非同期処理について(II)*

丸　山　武**

4. タスク間の競合

4.1 lock と unlock

ecb の制御のところで，同一の ecb に関する WAIT と POST が仮に同時に動作したとしたら，どんなことがおこりうるかについて述べたことを思いおこしていただきたい．WAIT ルーチンは

($w1$) ecb の c ビットを調べてみて，

($w21$) $c=0$ なら $w=1$ に設定して待状態にはいり

($w22$) $c=1$ ならイベントはすでに発生ずみで待合せは不要だから，ただちにもどる．

POST ルーチンは

($p1$) ecb の w ビットを調べてみて，

($p21$) $w=1$ なら $c=1$ に設定して，さらに，WAIT 中のタスクをいねむりからたたき起こして作業を続行させる．

($p22$) $w=0$ なら $c=1$ に設定するだけで，ただちにもどる．

つまり，WAIT ルーチンは

$w1\rightarrow w21$ または $w1\rightarrow w22$

の順に処理し，POST ルーチンは

$p1\rightarrow p21$ または $p1\rightarrow p22$

の順に処理する．WAIT, POST 両ルーチンがまぜこぜに走ることをとくに禁止しなければ，時間的順序で，無限の組合せがありうる．つぎに示す6とおりの組合せは，$w1$ と $w21$, $w1$ と $w22$, $p1$ と $p21$, $p1$ と $p22$ の移り目でしかまぜあわせがおこらないとした場合のものであり，無限の可能性のうちのほんの一部にすぎない：

$w1\rightarrow w21\rightarrow p1\rightarrow p21$

$p1\rightarrow p22\rightarrow w1\rightarrow w22$

$w1\rightarrow p1\rightarrow w21\rightarrow p22$

$w1\rightarrow p1\rightarrow p22\rightarrow w21$

$p1\rightarrow w1\rightarrow p22\rightarrow w21$

$p1\rightarrow w1\rightarrow w21\rightarrow p22$

これらのうち，正しく動作するのは，はじめの2つだけであり，それ以外のものは，POST ずみなのに永遠に WAIT がとけないことになってしまう．

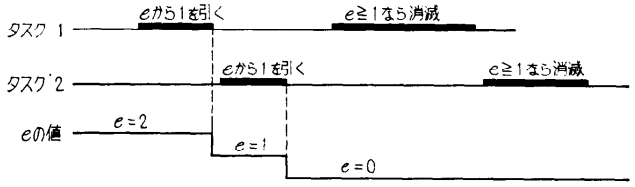
このことは，$w1\rightarrow w21$, $w1\rightarrow w22$, $p1\rightarrow p21$, $p1\rightarrow p22$ が，同じ ecb に関するものである限り，ひとつづきのものであって，途中に他の部分を介在させてはならないこと，1つを実行中に他の実行要求が発生しても，これを待たせるべきこと，つまり，シリアル (serial) にしか実行できないことを示している．

Conway 系においても同様で，JOIN(e) マクロについて

($j1$) e から1を引く

操作と，

($j2$) $e\geqq 1$ なら JOIN した制御の流れは消滅するにおける $e\geqq 1$ の判定とを分離するわけにはいかないのである．仮にそうでないとすれば，2つのタスク



タスク1,2とも $e=0$ と判定されるので，消滅するので生き残ってしまう．

第9図　誤った JOIN(e) の解釈

がほとんど同時に，第9図のような順序で JOIN(e) を実行することもおこらないことではない．この場合両タスクとも $e=0$ となるので制御の流れは2本とも生き残ることになり，1本に合流する（最後の1本だけを残す）という JOIN のねらいは実現できない．JOIN(e) と FORK(l, e, n) がほとんど同時に，異なるタスク間で実行された場合にも似たような矛盾を引きおこしてしまう．

もう1つの例を考えよう．一定の手続きの繰返しを通常のプログラムでは反復ループで処理しているが，これと同様に，同じ手続きを多重に同時に実行することが考えられる．FORTRAN における DO 文，ALGOL における繰返し文 (for statement) に対する parallel for の概念である．たとえば，配列 a[1 : 100] と b[1 : 100] の内積を計算するのに，仮に 100 個の演算機構が備わっているものとすれば，DO 文や

* A Guide to Asynchronous Processing (Part II), by Takesi Maruyama (FUJITSU LIMITED)

** 富士通(株)・ソフトウエア技術部

繰返し文を拡張して

$S:=0;$

for $i:=1$ step 1 until 100 do in parallel

$S:=S+a[i]\times b[i]$

というような文を設けるのがごく自然の発展だろう．ここに，do in parallel は，これに続く

$S:=S+a[i]\times b[i]$

を，制御変数 i のとりうる値すべて ($i=1,2,3,\cdots\cdots,100$)について，同時に（順序を問わず）並行に実行させてもよい，という気持を表現するものとする．ただし，$a[i]$ や $b[i]$ における i は，制御変数 i の現在値ではなくて

$S:=S+a[i]\times b[i]$

をタスクとして実行しはじめたときの i の値を表わすものとする．

さて，この場合，各タスクで変数 S に対して，$a[i]\times b[i]$ を加えるために，読み出し，書き込みの2回のアクセスが行なわれるが，この間ほかの i に関する S への加算操作を介在させてはならない．つまり，$a[i]\times b[i]$ の計算は，完全にパラレルに実行できるものであるが，S への加算(積算)はシリアルにしかできないのである．

以上述べた3つの例は，同一の対象に対して非同期的に施されるいくつかの処理手続きがあり，その1つが実施されているときには，他の処理手続きを施してはならない，という性格を持っている．このような処理手続きは，互いに他をインタロック (interlock) するもので，ときにクリティカル・セクション (critical section) とも呼ばれる．

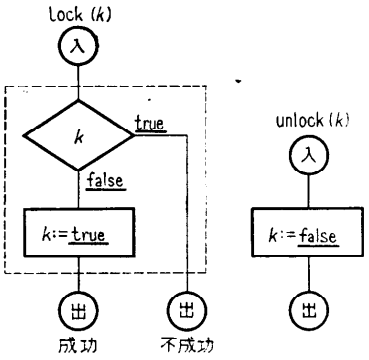
互いに他をインタロックするような手続き群は，そのうちの1つしか制御権を持てない．制御権は，関連するタスク群が競合する，いわば，一種の資材 (resource) であって，もし，制御権が他のインタロックする手続きに占有されていたら，返却されるまで待ち合わせなければならない．これを制御するためには，制御権のあき・ふさがりを示すインディケータを用意しておく必要がある．このインディケータをインタロック・キー (interlock key)，もしくは単にキーと呼ぶ．他をインタロックする手続きをはじめるときには，まず，このキーを調べてみる．キーがすでにかかっていれば，他のタスクが制御権を占有しているのだから，それがすむのを待ち合わせる．キーがはずれていることを確認したら，すばやく，キーをかけて，問題の処理手続きを実行する．実行が終わったら，キーをはずして，他のタスクの待合せを解除してやる．キー k をかける手続きを

lock(k)

逆にキー k をはずす手続きを

unlock(k)

と表わすことにしよう．



第10図　実行命令 lock と unlock

インタロック・キーは，ふつうのプログラム・スイッチとは少し違う．lock(k)操作は第10図のようにキー k の true, false を判定して，false なら true にセットするものであるが，例によって，k の判定とセットとは分離することができない．k に対する2回のアクセスの間に別の lock(k) 操作がそう入されては困るのである．多重 cpu 系では，実際に命令実行機構が多重にあるのだから，同じキー k に対して，2個以上の lock(k)操作をほとんど同時に行なおうとする事態もある確率でおこりうることである．それゆえ多重 cpu 系では，lock(k) が1個の実行命令で，しかもその間キー k (をふくむ) 主記憶モジュールに対する，他のアクセスを待たせるように作られている．もちろん，この命令の呼び名は，たとえば，

Philco* では Read & Clear
Burroughs D 825 では**
UNIVAC 1108 では Test & Set
IBM 360/67 では Test & Set
FACOM 230-60 では Load & Set

というぐあいに，計算機によって少しずつ違う．キー判定の結果の連絡方法なども違う．unlock(k) の方は，ビットをオフにセットしたり，語の内容を破算したりするための，ふつうの命令で代用している．

このように，lock, unlock 命令は多重 cpu 系では

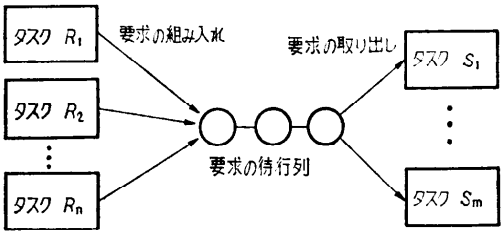
* 機種不詳
** 命令呼び名不詳

本質的に必要なものであるが，これをそのまま一般のプログラムで用いることには問題がある．まず，ロックしようとしたキーがすでにかかっていた場合，ダイナミック・ループしながらキーがはずされるのを待ち合わせるのでは，処理装置の使用効率上からも，主記憶に対する無効なアクセスを増すという点からも感心できない．いくつかのタスクが同じファイル上のレコードを更新するというような場合などでは，かけるべきキーは，ファイル・アクセスを含むかなりの長時間にわたってロックされるものである．キーがかかっていたら，処理装置をいったん別のタスクにあけわたすべきであろう．同様に，長時間キーをロックしているいる間，ずっと一切の割込みを禁止していたのでは，動的状況の変化にすばやく応答できないし，どだい，一般大衆プログラムに割込み禁止機能を持たせるのはむちゃであろう（たいていの機種では，割込み禁止機能は特権機能になっている）．さりとて，割込みを許したのでは，lock(k) と unlock(k) の間に他のタスクによって割り込まれて，このタスクが同じキーkをロックしようとすることも起こるだろう．すでに，キーはかかっているのだから，割込んだタスクの方はロックできず，むだにダイナミック・ループを繰り返す．一方，割り込まれた側は処理装置を奪われているから，このままでは，いつまでも unlock(k)に達しない．

というわけで，金物が用意した実行命令としての lock, unlock をそのまま用いるのではなく，管理システムが用意したつぎのようなマクロ命令 LOCK(k), UNLOCK(k) を用いることになる．一般に，キー・ロックの要求は2個以上のタスクから輻輳して発せられるから，要求の待行列をキーごとに形成する．LOCK(k) は要求をキー k の待行列に組み入れてその順番がまわってくるのを待たせる．また，UNLOCK(k) の方は，自分の順番がきて，その処理が終了したことを告げるものであり，キーkの待行列中で待ち合わせているつぎの要求の順番となる．マクロ命令としての LOCK, UNLOCK は，ロックする際，待行列が空きでなければ，いったん処理装置を手離して，他タスクにあけわたす点が，実行命令としての lock, unlock と違う．

4.2 待行列の制御

一時に1個の要求しかサービスできない部分処理系(窓口)に対して，2個以上のサービス要求が現われると，待行列を形成してサービスの順番を待ち合わせることになる．LOCK 要求はその例であった．



第11図 タスク間の要求待行列

単純なものは，2個のタスク間に形成される待行列であるが，一般には，同じ部分処理系に対して複数個のタスクからランダムにサービス要求が発せられる．さらに，サービスする側も複数個あって，各自，同じ待行列のなかから適宜要求を取り出してサービスするケースも存在しよう（第11図）．同等の機能を持つ cpu が複数個ある（対称型）計算機系におけるタスク・スケジュールはこの例である．

待行列の制御はきわめて基礎的なプログラム技法の1つである．しばしば出現する待行列のパターンとして，つぎの3種をあげることができる．

（1） 先着順 (first-in-first-out)

新着の要求は待行列の最後尾に組み入れられ，先着（先頭）のものから取り出されてサービスを受ける．

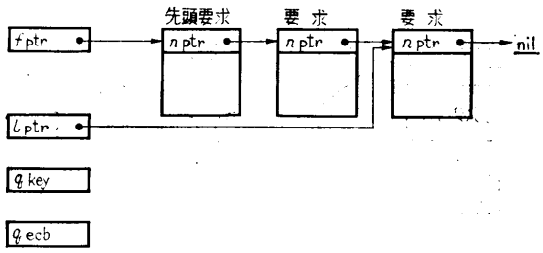
（2） 優先権順 (priority)

要求はそれぞれ優先権を持っていて，優先権の高い要求から先にサービスを受ける．同じ優先権を持つものの間では先着順である．

優先権順待行列は，通常，優先権ごとの先着順待行列からなる複合した待行列を形成する．新着の要求は，その優先権で決まる先着順待行列の最後尾に組み入れられる．また，空でない最高の優先権を持つ先着順待行列の先頭のものから取り出されてサービスを受ける．

（3） 順不同 (random)

先着順や優先権順と違って，要求のサービス順番が要求発生時点には決まらないような待行列である．扱い方はやや複雑になるが，多くの場合，システムの処理効率を高めるために用いられる．たとえば，磁気ディスクなどの大記憶に対するアクセス要求の処理手法の1つとして，全体としてアームの移動距離が最短となるような順に処理する方式がある．アームの進行方向で最短距離にあるアクセス要求を拾い出してサービスするのであ



第12図 先着順待行列の制御

る．このようにすると，要求のサービス順は動的条件で決まるのであって，要求の発生時点には決まらない．

最も基本的と思われる先着順待行列の扱い方を具体的に考えてみよう．新着要求は最後尾につけ，取り出すのは先頭要求だから，先頭，最後尾要求に対するポインタ（これをそれぞれ $f$ptr, $l$ptr とする）を用意する．要求の着順は，各要求のなかに，後続の要求に対するポインタ（これを $n$ptr とする）を記録しておくことで実現する（第12図）．新着要求の組み入れや先頭要求の取り出しは，これらのポインタを張りかえることを意味するが，例によって，組み入れ手続きと取り出し手続きとをまぜこぜにやったり，2個以上の新着要求を同時にいっぺんに組み入れようとしては正しい結果は得られない．そこで，インタロック・キー（これを $q$key とする）を採用しなければならない．また，サービスする側のタスクは，待行列があきになったら新しい要求の到着というイベント(これを $q$ecb で表わす）の発生を WAIT するものと考えて，新着要求組み入れの際，そのときまで待行列があきであった場合には，$q$ecb を POST することにする．結局，つぎのような手続きにまとめられる：

- 新着要求（ポインタ $c$ptr で示されるものとする）の組み入れ

```
nptr[cptr]:=nil;
disable interruption;
lock(qkey);
nptr[lptr]:=cptr;
lptr:=cptr;
if fptr=nil then
  begin fptr:=cptr;
        unlock(qkey);
        enable interruption;
        POST(qecb);
  end else
  begin
        unlock(qkey);
        enable interruption
  end
```

- 先頭要求の取り出し（ポインタ $q$ptr で示すようにする）

```
disable interruption;
lock(qkey);
qptr:=fptr;
fptr:=nptr[fptr];
if fptr=nil then lptr:=nil;
unlock(qkey);
enable interruption
```

ここに，

nil はさし示すものがないときのポインタの値．

disable interruption は割込み禁止状態に設定する手続き．

enable interruption は割込み禁止を解除する手続き．

を表わすものとする．その他については，ALGOL 60 の記法から類推されたい．

4.3 デッドロック

つぎのようなケースを考えてみる．2つのタスク t_A と t_B があり，それぞれ入力データを読み取ってファイル f または g のレコードを更新するときに，ファイル f のレコードの内容によっては，さらにファイル g のレコードにもアクセスしなければ，ファイル f のレコード更新が完成しないことがあり，逆に，ファイル g のレコードの内容によっては，さらにファイル f のレコードにもアクセスしなければ，ファイル g のレコード更新が完成しない，といったこともある．ファイル f，g は，タスク t_A, t_B が競合する共通資材であるから，例によって，アクセスするにあたっては，キーをかけあわなければならない．ファイル f，g のキーをそれぞれの kf，kg としよう．

1件の入力データについてファイル f，g の両方にアクセスしなければならないことはまれであるとすれば，いつも両方のキー kf，kg をかけるのはむだのようにみえる．まず入力データをみて，ファイル f にアクセスすることがわかったときにキー kf をかけ，ファイル f の所要レコードを読み取る．その内容によって，たまたま，さらにファイル g にもアクセスする必要があることが判明したとき，はじめてキ

ー kg もかけるという方が，効率がよさそうにみえる．だが，これはうまくいかない．つぎのようなことがおこりうるのだ．タスク t_A がキー kf に重ねて，キー kg をかけようとしたとき，すでにタスク t_B がキー kg をかけていたら，t_A は t_B がキー kg をはずのを待ち合わせなければならない．ところが t_B がファイル g のレコードを読み取った結果，さらにファイル f にもアクセスしなければならないことがわかったとすると，当然キー kf をもロックしようとするが，kf は t_A によってすでにロックずみであるから，今度は逆に t_B は t_A がキー kf をはずすのを待ち合わせなければならない．結局，t_A, t_B は互いに他のタスクがキー kf, kg をはずすのを待ち合わせるばかりで，永遠にキーははずれないことになる．ちょうど車1台しか通れないような小路の両側から車がはいってきて，どちらか一方が後退しない限り，にっちも，さっちもいかないといった事態と同じことである．このような状態をデッドロック (deadlock，すくみ) と呼ぶ．デッドロックは2つのタスク間に発生するばかりでなく，たとえば，タスク t_A が t_B を待ち t_B は t_C を待ち，t_C は t_A を待つというぐあいに，3つ以上のタスクの間で，一見それとわからない形で発生することもあるから油断がならない．

もう1つの例を考えてみる．多重ジョブ処理のシステムで，ジョブ j_A, j_B が，磁気テープ装置をそれぞれ3台，4台ずつ使用している．このシステムには全部で8台しかない．いま，j_A が新たに装置5台追加要求を発したものとすると，現時点でのあき装置1台に加えて，ジョブ j_B の装置4台が全部あきになるまで待ち合わせるしかない．ところが運悪く，j_B の方も新たな追加要求を発したとすると，事態は絶望的となる．j_A, j_B 間にすくみが発生したのである．j_A, j_B のどちらか一方を中断して，使用していた装置をあけわたさない限り，どうにもしようがない．同じことは主記憶の割当てについてもおこりうる．

デッドロックが発生したことを検出することは，しようと思えばできないことではない．しかし，問題はデッドロックを検出することにあるのではなくて，デッドロックを引きおこさないようなシステム設計を，速度効率的にも実用に耐えるような範囲で，実現することにある (速度効率を問題にしなければ，多重処理の必要はなく，したがってまた，デッドロックの発生するはずもない)．実際にデッドロックが発生する確率は低いことが多い．ほとんどは，幸いにも，杞憂に終わる．だから，もともとデッドロックの心配のない処理をも巻き添えにするようなことは極力避けたい．

デッドロック発生の可能性は，システム設計者が洗い出して，それぞれのケースに応じた対策を講ずるわけである．デッドロックを避ける方法は，一般解としては存在しないようだ．ただし，以下に述べるようないくつかの「十分」条件は知られている．十分条件としては，条件のゆるい方が適用範囲が広く一般性があるわけで，「十分すぎる」条件では，現実の問題に適用できるとは限らない．

- (一括要求)

  処理途上で必要になる資材を全部まとめて一度に要求する．全部がとれなかったときには，とれたものを返却して，あらためて一括要求を繰り返す．少しずつ動的に要求を発するからこそ，デッドロックは発生するので，いっぺんの要求ですませられるのなら，その必配はない．ただし，必要になるかもしれない，という可能性があるだけで，実際には不要のものまで要求してしまう点，ロスが大きいことがある．

  ジョブに対する入出力装置の割当てを，ジョブ単位に実行前に完全にすませてしまい，実行がはじまってからの動的な追加要求を認めない，という方式は，これであろう．そこまでいかなくても入出力装置の割当てをジョブ・ステップ単位に実行前にすませ，ジョブ・ステップ実行中の追加要求を認めない，というのはふつうの一括形ジョブ処理形態で採用されている方式である．

- (定順要求)

  資材に対する要求を，どのタスクも同じ一定順序で発するものと約束する方式である．ファイル f, g に関するキー kf, kg を，一方のタスクは $kf \rightarrow kg$ の順，他方のタスクは $kg \rightarrow kf$ の順にロックするというようなことはせず，どちらも定順，たとえば，$kf \rightarrow kg$ の順にロックするのである．

  せまい道路も一方通行にすれば「すくみ」は生じない道理である．

- 資材を途中で手離せるようなプログラム論理を組むことができれば，いざデッドロックが発生したときや，一定時間内に資材が確保できなかったときに，それまで握っていた資材をいったん手離して，あらためて一括要求を発する手がとれる．その際，もと握っていた資材がそのまま再びとれ

るとは限らないので，一般に，配置換え可能な資材に限定される．主記憶なども，ページ方式やベース・レジスタ方式の採用によって，配置換えできることが，この面から望ましい．

ジョブ・ステップ単位の入出力装置割当てを行なう際，磁気テープ装置など，確保できなかったら，逆に，先行するジョブ・ステップから受けついできた装置をも手離す，などの方式は，この例である．

- 必要以上のロックをかけない．たとえば，2つ以上のタスクからアクセスされるファイルは，レコードの更新をともなうアクセスであれば，他の更新だけでなく，一般に，読み取りだけのアクセスをもロックしなければならないのであるが，読み取りだけで更新をともなわないアクセスは，互いに他をロックする必要はなく，更新をともなうアクセスだけをロックすればよい．
- 共通資材を競合するタスク群を管理システムが把握していて，そのうちの2つ以上のタスクが同時に実行状態にならないように，タスク・スケジュールする．

4.4 プログラム・ミス

非同期系の動作論理は，通常のステップ・バイ・ステップの論理とやや趣が違って，健全な普通人には考えにくく，プログラム・ミスも犯しやすい．その例をまとめてみる．

- インタロック・キーの必要性を見落す．これは「すれ違い」か「衝突」事件を引きおこす．ただしすれ違いとは，相手タスクからの始動信号を正しく認識できず，しばしば休止（「ハングアップ」hungup）状態におちいること，衝突とは，予想しない干渉がおきて，プログラムが発狂状態になること，といった気持である．
- デッドロックの可能性を見落す．これは休止状態におちいる．
- わたすべき情報を完成しないうちに，相手側タスクを始動してしまう．もしくは，受入れ体制の整わないうちに，相手からの始動を受け入れてしまう．これは「追越し」を引きおこし，衝突の場合と同じように発狂状態におちいることが多い．

参考文献

1) Saltzer J. H.: MULTICS SYSTEM, 電子協 (1967), IV-1-87.
2) Havender J. W.: Avoiding deadlocks in multitasking systems. IBM Syst. J., 7, 2 (1968), 74-84.
3) IBM SYSTEM/360 Operating System. Supervisor and Data Management Services. C 28-6646 (1968), 32-36.
4) Dijkstra E. W.: The Structure of the "THE"-Multiprogramming System. Comm. ACM 11, 5 (May, 1968), 347-360.
5) Dennis J. B. and Van Horn E. C.: Programming Semantics for Multiprogrammed Computations. Comm. ACM 9, 3 (March, 1966), 143-155.
6) Conway M.: A Multiprocessor system design. AFIPS Conf. Proc. 24 (1963), 139-146.
7) Thompson R. N. and Wilkinson J. A.: The D825 Automatic Operating and Scheduling Program. AFIPS Conf. Proc. 23 (1963), 41-49.
8) Gill S.: Parallel Programming. Computer J., 1 (April, 1958), 2-10.
9) 富士通：FACOM 230-60 システムマクロ定義集，SP-045 (1968), 43-92.

（昭和44年12月4日受付）