論　文

プログラム・スタッキング技法*

塚　本　享　治**

Abstract

A technique named 'program stacking' is introduced and its applications are described. The technique pushes programs into parallel stacks, transfers control to them, and pops them up when completed their execution. The concept of program stacking extends 'execute instruction' to a program. Programs are protected and executed in parallel stacks. Therefore, this technique enable seven a program with program modification to be reentrant or recursive.

1. まえがき

現在の計算機はストアド・プログラム方式といわれ，手続きであるプログラムも手続きのための情報であるデータも計算機内部では同じ表現と記憶の形式がとられている．そのれい明期のころは，主記憶が狭いためもあって，プログラムとデータを混然一体として扱い，実行の途中でプログラムを変更しながら実行するいわゆるプログラム変更(program modification)という技法が用いられた．その後，この技法は用いられなくなったがその理由としては

（1） 多重プログラミングやオンライン・リアルタイム処理用のプログラムでは，プログラムは再入性を持つ必要があり，これを実現するためには割込みの禁止と解除や変更された情報の退避と回復などの処理を行わなければならない．

（2） 高級言語処理系などで要求される再帰性が実現しにくい．

（4） プログラム作成上のアルゴリズムの間違いをひき起こしやすい．

（4） プログラムの変更が動的なため虫取りがしにくい．

などがあげられ，『よくないプログラミング技法』[1]とさえいわれている．

ところが，目的コードにおとされたプログラムをデータとして扱うプログラム，例えばトラップ処理，シミュレータ，トレーサー[2]などにはプログラム変更はいぜんとして有効な技法である．この技法はほんらい再帰性と再入性を持たないが，使われるときにはとくに再入性が要求されることが多い．

本論文ではこのことを可能にする技法，すなわちハードウェアでスタック機構を持つ計算機を念頭において，プログラム変更された部分またはプログラム変更される部分をスタックに入れて実行するプログラム・スタッキング (program stacking)*** と名づける技法を提案し，その一般化と適用例について述べる．

この技法は，1語の命令に適用される実行 (Execute) 命令を割込みを許す形で複数語の命令からなるプログラムにまで拡張したもの，スタックまたはその拡張である動的記憶配置機構の使用をプログラムにまで拡張したものということができる．関連するものとしては，プログラムの一部でこの技法を用いる方法では混合コード方式[4),12)]があげられる．

2. プログラム変更

手続きとしてのプログラムとそのためのデータが，領域的，機能的に分かれていないときにプログラム変更が必要になる．

例 1. サブルーチン・コール

サブルーチンに引数を渡す最も簡単な方法の一つはプログラム中に引数を埋め込む方法である．プログラ

---

* Program Stacking Technique by Michiharu TSUKAMOTO (Information and Control Section, Control Division, Electrotechnical Laboratory)

** 電子技術総合研究所制御部情報制御研究室

*** 最初，スタックド・プログラミング (stacked programming)[3]と名づけていたが古川康一氏（電総研）の御指摘により変更した。

ムとデータが領域的に混在しており，新たな引数でサブルーチン・コールが行われるたびにプログラムが変更される．

例 2.

トレーサーやトラップ処理ルーチンなどでは，ほんらい手続であるはずのプログラムをデータとして扱い，それのふるまいを模擬することが行われる．このようなとき，実行の途中で命令を作りながら実行することが必要になる．1ワードの命令ならば実行命令によって1命令サブルーチンとして実行することができるが，この命令がない場合やあっても例1のように複数ワードで1命令と同じような働きをする場合には，命令を分析して解釈するか，次のステップに命令を埋め込んで実行しなければならない．

このようなときプログラム変更された部分は不要になるまでのあいだに重ねて変更されてはならない．プログラム変更は再入性と再帰性をそこなう原因となっており，そのことを可能にするためには，それぞれ再入の禁止と解除* や変更された部分の退避と回復が必要になる．ところが，前者の場合には他のプロセスをブロックすることになるし，変更される部分が数多くあったり1カ所であってもその大きさが変わるようなときには，後者のことさえむずかしいことになる．

3. プログラム・スタッキングの原理

計算機の処理単位であるプロセスの実行の仕方には，多重レベル割込みのように各々にあらかじめ優先度をつけておき，優先度順に処理する方法と，多重プログラミングのように決められた優先度を持たないで処理する方法がある．**3.2** では前者，すなわち低い優先度のプロセス**に対してのみ高い優先度のプロセスが割込めるのを許す場合を扱い，一般的な後者の場合は **3.3** で扱う．

3.1 スタック

処理の各段階で，あとの処理で使用されるプログラムやデータは活性状態にあるという．現在さしあたっては必要としていないが活性状態にある情報を記憶しておくための機構の1つにスタックがあり，次の2つの性質を持っている．

〔性質1〕 情報の記憶

〔性質2〕 記憶された情報の破壊からの保護

その機構は計算機上では次のようにして実現される[5),11)]．

(1) スタックは情報を記憶するためのスタック領域 (SA) と，それを管理するためのスタック・ポインタ・ワード (SPW) で構成される．

(2) スタック領域は主記憶上の連続した領域で，低番地から高番地へと使われる．

(3) スタック・ポインタ・ワードは次のものから構成される．

スタック・ポインタ (SP)：使用済みのスタック領域の最終番地を指す．

スペース・カウンタ (SC)：未使用領域のワード数

ワーク・カウンタ (WC)：使用済み領域のワード数

スタックの操作中に他のプロセスが割込んで同じスタックが使用されると，SPと対象となる情報やSCとWCの関係に矛盾が生じる．そこで，スタックに入れる操作 (push down，略してプッシュ) と出す操作 (pop up，略してポップ) はハードウェアで準備された命令であるとする．

このようにスタックを実現すると，情報の記憶するスタック領域が主記憶上にあって，その状況がスタック・ポインタで指示されるためにさらに次の2つのことが可能である．

〔性質3〕 スタック・ポインタと指標レジスタを使って，スタック内の情報をプッシュ，ポップしないでも直接に参照することができる．

〔性質4〕 スタック領域はスタック・ポインタ・ワードを介さないでみれば普通の記憶領域と同じであるから，実行プログラムをスタックに入れ，すなわちスタック領域にロードして，その中で実行することができる．

スタックの中に実行プログラムを入れ，それを実行する機構をプログラム・スタッキング (program stacking) と名づける．

3.2 基本型プログラム・スタッキング

プログラムの一部を切り出したもので，入口が最初で出口が最後にあるようなものを基本型ハンク (basic hunk)*** と呼び，基本型ハンクに対するプログラム・スタッキングを基本型プログラム・スタッキング (basic program stacking) と呼ぶことにする．以下に基本型プログラム・スタッキングのアルゴリズムを述べる．

---

* 例えば，割込み禁止と解除や Dijkstra の P-V オペレーション

** プロセスに優先度がつけられた場合，タスクと呼ばれるが，ここでは区別しないで両方ともプロセスと呼ぶ

*** Dawson は，入口が1つで，出口が最後にあるか，最後の命令がジャンプ，コール，リターンのどれかになるプログラムの単位を basic hunk (厚切れの意味) と呼んでいるが[4)]，ここでは Dawson の第1の型のものに限る

〔アルゴリズム1〕 基本型プログラム・スタッキング

(1) その時点のスタック領域で実行可能なプログラムをスタックに入れ，スタック中に基本型ハンクを作る．

(2) スタック中の基本型ハンクを実行し元の流れに戻るための命令をスタックに入れる．

(3) (1)，(2)の過程でスタックに入れたワード数から1減じた数の負数を指標レジスタXに入れる．

(4) 次の命令によって(1)で作った基本型ハンクの先頭に制御を移す*．

B *SP,X ; branch to (SP)+(X)

(5) 基本型ハンクを実行し，最後に(2)で作った命令を実行して元のプログラムに制御を戻す．

(6) (1)，(2)でスタックに入れたものを取り出し，スタックを元の状態に戻す．

プログラムの一部分において，いつ再入されてもよいとき，その部分で常時再入可能であるということにする．このとき次の定理が成立する．

〔定理 1〕 プロセスが優先度順に処理されるとき，基本型プログラム・スタッキングの技法は常時再入可能である．

証 明 あるプロセスの処理中に他のプロセスが割込むとき，暗黙のうちにプロセスの状態，すなわちプログラム・ステータス・ワード (PSW) やレジスタを含むプロセス・コントロール・ブロック (PCB) が交換されるから，各プロセスではレジスタをプロセス独自の作業用に用いることができる．また，活性状態にあるプロセスが優先度順に処理されるから，1つのスタックをプロセス間で共有して作業用に用いることができる．技法が常時再入可能であることを示すために，アルゴリズムの各ステップについて検証する．

(i) (1)，(2)でスタックに入れられるプログラムが再配置可能なものなら問題はないが，スタックに入れるときに修飾や変更があっても，そのときレジスタを作業用に用いるならばこの過程は常時再入可能である．

(ii) (3)で使用するのはレジスタだけ，(4)はもともとプログラムに埋め込まれている命令だからともに常時再入可能である．

(iii) (5)の途中で割込みがあっても，そのときのプログラムはスタック・ポインタによって保護されているから，この過程は常時再入可能である．

(iv) (6)は(1)の逆操作で常時再入可能である．スタックに入れて実行する単位は基本型ハンクだから，たとえ(5)の途中で制御が一時的にスタック外へ移ったとしても必ず(2)の命令とそれにつづく(6)の過程が実行される．したがって暗黙のうちに行われるプロセス・コントロール・ブロックの交換時を除けば常時再入可能である．

ところで，優先度順にプロセスを処理するとき，高優先度のプロセスは低優先度のプロセスに対する入れ子と考えることができる．この定理は制御構造が入れ子構造になっているときには，単一のスタックを用いて常時再入可能な形でプログラム変更とその実行が行えることを保証するものである．

3.3 一般のプロセスへの拡張

多重プログラミングのようにプロセスに優先度がつけられていない一般の場合，複数個のプロセスが同時に活性状態にあってその活性状態が解かれる順序は不定である．スタックで取り扱える制御構造は後入れ先出しの入れ子構造に限られるから，一般のプロセスに対しては単一のスタック上でプログラム・スタッキングを実現することはできない．それはプロセスの各々にプログラム・スタッキング用のスタックをそれぞれ1つずつ与えることよって解決される．この並列スタックは次のようにして実現される．

(1) スタック・ポインタ・ワードをプロセス・コントロール・ブロックに含ませる．

(2) スタック領域を動的記憶配置 (dynamic storage allocation) 機構によって配置する．

このとき次の定理が成立する．

〔定理 2〕 一般のプロセスの処理において，基本型プログラム・スタッキングの技法は常時再入可能である．

証 明 1つのプロセス中にあらわれる基本型プログラム・スタッキングは定理1の特別な場合である．複数個のプロセスが存在するとき，割込みが発生してプロセス・コントロール・ブロックが交換されると暗黙うちにプログラム・スタッキング用のスタックも交換され，プロセスごとに別々のスタックを使用することになる．したがって，プロセス・コントロール・ブロックの交換時を除いて常時再入可能である．

なお，この定理は動的記憶配置機構**の使用をプロ

---

* 記号命令については付録

** 動的記憶配置機構[2),6)]には再配置の必要なものと不要なものがあるが，再配置の必要な機構[7)]のときには，再配置にともないプロセス・コントロール・ブロックやスタック中のハンクの変更が必要になる

グラムにまで拡張したものとなっている.

以後スタックというときは，動的記憶配置機構によって実現された並列スタックの1つを指すものとする.

4. プログラム・スタッキングの拡張

プログラムはブランチ，コール，リターンなどいろいろなブランチ関係の命令を含んでいるので，プログラムの一部を基本型ハンクの形で切り出すのがむずかしい場合がある．そこでこの章では，一般のプログラムに対しプログラム・スタッキングを拡張する.

4.1 プログラム・スタッキングが適用されるためにプログラムが持つべき制御構造

プログラム・スタッキングによってプログラムが実行されるさいには，モジュール間* の制御の授受とそれに関連したスタックの管理が正しく行われなければならない．モジュール間の制御の授受に着目すると，制御構造は次の4つに分類される.

(1) 停留型： 制御が1つのモジュール中にあるとき，モジュール間に制御の授受はないから，制御に関連してスタックが用いられることはない.

(2) コール型： 一時的に他のモジュールに制御を移譲するとき，コールする相手によって2つに分けられる.

(2.1) スタック外のモジュールをコールするとき直接にはスタックは関与しない.

(2.2) プログラム・スタッキングが必要なモジュールをコールするとき，それをその時点のスタック領域で実行可能なモジュールにしてスタックに入れる.

(3) リターン型： (2)によって譲り受けた制御を元に戻すとき，何からリターンするかによって2つに分けられる.

(3.1) スタック外のモジュールからリターンするとき直接にはスタックは関与しない.

(3.2) スタック中のモジュールからリターンするとき，そのモジュールをスタックから取り出し，そのあとは (3.1) と同じ.

(4) 跳び出し型： 実行中のモジュールから抜け出し二度と同じモジュールに戻ってこない場合である．このとき非活性状態になったものがスタックにとり残される.

以上の考察から，プログラム・スタッキングを適用するためには，対象となるプログラムは制御が二度と戻ってこないような跳び出しを含んではならない．プログラム・スタッキングがコールと対応するリターンではさまれているようなプログラム，例えばサブルーチン，でなければならない.

4.2 プログラムの構成方法

前節で述べたような閉じた構造のプログラムは，一般には大きなもので，これをプログラム・スタッキングするのでは，スタックへのプッシュとポップに時間がかかり**，そのうえ大容量のスタックが必要になる．しかしながら，プログラムからの跳び出しがないように出口でのスタックの処理を確実に行えば，プログラムのうちプログラム変更される部分以外はスタック外で実行されても支障がないから，適当な方法でプログラムを分割し再構成しなおせば上記の不都合を無くすることができる．プログラムの一部を切り出したものを幾つか集めて1つにしたもので，出口がすべてブランチ関係の命令であるようなものを拡張型ハンク (extended hunk) と呼ぶことにする．以下に，プログラム・スタッキングを有効に適用するためにプログラムを再構成し，拡張型ハンクを構成する方法について述べる．ただし，入口はプログラムの先頭で，少なくとも1つの指標レジスタが作業用に使えるとする.

〔アルゴリズム2〕 プログラムの構成方法

(**Fig. 1** (次頁参照))

(1) プログラム・スタッキングを考えないでプログラム P_{r0} を作る (Fig. 1(a)).

(2) P_{r0} をプログラム変更される部分とされない部分に分割する．このとき論理的に分割不可能な部分，例えば複数ワードからなるマクロ命令や計算機の状態を保存する必要のある部分を細分割することはしない．分割されたプログラムを $\sum_{i=1,2,3\cdots} \mathrm{hunk}^i$ とする (Fig. 1(b)).

(3) 各分割 (hunk^i) の先頭にラベル($\boldsymbol{L}^i$)，最後に次の分割へのブランチ命令 ($\boldsymbol{B}\ \boldsymbol{L}^{i+1}$) をつけ加える．プログラムは $\sum_{i=1,2,3\cdots} \boldsymbol{L}^i$: hunk^i; $\boldsymbol{B}\ \boldsymbol{L}^{i+1}$ となる*** (Fig. 1(c)).

---

* スタック内外のプログラムを区別しない

** n ワードのプログラムをプログラム・スタッキングするためのオーバーヘッドは，例えば MELCOM 7700 のばあい

① 全レジスタを退避し，レジスタ経由でプッシュし，スタック・ポインタ・ワードの変更で結果的にポップするとき，約$(50\sim60)\times(3+[n/16])+2\times n$ μ 秒

② スタック・ポインタ・ワードを変更したのちブロック転送を行なって結果的にプッシュするとき，約$10+0.8\times n$ μ 秒

*** ただしプログラムの一番最後の $\boldsymbol{B}\ \boldsymbol{L}^{i+1}$ だけはとり除く

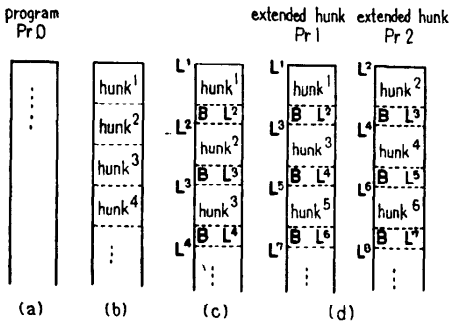

Fig. 1 Construction of two extended hunks
(a) Original program
(b) Partition
(c) Add labels and branch instructions
(d) Construct two extended hunks

(4) プログラム変更される分割同志，されない分割同志をそれぞれ連結し2つの拡張型ハンク P_{r1} と P_{r2} を作る(Fig. 1(d)).

$$P_{r1}=\sum_{i=1,3,5\cdots} \boldsymbol{L}^i : \text{hunk}^i ; \boldsymbol{B}\ \boldsymbol{L}^{i+1}$$

$$P_{r2}=\sum_{i=2,4,6\cdots} \boldsymbol{L}^i : \text{hunk}^i ; \boldsymbol{B}\ \boldsymbol{L}^{i+1}$$

P_{r1} をプログラム変更される拡張型ハンク，P_{r2} を変更されない拡張型ハンクとする．逆のときは P_{r2} の先頭に $\boldsymbol{B}\ \boldsymbol{L}^1$ をつけ加え，これを新たに P_{r1} とし，先の P_{r1} を P_{r2} とする．

(5) P_{r1} を参照する命令をその先頭を基準にした相対番地で参照する命令に書き直す．P_{r1} がスタックに入れられる拡張型ハンク，P_{r2} がスタック外におかれる拡張型ハンクである．

この手順で構成された P_{r1} がプログラム・スタッキングするうえで最小の拡張型ハンクとなっていることは明らかである．このようにして作られた2つの拡張型ハンクは分割を単位として互いに他を呼び出すような形になっており，これはコルーチンと同種の制御構造である．

4.3 拡張型プログラム・スタッキング

前節で述べたようにプログラム・スタッキングされる拡張型ハンクを参照する命令はその先頭からの相対番地で参照する命令になっていなければならない．拡張型ハンクはスタック領域中に動的に配置されるから，スタッキング時に番地修飾を行わないようにするためには，何らかの基準レジスタの役割をするものが必要である．そこでスタックを次のように拡張する．

〔**定義**〕 拡張型スタック

スタック・ポインタ・ワード (**3.1**) に次のワードを付加したスタック．

ハンク・ポインタ (HP): スタックに入っている最新の拡張型ハンクの先頭を指す．

ハンク・ポインタが基準レジスタの役割をするわけである．これによってプログラム・スタッキングは次のように拡張される．

〔**アルゴリズム 3**〕 拡張型プログラム・スタッキング

プログラム・スタッキングされる拡張型ハンクは，ハンク・ポインタを用いてアルゴリズム2によって構成されているとする．

Ⅰ．前 処 理

(1) 戻り番地をスタックに入れる．

(2) HP をスタックに退避する．

(3) HP を更新する．(*i.e.* HP:=SP+1)

(4) 拡張型ハンクをスタックに入れる．

Ⅱ．実 行

(5) 次の命令によってⅠでスタックに入れた拡張型ハンクに制御を移す．

B ***HP**; branch to (HP)

Ⅲ．後 処 理

(6) SP=HP になるまでスタックをポップする*,**.

(7) HP をスタックから回復する*.

(8) 戻り番地を取り出しそこに制御を戻す*.

これが常時再入可能であることは，定理1，定理2と同様にして証明される．ⅠとⅡはコール，Ⅲはリターンの拡張と考えられ，サブルーチン・コールまたはトラップを利用してマクロ命令の形で実現することができる[4]．

拡張型プログラム・スタッキングが拡張型ハンクを動的にスタックにロード，リンクし，また自動的に領域を解放することに着目すれば，これはダイナミック・オーバーレイ[8]の一種と考えることもできる．

5. 適 用

例 1. 駆動関数（ドライバー）

MELCOM 7500*** と PDP 12 間の交信用ハンド

---

* この処理が一般のプロセス処理中に並列スタック中で行われるときには問題がないが，1つのスタックを共有してプロセスが優先度順に処理されているとき，スタック中で行われると，(6)によって実行中のプログラム自身をスタックから取り出すことになり，再入に対する保護が解かれてしまうので，この処理はスタック外で行われる必要がある

** スタックのポップの開始以前には SP>HP となっており，ポップにつれてSP が1つずつ減って SP=HP になったら終了

*** モニタは RBM で多重レベルの割込みに直接結びつけられたプロセス（タスク）を単位とし処理する

ラー[9]の呼び出しのマクロ命令の引数はプログラムに埋め込みになっている．それゆえ幾つかのプロセスから共有して使用される駆動関数（LISP[10]の関数やFORTRANのサブルーチン）ではプログラム変更が必要となる（**Fig. 2**(a)）．

（1） 基本型プログラム・スタッキングの適用

アルゴリズム1にしたがって，ハンドラー・コールの命令，実行時に作った引数，元に戻るためのブランチ命令の順にスタックにプッシュし，指標レジスタとスタック・ポインタを使ってスタック中の基本型ハンクにリンクし，実行が終わって戻ったときスタックからポップする．このプログラムは Fig. 2(b) のようになる．

（2） 拡張型プログラム・スタッキングの適用

サブルーチン・コールとリターンに対応する拡張型プログラム・スタッキング用のマクロ命令をトラップ機構を使って実現し **PUSHL**(push and link), **POPR** (pop and return) とする．**PUSHL** はその実効番地に示される拡張型ハンク（そのサイズは実効番地＋1の内容）に対し，アルゴリズム3のⅠとⅡの処理を行い，**POPR** はアルゴリズム3のⅢの処理を行うものとする．このとき，再入されることを考慮しないで作られた駆動関数 (Fig. 2(a)) をアルゴリズム2にしたがって再構成すると Fig. 2(c) のようになる．ただし，その先頭にプログラム・スタッキングされる拡張型ハンクの番地とそのサイズを示す2ワードがつけられ，この駆動関数をコールする命令は **PUSHL** に書きかえられている．

例 2. トレーサー

この技法の適用例として，1語命令ずつを追跡する典型的なトレーサーをとりあげよう．

プログラムの挙動を同じ計算機のプログラムで追跡（トレース）する方法にはシミュレーターによる方法とプログラム変更によって1命令ずつトレーサー中に埋め込んで実行する方法がある[2]．ここでは複数個のプロセスを同時にしかも追跡中は常時再入可能な形で追跡できるように後者の方法のトレーサーを各プロセスがスタックを各々持っていると仮定して，プログラム・スタッキングの技法を用いて拡張する．

プログラム変更型のトレーサーでは次のことが要求される．

（1） 追跡を要求するプロセスとトレーサーは同じ環境，すなわちプロセス・コントロール・ブロックを共有しなければならない．

（2） ブランチ関係の命令によってトレーサーからの跳び出しが起きてはならない．

第1の点より，埋め込んだ命令を実行するたびに，PSWを交換することはできないから，命令番地をシミュレートし，コンディション・コードとレジスタを別々に退避し回復しなければならない．そのときレジスタが作業用に使えないときがあるので，プログラム・スタッキングにあたっては特別な工夫が必要である．第2の点については，ブランチ先がトレーサー内部にあるようにしておき，ブランチが起きたときは元の命令の実効番地を解釈（例えば，アラナイズ命令[5),11)]によって）することで解決される．以上のことを念頭において，例えば MELCOM 7500[11] 用のトレーサーとして **Fig. 3**（次頁参照）のようなものが構成された*．

この章では2つの例についてプログラム・スタッキングを適用したが，基本型プログラム・スタッキングは第1の例のようなプログラム変更の数が少なく制御構造も単純なばあいに適し，拡張型プログラム・スタッキングはプログラム変更の数が多かったり，制御構造が複雑なばあいに，オーバ・ヘッドが少ない点で適している．第2の例のように非常に複雑なばあいでも拡張型プログラム・スタッキングは少し変形して使用することができる．

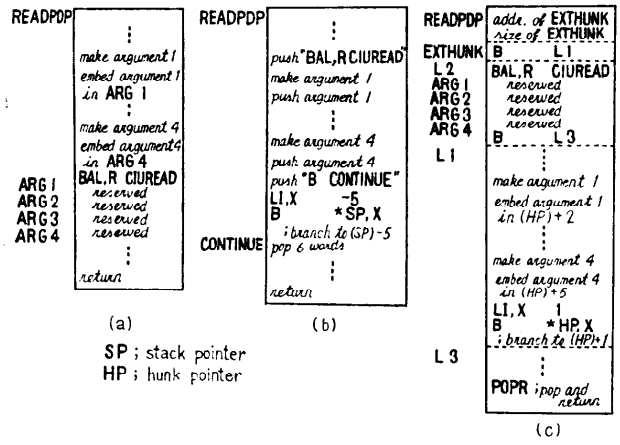

Fig. 2 Program stacking for driver
(a) Original driver
(b) Basic program stacking
(c) Extended program stacking

* これは実現されているもののモデルで，現実には特殊な BAL，実行命令，プログラム・ステータス・ワードの変換，トラップなどの処理が加わっている

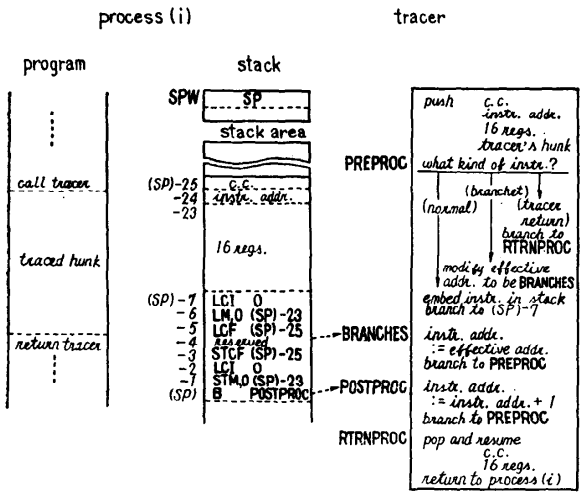

Fig. 3 Tracer for MELCOM 7500

6. むすび

ハードウェアでスタック機能を持つ計算機に対し，スタックまたはその拡張である動的記憶配置機構で実現される並列スタック中にプログラムを入れて実行するプログラム・スタッキングと名づける技法と，その技法を有効に適用するためにプログラムを再構成する方法について述べた．この技法の特長は以下の点に集約される．

（1） 1命令を対象にした実行命令の機能が拡張され複数命令（プログラム）を対象にするものになっている．

（2） スタックまたは動的記憶配置機構の使用がデータを対象にするものからプログラムを対象にするものにまで拡張されている．

スタックまたは動的記憶配置機構は，例えばサブルーチン・リンク・スタックのように，通常はデータの記憶と保護に用いられている．この技法によれば，その中に複数の命令をデータとして入れ（動的にプログラムを作成することになる），記憶と保護をしておき，そののちにその中に跳び込んで命令，プログラムを実行するから，プログラムもデータも外部からは完全に保護されていることになる．

すでにあるサブルーチンやモニタなどのインタフェースのようにハンクが小規模になるときには，基本型プログラム・スタッキングを直接適用することができ，すでにあるサブルーチンやモニタのインタフェースをかえないで見かけ上のインタフェースを変えることができる．この技法はプログラムに再入性，再帰性を持たせるようになることから，LISP 処理系[10] などのモニタ・コールなどの駆動関数の実現のさいに有効性を発揮している．拡張型プログラム・スタッキングで，拡張型ハンクを作る作業を手作業でするには，複雑すぎて虫の巣になることが予想される．これはアセンブラ等の前処理でアルゴリズム2に従って機械的に処理することで解決可能である．

最後に数多くの有益な助言と指針を頂いた電子技術総合研究所古川康一情報システム研究室主任研究官，長田正情報制御研究室長，井上博充システム制御研究室主任研究官，御討論頂いた制御部ロボット・グループの諸氏に厚く感謝致します．

参考文献

1) P. Wegner: Programming Language, Information Structures, and Machine Organization McGraw-Hill, New York, (1968)
2) D. E. Knuth: The art of computer programming, vol. 1, Addison-Wesley, Mass. (1968)
3) 塚本享治：スタックト・プログラミング技法，信学会，計算機研資，EC 74-31，(1974)
4) J. L. Dawson: Combining interpretive code with machine code, Comput. J., vol. 16, No. 3, pp. 216-219, (1973)
5) 三菱電機：MELCOM 7700 システム説明書
6) A. Van Wijngaarden (Ed.) et al.: Report on the algorithmic language ALGOL 68, Num. Math., vol. 14, pp. 79-218, (1969)
7) B. Wegbreit: A generalized compactifying garbage collector, Comput. J., vol. 15, No. 3, pp. 204-208, (1972)
8) R. J. Pankhurt: Program Overlay Techniques, CACM, vol. 11, No. 2, pp. 119-125, (1968)
9) 三菱電機：CIU Handler 説明書，(1973)
10) 塚本享治：Foreground」 LISP Interpreter について，情報処理学会第15回大会予稿集，pp. 653-654，(1974)
11) 三菱電機：MELCOM 7500 システム説明書
12) R. J. Daking et al.: A Mixed Code Approach, Comput. J., vol. 16, No. 3, pp. 219-222, (1973)

付 録

本文と図の中に表われる記号命令を**Table 1** に掲げる.

Table 1

Symbol	Meaning
B	Branch
BAL	Branch and Link
LCF	Load Conditions and Floating Control
LCI	Load Conditions Immediate
LI	Load Immediate
LM	Load Multiple
STCF	Store Conditions and Floating Control
STM	Store Multiple

(昭和50年8月12日受付)
(昭和51年7月12日再受付)