論　文

ミニコンピュータにおけるイベント処理指向型のリアルタイム・モニター*

山県　敬一** 牧之内　三郎**

Abstract

Recently, mini-computer has been used for various kinds of real-time processing. According to that situation, the productivity of control program is very important. This paper deals with a real-time monitor, which puts emphasis on the real-time and pseudo-parallel event processing in the small scale computer. Applying the fork-join mechanism to the logical model of data processing, flexible synchronizing operations are able to be described in user's function program.

In spite of that flexibility, the dispatching of the physical processor is implemented efficiently without using task control method. The dynamic linkage of each function program is also feasible and the means of program module replacement would give large adaptability for the variety of environment. As an application, DNC system with adaptive control feature is reported.

1. まえがき

最近のミニコンピュータの普及に伴い，その適用範囲は極めて広く，用途を制御用に限定しても，その対象には計測機器，特殊なデバイス，通信回線，プロセスの監視など，多くの性質の異なるものが含まれる[1]．このため，一般に用意されている汎用のリアルタイム・モニターでは使いにくく，使用環境に応じて，その都度専用のモニターと応用プログラムを開発することになり，ソフトウェアの生産性が問題になる．

この分野で重要なことは，プログラム・モジュールの portability であり，モニターの役目は強固な管理能力よりも，むしろ各モジュールの独立性と非同期処理の柔軟性を受け入れて，なおかつ，システム全体の混乱を防ぐ機能に重点がおかれるべきであろう．

この論文で扱うリアルタイム・モニターは，以上の趣旨に沿って，専用モニターの標準化という接近法をとり，独立なモジュールの積み上げによって[2]，必要な機能を付加して行く構成法を採用する．また，小型計算機の上でこの目標を実現するため，タスク管理の概念を使用せず，いろいろなイベントの発生に伴う処理要求の遷移を論理的モデルとして設定し，これに対して，簡潔な手法で CPU の割り当てを行う．

筆者らは，適応制御機能を持つ DNC システムの制御のために，NEAC-M 4 を使用しており，これがこのようなモニター開発の発端となった．これについては応用例として後に述べる．なお，ここでは原則として CPU は一つであるものとし，各プログラムは主記憶にあるものを対象として考える．プログラムの roll I/O は，ユーザの指示によってのみ行われるものとする．

2. ファンクション・プログラムの擬並行処理

2.1 論理的モデル

性質の異なるリアルタイム処理を一括して扱おうとするとき，タスク，イベント，メッセージ・キューなどの概念を用いるのが一般的であるが，モニターが大

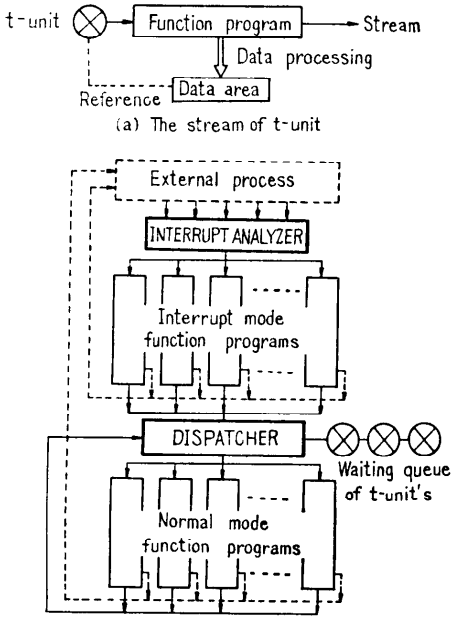
* A Real-time Monitor for Mini-computer Using Event Processing Model by Keiichi YAMAGATA and Saburo MAKINOUCHI (Department of Precision Engineering, Faculty of Engineering, Osaka University)

** 大阪大学工学部精密工学教室

きくなるのを防ぐためには，これらの概念の使い方に制約がつくことが多い．また一方，ユーザの必要とする同期操作の面からみると，複数個のイベント情報に対する論理判断のような操作が，プログラムの中で積極的に記述できることが必要である．

そこで，計算機の内部，外部を通じて発生するいろいろなイベントと，それらに結びついた処理要求のプログラム間での遷移に注目し，これを“t-unit”(transition unit) という概念でモデル化する．t-unit は 10～16 バイトのメモリを占有し，control register あるいは mail-box としての役割りを持つが，CPU とも間接的に結びついている．ここでは，タスク管理の概念は使用せず，ファンクション・プログラムの中を t-unit が通り抜ける“stream”によって，データ処理を定義する（**Fig. 1**(a)）．このように抽象的なモデルを設定するのは，処理形態の柔軟性と，実際的な CPU の割り当てを簡潔に行うことの二面性を両立させるためである．

一つのファンクション・プログラムに注目するとき，一般的には複数個の t-unit が出たり入ったりすることが許される．このことは，複数個のイベント情報を受け取り，論理判断を行った後でデータ処理に入る（または再開する）場合，あるいは，ファンクション・プログラムの内部にバッファ・キューを抱え込んでいるときなどに効果を発揮する．プログラムが厳密な意味ではリエントラント形式になっていなくても，t-unit の stream は，同時に複数個存在してかまわないのである．



(a) The stream of t-unit

(b) Schematic diagram of event processing

Fig. 1 A logical model of real-time processing

結局，システム全体がイベント処理指向型で構成され，ファンクション・プログラム相互の擬並行処理や cooperation も，stream の分流と合流，そして t-unit によるイベント情報の交信によって遂行される．ユーザ側はこれの使い方如何で，平易な形式の逐次処理のプログラムから，バッファ・キューを含む複雑なプログラムまで，その作成法に自由度が与えられ，モニター側は煩雑な同期操作の一部をユーザ側にまかせることによって，それ自身は簡潔になる．マクロ命令の実行によって，計算機内部では何らかの処理要求，またはイベントの通知を伴った t-unit が常時複数個動きまわることになる．この様子を Fig. 1(b) に示す．モニターの核としてディスパッチャがあるが，ユーザ側から見るとき，これは t-unit の分配機能を持つもので，CPU の割り当ては暗に行われるに過ぎない．この中には t-unit の待ち行列があって，基本的なスケジューリング機能を持ち合わせている．

また，小型計算機によるリアルタイム処理では，割込み処理の占める重みが大きいので，割込み処理プログラムを一般のファンクション・プログラムと同様に扱い，自由に差し換えができるようにする．このため，インタラプト・アナライザを独立させて，これをもう一つのディスパッチャとして扱う．ここには t-unit のキューはないが，かわりに周辺装置のインターフェースがあって，これが処理要求を発生し，イベント情報やデータの入出力もこれを介して行われる．逆の見方をすれば，t-unit はこれと等価なプログラム間のインターフェースの役割りを果す．

このように考えると，Fig. 1(b) に示したモデルでは，計算機の外部と内部，またハードウェアとソフトウェアの接合が滑らかである．たとえば，一つのファンクション・プログラムの機能をマイクロ・プロセッサで実現して，これをハードウェア・インターフェースに移行させるようなシステム変更が自然な形で処理される．

2.2 t-unit によるプログラム間の動的結合

すでに述べたように，t-unit として物理的にはメモリを使用する．その内容の具体例を **Fig. 2** に示す．

Byte no.	8 bits	8 bits
1, 2	pointer area for chaining	
3, 4	t-unit status	p_1 program no.
5, 6	order code to receiver	
7, 8	response code	p_2 program no.
9, 10	i return point to sender	
11, 12	(data area start address)	
13, 14	(data area end address)	
15, 16	(address counter of data area)	

p_1, p_2: priority for CPU assignment (2 bits)
i : return inhibit flag (1 bit)

Fig. 2 The contents of a standard t-unit on NEAC-M4

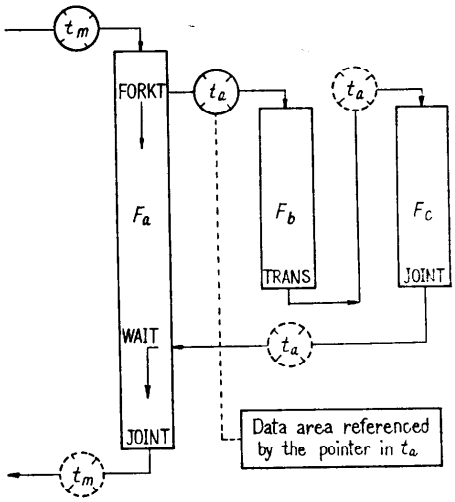
バイト番号* 4～6 は受信者の宛名と要求コード，7～10 は発信者に返す応答とそのもどり場所を指す．これらに先頭の3バイトを加えた 10 個のバイトはモニターが関与する部分で，それ以降の領域はファンクション・プログラムごとに適宜使用される．メッセージをこの直後に続けることにすれば，バイト番号1,2のポインター・エリアを用いてメッセージ・キューを構成することが可能である．なお，第3バイト目にはマクロ命令に応じたコードが書き込まれる．

リアルタイム処理の特徴から，ファンクション・プログラム相互の結合関係は，実行前には決まっていない．実際に何らかの処理要求が発生した時点で，t-unit によってデータ領域に対するポインターが渡され，共通の参照領域が作られる．

t-unit のプログラム間の遷移は，すべてマクロ命令により，ディスパッチャを経由して実行されるが，これらマクロ命令の基盤になっているのは，fork-join 系である．ただし，Conway の提唱[3] したものとは多少性質が異なる．まず，分流と合流の適用対象はプロセッサの流れではなくて，t-unit の stream である．さらに，それぞれの stream の間に従属関係があるわけではないが，t-unit とそれに伴うデータ領域の提供者は常に明瞭になっており，t-unit は原則としてその発信者にもどってもとの stream に合流する．したがって，プログラム間の関係は semi-coroutine[8] である．ファンクション・プログラム相互の結合が動的に行われるため，データ領域の提供者を常に明確にしておかないと，デッドロックを回避する手段が講じられなくなるからである．

以下の4種のマクロ命令について，その使用例を

* NEAC-M4 は8ビットを1語とし，4語までの可変語長が扱えるワード・マシンであるが，ここでは8ビットを1バイトと書く．



F_a, F_b, F_c: function program
t_m, t_a : t-unit

Fig. 3 Modified fork-join mechanism

Fig. 3 に示す．

FORKT (t_a, F_b)
TRANS (t_a, F_c)
JOINT (t_a)
WAIT

一つの t-unit t_m の F_a 内における stream から，FORKT によって新しく t_a の stream が作られ，F_b を通り抜けた後 TRANS マクロで F_c に移り，JOINT でもとの stream に合流する．ここで fork-join 系を用いることの特徴は，新しく作られた t_a の stream が，F_b に受け入れられるかどうかを見極わめることなしに，もとの t_m の stream が継続することである．マクロ命令実行時に，t_a はディスパッチャを経由するので，それが F_b に受け入れられる時点の判定はディスパッチャに管理されている．JOINT マクロについても同様で，このマクロ命令の実行によって F_c 内の t_a の stream は終了するが，それがもとの stream に合流する時点はディスパッチャに依存している．これにより，プログラム間の占有関係が作り出されないので，デッドロックの回避に役立つ．

複数個の t-unit を用意しておいて，FORKT マクロを続けて実行すれば，複数個の stream を作り出せる．この場合，合流点をそれぞれ別々に設定できるように，リターン・ポイントを t-unit 内に書き込めるようになっている (Fig. 2)．なお，ディスパッチャのスケジューリング機能に関連して，t-unit には2ビッ

トの優先度が指定できる．これは後に述べるように，CPU 割り当ての優先度を示すが，プログラム番号とは別に t-unit のそれぞれに指定できるので，同一のプログラムでも要求の種類に応じて異なった優先順位で実行することが可能である．

WAIT マクロは，何らかのイベント待ち（より具体的には t-unit 待ち）のとき，CPU をひとまず手放すために使われる．原則として，複数個の t-unit を順序不同で受け取れるようにするため，特定のパラメータを指定するようにはなっていない．

2.3 同期操作

ここでは，応用に即した形式についてのみ述べることとし，一般論は他の文献にゆずる[4),5)]．もともと t-unit のモデルを導入した目的は，リアルタイム処理特有の論理的判断を伴う同期操作を，ユーザ側で記述し易くすることであった．したがって，モニターとしては conditional critical region[7)] の一般形を処理する機能を持たず，t-unit 分配のスケジューリングに関連する操作だけを行うので，モニターの核は簡潔な構成になっている．以上の点を含めて同期操作を要約すると，つぎの3種類にまとめられる．

（1） ディスパッチャ内に，プログラム番号対応のエントリー・ポイント・テーブルがあり，ここにセマフォア変数が用意されているので，これを用いて t-unit 流入の同期をとる．

（2） すでに相互の共有エリアを持っているプログラムの間では，このエリアにセマフォア変数を用意することにより，cooperation を遂行する．必要ならば割込み禁止命令を併用する．

（3） プログラムの入口部分，ならびに stream の合流点において，一つ以上のイベント情報を受け取って論理判断を行う．

これらの中で，cooperation を実行する（2）の場合，相手側が WAIT マクロを実行した状態では，何らかの形で起動をかけることが必要である．この目的のために，RESUME マクロと SIGNAL マクロが用意されている．この両者は機能的には同じであって，プログラム結合時に使われた t-unit の遷移をもう一度引き起こし，wakeup の機能を果す．ただし，semi-coroutine の関係において，親から子には RESUME マクロを，子から親には SIGNAL マクロを使用する．

つぎに，複数個の t-unit をも扱う（3）の場合，より具体的には，一つの t-unit の流入についてその都度 CPU が流れ込んで来るので，論理的条件が満たされなければ，WAIT マクロを実行して他のイベントを待つ．また特に必要ならば，一つのファンクション・プログラムで，複数個のエントリー・ポイントを持つものを書いてもよい．この場合，見掛け上は別のプログラムとして扱われる．

ディスパッチャにおけるセマフォア変数の扱いは，スケジューリング機能に関連するので，次章以降に述べる．さらに複雑な同期操作をシステムで共通に使いたい場合には，上の手段を用いて，同期をとるためのファンクション・プログラムを独立に作り，これを登録することによって機能の追加を図る．

3. ディスパッチャの構成

ディスパッチャには二つの役目がある．一つは論理的モデルにおいて，マクロ命令の実行に伴う t-unit の分配機能であり，もう一つは物理的な CPU の割り当て機能である．

3.1 論理的モデルのスケジューリング

FORKT, TRANS, RESUME の3種のマクロ命令では，t-unit はエントリー・ポイントに送り込まれる．これらのマクロ命令は，原則として独立な並列動作の基盤の上で使用されるものであり，相手側の状況に無関係に実行されるのが立て前である．これに対して，システム全体が混乱に陥いることのないように，ディスパッチャ内に **Fig. 4** に示すようなエントリー・ポイント・テーブルを持っている．t-unit に書き込まれたプログラム番号が n に対応しており，実際の実行開始アドレスは，ここに一括して登録されている．したがって，プログラム差し換え時のアドレス変更，さらには RESUME マクロに対する処理再開アドレスの動的変更が可能である．

このテーブル内には，各エントリー・ポイントごとに一種のセマフォア変数 $S(n)$ があって，ディスパッ

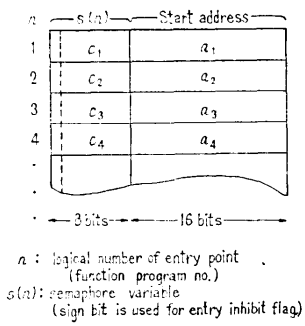


Fig. 4 Entry point table in dispatcher

チャとそれぞれのファンクション・プログラムの間で共通に参照され，また値の変更が行われる．ただし，Dijkstra の P/V 操作[6]とは多少性質が異なり，符号ビットとカウンターが独立に扱われる．カウンターは，受け入れ可能な t-unit の数を表わし，符号ビットは一時的に t-unit の流入を禁止する entry inhibit flag として使われる．$S(n) \leq 0$ ならば，対応する t-unit はディスパッチャ内の待ち行列につながれる．逐次的にしか使えないデバイスの制御プログラムでは，カウンターは1または0の値を取り，これによって要求を一つずつ受け入れる．なお，RESUME マクロでは，すでに一度 t-unit の流入が行われた後での操作であり，カウンターは無視されて inhibit flag だけが有効である．

JOINT, SIGNAL マクロによって発せられた t-unit は，それ自身の中に書き込まれたリターン・ポイントに送り込まれる．ここにも return inhibit flag があり (Fig. 2 参照)，一時的な stream 合流の抑止効果を持つ．

ディスパッチャ内の待ち行列を作る場合，各エントリー・ポイントに対応してキューを作れば，メッセージ・キュー[7]に近い形式となる．この手法では，各プログラムに対応した t-unit を取り出し易い利点はあるが，CPU 割り当てのための待ち行列を別に用意しなければならない．

ところで，ここで考えているモニターの構成法では，論理的条件を伴う同期操作をユーザ側にまかせているので，これに関連する t-unit はファンクション・プログラム内部に留まっているものが多く，ディスパッチャに入って来るのは真にプログラム間での処理要求が発生した場合に限られる．したがって，一時点でディスパッチャ内に存在する t-unit の数は，それ程多くはないと考えられる．

この特徴を考慮して，待ち行列は CPU 割り当ての面から，t-unit の優先順位別に作る．この場合，一つのキューに性質の異なる t-unit がつながれることになるが，その反面，それまで実行中であった優先順位より高い順位にあるキューだけを調べればよいという利点がある．この点は，CPU 割り当てのスケジューリングと関連するので，次節で詳しく述べる．

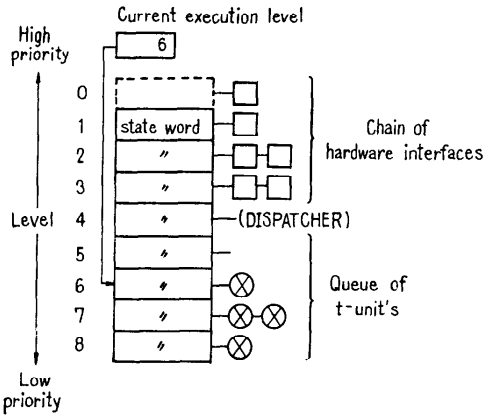
3.2 CPU の割り当て問題

一般的には，ディスパッチャの中に複数個の t-unit がまわって来るが，これらをそれぞれの中に書き込まれた優先度別にふるいわける．同じ優先度を持つものが二つ以上あれば，優先順位ごとにキューを作る．リアルタイム処理では優先処理は重要であり，CPU の preemption も行って優先度の高いものを先に処理する．また一方，同一優先順位に属するものについては，セマフォア変数の通過条件を満たす限り，要求発生順に処理するものとする．そこで，一つの優先順位に対応するキューを一種のコマンド列とみなし，これらについては逐次処理で良いと決めてしまえば，一つの優先順位ごとに一つの仮想プロセッサを割り当てることにして，state word も順位の数だけ用意すれば十分であろう．

実行中の優先順位は最後に CPU を割り当てた t-unit によって決まり，その後のマクロ命令の実行によって処理の順位が切り換わるときには，レジスタの待避復旧はディスパッチャにより自動的に行われる．ただし，WAIT マクロについては，同一の順位で次の t-unit の処理に移るのが原則であるから，ユーザ側で処理の再開に必要なレジスタの内容を保存しておく．これは一つの制約であるが，モニターのオーバヘッドを増大させないための妥協策である．

ディスパッチャのスケジューリング機能の中で，優先処理の機能と逐次処理の機能を明確にしておくと，ユーザ側では t-unit の優先度のつけ方如何によってこれら二つの機能を使い分けることができる．一つのファンクション・プログラムについて，複数個の t-unit が流入する場合や，stream の合流点で複数個の応答を受け取るときに，優先度が同じ順位に統一されていれば，WAIT マクロを実行しない限り t-unit の流入は起こらない．逆に優先度を変えておいて，エントリー・ポイントやリターン・ポイントを複数個設けておけば，一つのプログラムの中でも追い越し処理や割込み型の同期操作が可能である．さらに inhibit flag の併用を考慮すれば，プログラムの処理形態について，ユーザ側に大きな選択の自由度が与えられることになる．

t-unit の中に state word を持つ手法は，タスク管理法に近いもので論理的には興味がある．しかし，モデルの設定に柔軟性があるため，実際の処理がモニター側もユーザ側も煩雑になるので，この手法は敢えて採用しない．また，t-unit の stream と Lauesen の activity[9]には類似点があるが，activity は一つのモニター・プロセスの枠内で使われているのに対して，t-unit はいわゆるプロセスの枠を越えて，ユーザにより同期操作一般，ファンクション・プログラム間の動



Level 0～3: interrupt mode
(CPU assignment is executed by the interrupt analyzer.)
Level 5～8: normal mode
(CPU assignment is executed by the dispatcher.)

Fig. 5 Scheduling mechanism of CPU dispatching

的結合，優先処理の積極的利用など，汎用性のある道具として用いられる．システム全体の構成法という見地から，両者は性質を異にしているのである．

Fig. 5 には，多重割込み4レベルとノーマル・モードの優先順位が4レベルの場合を示してある．ディスパッチャは原則として割込み禁止で働くが，これにも優先順位を一つ与えておき，t-unit のキューについてセマフォア変数の判定を行う場合，一時的に割込みを受けつける．割込みモードには t-unit のキューはないが，必要に応じて，ノーマル・モードのキューに t-unit をつなぎ込むことが可能である．

実際にプロセッサが複数個ある場合にも，それらを別々の優先順位にわりふるものとすれば，このモデルはそのまま使えると思われる．ただし，非可分の test and set 命令ならびに相互の割込みのかけ方が明確にされなければならい．

一つの優先順位に属する t-unit の数に制約はなく，プログラムそのものはもともと優先順位とは無関係である．CPU の preemption を実施する優先順位の数は，デバイスその他の物理的資源が無尽蔵にあるわけではないので，必要最少限に止める方が望ましい．順位数を多くすると，過大負荷の状態に陥いる危険性があるので，待ち状態が発生する場合には，早い時点で待たせる方がシステムの効率は良くなる．

3.3 デッドロック対策

それぞれ独立に作られたプログラムに対して，組み合わせや差し換えに自由度を与えると，デッドロックに陥いる危険性も高くなるが，小型計算機でのモニターでは，それを動的に検出して防止することは期待できない．したがって，デッドロックの回避は，最終的にはユーザ側の責任である．

ただ，semi-coroutine に対する fork-join 系では，デッドロックに陥いる危険性は少ない．一般にデバイスなどの管理も，その制御プログラムのエントリー・ポイントにおけるセマフォア変数で処理されるが，2.2 でも述べたように，FORKT, JOINT マクロでは，ファンクション・プログラム間の占有関係はできないので，処理途中で cooperation を実行しない限り，デッドロックの原因を作り出さない．

たとえば，予め1レコード分のデータを用意しておいて，ディスク装置の転送要求が出されたような場合，ディスク転送処理は要求を出した側の状態と無関係に実行される．また，転送終了時には JOINT マクロによって，終了通知を返すことになるが，その通知が要求元へ帰ることを確かめずに次の要求を受けつける．したがって，このような転送要求の列は有限時間内に終了する．

ただし，共有エリアを用いて cooperation を実行するときには，暗に占有関係ができる．また，メモリ・スペースについても，モニターで一括管理する手法はとっていないので，それを管理しているファンクション・プログラムを常に明確にすることによって，ユーザが安全な手段を講じられるようにしておく．

4. オンライン DNC システムへの応用

広い意味でのモニターには，標準のデバイス制御プログラムや回線制御プログラムなども含まれるが，これらはユーザの作成するファンクション・プログラムと全く同様に扱われる．したがって，これまで述べて来たインタラプト・アナライザとディスパッチャを核として，これに標準のサービス・プログラムを必要に応じてシステムに登録し，ユーザのアプリケーション・プログラムを追加すれば，使用環境に応じたシステム構成ができ上る．

ファンクション・プログラム相互の結合は動的に行われ，マクロ命令によるディスパッチャ呼び出しのためのアドレスも，固定したアドレス・テーブルを用意すれば，個々のファンクション・プログラムの登録は独立に行われるので，システム・ジェネレーションは不要である．プログラム番号を決めておいて，ディスパッチャ（またはインタラプト・アナライザ）内のエントリー・ポイント・テーブルに実行開始アドレスを

設定すればよい.

この利点をいかして，たとえばタイプライタの制御プログラムでは，単純な入出力を行うものと，プログラム・デバッグ機能を持つものの2種類を用意して，使用状況に応じて差し換えると便利である．また，タイム・テーブルが必要なときには，タイマーによる割込みを利用して，予定された時刻に t-unit を発生するファンクション・プログラムを追加する.

このように使用環境に応じて，必要なプログラム・モジュールを寄せ集めてシステム全体を組み立てるのがここでの基本的な考え方である．このようなプログラム・モジュールの集りをライブラリとして整理することも，ある程度の範囲までは可能であろう.

現在，筆者の所で使用中のシステムを **Fig. 6** に示す．ミニコンピュータ NEAC-M4は，大阪大学大型計算機センターの TSS に接続されており，(1)通信回線の制御，(2)図形処理のためのタブレット型座標入力装置やプロッターの制御，(3)オンライン DNC，(4)加工プロセスの監視，などの多くの役目を担っている[10]．われわれは，TSS 側で計算した部品形状に対して，加工時における工作物の熱変形や工具の摩耗状況を監視し，高精度の加工を自動的に行うことを目指している．このような場合，かなり異質なプログラムを同時に働かせる必要があり，プログラムの差し換えや柔軟な同期操作が重要になる.

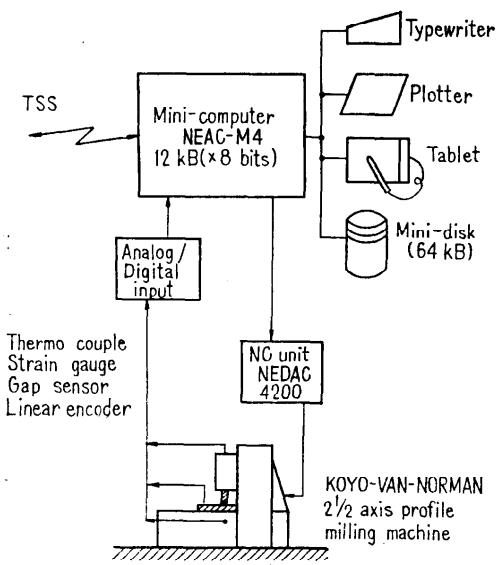


Fig. 6 The configuration of DNC system with adaptive control feature

メモリ占有量は，インタラプト・アナライザとディスパッチャの核の部分が，管理テーブルを含めて約1.5kバイトを占め，これにタイプライタとディスク転送の基本プログラムを加えて，約2.5kバイトで基本のモニターが働く．回線制御のためには，さらに2kバイトが必要である.

計測の面から考えるとき，割込み禁止による不感時間帯が，高速データ・サンプリングの障害になる．現在使用しているモニターでは，ディスパッチャ内の優先順位切換えの部分で，約 400 μsec の不感時間帯がある*．この値は，ノーマル・モードの順位数を4から2に減少させ，割込み処理主体の方法をとれば，もう少し短縮が期待できる．なお，すでに述べたように，待ち行列を調べるときには一時的に割込みを受けつけるので，全面的な割込み禁止時間としては 400 μsec が最長であって，待ち行列が長くなってもこの値は変わらない.

NEAC-M4 には，同一のレジスタ・グループが2組あるので，その一方を最高優先順位に専用に割り当て，ここで高速サンプリングを実行すれば，レジスタ待避復旧の負荷はそれ程大きくならない.

5. むすび

ここに述べたモニターは，およそ3年間の使用経験の中で，不要なものを切り詰め，必要な柔軟性に対しては機能を追加することによって，でき上ったものである．ミニコンピュータの場合，簡潔なモニター（これはむしろ server と呼ぶべきであろう）の下で，プログラム・モジュールの差し換えによって，広い適用範囲に応じられるようにすることが，有効な手段であると思われる.

なお，記述言語は現在のところアセンブラを用いているが，同期操作やポインター操作が容易に表現できる言語はやはり必要であろう．また，プロセッサが複数個ある場合の効率の問題は未検討であり，残された課題である.

最後に，本研究を行うに当り日本電気(株)，光洋機械工業（株）のご協力を得ており，日本電気小林淳人氏，光洋機械工業曽我部孟氏，ならびに日頃機械加工面でのご指導を賜っている，大阪大学工学部難波義治助教授に厚く御礼申し上げる．なお，計測側の機器については，昭和 49 年度科学研究費補助金を受けてい

* NEAC-M4では，16ビットのメモリとレジスタ間の移送に6 μsec を要する.

ることを申し添える.

参考文献

1) 情報処理学会編: ミニコン応用特集号, 情報処理, Vol. 15, No. 4, pp. 233～319 (1974).
2) 高橋秀俊, 亀田寿夫: オペレーティングシステムの一構成法, Vol. 11, No. 1, pp. 20～31 (1970).
3) M. E. Conway: A Multiprocessor System Design, Proc. AFIPS FJCC, Vol. 24, pp. 139～146 (1963).
4) 丸山 武: 非同期処理について (I), (II), 情報処理, Vol. 11, No. 1, pp. 41～48, No. 2, pp. 95～100 (1970).
5) 斎藤信男: OS の基礎理論 (1), 情報処理, Vol. 15, No. 11, pp. 887～899 (1974).
6) E. W. Dijkstra: The Structure of the "THE"-Multiprogramming System, Comm. ACM, Vol. 11, No. 5, pp. 341～346 (1968).
7) P. B. Hansen: Operating System Principles, p. 366, Prentice-Hall, (1973).
8) O.-J. Dahl and C. A. R. Hoare: Hierarchical Program Structures, in Structured Programming, pp. 175～220, Academic Press (1972).
9) S. Lauesen: A Large Semaphore Based Operating System, Comm. ACM, Vol. 18, No. 7, pp. 377～389 (1975).
10) 牧之内三郎, 山県敬一: TSS 端末用ミニコンピュータの制御プログラム, 情報処理学会, 第 14 回大会前刷 (1973).

(昭和 51 年 7 月 12 日受付)