルで融合した言語である. TAO それ自体は記号処理プリミティブの集合であり, 言語設計者やユーザが利用するプログラミング言語を構築するための基礎言語と位置付けられる. 言語の能力を損わずに言語設計の十分な自由度を備えるため, TAO はメカニズムとポリシーの分離を原則に設計した. 言語構築, 実行のための洗練されたメカニズムを用意するが, ポリシーは言語設計者, ユーザに開放されている.

TAO/SILENT の目的は, 大規模システムの自己安定性, 自己組織化といったヘテロ並列システムの研究, 知能ロボットの大脳部分, オブジェクト指向コンピュータグラフィックスエンジンなど, 実時間処理を必要とする実世界への適用をはかることである. これら実時間記号処理を実現するためには, 言語や OS のはもとより, 関数ひとつひとつにまで, コンカレンシ, 実時間性を充分考慮した設計が必要となる. 本稿では, TAO におけるプロセス間通信のプリミティブと, 同報や複数事象の待ちを実現するイベントボックス機構について述べる.

なお, 我々は以前, 記号処理専用マシン ELIS とマルチパラダイム言語 TAO からなる計算機システム TAO/ELIS に関わってきた. 本稿で述べる言語 TAO は同じ名前を持っているが別の言語である.

2. 並行実行制御と実時間処理

われわれは, TAO/ELIS において, 記号処理専用マシンでのマルチプロセス機能の有効性を確認した. 実世界への応用のためには, さらに強力で軽い並行制御機構と実時間性が必要となる. TAO の OS 部分の設計あたっては, 以下の点に重点を置いた.

- 軽いプロセスと高速なプロセススイッチ
- プロセスレベルの中粒度の並行処理
- 豊富な並行実行制御プリミティブとこれらを柔軟に組み合わせるイベントボックス機構
- 割り込み (イベント) への反応時間が予測可能であること
- 実時間 GC の実装

以下, これらについて述べる.

2.1 TAO のプロセス

TAO/SILENT はマルチユーザ, マルチプロセスのシステムである. TAO の式の実行はすべてプロセスの上で行なわれる. GC もプロセスとして並行に実行される. ただし, 並行に実行されるのはプロセスレベルであって式やいわゆるオブジェクトが実行主体となるわけではない. 例えば, 並行オブジェクト指向言語を TAO で実現するときにも, 実行そのものは TAO のプロセスのもとで走行しなければいけない. だが, TAO のプロセスはこの方式をとるのに十分軽い.

プロセスが実行時に使用するスタック領域はプロセスに固有であるが, それ以外のセル領域等のメモリ空間はすべてのプロセスに共通である. アドレス変換は行なわないので, ポインタ値はすべてのプロセスで常に有効である. 通信においてポインタを渡したときにも, ポインタの変換や指し示すデータのコピーは不要である. 共有データのアスセスには後述の排他制御機構を使用する.

プロセスの生成は関数 make-process で行なう. make-process で作られたばかりのプロセスは, 実行する式を持たず, スタックも割り当てられていない. これに対し, 関数 task でチョア (chore: 関数と引数の組をチョアと呼ぶ) を与えると, プロセスはアクティブになり, 実行が開始される. このチョアのことをとくにタスクと呼ぶ. 実行中のプロセスに対し, 関数 task でさらにタスクを与えることも可能である. プロセスは現在のタスクが終了し次第, 与えられたタスクを順に実行する.

プロセスには関数 interrupt でもチョアを与えることができる. これを割り込みチョアという. 割り込みチョアは task で与えられた本来のタスクに優先して割り込み実行される.

関数 task や interrupt で与えられた式の実行がすべて終了したときは, プロセスは休止する. また, 関数 quit-task を用いて, プロセスを強制的に休止状態にさせることができる. ただし, プロセスにタスクがいくつか与えられているときは休止せず, 次のタスクの実行が開始される.

```
(!p1 (make-process))
(task p1 (make-chore #'print-file "a.tao"))
```

! は代入を表わす.

TAO のプロセスは, 64 の優先度に基づいた先取り可能スケジューリング (priority based preemptive schedul-

ing) に従って管理される．優先度のより高いプロセスは低いプロセスに優先して走行状態となる．同じ優先度のプロセスが走行可能であるときは，タイムスライスによるラウンドロビン方式によりスケジューリングされる．プロセススイッチにかかる時間は，使用スタック領域に重なりがないとき約5マイクロ秒である．

2.2 プロセス間通信

TAOのプロセス間の通信機構は，同期制御，通信などのプリミティブと，これらを組み合わせたり，待ちが解けたときに指定の関数を実行するイベント機構からなる．まず，特徴的なプリミティブについて述べる．

セマフォとロッカ

セマフォは，プログラム実行の排他制御に用いる．TAOのセマフォは2値セマフォで，関数make-semaphoreで生成し，関数P, Vでそれぞれセマフォの取得と開放を行なう (TAOではシンボルの大文字と小文字は区別される)．Pでは，セマフォがすぐに取れなければプロセスは待ちにはいる．待ちにはいらないP-no-hangも用意されている．

これに対し，ロッカはビジィウェイト型のデータアクセスの排他制御機構である．排他制御はデータの占有権を制御することでなされる．セマフォはデータ更新を行なうプログラム部分の実行を排他制御するのに用い，ロッカは，データそのものに鍵をかける．セマフォと比較して，ロッカはメモリ使用量が少なく処理も軽い．よって，共有しているリストのすべての要素に個別にロックをかける，といった使い方が可能である．

データの占有権の取得の待ちは同期型の待ちではなく，ビジィウェイト，すなわち，頻繁に占有権の取得が許されるかどうかを調べにいくことで待つ．このため，あるプロセスの占有権の放棄と別プロセスの獲得は同期しては起こらない．ビジィウェイトの間隔，すなわち，鍵の検査の間隔は，待ちはじめは1ミリ秒以下であるが，待ち続けていると間隔が長くなる．最終的には，約1分に1回の検査しか行なわなくなる．

ロックにはR型とW型の2種類がある．R型のロックはロッカのデータを読み込むためのものである．正しく書かれたプログラムであれば，これを読んでいるあいだにほかのプロセスによってロッカのデータが書き変わることはない．(R型ロックを取っているときは，ロッカのデータは読み込みだけに限らないといけない — これはユーザの責任である.) W型のロックはロッカのデータ (あるいはその内部構造) を書き換えるために取るものである．正しいプログラムであれば，W型のロックを取ってから，それをアンロックするまでは，ほかのプロセスがそのデータを読んだり変更したりしない．R型のロックは複数のプロセスが同時に取りうる．一方，W型は，R型のロックもW型のロックもないときにのみロックを取ることができる．

データをロッカーに保持するには，関数make-lockerを使う．

(!shared (make-locker data))

R型ロック，W型ロックはそれぞれ (R shared) (W shared) で取る．すぐに取れなければビジィウェイトにはいる．ロックを放すには，それぞれ (UR shared) (UW shared new-data) を使う．UWの場合，データをnew-dataに置き換えてからロックを放す．

メールボックスとバッファ

メールボックスとバッファはデータの通信に用いられる．メールボックスを使って，任意のTAOデータを送ることができる．メールボックスに入れられたデータは，FIFOで取り出すことができる．また，内部的にはデータはリストで保持するので，任意個数の未受信データを保持できる．送るときには待ちはない．通信は，ポインタを送ることで行なうので，受取った構造データは送り側が保持していれば共有となる．メールボックスからデータを取り出すとき，データがなければプロセスは待ちに入る．データの送信に同期してこの待ちは解かれる．メールボックスのアクセスは，データの送り側と取り出し側のプロセスで別のポインタで行なう．

これに対し，バッファは送信データ保持のための固定長メモリしか持たない．このため，送信においても待ちが生じうる．バッファは典型的には，外部プロセッサとの通信において使用される．外部との通信では一般に大量のデータを1回で通信したほうが効率が良い．このために，メモリが一杯になったときに起動されるハンドラなどをユーザがバッファに対し付与できる機構がある．

セマフォ，メールボックス，バッファでは，送信データがない場合には，基本的に受信側はデータを待ってしまう．イベントボックス機構を用いると，データが到着した時点で特定の関数を起動できる．また，これを用いて，複数の受信の待ち，実時間の制御，ブロードキャスト型通

信なども可能である.

割り込み

あるプロセスに対し, 現在行なっている処理を中断して, 別の処理 (チョア) を実行させることができる. これを割り込みと呼ぶ. 割り込みで処理されるべきチョアを割り込みチョアと呼ぶ.

割り込みには二つの方法がある. 一つは, 別のプロセスから関数 interrupt を用いて割り込みチョアを与える方法である (これをプロセス間割り込みと呼ぶ). もう一つは自分で割り込み関数を指定し, イベントによって与えられた値を引数として割り込みチョアを合成して, それを起動させるものである. イベントには, メールの到着, 指定時刻の到来など, いくつかの種類がある.

関数 interrupt で割り込みチョアが, 指定されたプロセスの割り込みキューに入れられる. 割り込まれたプロセスがレディであった場合, 次に走行状態になるときにこのキューが調べられ, そこに入っているチョアが順に実行され, その後もとの実行が再開される. 割り込まれたプロセスが待ちであった場合, その割り込みキューの中のチョアの実行が終わった時点で再び待ち状態に戻る.

割り込みは禁止することができる (なにも指定しなければ, 割り込み可能である). 関数 set-interrupt-status を用いて陽に制御できる他に, 割り込みチョアの実行は, 割り込みが自動的に禁止された状態で行なわれる. 割り込み禁止状態であっても, 割り込みキューへのチョアの登録は行なわれる.

2.3 イベントボックス機構

イベントボックスは, 割り込みを使った非同期的通信, 同報通信, 複数の要因を同時に待つといった機能を実現するプリミティブである.

例をあげると, 他のプロセスとメールボックスで通信しているときに, キーボードからの入力も取得したいとする. これまでのプリミティブだけで実現するには, キーボード入力を待つ専用のプロセスを生成し, キーボードからのデータをメールボックスへ送るまた, timeout つきの待ちといった, 本来的に複数の事象を待つ状態がありうる. 逆に例えば, 3 つ以上のプロセスが同期をとろうとすると, すべてのプロセス間で互いにメールを送りあったり, 同期のためにプロセスを生成し, すべてのプロセスが待ちあわせ点に到達するのを待つといったことが必要となる. これについては, 1 つの事象を 3 つのプロセスが待つことができればよい. このような機構を実現するのがイベントボックスである.

入力待ちが解けるなど, なんらかの状態が遷移したことをイベントが発生したという. また, イベントを発生するような状態, または状態が定義されているものをイベントソースという. イベントソースには以下のものがある.

タイプ	ソース:
:semaphore	セマフォ: セマフォが取れているという状態
:mailbox	メールボックス: メールボックスにメールが来ているという状態
:buffer	バッファ: バッファからの入力がとれるという状態
:timeout	時間の経過: 指定した時間が経過したという状態
:user	状態を表わすデータ: ユーザ定義の事象が起きたという状態

イベントボックスの基本的動作は, これらのイベントの発生をトリガにしてプロセスに割り込むことである.

イベント割り込みの例

イベントボックスの動きを例をあげて説明する.

まず, イベントタイプ, イベントソースを指定してイベントボックスを生成する.

(make-eventbox *type source flag*)

flag はイベント発生を待っているプロセスをキューで扱うか, 集合として扱うかを指定する. デフォルトはキューである. 集合と指定すればイベントの同報となる (後述).

例えば, イベント「5 秒が経過した」の指定は

(!eb (make-eventbox :timeout 5000))

とする. この時点では, 5 秒の計測はまだ始まらない. イベントボックスで指定したイベントの発生を知るには, event-alert 構文を使う. イベントボックスにフックをかけるという. event-alert 構文は次のような形をしている.

(event-alert *eventbox hookfn BODY* ...)

フック関数 *hookfn* は 2 引数の関数である. *BODY* 実行中に *eventbox* で指定されたイベントが発生すると, プロセスに割り込みが起きて *hookfn* が実行される. *hookfn* の第 1 引数は *eventbox*, 第 2 引数はイベントに応じた値 (イベント値) である. event-alert 構文を抜けると割り込

みは発生しない．また，unhook-event で自プロセスの指定したイベントボックスによる割り込みを禁止できる．

```
(event-alert eb
   (op* (b n) (ring-bell))
   (do-something))
```

を実行すると，event-alert にはいってから 5 秒経過したところでイベントが発生し do-something を実行中のこのプロセスに割り込む．ここでは b に eb，n に 0 が代入されて ring-bell が実行される．同時に 5 秒の計測が再び始まる．n の 0 の意味は後述する．

タイムアウトはイベントボックスを使用することによってのみ実現できる．よって，タイムアウトを設定するその度にイベントボックスを生成することになる．このイベントボックスの使い捨てを防止するため，次のようなマクロ timeout-alert を用意する．

```
(timeout-alert interval hookfn body)
= (event-alert
     (make-eventbox :timeout interval)
     (op* (_ v) (_hookfn v)) body)
```

body の実行中にタイムアウトによって割り込みが発生する．割り込みの後，時間計測が再開されるので，最初に指定した時間間隔ごとに *hookfn* がくり返し実行されることになる．この割り込みチョアの中でユーザが与えた *hookfn* を実行するが，event-alert と異なり無引数の関数であり，ユーザの *hookfn* にはイベントボックス渡さない．つまり，timeout-alert マクロを使う限り，:timeout のイベントボックスをユーザが他の変数に代入したり，持ち出すようなことはない．よって，システムはこのように頻繁に使われるイベントボックスを event-alert 毎に生成するようなことをせず，別に保持し，再利用するようなプログラム変換を行なうことが可能となる．

次にメールボックスの受信の例を示す．メールボックスに送られたメールがないとき，メールボックスからの受信には待ちが生じる．だが，次のようにメールの到着をイベントで知ることにより，待つことなく，データを得ることができる．mb をメールボックスの取り出し側とする．

```
(!ebm (make-eventbox :mailbox mb))
(let ((data-list ()) (ndata 0))
  (event-alert ebm
     (op* (_ x) (!!cons x !data-list) (!!1+ !ndata))
     (do-something) ))
```

仮引数の _ は呼び出し時の実引数を使わないことを示す．!! は TAO の自己代入機能を表わしていて，(!!cons x !d) は (!d (cons x d)) の意味となる．

timeout と組み合わせるのは次のようにする．

```
(block
 (event-alert (make-eventbox :timeout 1000)
     (op* (_ _) (exit block #f))
     (P semaphore) ))
 (if (P? semaphore) ...)
```

この例では，1 秒を限度にセマフォ semaphore の P 操作の完了を待つ．P? は自プロセスがセマフォをとっているかどうかを判定するプリミティブ関数である．

event-alert をネストすることで複数のイベント発生を知ることができる．ネットワークからの入力，キーボードからの入力，及び，60 秒のタイムアウトの 3 つのイベントを待つなら，

```
(!net (make-eventbox :buffer (open-tcp-buffer ...)))
(!tty (make-eventbox :buffer *console*))
(!tim (make-eventbox :timeout 60000)
...
(block
  (event-alert tim
    (op* (_ _) (exit block 'timeout))
    (event-alert tty
      (op* (_ v) (exit block v) )
      (event-alert net
        (op* (_ v) (exit block v) )
        (wait) ))))
```

とする．

ユーザタイプイベントボックス

ユーザプロセス間の通信もイベントボックスを通じて行なうことができる．:user タイプのイベントボックスを生成し，event-alert 構文で使用する．イベントは，関数 raise-event で明示的に発生させる．また，イベントソースが undef 値でないとき，event-alert にはいったとき，まず最初に割り込みが起きる．

```
(raise-event ueb value)
```

value はイベントソースとなり，かつフック関数の第 2 引数に渡される．raise-event は割り込みが受け付けられたか否かによらずイベントボックスのイベントソースフィールドを書き換える．フック関数実行中に raise-event されれば，フック関数に渡された値と，イベントソースの値が異なることありうる．

同報と eventful

make-eventbox のフラグで，イベントを待っているプロセスを集合として扱うことを指定して生成したイベン

トボックスでは，待っているプロセスすべてで割り込みがおきる．ただし，そのイベントボックスにフックをかけているプロセスだけに割り込みが起きるのであるから，データを同報するような場合には，すべてのプロセスがevent-alertにはいるまで，送り側プロセスがデータを送らないよう，プログラムする必要がある．

イベント割り込みが発生した時点で，そのプロセスはそれ以上のイベント割り込みを受け付けない状態になる．これをイベント満腹状態 (eventful) という (反対は空腹 evempty)．満腹状態のときに起きたイベントはサスペンドされる．ただし，ユーザイベントの発生は最新以外は捨てられる．タイムアウトイベントは受け付けられなかった回数が記録され，受け付けられたときにその回数がイベント値としてフック関数の第2引数に渡される．フック関数を抜けると満腹状態は解消する．

3. 実時間処理サポート

TAOの実時間処理は，すべての処理のパスにおいて応答遅延時間を前もって保証することが求められるハードリアルタイムシステムにはならない．記号処理の適用される知的な処理は，ハードリアルタイムの前提と相容れない．TAOで目指す実時間処理は，適正な負荷のもとでシステムの応答遅延時間を保証 (目標は100マイクロ秒) し，また，アプリケーションプログラムの応答遅延時間の予測を助けるため，システムで用意するプリミティブ関数には，遅延時間を付記する予定である．

TAOは，長時間のシステム停止を避けるための実時間GCの実装や，GCの効率を高めるためのゴミの出にくい構造データを設計など，システム全体で，実時間性を高めるような設計となっている．

4. 終りに

記号処理カーネルTAO/SILENTの実時間記号処理のためのプリミティブとイベントボックス機構について述べた．プリミティブを柔軟に組み合わせることができるイベントボックス機構により，実世界の非同期なインターラクションを自然に表現できる．現在，試作ハードウェア上でTAOの実装を進めている．

[文献]

[1] Takeuchi, I: Concurrent Programming in TAO - Practice and Experience, Parallel Lisp: Language and Systems, LNCS 441, Springer-Verlag, 1989 pp.271-299

[2] 竹内，吉田，天海，山崎: 新しいTAOの設計，記号処理研究会, 56-2, 1990.

[3] 竹内，天海，山崎，吉田: TAOのオブジェクト，記号処理研究会, 65-1, 1992.

[4] 竹内，天海，山崎，吉田: TAOにおける余剰引数，多値，倍長数の計算などの見直し，記号処理研究会, 66-3, 1992.

[5] 山崎，天海，竹内，吉田: ヒープを使用する論理型言語でのトレール方式，記号処理研究会, 67-4, 1993.

[6] Yamazaki, K, Amagai, Y, Yoshida, M., and Takeuchi, I.: TAO: an object orientation kernel, Object Technologies for Advanced Software, LNCS 742, Springer-Verlag, 1993, pp.61-76