システムソフトウェアと
オペレーティング・システム 90−19
（２００２．６．２８）

動的リンクを利用した実行中プログラムの部分入替え法における状態把握法

山本 淳† 谷口 秀夫‡

†九州大学大学院システム情報科学府 ‡九州大学大学院システム情報科学研究院

〒812-8581 福岡市東区箱崎 6-10-1
TEL：092-642-3867
E-mail：†yamamoto@swlab.csce.kyushu-u.ac.jp ‡tani@csce.kyushu-u.ac.jp

あらまし

動作中のソフトウェアを変更できれば，24 時間無停止でのサービス提供を実現できる．そこで，我々は，実行中プログラムの一部分を入れ替える制御法を提案した．プログラム部分の動的入替えに際しては，当該プログラム部分を実行しているプロセスが存在しないことが必須条件である．このため，各プロセスのプログラム実行状態を把握する必要がある．本稿では，動的リンク機能を利用して，プログラムを部分分割し，動的リンカにおいて各プログラム部分間の呼出と復帰を検出する方式を提案する．提案方式では，状態把握のためのコードを動的リンカに局所化できるため，状態把握を意識したプログラム作成は必要なくなり，本入替え法の利便性が向上する．また，実装と評価により，提案方式によるプログラム部分間の呼出と復帰に要する 1 回あたりの処理オーバヘッドは約 1.1 μ秒であることを示す．

キーワード

オペレーティングシステム，プロセス，プログラム，動的リンク，プログラム入替え

A Method of Grasping the Execution States of Program on Mechanism for Exchanging Parts of Executed Program Using Dynamic Linking

Atsushi YAMAMOTO† and Hideo TANIGUCHI‡

Graduate School of Information Science and Electrical Engineering, Kyushu University

6-10-1 Hakozaki, Higashi-ku, Fukuoka-shi, 812-8581 Japan
TEL：+81-92-642-3867
E-mail：†yamamoto@swlab.csce.kyushu-u.ac.jp ‡tani@csce.kyushu-u.ac.jp

abstract

We have proposed the mechanism for exchanging parts of executed program. Exchanging is possible only when the program part is unused. Therefore the execution states of program must be grasped. In this paper, we suggest a method of grasping the execution states of program using dynamic linking. In this method, a program is divided into load segments at compile-time and every call/return between load segments is detected by dynamic linker at run-time. This method does not force programmers to change application codes for the state grasp. We found that the overhead of this method was about 1.1 micro seconds per each call/return between load segments.

keywords

operating system, process, program, dynamic linking, exchanging program

1　まえがき

常時接続のネットワークの普及により，利用者の計算機システムに対する要望として，24 時間無停止でのサービス提供を要求する声がますます高まっている．一方，計算機システムの利用者の増加に伴い，その利用形態も多様化し，提供サービスの保守作業を頻繁に行う必要が生じている．この 2 つの要件を同時に満たすためには，提供中のサービスを停止させることなく，そのプログラムを変更する方法の確立が不可欠である．

このような背景から，我々は，サービスを提供しているプロセスを終了させることなしに，そのプログラムの一部分を動的に入れ替える方法を提案している [1]．我々の提案法は，従来法 [2, 3] の問題点を解消し，次の大きな 2 つの特徴を持つ．1 つは，サービスプログラムが入替えの契機を意識する必要がない点である．もう 1 つは，入替え対象のプログラム部分が入替え対象でない別のプログラム部分を呼び出している場合も考慮している点である．さらに，複数のプロセス間で共有されたプログラム部分の入替え法を示し [4]，入替え要求から入替え処理を終了するまでの時間（以降，入替え時間と呼ぶ）の定式化 [5]，および入替え時間を短縮化する方法を示した [6]．

一方，プログラム部分の動的入替えに際しては，当該プログラム部分を実行しているプロセスが存在しないことが必須条件である．このため，各プロセスのプログラム実行状態を把握する必要がある．具体的には，各プロセスのプログラム部分の呼出と復帰を検出し，それをオペレーティングシステム（以降，OS と略す）に通知する必要がある．先の研究 [1, 4, 5, 6] では，入替えの制御メカニズムの検討に重点を置いていたため，各プログラム部分間の呼出と復帰を検出する方式については，プログラム作成時にプログラム部分の呼出と復帰を通知するシステムコール（以降，状態把握用システムコールと呼ぶ）をアプリケーション（以降，AP と略す）のソースコードに埋め込む方式を採用していた．しかし，当然のことながら，状態把握を意識したプログラム作成は必要ないことが望ましい．文献 [7] では，動的リンク機能を利用してプログラムを部分分割することにより，動的リンカにおいて各プログラム部分間の呼出と復帰を検出できることを示唆している．しかし，具体的な課題と対処および評価を示してない．

そこで，本稿では，動的リンク機能を利用して，

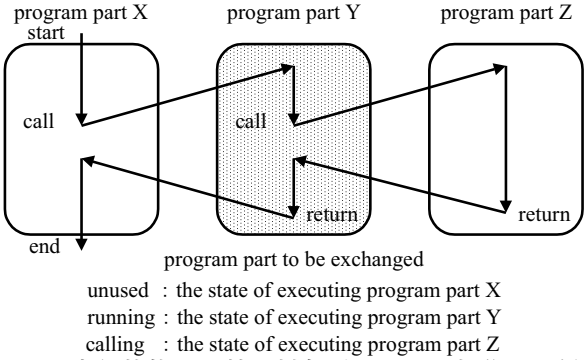

図 1　実行状態と入替え対象プログラム部分の関係

プログラムを部分分割し，動的リンカにおいて各プログラム部分間の呼出と復帰を検出する方式を示す．具体的には，動的にロード / リンクされるロードセグメント（以降，LS と略す）を単位として，プログラムを部分分割する．これにより，入れ替えるプログラム部分の作成を容易にでき，かつ入替え内容の柔軟性も確保できる．また，各プログラム部分間の呼出と復帰の検出に関しては，実行時の動的リンカによる関数シンボルのアドレス解決時の処理メカニズムを利用して，各プログラム部分内で定義された関数の呼出と復帰を検出し，それを状態把握用システムコールにより OS に通知する．これにより，状態把握のためのコードを動的リンカに局所化でき，状態把握を意識したプログラム作成は必要なくなる．また，実装と評価により，状態把握用システムコールを AP のソースコードに埋め込む方式と比較して，状態把握用システムコールを動的リンカに組み込む方式によるプログラム部分間の呼出と復帰に要する 1 回あたりの処理オーバヘッドは約 1.1 μ秒だけ大きいことを示す．

2　プログラム実行状態の分類

図 1 に示すように，プログラム部分の実行状態は 3 つの状態に分類できる．すなわち，入替え対象のプログラム部分に対し，以下の 3 状態である．

（1）未使用：実行する前または実行を終えた状態
（2）走行中：実行中の状態
（3）呼出中：別のプログラム部分を呼び出している状態

図 1 において，入替え対象のプログラム部分はプログラム部分 Y である．未使用の状態とは，入替え対象のプログラム部分を実行する前または実行を終えた状態である．図 1 では，プログラム部分 X の位置を実行している状態である．走行中の状態とは，入替え対象のプログラム部分を実行中の状態である．

図 1 では，プログラム部分 Y の位置を実行している状態である．呼出中の状態とは，入替え対象のプログラム部分が別のプログラム部分を呼び出している状態である．図 1 では，プログラム部分 Z の位置を実行している状態である．3 状態のうち，走行中の状態では，内部変数値が過渡的であるため，当該プログラム部分の入替えは不可能である．

図 1 を見てわかるように，プログラム部分の実行状態は，別のプログラム部分から当該プログラム部分の呼出と復帰時，および当該プログラム部分から別のプログラム部分の呼出と復帰時に遷移する．すなわち，各プログラム部分間の呼出と復帰を検出できれば，各プロセスのプログラム実行状態を把握でき，当該プログラム部分の入替え可否を判別できる．なお，各プログラム部分間の呼出と復帰を検出できた際のプログラム部分の状態管理方式については，文献 [1] と [4] において，フラグ方式とスタック方式を示している．

3 プログラム実行状態の把握法

3.1 動的リンク機能の利用

動的リンク機能を利用する場合，1 つのプログラムは複数の LS から構成される．各 LS はプログラムの実行時に動的ローダによってロードされ，各 LS 間のシンボルによる参照は実行時の動的リンカによって解決される．

動的リンク機能を利用して，プログラムを LS 単位で部分分割したものとし，LS 単位での入替えを可能にすることにより，以下の 3 つの利点がある．

（1）LS 毎にコンパイルできるため，高級言語で入れ替えるプログラム部分を作成することが可能になる．

（2）動的ローダを利用した LS の入替え処理を実現することにより，入れ替えるプログラム部分の大きさに制限がなくなる．

（3）各 LS 内で定義された関数シンボルのアドレス解決時の処理メカニズムを利用することにより，動的リンカにおいて各プログラム部分間の呼出と復帰を検出することが可能になる．

ここでは，(3）を実現する上での課題と対処を述べる．

3.2 課題と対処

図 2 と図 3 に，実行時の動的リンカ（rtld）による関数シンボルのアドレス解決時の処理の流れを示す．図 2 は関数シンボルの 1 回目の参照，図 3 は 2 回目以降の参照の様子である．なお，図 2 と図 3 では，LS（refobj）から LS（defobj）内で定義された関数シンボル（function）を参照する場合を想定している．

関数シンボル（function）の 1 回目の参照は，LS（refobj）が保持する symbol-location 表の当該エントリは未解決であるため，動的リンカ（rtld）に制御が移る．このとき，引数としては refobj-symbol 情報が渡される．動的リンカ（rtld）は，引数として渡された refobj-symbol 情報をもとに，ロードされている全 LS を検索してその定義を見つけ出しアドレス解決する．そして，解決したアドレスを LS（refobj）が保持する symbol-location 表の当該エントリに登録し，その解決したターゲットアドレスにジャンプする．これにより，当該関数シンボル（function）の 2 回目以降の参照は，動的リンカ（rtld）を経由せずに直接当該関数へ制御が移る．

以上のように，LS 内で定義された関数の呼出において，動的リンカを経由するのは，当該関数の 1 回目の呼出時だけであり，2 回目以降の呼出時には動的リンカを経由せずに直接当該関数へ制御が移る．また，いずれの場合も，当該関数からの復帰時には動的リンカを経由しない．

動的リンカにおいて各プログラム部分間の呼出と復帰を検出するためには，各プログラム部分内で定義された関数の呼出と復帰時に必ず動的リンカを経由しなければならない．すなわち，以下の 2 つの課題に対処する必要がある．

（1）2 回目以降の関数呼出時にも動的リンカを経由するようにする．

（2）関数からの復帰時にも動的リンカを経由するようにする．

上記 2 つの課題への対処の様子を図 4 と図 5 に示し，以下に説明する．

2 回目以降の関数呼出時に動的リンカを経由しないのは，図 2 の（3）において，symbol-location 表の当該関数シンボルのエントリに解決したアドレスを登録するためである．したがって，解決したアドレスを symbol-location 表の当該エントリに登録しないようにすることで，必ず関数の呼出時に動的リンカを経由するようにできる．これにより，動的リンカにおいて各プログラム部分の呼出を検出できる．しかし，毎回，関数呼出時に当該関数シンボルのアドレス解決のために，ロードされている全 LS を検索するのは処理オーバヘッドが大きい．そこで，

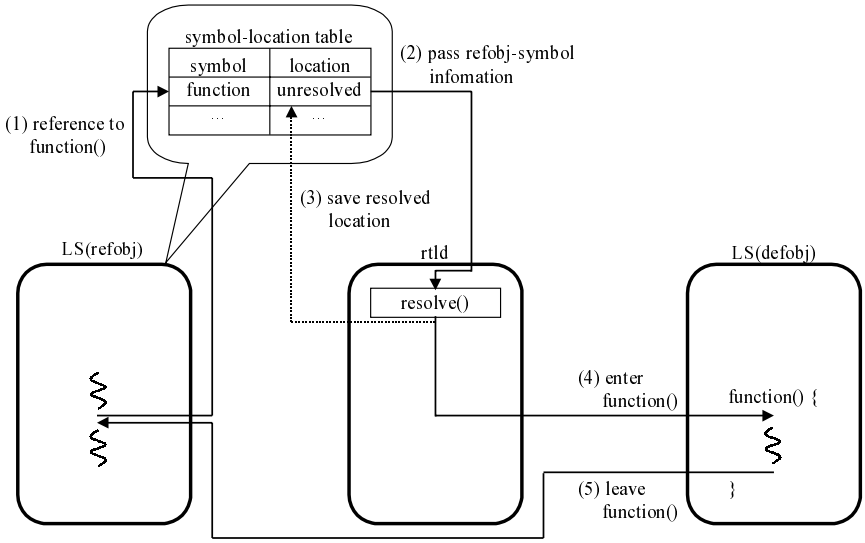

図 2　アドレス解決時の処理流れ（1 回目，対処前）

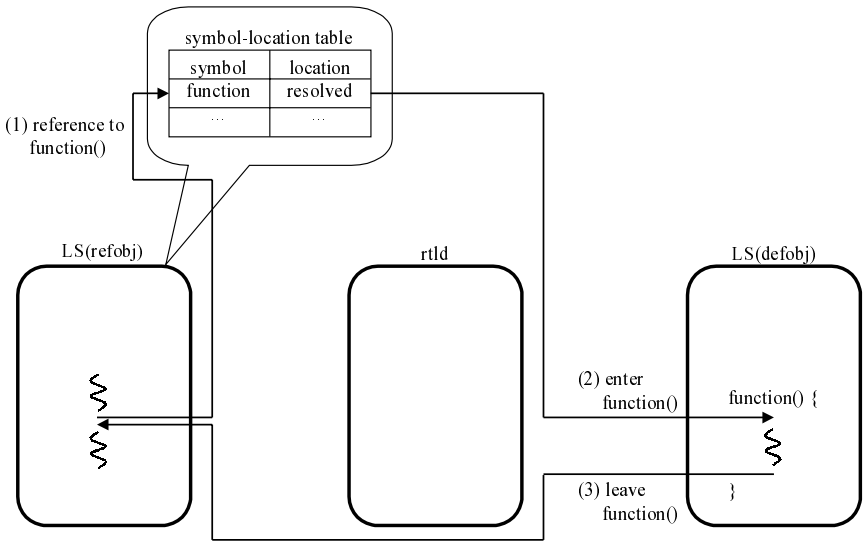

図 3　アドレス解決時の処理流れ（2 回目以降，対処前）

図 4 と図 5 に示すように，関数シンボルの参照と定義の対応を動的リンカで管理することにする．具体的には，(refobj，symbol，defobj，location）の組をセットで管理する．動的リンカは，引数として渡される refobj-symbol 情報から当該関数シンボルがすでにアドレス解決済であるかを対応表から調査する．対応表に登録されていなければ，1 回目の参照であるため，ロードされている全 LS を検索してアドレス解決し，その結果を対応表に登録する（図 4 の（3））．対応表に登録されていれば，2 回目以降の参照であるため，当該の解決済アドレスを使用する（図 5 の（3））．これにより，2 回目以降の関数呼出時の処理オーバヘッドを削減できる．

関数からの復帰時に動的リンカを経由するようにするためには，関数シンボル（function）のターゲットアドレスを call 命令で呼び出すようにすればよい．これにより，動的リンカにおいて各プログラム部分の脱出も検出できる．しかし，この場合には，LS（refobj）内の真のリターンアドレスが上書きされてしまう．このため，真のリターンアドレスを動的リンカで保存する必要がある．LS（defobj）が LS（refobj）にもなりうることを考慮すると，リターンアドレスはプロセス毎にスタックで管理する必要がある．図 4 と図 5 において，push_retaddr() は真のリターンアドレスをスタックにプッシュして保存する処理，pop_retaddr() は真のリターンアドレスを

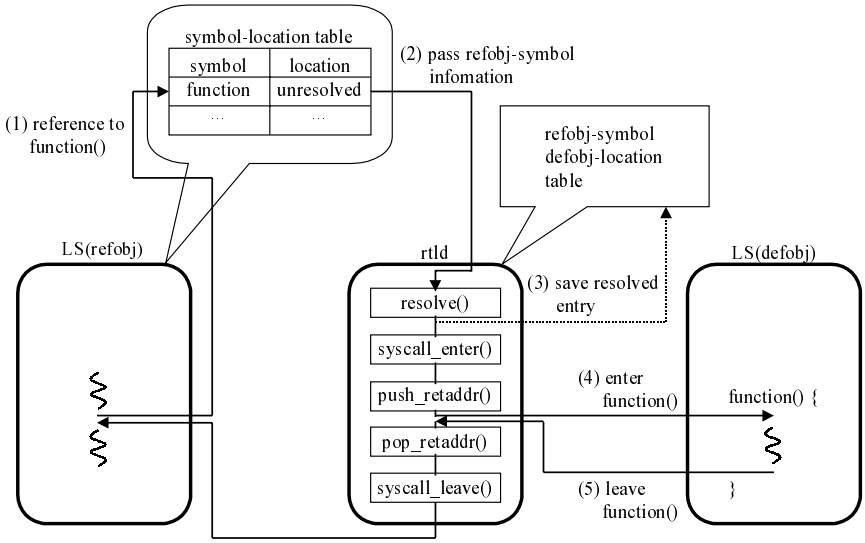

図 4　アドレス解決時の処理流れ（1 回目，対処後）

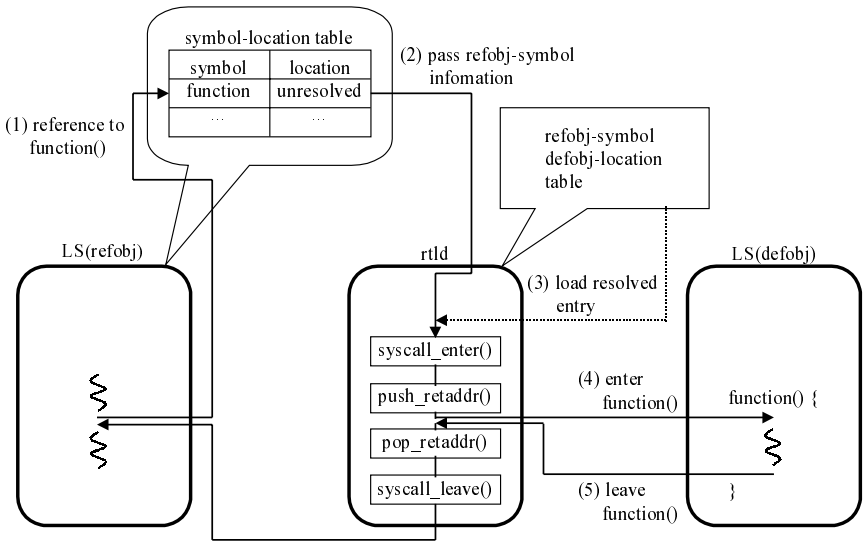

図 5　アドレス解決時の処理流れ（2 回目以降，対処後）

スタックからポップして回復する処理に相当する．

図 6 に，動的リンカ（rtld）の処理手順を示す．最初の条件分岐（resolved entry?）は，先に述べた当該関数シンボルの参照が 1 回目であるか，2 回目以降であるかによる分岐である．syscall_enter() と syscall_leave() は，それぞれ，プログラム部分の呼出と復帰を通知するシステムコールに相当する．push_retaddr() と pop_retaddr() は，先に述べた図 4 と図 5 における意味と同様である．図 6 に示した処理手順により，動的リンカにおいて各プログラム部分間の呼出と復帰を検出できる．すなわち，それを契機として状態把握用システムコールを発行することにより，OS において各プロセスのプログラム実行状態を把握でき，当該プログラム部分の入替え可否を判別できる．

4　評価と考察

4.1　測定条件

図 7 に，測定に用いたテストプログラムの処理の流れを示す．メインプログラム（a.out）は，ライブラリ（libtest）内の関数（test()）の呼出を 10 回繰り返す．ライブラリ関数（test()）は，何も処理を行わず，復帰するのみの関数である．プログラム部分間の呼出と復帰に要する処理時間として，ライブラリ関数（test()）の呼出直前と復帰直後のハードウェアクロックを取得し，その差分のクロック数を測定

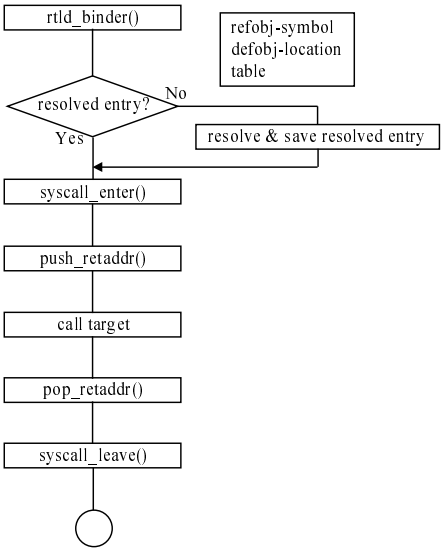

図 6 動的リンカの処理手順

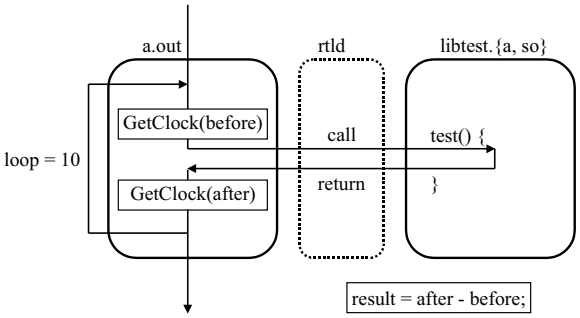

図 7 テストプログラムの処理の流れ

した．プログラムのリンクには，静的リンクと動的リンクがあるため，ライブラリ（libtest）が静的リンクライブラリ（libtest.a）の場合と動的リンクライブラリ（libtest.so）の場合について測定した．

状態把握による処理オーバヘッドを明らかにするため，状態把握なし（default）の場合と状態把握あり（grasp）の場合について測定した．状態把握あり（grasp）の場合の実現方式としては，静的リンクの場合にはライブラリ関数（test()）の最初と最後に状態把握用システムコールを埋め込む方式（injection），動的リンクの場合には動的リンカに状態把握用システムコールを組み込む方式（built-in）とし，両方式のプログラム部分間の呼出と復帰に要する処理時間を比較した．なお，文献 [4] では，状態把握用システムコールの処理オーバヘッドを削減するため，状態把握開始と状態把握終了を宣言するシステムコールを実現し，その有効性を示している．そこで，状態把握あり（grasp）の場合の測定については，それぞれ，状態把握を無効にした場合（grasp_off）と有効にした場合（grasp_on）について行った．測定機

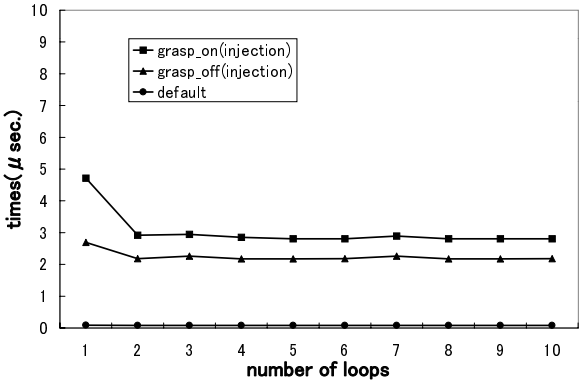

図 8 プログラム部分間の呼出と復帰に要する処理時間（静的リンク）

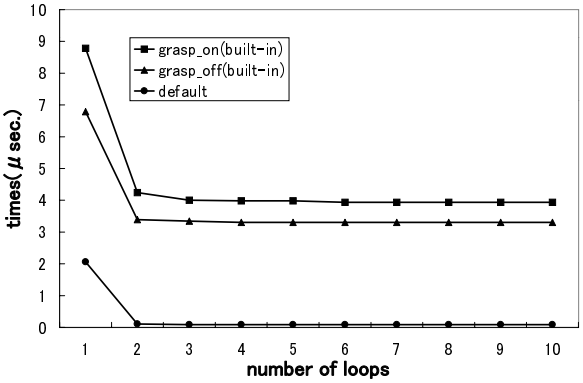

図 9 プログラム部分間の呼出と復帰に要する処理時間（動的リンク）

としては，静的リンクと動的リンクの両方をサポートしている FreeBSD 4.3-RELEASE が走行する計算機（PentiumIII 550MHz）を使用した．

4.2 結果と考察

測定結果を図 8 と図 9 に示す． 図 8 は静的リンクの場合，図 9 は動的リンクの場合のプログラム部分間の呼出と復帰に要する処理時間である．また，表 1 に静的リンクと動的リンクの場合の処理時間比を示す．

まず，図 8 から以下がわかる．

（1）状態把握なしの場合と状態把握ありの場合を比較したとき（2 回目以降），状態把握ありの場合のほうが把握無効時で平均 2.12 μ秒，把握有効時で平均 2.77 μ秒だけ要する処理時間が長い．この差分の処理時間は，そのまま，プログラム部分の呼出と復帰を通知するシステムコールの処理時間に相当すると考えられる．

（2）1 回目よりも 2 回目以降のほうが要する処理時間が短い．これは，命令キャッシュの影響であると考えられる．同じ処理を繰り返し実行するため，命令キャッシュのヒット率が上が

表 1 プログラム部分間の呼出と復帰に要する処理時間比

	static(injection) [μ sec.]		dynamic(built-in) [μ sec.]		dynamic/static [ratio]	
	1st	2nd -	1st	2nd -	1st	2nd -
default	0.09	0.08	2.06	0.09	22.9	1.13
grasp_off	2.70	2.20	6.79	3.32	2.51	1.51
grasp_on	4.72	2.85	8.79	3.99	1.86	1.40

り，要する処理時間が短くなったと推察する．

（3）把握無効時と把握有効時を比較したとき，その差分の処理時間は，2 回目以降は 0.65 μ秒でほぼ一定であるが，1 回目は 2.02 μ秒と 2 回目以降に比べ 1.37 μ秒だけ長い．これは，把握有効時には最初のプログラム部分の呼出の際に，当該プロセスのプログラム実行状態を管理するための管理表を作成する処理を行うためである．

次に，図 9 から以下がわかる．

（1）状態把握なしの場合と状態把握ありの場合を比較したとき（2 回目以降），状態把握ありの場合のほうが把握無効時で平均 3.23 μ秒，把握有効時で平均 3.90 μ秒だけ要する処理時間が長い．この差分の処理時間は，プログラム部分の呼出と復帰を通知するシステムコールの処理時間，および動的リンカの処理内容の変更による処理オーバヘッドであると考えられる．

（2）1 回目よりも 2 回目以降のほうが要する処理時間が短い．これは，先に述べた命令キャッシュの影響と，1 回目には未解決シンボルのアドレス解決処理を行うためである．

（3）把握無効時と把握有効時を比較したとき，その差分の処理時間は，2 回目以降は 0.67 μ秒でほぼ一定であるが，1 回目は 2.00 μ秒と 2 回目以降に比べ 1.33 μ秒だけ長い．これは，先に述べた当該プロセスのプログラム実行状態を管理するための管理表作成処理に要する処理時間であると推察する．

最後に，図 8 と図 9 の比較から以下がわかる．

（1）状態把握なしの場合について比較したとき，動的リンクのほうが静的リンクに比べ，1 回目は 1.97 μ秒，2 回目以降は平均 0.01 μ秒だけ要する処理時間が長い．これは，動的リンクの場合には，1 回目には未解決シンボルのアドレス解決処理を行うためである．2 回目以降には解決済アドレスを使用するが，静的リンクと比較すると間接呼出を 1 回行うため，動的リンクのほうが要する処理時間は長くなる．

（2）状態把握ありの場合について比較したとき（2 回目以降），動的リンクのほうが静的リンクに比べ，把握無効時で平均 1.12 μ秒，把握有効時で平均 1.14 μ秒だけ要する処理時間が長い．これは，プログラムのリンクの違い，および動的リンカの処理内容の変更による処理オーバヘッドであると推察する．プログラムのリンクの違いによる処理オーバヘッドは，把握無効時で平均 0.03 μ秒，把握有効時で平均 0.05 μ秒であった．したがって，残りの平均 1.09 μ秒は，動的リンカの処理内容の変更による処理オーバヘッドである．

（3）状態把握ありの場合について比較したとき（1 回目），動的リンクのほうが静的リンクに比べ，把握無効時で 4.09 μ秒，把握有効時で 4.07 μ秒だけ要する処理時間が長い．これは，未解決シンボルのアドレス解決処理に要する処理時間，および動的リンカの処理内容の変更による処理オーバヘッドである．

以上のことから，built-in 方式は injection 方式に比べて，状態把握を意識したプログラム作成は必要なくなるという反面，プログラム部分間の呼出と復帰に要する処理時間は動的リンカを経由する分だけ長くなるといえる．具体的には，2 回目以降のプログラム部分間の呼出と復帰時には，built-in 方式は injection 方式に比べて約 1.1 μ秒の処理オーバヘッドがあり，injection 方式の約 1.5 倍の処理時間を要することがわかる．この処理オーバヘッドが性能上の問題になるか否かは，各プログラム部分の粒度と各プログラム部分間の移動頻度に依存する．すなわち，各プログラム部分の粒度が約 1.1 μ秒に比べてずっと大きく，各プログラム部分間の移動頻度も低い場合には問題にならないが，その逆の場合には問題になる可能性がある．

5 むすび

実行中プログラムの部分入替え法において，プログラム部分の実行状態を把握する方法について述べた．具体的には，動的リンク機能を利用して，プログラムを LS 単位で部分分割し，動的リンカにおいて各プログラム部分間の呼出と復帰を検出する方式を提案した．提案方式では，状態把握のためのコードを動的リンカに局所化できるため，状態把握を意識したプログラム作成は必要なくなり，本入替え法の利便性が向上する．

また，実装および評価を行い，提案方式によるプログラム部分間の呼出と復帰時の処理オーバヘッドを示した．具体的には，従来方式と比較して，プログラム部分間の呼出と復帰時には，約 1.1 μ秒の処理オーバヘッドがあり，従来方式の約 1.5 倍の処理時間を要することを示した．すなわち，提案方式は従来方式に比べて，状態把握を意識したプログラム作成は必要なくなるという反面，プログラム部分間の呼出と復帰に要する処理時間は動的リンカを経由する分だけ長くなるというトレードオフがあることを示した．

残された課題として，提案方式におけるプログラム部分間の呼出と復帰に要する処理時間の短縮化，および動的ローダを利用した入替え処理方式の実現がある．

参考文献

[1] 谷口秀夫, 伊藤健一, 牛島和夫, “プロセス走行時におけるプログラムの部分入替え法,” 信学論 (D-I), Vol.J78-D-I, No.5, pp.492-499, May 1995.

[2] B.Snead, F.Ho, and B.Engram, “Operating system features real time and fault tolerance,” Computer Design, pp.177-185, Aug. 1984.

[3] 郷原純一, “OS プログラム入替処理とユーザアクセスの競合制御に関する一考察,” 1989 信学春季全大, D-356, March 1989.

[4] 谷口秀夫, 後藤真孝, “走行中のプロセス間で共有されたプログラムの部分入替え法,” 信学論 (D-I), Vol.J80-D-I, No.6, pp.495-504, June 1997.

[5] 谷口秀夫, 後藤真孝, “実行中プログラムの部分入替え法における入替え時間の評価,” 信学論 (D-I), Vol.J82-D-I, No.8, pp.998-1007, Aug. 1999.

[6] 中島雷太, 谷口秀夫, “実行中プログラム部分入替え法における入替え処理時間の短縮,” 情処学論, Vol.41, No.6, pp.1734-1744, June 2000.

[7] 伊藤健一, 箱守聴, 横山和俊, 谷口秀夫, “走行中プログラムの部分入れ替え法,” 情処学コンピュータシステム・シンポジウム論文集, pp.71-78, Nov. 1994.