Programming System の発展 (4)

戸島 潤

目　次

1. 初期の programming system
2. IT compiler
3. FŌRTRAN の出現
4. ALGŌL 小史
5. Data-Processing Compiler について
(以上,「商学討究」, 第20巻第3, 4号)
6. FŌRTRAN II と段階的処理
7. FŌRTRAN II Monitor System (FMS)
8. SHARE Operating System (SŌS)
9. IBSYS Operating System
10. 最近の Operating System の2, 3の概念
(以上,「商学討究」, 第22巻第1号)
11. Task について
12. Multiprogramming

11. Task について

10. でごく簡単に job と task の説明をしたが, 最近の ŌS の機能を理解するためには task という概念の明瞭な把握がどうしても必要なので, ŌS の種々の management 機能をのべるのに先立って, まず, ここでは task という概念をもっとあきらかにしておくことにしよう。最初に job の説明からはじめるのが適当である。job は内容的には, 計算機 system に処理させる論理的に独立な仕事の単位であると定義することができる。ここで, 論理的に独立ということが重要であって, このことは job を行なわせる user の側からみたとき, 個々の job が計算機 system によって独立に取り扱わ

れてもかまわないことをいみする。しかし，job はもっと形式的に考えることができる。job は何らかの媒体を通じて計算機 system に指定される。たとえば，入力媒体が card であれば，適当な job 制御文によって区切られた card deck がひとつの job を指定する。多くの ŌS でその役目をはたす job 制御文は job 文と job 終了文の 2 つである。すなわち，この 2 つの job 制御文によって区切られた card deck を ŌS はひとつの job を指定するものとみなすのである。job 文は job の開始を示すばかりでなく，job 全体の性格をきめるためにつぎのような情報を parameter として指定することができるのが普通である。

登録番号
利用者名
会計番号
会計用追加情報
合言葉
message level（どの程度の message を出力するかについての情報）
job 開始時刻
job 優先権
実行優先権
主記憶優先権
CPU 使用時間
主記憶占有時間
印刷 page 数，印刷行数，出力 card 枚数
主記憶占有語数
job 処理の実行継続に関する判定条件

なお，job 文であることを示すために JŌB という string が所定の場所におかれなければならない。たとえば，ŌS/360 では job 名のうしろに JŌB という string があって各種の parameter がそれにつづくものを job 文といっている。一方，job 終了文には通常 parameter がなく JEND という string

とか job 制御文の mark だけからなる。ŌS/360 の job 終了文は job 制御文であることを示す 2 つの slant だけからなっている。このように job 文から終了文までの card deck が形式的にはその内容にかかわらず job を指定するものとなる。身近な job の例として，つぎのようなものをあげることができる。FŌRTRAN 語でかかれた source program を FŌRTRAN compiler で object module に変換し，それを linkage editor で load module に結合編集し，最後にその load module を主記憶装置に load して実行する。この job を指定する card deck は job 文から始まり job 終了文で終るが，job 制御文として通常このほかにさらに実行文といわれるものが必要になる（他にも必要な job 制御文があるがここではふれない)。実行文はひとつの job をいくつかの処理の集まりからなるように区切った場合に，そのひとつの処理の始まりをあらわすと共に，そのときに使用される処理 program が何であるかを示す。実行文のそのほかの parameter にはつぎのようなものがある。

処理 program に対する parameter 情報
実行優先権
主記憶優先権
CPU 使用時間
主記憶占有時間
主記憶占有語数
会計用追加情報
実行と memory dump に関する判定条件

これらの parameter の指定は job 文の parameter の指定とは矛盾してはならない。指定がなければ job 文の指定が採用される。うえの job では compile, 結合編集，実行のそれぞれを指定するために 3 つの実行文が必要になる。一般に job はその論理的内容からさらに細分化されうる。これを job step という。そこで，job step の始まりは実行文によって指定されるということができる。job step は job の論理的内容からきまってくるばか

りでなく，用意されている処理 program にも依存しており，ひとつの job 中でどのような処理 program が，どのように動作するかということが，job がどんな job step にわけられるかをきめる大きな要因である。このいみで，job step は必ずしも固定したものでないということができる。また，ひとつの job を構成するいくつかの job step は job の論理的内容から何らかの順序関係をもっている。たとえば，compile は必ず実行に先立たなければならない。そこで，job は利用者の実行文の指定にしたがって，その順序通りに job step 単位で処理が行なわれる。job step では必ず何らかの program が動作するが，そのほかに，CPU，主記憶装置，2 次記憶装置，入出力装置，data set (file) などが必要とされる。これらのものがすべて必要とされるだけ確保されてはじめてひとつの job step の処理は実行されるのである。このように，job step の処理で利用される hardware, software 類は **10.** でのべたように資源といわれるが，もしも，計算機の制御の流れがひとつであれば，こうした資源の確保はそれほど問題にはならない。なぜならば，このときは全資源がひとつの job step の処理のみに利用されることになるからである。この場合，ひとつの job step で動作する処理 program がいくつあっても同じことである。ある program が他の program を引用しても引用された program はやはり全資源を利用することができる。このことは，たとえば，FŌRTRAN 語の program を考えてみれば容易にわかる。FŌRTRAN 語の program で subroutine を引用すると制御は subroutine に移ってしまい，subroutine の実行が終って制御が再び引用した program にもどるまでは引用した program のつぎの文は実行されない。これが計算機のごく普通の制御の流れであって，決して引用した program と subroutine が同期をとることなく勝手に動作するようなことはない。ところが，task という概念はこのような普通の制御の流れからだけでは理解することができないのである。まず，つぎの例をあげよう。いま，PL/I 語の program で

```
CALL SUBR EVENT (E);
```

とかけば，計算機は手続き SUBR をもとの program とは無関係に実行する。逆にいえば，SUBR の実行の完了をまたずにもとの program が実行される。これは FORTRAN にはない PL/I の新しい機能であるが，このように並行した複数個の制御の流れによる複数個の動作のことを非同期動作という。これはいくつかの制御の流れが本来的には他と同期をとることなく勝手に流れるところからつけられた名前であろう。なお，この場合，必ずしも異なった program が動作するとは限らず，program の構造によっては同一の program が非同期に動作することもありうるが，この点に関連した program の概念については後述する。また，非同期動作といっても，同時に複数個の処理が複数個の CPU で進行するのではなく，いくつかの理由によってひとつの処理の実行が中断されるので，その間に別の処理が進行するという形をとる。この点については **12.** でふれる。ところで，こうした非同期動作が行なわれるということになれば，もはや，全資源がひとつの処理のみに奉仕するというわけにはいかない。いくつかの処理が同期並行して進行するのであるから，当然，そこで資源をどのように，いくつかの処理の間に配分したらよいかという問題がおこってくる。いいかえれば，各処理は互いに資源を求めて競合する。この点に注目して task を定義するとつぎのようになる。task とは互いに資源を求めて競合し，かつ，他と非同期に実行される処理単位である。しかし，これだけでは task という概念は少しもあきらかではないので，さらに説明をつづけよう。task は計算機の制御の流れを分岐させるという形で新しい task を生成することができる。このとき，task を生成した task を親（主）task，生成された task を子 task という。親 task は子 task をいくつも生成しうるし，その子 task 自体が，さらに，子 task（親 task からみれば孫 task）を生成しうる。こうしてひとつの親 task から始まってつぎつぎに生成される task の全体を同一系列にある task（または，task 群）とよぶ。

job と task はつぎのような関係にある。job は，すでにのべたように，job step 単位で処理されるが，それは task として実行される。これを job

step task という。すなわち，job step は実際に処理が行なわれるときは，それがひとつの処理単位となり他の task と競合しつつ資源を要求して，job step task として実行される。しかし，このことはひとつの job step がただひとつの task を生成するということでないことに注意しなければならない。task は task を生成することができるから，ひとつの job step の中で並行処理が可能な部分をいくつかの task によって並行して実行することもできるのである。親 task としての job step task は job 制御文である実行文によって生成されると考えることができる。うえで実行文は job step の開始を指示するとのべたが，job step は必ず job step task として実行されるからこのように考えてもよい。子 task の生成のためには macro 命令が用意されていることが多い。たとえば，OS/360 では，ある場合に限られているけれども，ATTACH という macro 命令によって新しい task を生成することができる。以上は user の job から生成される task を問題にしたものであるが，こうした観点からは task とはあらゆる処理実行のひとつの単位とみなすことができる。すなわち，計算機 system に user がどんな処理を依頼してもすべて task という単位で実行されるのである。そのさい，具体的に task という単位を形成するものが，job step task であり，また，その task から生成される子 task であると理解することができるであろう。したがって，この場合，task の内容はまえもって不定であって，user の job の内容によってさまざまな task が生成されうることになる。

ところが，一般に task といったときには，このような user job の job step に関連する task ばかりではない。このほかに制御用の task が存在するのである。これを区別するために，以下では user の job step task とそれから生成される task を user task とよび，それ以外の task を control task とよぶことにしよう。user task は上述のようにあらかじめ処理内容がきめられていないのに反して control task はほぼ処理内容がきまっている。とえば，system 入出力を行なう task とか，job step を initiate させたり terminate させたりする task は control task の例である。こうした control

task は ŌS の始動時に生成されて，ŌS が動作しているあいだ中 system の中につくりつけになっており，必要に応じて起動されて種々の service を行なう。したがって，control task を特定の service を行なう処理単位とみなすことができる。user task は program の実行が終了すれば task として消滅するが，control task は ŌS が動作していれば決して消滅することはなく，system の停止によってはじめて消滅すると考えられる。さて，時間を限って計算機 system の中を眺めてみれば user task と control task が入り混って多数の task が存在して，それらは互いに非同期に動作しているが，完全に非同期に動いているわけではない。このことはつぎの例でわかる。いま，ある task が control task に service を依頼したとする。しかし，control task は必ずしも直ちにその service を果たせるとは限らない。なぜなら，多数の task が CPU による実行を要求して競合していれば，一定の rule に基づいて順次に task を実行するほかないからである。具体的には，実行優先権によって task の待ち行列をつくるなどの方法がとられることが多い。さらに，task の実行終了までには一般に時間経過を必要とする。そこで，もし，control task に service を要求した task はその service が終了しなければ先に進めない性質のものであれば，要求元の task は control task の service 終了を待たなければならない。すなわち，要求元 task は同期をとらざるをえない。このようにして，task は非同期に動作しているがゆえに同期をとる必要性がおこってくるのである。task の進行が何らかの事象の生起を条件としているとき，条件になっている事象を event とよんでいる。data の読み込みとその処理が別々の task であれば，data の読み込みが終了しなければ処理 task を実行することができないなどはこの例である。そこで，task の状態を 3 つに分類して考えることができる。まず，第 1 は実行状態 (EXECUTE) にある task である。これは現に CPU によって処理が行なわれている task である。ただし，その処理の中にその task が要求を出して行なわせている，task としては動作しない control program の処理（この点は後述する）を含めて考えることにする。第 2 に実行

待ち状態 (READY) にある task である。これはいつでも CPU によって処理が開始されてもかまわない task で，処理の順番がまわってくるのを待っている。うえでふれたように，多くの場合は task 実行の優先権と実行待ち状態になった順序にしたがってこれらの task は待ち行列をつくっている。第3は待ち状態 (WAIT) にある task で，これは event の発生をまっている task である。したがって，CPU があいてもこの状態の task は決して実行されない。実行待ち状態にある task は CPU さえあけば待ち行列の先頭から実行状態になる。あるいは，状況によっては逆に待ち状態になることもある。実行状態にある task は一定の rule にしたがって実行待ち状態になると共に event を待つために待ち状態に入ることもある。待ち状態にある task は event が発生すると実行待ち状態になる。task は必ずこの3種類の状態のいずれかに属しながらしかも状態をたえずかえつつ計算機の中に存在しているのである。

実行状態にある task の実行中断は大別して2通りの方法で行なわれる。ひとつは，うえでのべたように event の発生をまつために待ち状態になる場合である。このときは中断の原因が task の中にあることになる。もうひとつはまったく実行中の task とは無関係に実行中断がおこる場合である。これは後述する external interruption (外部割り込み) によっておこる。いずれの場合も task の実行中断時の CPU の register 類の状態を待避させなければならない。もし，中断時の CPU の状態が保存されていなければ，つぎにその task が実行状態になったときに正しく実行を再開できないことになってしまう。このために，task は task control block (TCB) といわれる制御表をもっている。TCB に格納される情報は大略つぎのようなものである。

実行待ちにあるつぎの task の TCB の所在

task の状態

task の優先権 (実行優先権)

task の種類 (user task か control task か)

register や timer の値 (退避領域)

task で使われている資源の情報

他の制御表の所在

task の実行が中断されると CPU の状態が TCB の退避領域に格納され，逆に task が実行状態になると TCB から中断時の CPU の状態が回復される。このような観点に立って task を定義づければつぎのようになる。task とは event の発生を待つために一時実行を中断したり，外部割り込みによってまったく非同期に実行が中断させられたりする処理単位で，このために task は実行中断後の再開にそなえて CPU の状態を退避させる TCB をもつ。そこで，ŌS の内部構造に即して簡単にいえば，task は TCB をもつ処理単位であるということができる。task をその属性からみたこのような定義がおそらく task の定義としてはもっとも適当なものであろう。ここで最初にあげた task の定義を振りかえってみれば，資源をもとめて競合し，かつ，非同期に実行される処理単位といういみが明瞭に理解されるであろう。そこで，task を複数個の計算機 system が必要に応じて communicate しあいながら独立に仕事を進めて行く状態をひとつの計算機 system で simulate するために考え出された概念であると考えることもできる。このいみで task のことを virtual machine ということがある。最近の ŌS はすべてこうした task という概念を基礎に構成されているが，必ずしも task という呼び方がつねに使われているわけではない。task というのは ŌS/360 の呼び方であって，MIT の MULTICS では process, THE では sequential process などといっているようである。そして，名前が違うのに応じて，その概念にも微妙な違いがあるが，うえで大ざっぱに task の性質として説明したことは，ほぼ共通しているように思われる。

ところで，計算機の中の処理は ŌS のもとでは，すべて task として行なわれるのかといえば，実は task でない処理も存在するのである。ここでは task であるかいなかをもっぱら TCB の存在の有無によって判定している。たとえば，ある ŌS では task の生成，消滅，同期などを制御する program

の実行は task として行なわれないようになっている。すなわち，これらの処理に対応する TCB は存在しないのである。したがって，これらの処理が行なわれているときは，external interruption も event 待ちもおこらないようになっている。もちろん，task ではない処理はすべて control program によるものであって，user の job の job step が task として実行されるということに例外はひとつもない。control program の実行がすべて task として行なわれるとは限らないということは，つぎの例によってあきらかであろう。実行状態の task が待ち状態の task になるときには，そのことを switch を用いて表示するが，その表示の前に event が発生しているかどうかを確認する。もし event が発生していたら直ちにもとの task にもどるが，event が発生していなければ switch を ON にして，その task を待ち状態にする。一方，event が発生すれば，その event の発生を待っている task があるかどうかを switch をみて調べ，もしあれば，待ち状態の task を実行待ち状態にするが，なければ，やはり event の発生を示す別の switch を ON にして終る。ここでもし割り込みが許されていればつぎのような困った事態がおこりかねない。いま，event 待ちの task に対して event が発生していないことを確認した瞬間に event 発生によって external interruption がおこったとすれば，まだ，event 発生を待っていることを表示する switch が ON になっていないので，event 発生を示す switch が ON にされるだけである。一方，この処理が終ったあと中断が再開された方の program は event が発生していないことを確認したあとの処理を続行する。すなわち，event 待ちを示す switch を ON にして task を待ち状態にする。こうして，この task は実際には event が発生しているにもかかわらず永久に待ち状態をつづけることになるのである。これは task 間の同期をとるための control program が task として動作するために起る矛盾であるから，これを防止するにはこの program を task として動作させないことにすればよい。このようにして，control program の中には task として動作してはならないものがある。しかし，control program の中で何が task として動作

し，何が task として動作しないかは必ずしも一意にきまっているわけではない。現状では ŌS を設計する者の判断がひとつの処理を task にするか，しないかの key をにぎっている場合が多いようである。もちろん，設計者は判断にあたって能率性などを考えるわけであるが，それにしても，control task が個々の ŌS の設計と緊密に結びついていることは否定しえないであろう。この段階になると task は概念上はともかく実際上はかなりあいまいになってくるように思われる。これは ŌS の評価にもつながる問題であるので簡単には論じられない。

以上でほぼ task という概念があきらかになったと考えられる。ここでは task 制御の algorithm についてはほとんどふれなかったが，それらについては task management として後述する。

12. Multiprogramming

10. で現在の ŌS のもっとも大きな特徴は multiprogramming であるとのべたが，これは具体的にいえば，複数個の task を並行的に実行することがその内容となる。このために multiprogramming はまた multitasking ともよばれている。それでは multiprogramming はどのようにして行なわれるのであろうか。ここでは task 間の制御の転移がどんな仕方でなされるのかを，control program の内部構造にも多少ふれながら素描してみることにしよう。まず，multiprogramming が可能であるためには hardware の方もいくつかの機能をそなえていなければならない。第 1 に入出力動作と CPU の処理動作が並行して行なわれるという機能がある。CPU は入出力命令の解読と実行を channel 制御装置にまかせてそれ自身はただちにつぎの命令に進む。ところが，これでは入出力動作がいつ終了したかを知ることができないので，第 2 に hardware は割り込みという機能をもっている。これは一定の原因によって，そのとき実行している program を一時中断して他のあらかじめ定められた場所に自動的に（すなわち，飛び越し命令なしに）制御を転移させるという機能である。そのさい，割り込み原因としては通常つぎのよ

うなものが考えられている。

(1) 機械の誤動作

(2) program の誤り

(3) control program が呼ばれたとき

(4) 演算結果の異常

(5) external interruption

(2)は使用してはならない命令を使用したときなどであるが，これはつぎのようなことである。最近の計算機は命令のうちのいくつかのもの（特権命令）の使用を禁止したり，特定の記憶域へのかき込みを禁止したりした状態で動作しうるようになっているものが多い。これを user mode とか slave mode などと呼び，禁止事項のない状態を master mode といっている。そこで，user mode のときに禁止事項を侵犯すると割り込みがおこるのである。(3)は control program に service を要求するため実行中の program が control program を呼ぶことである。(4)は overflow がおこったとか0で割算が行なわれたとかの場合である。(5)は入出力動作の終了，時間切れ，複数個の CPU がある場合には，他の CPU からの信号，入力電源異常などである。(1)から(4)までは現在実行中の program の中に原因があるので，とくに割り出しとか internal trap ということがある。これに対して(5)は実行中の program とは無関係に生起するので，これだけをさして割り込みということがある。このような（(1)～(5)の原因による）割り込みの機能を利用すれば，割り込みをおこした原因を解析してそれに対して適当な処理を行なうことができるので，後述するように control program はこの割り込み機能を中心に組み立てられている。multiprogramming を可能にする hardware の第3の機能は上述した記憶域の保護機能である。一般に OS の制御のもとでは主記憶装置の中には多数の独立した program が格納されるが，ひとつの task の実行中に他の task で使われる program が破壊されるようなことがあっては困るし，また，control program がこわされるようなことでもあれば計算機は暴走したり停止したりして，以後，計算の続行が不可能になってしまう。こ

れをさけるために一定の記憶域に user mode では，かき込みを禁止することができるようになっている。これらの3つの機能が multiprogramming を可能にしているのであるが，以下では(3)を原因とする割り込み機能が control program によってどのように利用されているかを概観することにしよう。

11. でのべたように，実行状態にある task は external interruption と event 発生をまつために待ち状態になることによって実行が一時中断される。後者はさらに2つにわけて考えられる。ひとつは実行中の task が自ら待ち状態に入る場合であり，他は実行中の task が control program に何らかの service を依頼した結果，強制的に待ち状態にさせられる場合である。自ら待ち状態に入る場合もそのための service を control program に依頼しなければならないから，結局，待ち状態に入ることは自発的か強制的かの区別はあっても control program に service を依頼することによっておこると考えてよい。そして，これは実は(3)を原因とする割り込みによって処理されるのである。したがって，task の実行中断は中断原因が task の中にあるか外にあるかの区別はあっても，すべて割り込みによっておこるように統一されていることになる。さて，(3)によって割り込みをおこす場合には，control program にどんな service を要求をするかということを明示しておく。control program は要求される service 毎にそれぞれの service program を用意していて，後述するように，要求にしたがって適当な service program を動作させるのである。実行中の task が待ち状態になるためには，実行再開にそなえて CPU の register 類の内容を TCB に退避させるばかりでなく，どんな event を待つかということを適当な制御表に用いて表示すると共に，task が待ち状態にあるということをあきらかにしなければならない。いいかえれば task が実行状態にあるという表示をやめなければならない（実際は実行待ち状態の task の待ち行列からその task をはずすという処理などがなされるようである)。これらのことはすべて専用の service program が行なう。

割り込みがおこると割り込み原因別にそれぞれの割り込み受付け program に制御が移るようになっているのが普通である。この段階で user mode は master mode にかわり，さらに，つぎに task に制御が移るまで external interruption を保留にしてしまう。なお，いくつかの割り込みが同時に発生したときに備えて割り込みにも優先権があたえられている。通常，external interruption は internal trap より優先権が低い。割り込み受付け program に制御が移ると TCB の退避領域に register などの内容を格納して，つぎに割り込み処理 program に制御を移す。この program も，また割り込み原因別につくられていて，たとえば，何らかの service が要求されている場合であると，適当な service program をえらんでそれに制御をわたすという処理を行なう。service program には task として動作するものとそうでないものがあるが，event 待ちを制御する service program は 11. でのべたようないな理由から task としては動作しない。service program の呼び出しには，あらかじめ control program の中につくりつけになっている service program の一覧表と呼び出し program が使われることもある。external interruption の割り込み処理 program はこれとは少し違うばかりでなく，入出力動作終了の場合とそれ以外の場合でも処理の仕方が異なるが，いずれも service program を呼び出して service させる点では同じである。ただし，入出力動作の終了は event の発生であるから，その event をまっている task があるかどうかを制御表を調べて確認し，もしあればその task を実行待ち状態にしなければならない。なければ event 発生の表示だけをする。したがって，そのような処理を行なう service program がよばれることになる。このようにして，ともかく割り込み処理 program が service program に service させたあと制御は task への復帰を行なわせる control program に移る。この control program ではどの task に制御が移るかがきめられる。通常，実行待ち状態にある task は実行状態の task があればそれを先頭に優先権と実行待ちになった順番にしたがって待ち行列をつくっているが，割り込みによって control program が処理を行なえばこの待ち行列は変化す

る可能性がある。task への復帰を行なわせる program はつねにこの行列の先頭の task に制御をわたす。したがって，もし待ち行列にまったく変化がなければ，割り込みによって中断されていた task の実行が再開されるが，もし待ち行列の先頭の task がかわっていれば，新しい task に制御が移る。そして，いずれの場合も TCB に格納してある register 類の内容が回復される。要するに，いつでも実行待ち状態の task の待ち行列の先頭の task がつぎに実行される task となるのである。もし実行待ち状態の task がひとつもなければ，特別につくりつけになっている割り込み待ちの idle task に制御がわたって，計算機が停止することを防ぐ。なお，task として動作する service program への要求の発生はその前にその program への要求が待ちになっていなければ event の発生となるから，この task は実行待ち状態の task の待ち行列の適当な場所におかれる。もし，service program への要求が複数個あればそれは到着順に待ち行列をつくる。したがって，task として動作する service program に service を依頼しても ただちに service がえられるとは限らない。これが service を依頼した task が時には強制的に待ち状態にさせられるひとつの理由であることは前述した。また，うえでのべた割り込み受付け，割り込み処理，task 復帰の各 control program はすべて task として動作しないことはいうまでもない (external interruption が保留される)。

以上が task の実行中断にともなう task 間の制御の転移の大体の様相である。（未完）

(1971. 8. 30)