オペレーティングシステムの一構成法*

高 橋 秀 俊** 亀 田 壽 夫**

Abstract

Some underlying concepts of operating systems are discussed. Simplification and further generalization of them are proposed. Interrupts (i. e. external interrupts, supervisor calls and traps) are hardware functions indispensable to operating systems. The handling of external interrupts and supervisor calls leads naturally to the notion of sequential process. The tree-like system hierarchy is introduced which changes the distinction between system programs and user programs from absolute to relative. New basic operations on processes are proposed which can be handled in the same way as traps. A generalized master contral program which incorporates these ideas is sketched. An experimental TSS constructed upon the MCP for the HITAC 5020 is roughly described. The above ideas have proved to be successful in the actual experiment. Finally, some experiences in using a procedure-oriented-language in the description of system programs are presented.

1. まえがき

電子計算機のバッチ処理，リアルタイム処理，時分割処理 (TSS) の各使用法を通じて，operating system (executive, control program, supervisor, monitor などとも呼ばれる) が，重要な役割を演じている．しかし，operating system のプログラムは，一般に複雑な機能を有するので，作成もメインテナンスも容易ではない．また，具体的なシステムそれぞれに応じて，外部仕様が様々に異なることが要求され，別々の operating system として作らなければならないことがさらに複雑さを増す．これを防ぐために，何らかの統一的取り扱いが要求される．

ここで，operating system のプログラムが，すべて，普遍的な，かつ単純ではっきりした概念で構成されていると，その概念を把握することにより，プログラムの作成や理解が容易になるだろう．そして，その概念の多くは，どのような operating system をも，その上に組み立てることができるような，一般的な基礎プログラムの存在と，密接な関連があるはずである．

問題は，そのような普遍的な概念を明らかにし，発展させることにある．しかし，その概念，基礎プログラムの形に束縛されて，かえってプログラム作成が複雑になったり，効率がひどく悪くなったりするようではいけない．正しい概念は，在来のシステムを構成するものから，あまりかけはなれたものであるはずがない (すなわち，経験的妥当性がなければならない)．

上のように構成概念をはっきりさせて，operating system を作ろうという一つの試みは，有限オートマトンの諸概念 (state diagram, transition matrix など) を用いて，operating system の機能を明確に表現しようとするものであるが[5]，在来の operating system について，一般的に用いられて経験的妥当性が示されているかどうかには疑問が残る．

また別の試みは，operating system を virtual machine の概念で構成しようとするものである．virtual machine, task,[15] process,[12] sequential process[1] などの概念は，異なったところもあるが，非常に類似した概念である．本論文もその方向に沿って検討し，概念の意義を明らかにして発展させ，operating system の構成法に対する，一つの考え方を提出しようとするものである．その経験的，かつ具体的な裏付けとして，実験的 TSS を作った (東大大型計算機センター

* A Conception of Operating System Construction, by Hidetosi Takahasi and Hisao Kameda (Fac. of Science, Univ of Tokyo)

** 東京大学・理学部

HITAC 5020 使用).

2. operating system の構成

昨今の operating system は，割り込みや周辺装置の制御を取り扱う一般的な基礎プログラム (master control program MCP) の上に作られるのが普通である．この基礎プログラムによって，multiprogramming が容易にできる．そして，virtual machine, task, sequential process, process などは，この multiprogramming や割り込み処理に関連した概念である．

2.1 既存の概念の検討とその意義の明確化

2.1.1 割り込みの処理

汎用計算機は，情報を逐次的に処理するので，"制御の流れ (flow of control)" の概念が基本的である．このことは flow diagram がよく使われることにも現われている．"割り込み" は，この制御の流れの変更の一種である[9]．

複雑なプログラムを構成するのに，これを比較的独立な幾つかの部分 (module) に分けて表現して，わかりやすくすることがよく行なわれるが，普通は，この分離は制御の流れに関して行なわれる．つまり，おのおのの module 内の流れが，module 間の流れから分離され，各 module 内の流れは，他の module 内の流れと独立になるように作られるのである． operating system も，これをわかりやすくするために，幾つかの独立な module から構成することが行なわれる．ここで，module 間の制御の転移としては，subroutine jump, supervisor call, 割り込みなどがある．

割り込みの場合には， jump の直後に， program counter, accumulator, index register などの割り込み前の中央処理装置の状態を定める値 (stateword と呼ばれる) が退避され，もとの module に再び制御が移される直前に，その stateword が回復される．それにより，もとの module は，あたかも途中に割り込みがなかったかの如く処理を続行する．

supervisor call の場合も，割り込みと同様である．subroutine jump の場合にも，jump 直後に，program counter, accumulator, index register などの値が退避され，return の直前に，それらの値が回復されて，もとの module に制御がもどる．それにより，もとの module は，あたかもそのような制御の転移がなかったかの如く処理を続行できる．

以上のように，上の3つでは類似した方法で module 内の流れが，お互いに独立な形となる．実際，多くの計算機には subroutine jump 命令として，jump の際に，もとの program counter の内容などを自動的に退避する命令があり，割り込みや supervisor call の際にも，同じ金物を利用して program counter の内容などを，自動的に退避するようになっている．

subroutine jump と割り込みなどとのこのような類似性から，多重レベル割り込みのハードウェア機能を備えたある種の機械では，割り込みを subroutine jump 命令の機能を利用して処理するようにしている[2),10)]．つまり，$n>m$ の場合に，また，その場合に限り，優先順位第 n レベルのプログラムが第 m レベルのプログラムに割り込まれ，その stateword が第 m レベルのプログラムによって，退避回復されるようになっている．

多重レベル割り込みを持たない機械では，どの原因による割り込みでも，常に同じ番地に jump するようになっているものが多い．この場合に，幾重にも起きる割り込みを処理するのには，subroutine jump との類似性から，last in-first out の push-down-stack を用いて簡単にしようとする方法が考えられる．しかし，この方法には，つぎのような問題点がある．割り込まれて stack に積まれた module の中で，もう実行させたくないものが生じた場合には，stack を scan して，その module に対する data を引き抜かなければならない．また，stack に積まれている module の幾つかよりも，緊急度が高い module の実行が要求されると，それを stack の中にさし込まなければならない．このようにして，push-down-stack の last in-first out の利点は失われる．つまり，recursive subroutine call などの場合には，module 間の前後関係という論理的関係が，stack 内の上下関係として表現されるわけであるが，割り込みの場合には，そのような論理関係にはあまり意味がないことが示されたわけである．割り込みの場合には，各 module に対する stateword は，むしろ，それぞれ独立に扱った方がよい．そして各 module の処理順を適当な queue で表現するのが適当である (stack で処理しても，結局，queue のように用いられることになる)．つまり，割り込みの際，割り込まれたプログラムが queue の適当な位置に入れられて，いつか順番がまわってきて，制御がもどされるというようになっていればよい．すなわち，この点において，subroutine jump と割り込みの違いが現われてくる．両者は同じように module 内の流れを，お互いに独立にする方式ではあるが，subroutine jump

の場合は，module 間の論理的結合度が高いのに対し割り込みの場合には，結合度ははるかに低いか，あるいは全くないという違いの結果である．

【前述の多重レベル割り込みの処理も，つぎの点で論理的にすっきりしていない．つまり，割り込まれた module の stateword が，これと論理的結合性が必ずしもないところの割り込んだ module によって，退避回復されるという点である】

各 module を独立に扱う割り込み処理方式は，まさに1台の計算機を幾つもの独立な仮想的計算機の集まりのようにみせ，各 module に各1台の仮想的計算機を与えるもので，また，multiprocessing にも直結する方式であり，virtual machine, task, sequential process などと呼ばれる概念をもたらす〔Multics の process[12] の概念には，time slicing や scheduling などの time sharing の機能や，segment mechanism による virtual address の機能まで含まれている点が少し違う〕.

割り込み処理方式を，さらに深く検討するために，割り込みなどの論理的結合度について，もう少し考えてみることが必要である．

2.1.2 制御の転移における論理的結合性

割り込みには，タイマー，周辺装置などからの外部的要因によるもの（external interrupt）と，記憶保護，特権命令違反などの内部的な原因によるもの（trap, 割り出し[14]）がある．external interrupt の場合の転移は，割り込まれたプログラムが引き起こすものでないから，全く論理的な意味を持たない（たとえば，計算中の program module と，磁気テープ装置からの割り込みで，制御が移された磁気テープ制御の module とは，その制御の転移の際に，何らの論理的結合性も見い出せない）．trap の場合には，割り込まれた module は trap の原因を引き起したので，program error 検出のために停止している必要がある．すなわち，trap は，あらかじめ予定されていないという違いはあっても，むしろ，subroutine jump の方により近い．一方，supervisor call の場合には，割り込まれた（つまり，supervisor call を出した）module の方は，要求を満足させるために，呼び出された module とは関係なく，処理を続けていってよい場合が多いので，両者の中間に位する（たとえば，get, put などの機能を考えてみるとよい）．

要約すると，module 間の制御の転移は，論理的結合度（および同期性の強さ）の順に並べると，つぎのようになる．

（1） subroutine jump
（2） trap
（3） supervisor call
（4） external interrupt

（2），（3），（4）はハードウェアで同様に取り扱われる．したがって，master control program にもこのことを反映させて，その中で単位となる module から，（2），（3），（4）の制御の転移を分離し，それらを MCP に一般的に扱わせることが考えられる．その結果，MCP によって，interrupt logic の取り扱いを，operating system の他の部分から切り離して，論理的意味のない制御の転移を，MCP にゆだねることによって，単純化をはかることができる．そして，割り込みや supervisor call は，module 間の control communication という operation に置き換えられる．

2.1.3 external interrupt と supervisor call に対する module (process) 間の control communication

external interrupt による制御の転移は，何の論理的意味をも持たないが，それを引き起こした外部装置と，その service routine との間には論理的結合がある．それは，外部装置からの supervisor call によって，service routine module が呼び出されたというように考えられる．

この意味で，external interrupt も supervisor call も，master control program によって，対応する service routine module を ready にする "wakeup" 信号に変換されるべきものである[12]．また，周辺装置を駆動するための supervisor call も，その装置を "wakeup" するための信号とみなせる．このように，external interrupt も，supervisor call も，device への request も wakeup という control communication の primitive と考えることができる．

要求された仕事を完了した module は休止して，つぎの要求がくるのを待つ．この状態は MCP に特別な要求 "block" を出すことによって得られ，この module が MCP 内の処理待ち行列からはずされる．

wakeup と block は[12]，概念的には，POST と WAIT に[15]それぞれ一致している〔block は，どの wakeup 信号でも解除されるが，WAIT はどこからきたかを見て選択するという違いがある〕.

上のように，external interrupt, supervisor call の扱い方は，在来の方式できれいに統一されるが，trap

の処理方式については，新しく検討しなければならない．それは**2.2.2**で行なう．

2.2 われわれの方式

一般的にいって，TSS にもバッチ処理にも適する基礎プログラム (MCP) には，つぎのような新しい要求が実現されていなければならないということを大前提として考える．

I) man-computer communication に必要なことができるようになっていること．

II) system program の中でも，変更・拡張が容易にできること，つまり，お互いに独立な部分は，独立性を保った表現ができ，干渉しあわないこと．

III) system program に許される機能と，user program に許される機能の違いを絶対的なものとせず，単なる相対的な違いとすること．つまり，両者の形式を同一にし，扱い方も統一することによって，単純化と一般化をはかること．

I) man-computer communication に必要なことは，つぎのことである．

i) 普通のプログラムでは，プログラムの方から入力要求を出さなければ入力ができず，計算機を主とし，入力装置を従とすることしかできないが，man-computer system においては，入力を人間が強制的に行なうことができる必要がある．それには，人間が一種の割り込み (external interrupt) をかけることが，常にできるようになっていなければならない．

ii) さらに，on-line でプログラムの debug をする場合には，Dynamic Debugging Technique (DDT) のようなプログラムを用いるわけであるが，debug されるプログラムによって，それが破壊されることが生じないようにしたい．また，user program で生じた記憶保護侵犯などによる割り出し (trap) を，user program 自身で処理できるようにもしたい．

i)，ii) は user program にも external interrupt や trap が扱えるようにしたい．つまり，モニタ動作ができるようにしたいということである．user program が supervisor part と user program part に分けられると，さらに，この user program part にも supervisor part を持たせたいことが生ずる．モニタ動作は玉ねぎの皮のように，幾重にもできることが望まれる．

II) が要求されるのは，一般に，設置される機械の記憶容量や周辺装置の種類によって，operating system として異なったものが望まれるし，使用者側の事情によって，その後の変更改善が度々要求されるからである．つまり，変更に柔軟に即応できることが望まれるのである．

III) の要求は，I)，II) と関連し，概念的に I)，II) をも満足させるために考え出されたものである．このことのより厳密な表現，および解決法は次節で検討する．

以下に，上のような要求を基礎にして考えられた一つの新しい方式について述べよう．

2.2.1 system hierarchy

この方式では，system program も user program も，幾つかの module (process) から成り立つ．これらの module は MCP により，同じ単位として同等に扱われる．しかし，システム全体が安全に活動を維持するためには，system module と user module の間に力の差がなければならない．つまり，user module は system module を破壊させたり混乱させることができないようになっていなければならない．しかし，前節 III) で要求されるように，system program と user program の絶対的区別は望ましくない．絶対的区別をせず，相対的区別をするとは，つぎのようなことである．

ある module (たとえば Y) が，他の module (たとえば Z) を監視できると同時に，Y が他の module (たとえば X) から監視される [これは I) の多重モニタの考えを満足させる]．この場合 Y は Z よりも system 的であるが，X よりも user program 的である．そして，Z が支配できる記憶領域は，すべて Y も支配できるが，Y が支配できる領域で，Z が支配できない領域があり得，また，X は Y, Z の支配できる領域を，すべて支配できるということになる．

module 間に“全順序関係”を持ち込むことによって，上の要求は満たされる．つまり，すべての module 間に，より system 的か，より user program 的かの順序関係が，定義されるようにする．その結果，すべての module に，力の順序を示す番号をつけることができる．つまり，この関係は “level” 概念の導入につながる．同じ level にある module は，同じ力を付与される．より高い level の module は，より低い level の module すべてを制御できる．ある level の module は，より高い level の module すべてから制御できるが，自分はそれらのどれも制御できない．全順序関係，level の概念は，Multics の protection ring[4] にも含まれているものである．

しかし，この全順序関係では，前節II）の要求は満足されない，すなわち，独立性を保つために，ある2つの独立な module 間には，制御したりされたりする関係が全くないようにしたい．つまり，お互いに全く干渉できないようにしたいという要求が満足されない．必ずしもすべての module 間に，順序関係が定義されないような順序関係は“半順序関係”である，したがって，system hierarchy には半順序関係の方がより適している．

さらに，II）はもう一つの要求をする．頻繁に変更したり改善したりする module は，システム全体の安全性のためには，できるだけ順序の下の方に置く必要がある．そして，お互い同士できるだけ独立にして，プログラム誤りがからみあって，複雑になることが避けられるようにしたい．このような要求は，tree 構造で表現される単純な順序関係によって満たされる．この tree 構造の system hierarchy は，level の概念に代わるものである．つまり，tree の中のどの深さ (level) にあるかという制約の強い区別の代わりに，より深いか浅いかという制約の弱い区別だけを問題にすればよい．この system hierarchy 内の順序と処理の優先順位（本来全順序である）とは，一応独立にできる．

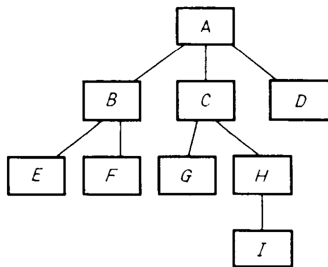

Fig. 1 An example of the tree-like system hierarchies. Each box represents a module (process). Alphabetical characters denote the names of modules.

Fig. 1 に system hierarchy の例を図示した．module (process) A は，B, C, D, E, F, G, H, I を支配でき，C は G, H, I を支配できる．F と G は互いに独立である．また，A の支配する記憶領域には，B, C, D の支配する記憶領域が完全に含まれるが，たとえば，B の領域と D の領域は，全く独立にできる，等々．こうして H は I のモニタに，C は H の，A は C のモニタになりうる．

2.2.2 trap に対する処理

trap（とくに，記憶保護，特権命令違反など）は，普通のシステムでは supervisor への制御の転移を引き起こす．このことは，われわれのシステムでは，当の module の直上の module（たとえば，Fig. 1 で E から B, H から C など）が，自動的に wakeup されることに対応する．しかし，trap の原因を起こした module の状態は，誤り検査のために保存される必要があり，そのため，その module は凍結 (freeze) され，検査が済むまでは，他のどの module (process) からの wakeup 信号によっても，活動を開始してはならない．一方，後に凍結を解かれた (unfreeze) ときに，凍結中にきた wakeup 信号の効果が，失われていてはいけない．したがって，module (process) の状態として，ready/blocked の区別のほかに，frozen/unfrozen の区別が要求される．

もし，trap の原因を生じた module が，その下に別の module を幾つか支配している場合には，trap の原因が1つの module にあると考えるよりも，その module 集団全体の中に，誤りがあったとみなすべきであり，その集団全体の状態が保存されることが望ましい．そこで，freeze は module 1つでなくて，module group に対する作用とする．そして，各 module の ready/blocked の状態は，再開のときのために保存される（blocked module に，freeze 中に，wakeup 信号がくると ready に移るが，frozen の状態を続け，実際に活動することはしない）．たとえば，Fig. 1 で C が trap を引き起こすと，C, G, H, I が freeze される．

freeze, unfreeze の操作は，trap が生じない場合でも，debug などに有用であるので，module (process) 間の control communication に加える（後述するように，TSS にも有用である——**2.4.1**）．当然のことであるが，freeze, unfreeze は下から上に作用できるべきではない．

trap は external interrupt や supervisor call よりも（同期性において），subroutine jump に近いので，trap だけを分離して conditional subroutine jump として扱う処理法もありうる[12]．しかし，われわれの場合に，trap も supervisor call などと同様に扱う方がよい理由は

i）system はできるだけ多くの module に分けるのが望ましいこと（したがって，1 module に対応する仮想的計算機に割り込みの機能をもたせることによって，1 module 内での割り込みを許し

て，その結果 1 module 内で，幾つかの独立な処理の流れが扱われるようにするよりも，割り込みに関係する幾つかの独立な処理の流れは，おのおの別の module にして扱う方がよい）．

ii）割り込みは皆 hardware で同様に扱われること．

iii）trap を分離する方式は，とくに segment mechanism がある場合には単純であるが，ない場合には複雑になること．

iv）trap は誤りのために生ずるものであり，頻繁に生ずるものでないので，簡単な処理法を特別に用意するほどの必要はないこと．

iv）trap によって自動的に作用される freeze 操作は，control communication としても有用なものであり，こうする方が，システムをより単純化すること．

などである．

なお，trap ばかりでなく， external interrupt も user に与える 1 process 内でできるようにして，**2.2** の要求 I）(多重モニタ動作) を満足させようとするシステムもある[3]．そのようなシステムでは，user process に対応する仮想的計算機には割り込みの機能があり，割り込みの禁止される master mode と，割り込みの禁止されない slave mode の区別がある．しかし，system process に与えられる仮想的計算機には，割り込みの機能も mode 区別もない．

これには，上の i），ii）の理由および

・ process の概念が，本来 external interrupt を処理しやすくするためのソフトウェア的概念であること．

・ process に種別があることは，統一性を少し欠く．

・ system process と user process との 2 つにはっきり分けることは絶対的区別であり，相対的区別でない．

などの理由で，統一化・単純化の傾向に反するが，ハードウェアの用い方によっては，この方が有効であるかも知れない．それには，さらに具体的検討が必要である．

なお，ここで述べた freeze, unfreeze は，おのおの Lampson の suspend, release[8] と類似しているが，後者はその中に system hierarchy の概念が含まれていないこと，単一 process に作用するのみで，process group に作用するものでないことなどにおいて，前者と異なっている．

2.2.3 system hierarchy と記憶保護

前述のように， system hierarchy の上位にある module の支配可能な記憶領域は， 下位にある支配可能領域を完全に含んでいる．たとえば，**Fig. 2** で A は B, C を含み，C は D, E を含む．しかし，A が

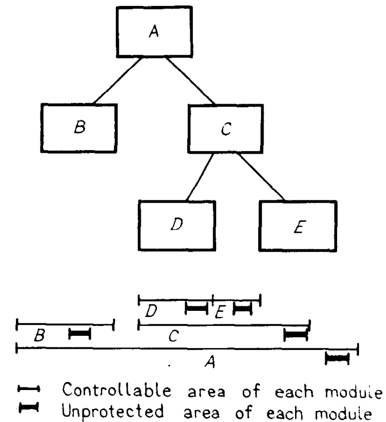

Fig. 2 The relation between the system hierarchy and memory protection.

通常用いる領域は，むしろ B, C を除いた A 固有の領域である．したがって，A が誤って B, C を書き替えたり，参照したりすることを防ぎたい．

このためには，支配可能領域と，実際に記憶保護をはずしてある領域とを別なものとし，前者は広くしておき，後者をできる限り狭くするようにすればよい．そして，後者は前者の範囲内で自由に変更できるようにする (特別な supervisor call によって)．すなわち，controllable area の概念と，unprotected area の概念を分離する．この unprotected area を狭くし，かつ自由に変更する方法は，system program の debug にも user program の debuq にも有用である．

【IBM 360 の記憶保護システム[6]は，われわれの目的には粗雑過ぎる．記憶領域全体が 15 のお互いに独立な比較的固定した領域に分けられるので，system hierarchy との関係が付けにくい．実際 user に与えられる task の概念には，上述のような system hierarchy の概念がなく，皆同じ力を持つ】．

上述のことは，読み書き実行の各保護について，それぞれ別に行なうようにすればよい．controllable area 内の unprotected area の変更は，ハードウェアで行なっても簡単であるならば，その方が overhead が少なくなり，より自由に使えるようになるであろう．また，Multics のように，segment mechanism がある

場合には，file が segment として主記憶装置と同一レベルで扱われるので，file の tree 構造に system hierarchy を埋蔵させることが，自然に考えられる[7].

2.3 具体的なシステムの説明

ここでは，以上の考察を実現する具体的な構成法をわれわれが実際に作成したものの内容に即して記述する．ハードウェアとしては，つぎの性格のものを想定する．

- master/slave の mode 区別がある．
- 記憶保護は境界をできるだけ細かく指定することができ，かつ全体の記憶保護の様相が簡単に変更できるもの．たとえば，上界・下界のアドレスを指定するレジスタを設けるものなどでもよい．
- paging および segment mechanism は，とくに考えない（relocation register 方式には容易に拡張できる）．

2.3.1 master control program の機能

ここで述べる master control program (MCP) はバッチ，リアルタイム，TSS に通用する一般的なものである．この MCP により user program も system program も同じように表現・管理され，相対的にのみ異なるが，絶対的差異のない能力が付与される．

この MCP によって，主記憶装置を共用する virtual machine が，何台ももたらされたようになり，独立したプログラム module（たとえば，各入出力装置管理プログラム，user program など）は，それぞれ各1台の virtual machine によって実行される形となる．

各 module (process) は，つぎのような MCP call を出せる（以下で wakeup, block, change protect 以外は，当の module よりも下にある module にのみ作用できる）．

- wakeup (*n*)　*n* で指定される module を ready にする．*n* が周辺装置ならば，それに対して要求を出す．
- block　自分自身の活動を止め，制御を他の module に渡す要求．process が ready 中に wakeup 信号を受けると，そのことが MCP で記憶され（wakeup waiting switch[12]），block 要求を出しても，その switch が reset されるのみで，blocked にならずに，ready のまま活動を続行する．Lampson によれば，この場合 wakeup waiting switch が 1 bit でよい[8]（つまり，ready 中に wakeup 信号が幾つきても，1つだけ記憶しておけばよい）が，このことはわれわれのシステムにおいても採用したが，まだそれによる不都合は生じていない．
- change protect (*a*)　自分自身の支配可能領域の範囲内で，記憶保護の状態を *a* で指定される内容に変更する要求．
- change stateword (*n*, *a*)　*n* で指定される自分より下の module の stateword を，*a* で指定される内容に変更する要求．変更できる内容は program counter, accumulator register, index register, hardware switch のうち，プログロムで使えるもの（overflow などの mask を含む），protection register, 支配可能領域などである．支配可能領域はこの要求を出した module のものをはみ出てはいけないし，要求された unprotected area が要求された支配可能領域をはみ出てもいけない．こうして，system hierarchy の秩序が保たれる．なお，module が自分自身で protection を変更できるようには，overflow などの mask も変更できようにしなかったのは，後者の変更が前者に比べて，はるかにまれであるので，overflow などの mask を直上の module に要求を出して変えてもらっても，大して overhead にひびかないからである．
- freeze (*n*)　*n* で指定される自分以下の module, およびそれに支配されるすべての module の活動を凍結するための要求．たとえば，Fig. 1 で *A* が *C* を freeze すると，*C*, *G*, *H*, *I* すべてが即座に凍結され，その中のどの module にも制御が移されなくなる．
- unfreeze (*n*)　*n* で指定される自分以下の module, およびそれが支配する module すべての活動が，freeze 要求，あるいは trap によって凍結されている場合に凍結を解除し，もとのまま活動を再開させる要求．

freeze, unfreeze は，もとの状態を保存し，回復させる点に特徴があり，frozen/unfrozen の状態も保存回復する．たとえば，Fig. 1 で，もし，*A* によって *C* 以下が freeze される前に，別の原因で *H* 以下が，すでに freeze されていた場合には，*A* が *C* 以下を unfreeze しても，*H* と *F* は freeze されたままである．

また，freeze の状態では，外からくる信号も無視せず記録しておき，unfreeze されたときに，その効果が現われるようにする．たとえば，*C* 以下

が freeze されていた間に，A から C に wakeup 信号がくると，C が blocked のときには C が ready にされ，ready のときは wakeup waiting switch が on になされる.

- create (n, m)　自分の下に新しく module を作り，支配下に置く要求. ATTACH (OS/360), create process (Multics) に相当する．しかし，われわれの場合には，system hierarchy の下の方の module が，勝手に沢山の module を create することを防ぐために，各 module ごとに create できる module の最大数を付与し，その module の直下にあって，直接に支配される module に与えられた数の和が，その module に与えられた数を越えられないようにした. m は create される module に与えられる create できる最大 module 数である．もし，m が許される数を越えた場合には，m を許される最大の数に縮めて create が行なわれる.
- delete (n)　自分の支配下にある module で，不要になったものを消す要求. 消すべき module の下にある module もすべて一緒に消される．つまり，その module に導びかれていたグループの活動全体が消滅させられるのである.
- change priority (n)　n で指定される module group の処理の優先順位を変える要求. その group 内部の順位は，group 内部で変更される.
- trap の処理　ある module の中でプログラム実行中に，記憶保護，特権命令違反，overflow などの誤りが生じたとき，それはその module が導びくグループの誤りとみなされ，グループ全体の活動が凍結され，その module の直上の module が wakeup される．この凍結は freeze による凍結と全く同じで，unfreeze によって解除される.

wakeup, change stateword, freeze, unfreeze, trap 処理などは，人間が普通の計算機ハードウェアを制御するために，制御卓からなしうる諸操作に対応することに注意されたい．たとえば，wakeup は restart, freeze は halt, unfreeze は halt をはずすこと, change stateword は諸レジスタを変更することに相当する.

なお，われわれのシステムでは，独立性の表現（独立なものが独立な形に表現できること）と単純さのために，wakeup の operation は直上直下の module に対してのみ作用でき，change stateword, freeze, unfreeze, create, delete, change priority は，直下の module に対してのみ作用できるとした．たとえば，Fig. 1 で C は A, G, H のみ wakeup でき，G, H に対してのみ freeze などの operation が作用できる. このことにより，たとえば，Fig. 1 で E として B に接続されている入出力機器が，F や G や H などから勝手に駆動されることも防げるし，H が F を誤って wakeup するような誤りも防げる．また，change stateword などを C が I に対して作用させたい場合には，C が H を制御し H に代行させることができる．上の制限はきびし過ぎるかも知れないが，system に十分の力が付与されていることには変わりがない.

2.3.2 master control program の構造

上述の機能を実現する master control program は，pure procedure であるプログラム部と，これによって書き替えられる table 部とから成る．table 部には，各 module に対して，1つの data block があり，wakeup などの control communication はすべて，MCP によって, 各 module に対応する data block を書き替えることとして実現される．MCP は各 module に対し，各1台の仮想的 CPU が与えられているようにみせるが，ready 状態を続けることの多い module (たとえば，user program) に，現実の CPU を占有されてしまって，時々 ready になる module (system の module が多い) に，CPU の制御が移らなくなってしまわないようにするため，module 間には処理の優先度が設けられ，MCP の処理待ち行列内の順序として，それが表現される．MCP は常に，ready 状態にある module のうちで, 最も優先順位の高い module が実行されるようにする (すなわち，master mode から slave mode に帰るときに，最優先の module に制御を渡す)．各 module に対応する data block は，つぎの情報を含む.

- frozen/unfrozen の状態 (queue の link).
- ready/blocked の状態，および wakeup waiting switch.
- stateword (program counter, accumulator register, index registers, protection register, controllable area, mask register, hardware switches).
- create できる process の最大数.
- system hierarchy を表現する list. (直上の module の data block をさす pointer と，直下の modules の data blocks をさす pointers).

プログラムの大きさは，MCP のプログラム部の語数が約 1K 語であり，table 部は必要となる module 数に応じて大きさが定まる（1 module に対し約 40 語必要）.

wakeup, block などの control communication のアルゴリズムの大体の複雑さをプログラムの語数で示すと

- wakeup, block, change protect　約 20 語
- freeze, unfreeze　約 30 語
- change stateword　約 50 語
- create, delete, change priority 約 100 語以上

で，頻繁に使われるものほど単純になっている．そして，create などは処理時間もメモリも多く要するのにもかかわらず，使用頻度は他に比較してずっと低いので，特殊目的のリアルタイム処理の MCP などの場合には，はずしてもよい．

上述のような割り込み処理は，multiprocessing に容易に拡張できる性格を有するが，実験はそこまでできなかった．しかし，上述の方式は，少なくとも multiprogramming system の記述を十分単純化し，かつ一般化するものとして，有効と考えられる．

2.3.3 module (process) 間の通信方式

module 間の通信は，wakeup を用いて行なわれるが，どの module 間の通信においても，wakeup された module が，どの module から wakeup されたかがわからなければ，情報の受け渡しができない．そこで，通信をする module の対ごとに，flag を 1 つ用意する（この flag の存在および場所は，関係する module 同士の間でのみ了解があればよい．したがって，MCP や他の module は，そのことについて全く関与しなくてよい）．そして，各 module は呼ばれる module に対応する flag を set し，その呼ばれる module を wakeup する．wakeup された module は，各 module に対応する flag が set されているかどうかはじから順に調べ，set されていれば reset し処理する．処理し終わって，もう一度すべての flag を調べ，再び set されていれば処理し，set されていなければ block する．これは data channel と CPU の間の通信に，一般に使われている logic の応用である．

2 つの module 間の相互通信には，両方向に対し，それぞれ 1 つずつ計 2 つの flag を用いる．それらの flag はいずれも，set する方からは reset せず，reset する方からは set しないものとする．

2.4 応用例——TSS の operating system

上述の MCP に，slave mode で動く幾つかの module を加えて作ったわれわれの TSS の operating system を略述する．

2.4.1 TSS の operating system の構造と機能

その system hierarchy は **Fig. 3** のとおりである．

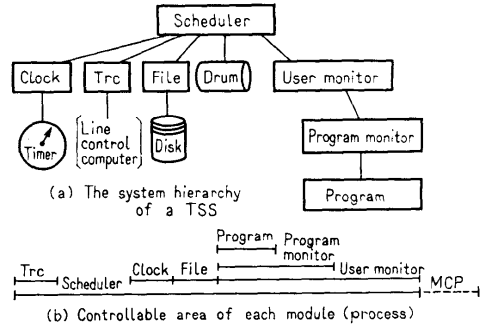

Fig. 3 The structure of an experimental time-scharing operating system.

実験的なものであり，構成も簡単になっている．全体は，全 user に共通な部分と，各user ごとに設けられた roll in, roll out される部分（Fig. 3 で user monitor 以下）とに分かれる．全 user に共通な部分は，各 user に対して Fig. 4 のような system hierarchy による virtual operating system が与えられたようにみせる．

user monitor 以下の部分は，user ごとに種々の system を採用することができる．しかし，標準的な user monitor は，command language interpreter を含み，バッチシステムの場合の Job monitor によって行なわれる機能に類似した機能を遂行するものである．

以下では，wakeup などの，MCP によって扱われる module 間の control communication が，具体的にどう使われたかを示す意味で，全 user に共通な部分の各 module の動きを述べる．

- clock　timer 制御の module である．それは hardware timer から wakeup され，それが 1 user program に与えられた time quantum の終わりであれば，scheduler を wakeup し，roll in, roll out を行なわせるなどの働きをする．
- trc　回線制御用計算機[13]との通信を媒介するための module である．trc が回線制御用計算機から wakeup されると，それがどの端末からの割り込みか調べ，対応する端末番号の旗を set して scheduler に知らせるが，これを wakeup はしな

い．なぜならば，set された旗に対応する user program が読み込まれることは，現在の user program の time quantum の終わりがきて，scheduler が wakeup される前に起こり得ないはずだからである．同じことは clock についてもいえる．つまり，user program から要求される絶対時間目ざましは，精度が time quantum 以上には上がらない（つまり，目ざましのベルが鳴っても，その効果は当の user program に処理の順番がくるまで待たされる）ので，同じように scheduler に知らせる旗は立てるが，wakeup はする必要がない．

- scheduler　おもに user program の roll in, roll out を扱う．つまり，これにより各 user に，各1台の計算機（幾つかの module から構成される）と，memory space が与えられる．また，端末機器や timer やファイル装置などと，ドラムに滞在することの多い user program との媒介をし，各 user program に対し，各1台の端末装置，timer（絶対，相対時間），個人用ファイルなどがあるかのようにみせる．この結果，各 user は **Fig. 4** のような system hierarhy による virtual operating system が与えられたと同じことになる．

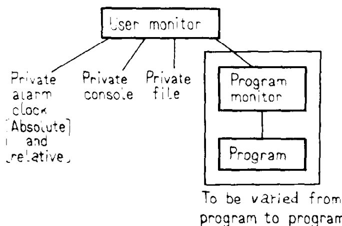

Fig. 4 The structure of a virtual operating system given to each user.

Fig. 3 からわかるように，user monitor の支配可能領域内に，user program の他の module の領域が完全に含まれているので，roll in, roll out を user monitor の支配可能領域にのみ注意して行なえば，他 module も自動的に roll in, roll out される．roll in, roll out 中は，user progrom のどの module にも制御が移らず，状態が保存されていなければならないが，そのためには freeze, unfreeze operation が有効に使われる．

TSS の time slicing, scheduling (user program の処理待ち行列の扱いなど）は，この scheduler で行なわれる．MCP は1つ1つの module の処理待ち行列のみを扱うものであり，上のことは取り扱わないので単純ですむ．すなわち，scheduler は clock module から wakeup され，それが time quantum の終わりであるならば，user program を freeze し，drum を wakeup して roll out を行ない，trc（や clock）からの旗を調べ，旗が set されていれば，対応する user program を user program の処理待ち行列に加え，つぎに実行されるべき user program を決定し，drum を wakeup して roll in を行ない，その後で置き換わった user program を unfreeze すると，その user program の活動が再開される．

scheduler は，旗が trc（や clock）から set されている user program が処理されるときに，その user の user monitor module を wakeup して知らせ，旗を reset する．

また scheduler は，user monitor から wakeup され，それが trc への要求であれば，trc module を wakeup して要求を伝え，file への要求であれば，file module を wakeup して要求を伝えるという具合に，user からの要求も媒介する．user からの要求は，すべて wakeup によって伝えられる．

2.4.2 TSS の operating system の特徴

ここでは，**2.3** の MCP の機能が，この TSS でどのように生かされているかについて説明する．

われわれの TSS では，user program の module の集団を処理待ち行列に加えることと，MCP で module を ready にすることは別のことである．しかし，scheduler により，両者は，つぎのように対応づけられる．

user program の状態	各 user module おのおのの状態
(B) Blocked ……………… (処理待ち行列からはずされる)	全部 blocked, あるいは一部 ready でも freeze されているとき．
(R) Ready………………… (処理待ち行列にはいる)	どれかが ready であり，かつ，それが freeze されていないとき．

いま，(B)の状態，つまり，user program の処理待ち行列からはずされているとき，trc や clock からの旗が立つと，scheduler はそれを user monitor を wakeup すべきことと解釈し，(R)の条件が満たされるので，その user program を処理待ち行列に加える．こうして，われわれのシステムでは user monitor 以下を除く scheduler の段階まででは，処理待ち行列

からはずす原因が何であっても，区別せず同等に扱われ，また，行列に加える原因が何であっても，同じく同等に扱われる．したがって，scheduler の段階において，input wait, output wait, waiting command, working, dormant など[11]に細かく状態分けをする必要はない．このことは，その段階においてあつかう概念の数や旗の種類を少なくしシステムを単純化し，かつ柔軟なものとする．たとえば，上のことにより，user program が入出力要求を出しながらも，ready の状態にいることもできる（つまり，入出力などとの並列処理ができる）という点でより柔軟である．

2.2 のＩ）の要求（多重モニタ動作）は，つぎのようにして満たされる．Fig. 4 で，通常の計算中は "user monitor" も "program monitor" も blocked 状態であり，"program" のみが ready 状態で活動する．しかし，端末（テレタイプ）から人間が強制的入力をしたい場合には，Quit 以外の何のコードでもよいから打鍵すると，user monitor が wakeup され，それがコードを調べ，それが Quit コードでないので，コマンド要求と解釈されずに，さらに，program monitor が wakeup される．program monitor はそのコードを見て，適当な出力入力要求を user monitor を通して出す．そこで，人間が何らかの入力を与えると，program monitor が program の動きをそれに従って制御する．ここに至るまでは，"program" も上の動きを全く知らずに計算を続けていることができる．user program は program ごとに別な構成にすることもできるし，さらに層を重ねることもできる．たとえば，誤り検査のための（DDT のような）プログラムを，user monitor と program monitor の間の interface module としてそう入することは容易であるし，そのために program monitor 以下に特別な変更が要求されることもない．しかも，この interface module は program monitor 以下から完全に保護できるのである．

2.2 のⅡ）の要求（system program の各 module の間が独立にできること）は，つぎのように実現されている．

まず，clock, trc, user monitor などの各 module は，支配可能領域が重なり合わないので［(Fig. 3(b)］，お互いに破壊しあうことができないようになっている．interval timer は clock module の直下に，回線制御用計算機は trc module の直下に付着されており，また一般に，wakeup は直上直下に対してしか出すことができないので，誤まって trc module が interval timer の set 要求を出したり，clock module が回線制御用計算機に要求を出したりするようなことは起こらないようになっている．freeze, delete などの operation が直上の module に対して作用できないことは，clock module などが誤って全システムを凍結したり，消滅させたりすることを防ぐ．scheduler module は一番強力であるが，自分自身で必要とする領域以外は記憶保護をかけているようにしているので，誤って他の module を書き直したりすることは起こり得ないようになっている．また scheduler は，interval timer や回線制御用計算機がその直下にないので，誤まって直接に要求を出すことはできない．

このようにして，scheduler の下に新たに system module を付着したり，既存の system module を変更したとしても，それらの中にある誤りがめぐりめぐってシステム全体に影響を及ぼすようになり始める前に，事前に検出されるようになっている．

clock や trc module などは，scheduler を破壊することが全くできず，逆に制御されるという点で，user program に近いといえよう．また，user program が clock や trc や scheduler と全く同じ MCP call および命令のセット(組)を有し，また，記憶保護や trap の機能が同じように備わっており，多重モニタ動作ができるという点で，system program と差別がない．このようにして，system program と user program の差の相対化が実現される．

2.4.3 記述言語

system program と user program の絶対的差別は無いという原則の一つのあらわれとして，system program も，支障のない限り procedure oriented language [FORTRAN Ⅳ (HARP)] を用いて表現する方針がとられた．これにより，プログラムを作るのが非常に容易になり，また，assembler を用いるときにありがちな，つまらない記法の誤りによるプログラム誤りがほとんど避けられた．また，assembler で書かれたものよりも，プログラムがはるかに見やすくなり，プログラムの変更も容易になった．しかし一方，部分的に検査した結果によると，hand-coded のものの 2～4 倍の長さになっていることがわかった．

HITAC 5020 の HARP は，東大大型機センターにおいて非常によく用いられ，その結果，compiler の debug が非常によくなされてあるものであるので，われわれの場合には，system 表現に用いた procedure-

oriented-language の compiler の誤りが，operating system 作成に，どのような影響をおよぼすものかについては，経験が得られなかった（むしろ，誤りは assembler にあったのだが，assembler を全く疑っていなかったために，その誤りの発見はそう容易ではなかった）．

とにかく，compiler がよく debug されていれば，procedure-oriented-language を用いて，比較的小さいシステム（とくに，リアルタイム処理のような特殊目的のもの）を，すばやく低価格で作成することができるだろうし，その maintenance も容易になることが確言できる．したがって，前述してきたような，一般性のある単純な規格化された MCP と，その MCP の上に system program を作ることを容易にする procedure-oriented-language があれば，その software を用いて，種々の operating system を作ることが，非常に容易になることは明らかである．

われわれの TSS の operating system の，MCP を除いた，user mode で走る，全 user に共通な部分が約 2K 語ほどである．

3. むすび

割り込み（external interrupt, supervisor call, trap など）は，operating system にとって欠くことのできないハードウェア機能であり，そのハードウェア機能を処理して普遍的に有用なソフトウェア概念をもたらす MCP は，どの operating system にも通用する基礎プログラムになりうるわけである． external interrupt と supervisor call の処理は，process, task, sequential process などの概念に似た概念で扱われる．その概念を用いて，安全な operating system を作るために，さらに，system hierarchy の概念が導入され記憶保護システムと関連した tree 構造の hierarchy で種々の要求が満たされることが示された．また，その system hierarchy において，freeze, unfreeze という操作の導入が有用なものであり，trap の処理も freeze 操作と統一して扱えることが示された．

われわれは実際に，一般的 MCP の具体的形を決め，その MCP を用いて実験的 TSS を作って検討した．1 回の MCP call や割り込みで数十ステップ程度のプログラム動作が生じ，ほとんどが 1 msec よりも小時間に納まり，overhead にもさして問題がなさそうなことがわかったが，量的な検討は，今後の問題として残されている．

この実験的 TSS は，上の MCP の特徴を生かして，種々の要求が満足されるようにしたものであり，それについても略述した．

本研究は，工学部和田英一助教授，佐久間，坂本，牛丸氏らとの TSS に関する共同研究の一環として行なわれた．さらに，日立中研 TSS グループ，工学部野口健一郎氏とも議論を繰り返した．他に大型計算機センターの方々には，いろいろお世話になった．以上の方々に対し深く感謝する次第である．

参考文献

1) Dijkstra, E. W.: The Structure of "THE" Multiprogramming System. Comm. ACM. Vol. 11, No. 5 (May 1968), pp. 341～346.
2) FACOM 270-20/30 ハードウェア解説書，EX-011-1/1-3 (1968 年 5 月).
3) 淵 一博，田中穂積：ETSS における GPP-Supervisor. 電気試験所彙報，ETSS 特集，Vol. 32, No. 8 (1968), pp. 750～760.
4) Graham, R. M.: Protection in an Information Processing Utility. Comm. ACM, Vol. 11, No. 5 (May 1968), pp. 365～370.
5) Heistand, R. E.: An Executive System Implemented as a Finite-state Automaton. Comm. ACM, Vol. 7, No. 11 (Nov. 1964), pp. 669～677.
6) IBM System/360, Principles of Operation. Form A 22-6821-5.
7) 亀田壽夫：計算機システムの設計．東京大学修士論文（未公刊），(1967 年 1 月).
8) Lampson, B. W.: A Scheduling Philosophy for Multiprocessing Systems. Comm. ACM, Vol. 11, No. 5 (May 1968), pp. 347～360.
9) Leiner, A. L.: System specifications for the DYSEAC. JACM, Vol. 1, No. 2 (April 1954), pp. 57～81.
10) PDP-9 USER HANDBOOK. Digital Equipment Corporation, Marynard, Massachusetts (Jan. 1968).
11) Saltzer, J. H.: CTSS Technical Notes. MAC-TR-16, M. I. T. (March 1965).
12) Saltzer, J. H.: Traffic control in a Multiplexed Computer System. MAC-TR-30 (Thesis), M. I. T. (June 1966).
13) 和田英一：TSS 用通信制御装置のプログラム，情報処理，Vol. 9, No. 6 (1968 年 11 月), pp. 335～343.
14) 和田英一：モニタシステム，情報処理，Vol. 3, No. 5 (1962 年 9 月) pp. 267～277.
15) Witt, B. I.: The functional structure of OS/360 Part II Job and task management. IBM Syst. J. Vol. 5, No. 1 (1966), pp. 12～29.

(昭和 44 年 6 月 26 日受付)