ソフトウェアからみたマルチプロセッシングシステム

Multiprocessing Seen From Software Side

和田英一
Eiiti WADA
東京大学 工学部 計数工学科
Department of Mathematical Engineering, University of Tokyo

[1] はじめに

ブリンク・ハンセンの「オペレーティングシステムの原理」のなかに次のような一節がある.

「人間は, それぞれ独立の速度で同時に進行する複数個の活動体全体の動作を理解することに非常な困難さを感じるものである. 今回はアメリカの歴史, 次回はフランスの歴史というようにして民族の歴史を学んできた者にとって, ある国で急激な変動の起きた重要な時期に, 他の国で起きていた事件を記憶しておくことがどんなにむずかしいことであったか思い出すとよい(田中, 真弓, 有沢訳から)」

M.I.T.のヒューイットとこの点について次のような会話をしたことがある. ヒューイットから.

「ケネディとドゴールとは, どちらが先に死んだか」

「ケネディの方だと思う」

「どうして」

「ケネディの葬儀にドゴールが出席したのではなかったかしら」

「そうだ」

つまりこのような交流に基いて覚えておかなければ, 覚えられるものではないし, またもし交流がなければ, どちらが先に死んだかはあまり重要なことではない.

マルチプロセッシングのソフトウェアを書くのは, ちょっとやってみても, 交流とその効果にいろいろ気をつけなければならないので, 相当厄介であることがすぐにわかる. わり込み禁止やtest & setの命令がなければまず不可能だからこれはハードウェアの方でつけて貰うけれども, これらの機能はgoto命令のように基本的であり, これを組み合わせて, シングルプロセッシングよりむずかしいマルチプロセッシングのソフトウェアを作るのは, 至難なわざであるので, ソフトウェアの側としては, プログラム言語を設計するのと同じような発想で, マルチプロセッシングにむくような機能をいろいろと考えている. ここでは最近の, この方面の努力を, ひとつかふたつ, 紹介してみたい.

[2] 並行処理のごみ集め

まず, マルチプロセッシングの使用例をひとつ紹介する. これはダイクストラの考えたLispのごみ集めである. Lispではデータ構造を構成している単位がセルで, それがリンクで結ばれている. リンクをつなぎかえたりするので, 不要になったセルが, 段々とごみとして記憶装置の中にたまってゆく. 一方データ構造は新しいセルを次々と使って処理が進んでゆき, そのうち新しいセルがなくなると, ごみの回収がはじまり, 新しいセルを復活させる. 普通のLispでは, 同じ処理装置がリスト処理とごみ集めを担当し, ごみ集めの最中では当然リスト処理はおこなわれない.

しかしこの並行ごみ集めは, 処理装置をふたつ用意し, ひとつがリスト処理をやっているあいだ, もうひとつがごみ集めを行う, これは衝

で，一般の市民が生活しているあいだに，ごみ集めのトラックが走りまわっているのと同様に，市民は必要なものをごみとしないように注意しなければならない．

ダイクストラの方法では，各セルは何本かのリンクの出発点となるほか，白，灰，黒，緑のいずれかの色を覚えておくことができる．緑色はそれが新しいセルであることを示している．このLispの働作は交代するふたつの位相i)白→黒，ii)黒→白のくりかえしの中でおきる．いま白→黒の位相にいたとして，リスト処理の処理系は，リンクをかえるたびに

i) 既存のセルにリンクをのばすなら，その既存のセルを，少くとも灰色にする（つまり白なら灰色に，灰か黒ならそのまま）

ii) 新しいセルにリンクをのばすなら，その新しいセルを黒にする．

という塗り替えをする．一方ごみ集めの処理系は

すべてのルートのセルを少くとも灰色にする；
灰色のセルが存在するかぎり
　[すべてのセルについて，もしあるセルが灰色なら，そのセルからのリンクの指しているセルをすべて少くとも灰色にし，そのあるセルを黒にする]をくりかえす；
灰色のセルがなくなったら，白のままのセルはごみであるから，緑色にする；
リスト処理の処理系に合図して，黒→白の位相へ進む．

黒→白の位相では，上の規則の白と黒を交換する．

この働きをもう少し説明してみる．リスト処理の処理系が動いていないとすれば，ごみ処理の方は，ルートを灰色にし，次々と灰色のセルの先を灰色にし，灰色だったセルを黒にしてゆくから，ルートからつながっているセルはいずれすべて黒になる．次の位相では，ルートを灰色にし，次々とルートの側から白にしてゆく．これを永遠にくりかえしているのである．それに対してリスト処理は白→黒の位相では，必要なセルは黒にするのだから，新しいセルをとったら黒にする．もし白にすれば，新しいセルに手をのばした元のセルが，ごみ集めで必要の印がついた後なら（黒になっているなら），黒から先には必要の情報は伝播しないから，ひとりでに必要の印がつくことはない．もし黒になる前なら必要の印はつくが，新しいセルはリストになっているので，それを先に印をつけておかないと，新しいセルはすべて必要の印がつけられてしまう．そこで黒にしなければならない．既存のリストにのばすとき，少くとも灰色にするのは，つなげたというのは先が必要なセルであることなのだが，そのままならいつかは必要の印がついてくる筈だが，それがくる前に元の方で切られて，ひとりでに印がつかない恐れがあり，それを防ぐで，積極的に印つけを開始するためである（図1参照）．

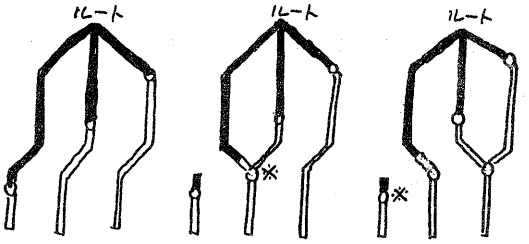

図1

図1 aのようにルートから始まって途中まで黒になっているデータ構造があるとする．丸印は灰色のセルを示す．bのようにリンクが切りかわると，切りかわった先(※印)を灰色にしここから黒化がすすむ．ルートから中央のリンクを進んできた印つけが期待できないのは，図cのように，この地点まで黒になる前にリンクがよそへ向いてしまう可能性があるからである．図cの※印のごみは，必要な印つけが進行する．つまり，ごみ集めが一部を必要とみてとったあと不要になったからで，今回はごみなのに回収されない．次の位相では黒のまま残っているのでごみとして回収される．

このように4色に塗り分けて考えるのはいかにもダイクストラ流である．この方式の特徴は同じセルに交対むきの色をつけようとしない点にある．白のごみを緑にした途端にリスト処理が黒にすることもあるが，緑はあまり本質的な働きをしていない．白→黒の位相では，黒にすることと，少くとも灰色にすることしかないから，困ることはない．困ることはしかし，どうしてこれを実現したらよいかわからないことである．白，灰，黒を2ビットの組み合わせであらわし，少くとも灰色は and もorの命令を実行すればよいともいわれているが，それにす

い分時間をとられそうな気もする.

2年ほど前にガイ・スティールの考えたマルチプロセッシングのごみ集め[1]は, 必要なセルを一ヶ所にまとめる機能までもったものである. これはpush命令や, ごみ集めの専用のスタックなどをもったやや複雑なシステムである. 実際に作ったのかどうかはわからないが, これを考えた当時, かれが学部の3年生であったことを考えると, 相当の腕まえといわざるを得ない. 並行処理のごみ集めの長所, 短所などもよく考察してあるので, リスト処理のごみ集めに関心をもつ方には一読をすすめたい.

[3] モニター

ここでいうモニターは, ひとつのようなもののモニターではない. マルチプロセッシングのプロセス間で共有されるデータの保全をはかり, プロセスの交通整理をするための, データと手続きの集まりである. このモニターの考えを採用して ひとつ(やその部分)を設計したひとりはブリンク・ハンセンであるが[2] かれはそれを実験するためにモニターの機能をもった Concurrent Pascal という言語とその処理系を製作し, その上になんつのプロセスの走るSoloという名のOSを試作した. このSoloは比較的容易に他の機種へ移植できるのが特徴的だがシングルユーザーのためのOSだから, てう面白いことはできない.

モニターなる構造を推進しているもうひとりはホアーで, かれのグループもPascalにモニターの機能をとり入れた言語Simoneとその処理系を製作して[3], いろいろな実験をやっているようである.

この両者のうち, ホアーのそれの方が簡単だと思われるので, 次にそのモニターについて, 少し説明をしてみる. ホアーによれば, モニターの構造はPascal風で, 次のようになっている(モニターという題の論文[4]とSimone[5]とでは形が若干異るので, 前者に従う.)

```
モニター名: monitor
  begin データの宣言;
    procedure 手続き名(…仮引数…);
      begin … 手続き本体 … end;
    …他の手続きの宣言;
    … データの初期値設定部分 …
  end
```

ひとつの例は次のようなものである.

```
singlresource : monitor
begin busy: Boolean;
      nonbusy: condition;
  procedure acquire;
    begin if busy then nonbusy.wait;
          busy:=true
    end;
  procedure release;
    begin busy:=false;
          nonbusy.signal
    end;
  busy:=false
end
```

最後の方の busy:=false がデータの初期値設定部分である. このはじめの方で宣言されたデータ busy は型がBooleanで, ここで宣言されたデータは(他にあったとしても, すべて)モニター内部の手続きでしか読まれたり書きかえられたりしない. またモニター内部の手続きは, モニターの外のデータを読んだり書きかえたりしてはならない. さてある種のリソースがひとつしかないとして, 沢山のプロセスがそれを使おうと狙っているとする. どのプロセスも, 使うまえに singlresource.acquire として第一の手続きを呼んで, リソースを確保する. すんだら singlresource.release とすぐに呼んで, そのリソースを手放す. あるプロセスがいずれかの手続きを実行中だと, 他のプロセスは, どちらの手続きと呼んでも待たされることになっている. A, B, このみっつのプロセスがこの順にほとんど同時にリソースを確保しにきたとしよう. Aは最初だから無事に入れて, busyをみるが, falseに初期値設定してあるので, nonbusy.wait は実行せず, 下の busy:=true に行って帰る. 帰るとAが出るのをまっていたBが acquire に入ってくる. 今度は busy=true 故 nonbusy.wait を実行する. nonbusy は condition というモニター専用の変数で, これに .wait や .signal をつけてモニターの手続き内で使うことになっている. wait を行うと, そのプロセスはこの条件の名のもとに待ち行列につなげられる. 外へは出ないが,

これによって C はモニター手続き acquire に入ることができるようになる。しかしこれも同じく nonbusy の待ち行列につなげられる。やがて A が release を呼んでくる。リソースが不要になったから busy を false にし、nonbusy.signal を実行する。もし条件 nonbusy に待ち行列がなければ、この実行は無効だが、待っているものがあれば、A はこの時点で停止させられ、待ち行列先頭の B が走りだす。B はさっそくリソースを得て外へ出、A もつづけて走ることができるようになり、外へ出る。

モニター手続きの出入り口、wait や signal でプロセスの停止進捗をどう制御するかは、P, V 命令を使ったプログラムの形でホアーの論文に記述されている。

条件.wait の実行には、整数値のパラメータをつけることもでき、そうすると整数値の小さいものほど先頭になるように行列につなげられる。また条件には .queue をつけることもできる。これは待ち行列に待っているものがあれば true, 空なら false の値をかえす。queue を利用した例として readers and writers の解を引用しておく。

readers and writers の問題は、沢山のプロセスがある場所にデータを書いたり、そこを読んだりするのだが、読むのは複数のプロセスで同時にできる。しかし書く方は読んでいるプロセスがあったり、他のプロセスが書いていたりしていると待たされる。書き込み中は読むのもまたされ、書き込みは読みだしに優先する。プロセスは読みたい、読み終り、書きたい、書き終りというときにモニターの手続きを呼ぶ。読みたい、書きたいは、手続きの中で wait にひっかかり、待たされることがある。

```
readers and writers : monitor
begin readercount : integer;
      busy : Boolean;
      Oktoread, Oktowrite : condition;
 procedure startread;
  begin if busy V Oktowrite.queue
              then Oktoread.wait;
        readercount := readercount + 1;
        Oktoread.signal
  end;
 procedure endread;
  begin readercount := readercount - 1;
    if readercount = 0 then Oktowrite.signal
  end;
 procedure startwrite;
  begin if readercount ≠ 0 V busy
              then Oktowrite.wait;
        busy := true
  end;
 procedure endwrite;
  begin busy := false;
    if Oktowrite.queue then Oktowrite.signal
                       else Oktoread.signal
  end;
  readercount := 0;
  busy := false
end
```

プログラムは簡単だから説明の要はあまりないと思う。読みたい、つまり startread なら次のように読む。あるプロセスが書いているか (= busy) 書きたいというプロセスが待っているか (Oktowrite.queue) なら Oktoread になるまで待つ。誰かが走らせてくれたか待つ必要がなければ下へ進んで readercount をふやす。つまりこれだけのプロセスが一斉に読んでいるわけである。そして他に Oktoread で待っているのがあれば起動する。

このモニターは、マルチプロセスの制御には比較的簡単でよいという気がするが、今後マイクロプロセッサーの多重処理には少しマクロすぎるようにも思う。マイクロプロセッサーに手頃な機能を探すのがこれからの課題であろう。

参考文献

[1] G.L. Steele Jr: "Multiprocessing Compactifying Garbage Collection", CACM, Vol. 18, No. 9 (Sept. 1975), pp. 495-508

[2] C.A.R. Hoare: "Monitors: An Operating System Structuring Concept," CACM, Vol. 17, No. 10 (Oct. 1974), pp. 549-557.

[3] W.H. Kaubish 他: "Quasi parallel Programming", Software-Practice and Experience, Vol. 6, No. 3 (July-Sept. 1976), pp. 341-356.