記 号 処 理 69－3
（1993．6．18）

TAOのコンカレンシ・コントロール

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

記号処理カーネルSILENT上の言語TAOのコンカレンシ・コントロールについて述べる.

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. このためには, 言語やOSの基幹部分から, コンカレンシと実時間性を充分考慮した設計が必要になる.

TAOではこれを, 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ヴと, 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.

Concurrency Control in TAO

Yoshiji Amagai Kenichi Yamazaki Masashi Nakamura
Masaharu Yoshida Ikuo Takeuchi

NTT Basic Research Laboratories, NTT Human Interface Laboratories

This paper describes concurrency control of the TAO language on SILENT.

A TAO/SILENT system is a dedicated real-time symbolic processing kernel for applying AI to real world problem, such as autonomous intelligent robotics, object-oriented computer graphics. For this end, it is necessary to concider a concurrency and real-time support mechanism in design of basic language and operating system.

TAO has primitives for concurrency control such as inter process communication, synchronization, mutual exclusion and remote funcion execution. And there is an "event-box" mechanism that enable one process to wait multiple events and one event to interrupt multiple processes.

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マシン SILENT 上に, 豊富な記号処理プリミティブとコンカレンシ・プリミティブを備えた機械語 TAO がのったシステムである. また, 三次元ベクタ指向浮動小数点演算器 TARAI を備え, 記号処理と大規模数値演算の統合をはかっている. 現在, SILENT 試作機の試験と TAO の詳細設計を行なっている.

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による機能の質の向上をねらった計算原理の探究のためのプラットフォーム, 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クスエンジン等, 実時間処理を代表とする実世界への適用をはかること, である.

この目的を果たすには, 言語や OS の基幹部分から, コンカレンシ, 実時間性を充分考慮した設計が必要になる.

本稿では, TAO/SILENT のプロセス, コンカレンシ機能について述べる.

2. TAO/SILENT の概要

まず, TAO/SILENT について簡単に述べる.

SILENT は, 8 ビットタグと 32 ビットポインタを 1 語とするタグアーキテクチャの専用 VLSI チップ (SILENT チップ) を搭載したボードコンピュータである. タグ部, ポインタ部それぞれが専用の ALU をもっており, ポインタ部の比較や演算と同時に, タグ部分の比較や, タグ生成ができる. SILENT チップは, 2 語からなるセルを一度に読み書きできるよう, 80 ビット幅のシステムバスを通して, 主記憶, 浮動小数点演算器 TARAI 等と通信する. SILENT チップ制御のマイクロコードを収める WCS, スタックメモリはそれぞれ専用の高速メモリチップを用いる. どちらも主記憶のアドレス空間にアドレスをもつが, SILENT チップ直結のバスも備えている. つまり, 主記憶, マイクロコード, スタックを同時にアクセスできる 3 バスアーキテクチャとなっている.

TAO はこの SILENT の核言語である. 豊富な記号処理プリミティブとコンカレンシ・プリミティブを備え, 実時間処理に対処している. 機能的には, TAO/ELIS で提唱した Lisp, オブジェクト指向, 論理型のパラダイム融合をより深め, さらに, これら記号処理と TARAI チップによる実世界向き 3 次元数値処理との統合も狙っている.

3. TAO のプロセス

TAO の CPU リソース, すなわち計算時間はプロセスに対してのみ与えられる. すなわち, CPU リソースを使うすべての式の実行は, いずれかのプロセスで行なわれる. プロセスは, 固有のスタック領域と各種レジスタ値等のコンテクストをもつ. 構造データが格納されるメモリはすべてのプロセスに共通である.

プロセスは次の四つの状態のいずれかをとる.

- 休止 (dormant): 実行すべき式をなにももたない状態.
- レディ (ready): 走行可能であるが, CPU リソースが割り当てられていない状態.
- 待ち (waiting): 外部からの入力, 信号, 合図を待っている状態.
- 走行 (running): 実際に CPU リソースを消費している状態, すなわちなんらかの実行をしている状態.

休止以外の状態にあるプロセスを活性プロセス, あるいはアクティブなプロセスと呼ぶ.

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休止状態にある. 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 (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. 活性プロセスに対し, 関数 task でさらにタスクを与えることも可能である. task でタスクを与えたときは, 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 (アクティブか休止かにかかわらず) 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, プロセスは休止状態に戻る. また, 関数 abort-process を用いて, 活性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状態にはならず, 次のタスクの実行が開始される.

4. プロセススケジューリング

TAO/SILENT は実時間処理を行なうことが目的のひとつである. よって, プロセススケジューリングについては, 事象が発生してから, それに応答するまでの時間が予

測できることが必要である.

TAO のプロセスは, 32 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 scheduling) に従って管理される. 優先度のより高いプロセスはレディ状態になるとただちに走行状態となる (先取りが起こる). それまで走行していたプロセスはレディ状態となる. この先取り可能スケジューリングにより, 応答時間の予測がやりやすくなる. 優先度は実行時に動的に変更可能である. 動的な優先度の変更の結果, 先取りが起きることもある.

同じ優先度のプロセスが二つ以上レディ状態にあるときは, 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. ラウンドロビン・スケジューリングは同じ優先度を持つプロセス群に対してのみ適応される. 指定した CPU リソースをあるプロセスが消費すると, すなわち一定の CPU 時間を消費すると, そのプロセスはレディになり, 同じ優先度をもつ別のレディプロセスが走行状態になる. 同じ優先度のレディプロセスがある限り, 順に CPU リソースが割り当てられていく. この優先度のレディプロセスがなくなるまで, これが繰り返される. 関数 yield を実行することで, 陽に CPU を明け渡すことができる. 逆に, 構文† (without-process-switch • <*body*>) を使えば, 暗黙の seq‡ *body* の逐次実行の間, プロセススイッチを禁止することができる. ただし, 約 50 μ 秒以上 without-process-switch 構文を実行するとエラーとなる. つまり, 50 μ 秒以上プロセススイッチなしに走らせることは許されない. また, *body* の実行中に待ちに入ったり, 休止状態になるときは, 自動的にプロセススイッチ禁止が解除される.

なお, 関数 wait を実行することで, 陽に待ち状態に入ることができる. これはイベントボックスあるいはプロセス間割り込みを待つときにのみ使用する.

5. プロセス間通信

TAO/SILENT では, 主記憶のアドレス空間をすべてのプロセスで共有しているので, 基本的にはデータの受け渡しはポインタをそのまま渡してやればよい. これがもっとも高速かつ記憶効率のよい方法ではあるが, 同期や排他制御を行なうための OS でサポートされた通信機構は必須である.

† 構文とはいわゆる特殊形式のことである.
‡ implicit progn に相当するもの.

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.

機構はは次の 6 個である.

セマフォ:	排他制御
メールボックス:	同期制御, データ通信
バッファ:	データ通信
ロッカ:	ビジィウェイト型排他制御
割り込み:	遠隔関数実行
イベント:	待ちが解けたときに指定の関数を実行する機構

以下, 各々について述べる. なお, プリミティブ関数のすべてをあげているわけではなく, 特徴的なものを選択して説明する. 例えば, 生成関数やテストの関数は省略した.

5.1 セマフォ

セマフォは, データ通信ができないので, もっぱら同期, 排他制御のために使用される. プロセス間でセマフォを共有するには, メールボックスなどで送るか, プロセス実行のためにプロセスに与える task の引数として渡す.

(P *semaphore*) 関数
(V *semaphore*) 関数

P はセマフォ *semaphore* を取る. すぐ取れなければ待ちに入る. 自分がすでに取っていたときはエラーとなる. V は自分が取っているセマフォ *semaphore* を離す. なお, TAO はシンボルの大文字 / 小文字を区別する.

(P-no-hang *semaphore*) 関数

セマフォ *semaphore* を取る. 自分がすでに取っていればエラー.

(sys:V *semaphore*) システム関数

強制的にセマフォ *semaphore* を離させる. バグによりデッドロックに陥ったプロセスの救済等に用いる.

P と V の操作でクリティカルセクションを囲むには, 次ようなマクロを使うとよい.

```
(critical-section semaphore • <body>)
  = (unwind-hook (seq (P semaphore) • body)
       (op* (#:ignore) (V semaphore)) )
```

unwind-hook は unwind-protect に相当するが, 脱出時に実行されるべき関数を与えるところが異なる. op*は動的スコープの無名関数で, lambda に相当する.

5.2 メールボックス

メールボックスはデータ通信が可能である．任意のTAOデータを送ることができる．送るときには待ちはない．また，送信にあたってデータをコピーをするようなことはなく，ポインタをそのまま送信する．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．さらに，送信に同期して受信の待ちが解かれるので，同期制御を行なうことも可能である．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．

(send-mail *mailbox value*) 関数

メールボックス *mailbox* に *value* で示されたメールを送る．値は *mailbox* に受けを待っているプロセスがいたら #t，いなかったら #f である．(TAOではブール値として，#t, #f, _ (未定義を表わす) の3値をもつ)

(receive-mail *mailbox*) 関数

メールボックス *mailbox* にメールが送られていれば，それの先頭を取る (FIFOキューからデキューする)．なにも送られていなければ，送られるまで待つ．

(peek-mail *mailbox*) 関数

メールボックス *mailbox* にメールが送られていれば，それの先頭のメールを主値，#t を副値とした多値を返す．メールボックスのFIFOキューはデキューしない．なにも送られていなければ，主値が _ (undef)，副値が #f の多値を返す．

(receive-mail-no-hang *mailbox*) 関数

メールボックス *mailbox* にメールが送られていれば，それの先頭のメールを主値，#t を副値とした多値を返す．メールボックスのFIFOキューはデキューする．なにも送られていなければ，主値が _，副値が #f の多値を返す．

図1のプログラムは，プロセス p を生成し，そこでオブジェクト obj にメッセージ msg を送る TAO の式 [obj (msg arg …)] を実行する．実行の結果は変数 v をアクセスすることで得られる．v は receive-mail を遅延評価するので，値が送られていなければ，そこで待ち状態となる．いったん受取った後は，v を何度アクセスしても待たない．なお，_m の下線は本来は評価しないメッセージ部を強制評価することを指示する．仮引数部のドットは，それ以降が余剰引数であることを指示する．実引数部 (_m . args) にあるドットは TAO の意図的ドットと呼ばれるもので，(a . (b c)) と (a b c) は TAO の式の解釈 (フォーム化) では区別される．

```
(!mb (make-mailbox))
(!v (delay #'receive-mail (list mb)))
(!p (make-process))
(task p #'(op (b o m . args)
              (send-mail b [o (_m . args)]))
   (list mb obj msg arg …) )
```

図1. 未来型メッセージ送信

5.3 バッファ

バッファは，任意個の未受信データを保持できるメールボックスと異なり，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．バッファについては稿を改めて報告したい．

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．

5.4 ロッカ: ビジィウェイト型排他制御

ロッカはビジィウェイト型排他制御機構である．これは，任意のTAOデータに対する排他制御を行なう機構である．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使用するのが普通だが，ロッカは，データそのものに鍵をかけるものである．よって，ロッカによる排他制御では，複数のプロセスが同じプログラム部分を走行するとき，同じ (eq な) データをアクセスするときには排他的だが，違うデータをアクセスするなら同時に走行することができる．また，セマフォと比較して，ロッカはメモリ使用量が少なく1つ1つの処理も軽い．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起こらない．これはもっぱらデータの排他制御に用いられ

る機構である.

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.

ロッカによる待ちは, 次の二つの場合に起こりうる.

- 他のロックが取られているときに, W型のロックを取ろうとした場合.
- W型のロックが取られているときに, R型のロックを取ろうとした場合.

ロッカによる待ちは, セマフォ, メールボックスなどとは異なる機構である. ロッカを待っているプロセスは微小ではあるが定常的にCPU資源を消費する. そのため, システムの定める一定時間 (busy-wait-quantum) の倍数を経過してもビジィウェイトを続けているプロセスはエラーを起こす (関数Wまたは関数Rがエラーを起こした形になる). 倍数はロッカのデータに対する処理の粒度によって調整できる. この倍数をグレイン (grain) と呼ぶ. なお, このエラーによりデッドロックの検出が可能になる.

(W *locker* [*grain* 1]) 関数

ロッカ *locker* のW型ロックを取る. 値はロッカの中のデータ. すぐ取れなければビジィウェイトに入る. Wはデータを変更する目的でアクセスするための関数である. 自分がすでに *locker* を取っていればエラーである.

(R *locker* [*grain* 1]) 関数

ロッカ *locker* のR型ロックを取る. 値はロッカの中のデータ. すぐ取れなければビジィウェイトに入る. Rはデータを読むだけの目的でアクセスするための関数である.

(UW *locker new-data*) 関数

W型ロックを取ったロッカ *locker* のデータを *new-data* に置き換えてから, W型のロックを外す.

(UR *locker*) 関数

R型ロックを取ったロッカ *locker* のR型ロックを外す.

ロッカは, 助言型 (advisory) の排他機構であるから, 対象データにアクセスするであろうプログラム部分で正しくロッカを取り, また外さなければいけない. 軽いロックにRやWといったシンタクス的にも短い関数名をつけることは正しいプログラムを書かせる上でも重要なことと考える.

5.5 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, 別のプロセスから割り込みチョアを与える方法で, 関数 interrupt を用いる (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 (次章).

割り込みが起きると, 割り込みチョアが, 割り込ま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, そのプロセスは割り込みチョアを実行することについてはレディ状態になる. そして, 走行状態となり, 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. 割り込まれたプロセスが休止であった場合, 関数 interrupt は関数 task と同じ結果をもたらす.

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ほかに, 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.

(interrupt *process fn args*) 関数

fn と *args* で示されたチョアを *process* の割り込みキュー

に入れる (*process* が割り込み禁止状態であってもよい). interrupt を実行したプロセスが割り込み先の *process* よりも高い優先度をもっていたときは, このチョアが割り込みキューの先頭に入れられる. 同等以下の場合は後ろに入れられる.

6. イベント機構

これまで述べてきたプリミティブは, どれも単機能である. いずれかの要因で待ち状態にはいると, 外部からの割り込み以外の要因に反応することができない. 例えば, 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得たいような場合, キーボード入力を待つ専用のプロセスを生成し, メールボックスへ送る, といったことが必要になる. 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. これは, 1 つの事象を複数のプロセスで待つ, すなわち事象のブロードキャストの機構があれば解決する.

6.1 イベントとイベントボックス

TAO では, プロセスが複数の事象の発生を待てること, 1 つの事象を複数のプロセスが待てること, を可能にするためイベントとそれを操作するイベントボックスのメカニズムを用意した.

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.

イベントはイベントボックスを介して操作される. まず, イベントタイプとイベントソースを指定して, イベントボックスを作成する. イベントタイプとイベントソースには次のようなものがある. :user タイプのイベントの場合, イベントソースの内容が状態を表わすと解釈する. イベントボックス生成時にはその初期状態をイベントソースとして指定する. なお, ロッカはイベントソースとしては使えない. また, タイムアウトの機能はイベントボックスでのみサポートされている.

タイプ	ソース:
:semaphore	セマフォ: セマフォが取れたという状態
: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たという状態
: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

イベントボックスは構造データであり, イベントタイプ, イベントソース, イベント待ちのプロセス集合, 補助フィールドからなる. イベント待ちプロセス集合は, このイベントボックスのイベントの発生を待っているプロセスの集合である. 補助フィールドはユーザが自由に使用できる. イベント待ちプロセス集合以外は, それぞれのフィールドの中身を知る関数が用意されている.

6.2 イベント発生の検知

イベントの発生を知るには event-hook 構文を用いて, フォームの実行中にイベントが発生したときに割り込みをかける. このことをイベント (あるいはイベントボックス) にフックをかけるという. event-hook 中のフォームの中で別のイベントにフックをかけることもできるから, 複数のイベントの発生を, 同時にかつ非同期的に待つことができる. また, 一つのイベントボックスに複数のプロセスがフックをかけることもできる. この場合, すべてのプロセスで割り込みが発生する. イベントのブロードキャストにあたる.

event-hook 構文は次のような形をしている.

(event-hook <*form*> *eventbox hookfn*)

form を評価するが, 評価中に *eventbox* のイベントが起こったときは, 自分自身に *hookfn* と引数のリストをチョアとする割り込みが起こる. *hookfn* は *eventbox* とイベントタイプごとに決められた値を受け取る 2 引数の関数である.

フックをかけたときの動作は, イベントタイプによって異なる. まず, 一つのプロセスだけがフックをかけた場合について説明する.

:semaphore P 操作が行なわれ, 成功したときに割り込みが発生する. イベントソースであるセマフォアはこのイベントボックスが占有することになる (セマフォアは, プロセスだけ

でなく，イベントボックスも取れる)．#t が値として *hookfn* の第2引数に渡される．このセマフォアの解放は自動的にはなされない．ユーザがV操作を行なう必要がある．

`:mailbox` receive-mail が開始され，実際にメールを受信したときに割り込みが発生する．すでにメールが届いていたときは，直ちに割り込む．receive-mail された値が *hookfn* の第2引数に渡される．

`:buffer` input が実行され，バッファから情報が入力できたとき割り込みが発生する．input された値が *hookfn* の第2引数に渡される．

`:timeout` 時間の計測が開始され，指定時間が経過したら割り込みが発生する．割り込みが発生した時刻が *hookfn* の第2引数に渡される．割り込みが発生したあと，再び時間の計測が始まる．

`:user` イベントボックスと状態を表わすユーザ定義のデータを引数として raise-event を実行したときに割り込みが発生する．raise-event で与えられた値が *hookfn* の第2引数に渡される．また，イベントボックスのソースフィールドに保持される．:user 以外のタイプに raise-event を行なうとエラーである．:user 以外のタイプのイベントボックスに対する raise-event はシステム内部のみで行なわれる

一つのイベントボックスに複数のプロセスがフックをかけた場合，イベント発生の時点にイベントボックスのイベント待ちプロセス集合に登録されていたプロセスすべてに割り込みがおきる．これがイベントのブロードキャスト機能である．

例えば，:mailbox タイプのイベントボックスに複数のプロセスがフックをかけたとき，メールが届いたときに割り込みが発生し，その時点でフックをかけていたプロセスの割り込み関数が実行され，receive-mail で受取った値は各々の割り込み関数に引数として渡される．ただし，この値はイベントボックスに保存されるわけではないので，これより後に同じイベントボックスにフックをかけたプロセスはこの値を受取ることはできない．フックをかける前にメールが届いていたときは，最初にフックをかけたプロセスだけに割り込みが発生し，値を受取ることになる．

TAO のプリミティブとしてのイベントボックスは，その効果がプログラムの動的状況に強く影響される．グループプロセスに割り込みで確実にメールをブロードキャストする，といった古典的なコンカレンシコントロールを行なうためには，フック関数やフックの前後での処理が必要である．どこまでをプリミティブがサポートするかについては議論の余地がある．

なお，イベントによる割り込みは，プロセス間割り込みと同様に一時的に禁止できる．割り込み処理を行なっているあいだは，自動的に割り込み禁止状態である．一方，イベントによる割り込みを完全にやめるには，関数 disable-event を実行する．フックをかけた event-hook 構文が正常終了あるいは脱出によって終了したときは，自動的にそのイベントボックスに disable-event が実行される．つまり，そのプロセスにとって，イベントによる割り込み受け付けのスコープが閉じるわけである．

例を示そう．

1: 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つの事象を待つ

event-hook をネストすることで3つの事象を待つことができる．タイムアウトのときは throw するとする．

```
(!net (make-eventbox :buffer (open-tcp-buffer ...)))
(!tty (make-eventbox :buffer *console*))
(!tim (make-eventbox :timeout 60000)
...
(block
  (event-hook
   (event-hook
    (event-hook
      (wait) ; もっとも内側では待つだけ．
      tim (op* (x v) (throw 'timeout)) )
    tty (op* (x v) (exit block v) ))
   net (op* (x v) (exit block v) )))
```

この例では，プリミティブをそのまま使用した．タイムアウトもイベントボックスを待つごとに生成するようになっている．

タイムアウトについては，短い時間だけ待つような場合も，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-hook を用意する．

```
(timeout-hook form interval hookfn)
= (event-hook form
    (make-eventbox :timeout interval)
    (op* (#:ignore #:ignore2) (_hookfn) ) )
```

form 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-hook 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-hook マクロを使うかぎり, :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-hook 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.

2. 複数プロセスの待ちあわせ

待合せをするプロセスが, :user タイプのイベントボックスを共有しているとする. また, 待合せるプロセスの数は既知とする (この場合 3). イベントボックスは次のように生成する. 待合せ場所に到達したプロセスの数を状態とする. 初期状態は 0 である.

```
(!box (make-eventbox :user '0))
```

待合せのところで, 各プロセスが

```
(rendezvous box 3)
```

を実行するようにする. 関数 rendezvous を次に示す.

```
(defun rendezvous (eb n)
  (event-hook
   (seq (without-process-switch
          (raise-event eb
            (+ 1 (event-source eb)) )
          (wait) )
    eb
    (op* (x v)
      (cond ((= n (event-source x))
             (exit rendezvous #t) )))))
```

event-source はイベントボックスのソースフィールドのアクセス関数である.

3 つのプロセスが順不同に 3 回実行するが, 3 回めでは, 3 つのプロセスすべてで割り込みが発生し, 同期して rendezvous を脱出する. 割り込みは, 自分自身でも発生することに注意してほしい.

7. おわりに

本稿では, TAO/SILENT のコンカレンシコントロールのためのプリミティブと, それらを組み合わせるイベント機構について述べた.

TAO/SILENT ではここで述べたもの以外にも, 実時間 GC や, メモリを陽に返却できるようなデータ構造を使って GC の負荷を軽減を図るなど, コンカレンシ機能をいろいろな面からサポートしている. 今後, ハードウエアのデバッグ, TAO の実装を進めていく予定である.

[文献]

[1] 吉田, 竹内, 山崎, 天海: 新しい記号処理カーネル SILENT の設計, 記号処理研究会, 56-1, 1990.

[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] 天海, 山崎, 竹内: 新 TAO のメッセージ伝達式, オブジェクト指向計算ワークショップ, 1991.

[4] 天海, 竹内, 吉田, 山崎: TAO/SILENT のソフトウェアアーキテクチャ, 日本ソフトウェア科学会第 8 回大会, pp.57-60, 1991.

[5] 山崎, 天海, 竹内, 吉田: TAO/SILENT の論理型プログラミング, 記号処理研究会, 64-1, 1992.

[6] 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.

[7] 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.

[8] 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.