システムソフトウェアと
オペレーティング・システム 84−15
（２０００．５．２５）

動的再構成をサポートするための Portal 生成法とその評価

小林 良岳　　前川 守

電気通信大学大学院 情報システム学研究科 情報システム設計学専攻

E-mail: {yoshi,maekawa}@maekawa.is.uec.ac.jp

要旨

実行中のプログラムに対し，その一部を動的に差し替えあるいは拡張するにはプログラムの実行状態を監視する手段が必要である．しかし，状態監視のためにソースコードに変更を加えることは，プログラム作成者の負担となり思わぬミスを招く．そこで本稿では，*Portal* と呼ばれる状況監視のためのコードを，関数の呼び出しポイントに対して自動生成する Portal Creator(*PoC*) を実装し評価を行った．*PoC* は *Portal* 生成時に，個々のプログラム部品が内部に保持しているシンボルやデータを管理するためのリストも生成する．また，*Portal* はロックが可能であり，これを OS の機能として提供することにより，プログラムの実行をより細かく制御することが可能になる．最後に実験を行ない関数呼び出しに対する *Portal* 付加による処理速度への影響を測定した．

Portal Creation for Dynamically Reconfigurable Systems and Its Evaluation

Yoshitake Kobayashi　　Mamoru Maekawa

Department of Information Systems Science, Graduate School of Information Systems,
University of Electro-Communications

Abstract

In order to dynamically replace and extend programs while they are running, it is necessary to have some way to monitor the current status of them. However, making modifications to the programs for monitoring is difficult and a burden for programmers and also it paves a way for introducing errors into the programs. In this paper, we introduce Portal Creator(*PoC*) which automatically generates a *Portal* for each function entry point to achieve the above purpose. To control and manage the symbols and data in each program component, *PoC* organizes data structures to be accessible to systems while generating the *Portals*. Further it is possible to control running programs at a finer level by including this feature in operating systems since locking is possible with *Portals*. Finally, we evaluated the effects of *Portals* on performance of program execution.

1　はじめに

システムが動的に変更可能であれば，モバイル環境など周囲の状況が変化する場合でも，動作しているプログラムに対し機能の追加／削除や変更を行うことで対応できる．これをサポートするために OS のツールキット化 [1] を行い，それを利

用してOSのコンポーネントをカーネルに動的に組み込むシステムも考えられている[6]．しかし，動作中のプログラムはそれぞれ部品が何らかの状態を持っているため，プログラムの詳細な実行状態を監視し，差し替え直前までの状態を保持したまま変更可能なシステムを提供する必要がある．

実行状態を監視するシステムの実現法としては，モニタするための専用のシステムコールを作成しているもの[4]，リフレクションを用いるもの[5]，最初からシステムの構成を1つのインタフェイスに基づいて記述するもの[3]などがある．しかし，どの場合もプログラムに対して変更を加えねばならず，その作業量は無視できない．

そこで本稿では，プログラム部品の実行状態を判別するための機構を，プログラムのコンパイル時に，全てのプログラム部品に対して自動的に付加することを提案し，その実装と評価を行う．これによりユーザはソースコードを一切変更することなく，個々のプログラム部品の実行状態を把握することが可能なシステムを作成できる．

以降の議論では，実装を行う対象となるプログラム言語をC言語とし，単に関数といった場合はC言語の関数を指す．またモジュールとは，1つのソースファイルをコンパイルしたものを指し，これらモジュールをリンクすることにより実行可能イメージができる．そして，実行可能イメージをOS上で実行する際の単位をタスクとする．モジュール差し替えの最小単位は，モジュール内部の関数である．また，モジュールの状態とは，そのモジュールで保持している変数の値を指す．そして，差し替えの対象となるモジュールが持つ変数の値を新しいモジュールに対してコピーすることを，状態の再現という．

2 関連研究

2.1 動的再構成の実現法

システム全体を，1つのアブストラクションに基づいて構成しているものとしては，Kea [3]がある．Keaでは，Portalと呼ばれるモジュールへの出入口をカーネル内部に置き全てのサービスを管理し，Portalから呼ばれる部分を，それまで使用していたモジュールとは別のモジュールに変更することによりシステムの再構成が可能である．しかし，変更に際して状態の再現についてはサポートされていない．

タスクの実行を停止せずに，その一部分を変更する手法も提案されている[4]．しかし，差し替えるコードは差し替えの前後で関数の引数および戻り値の型，さらにリターンアドレスも一致している必要がある．また，差し替えの対象となるコードがその時点で実行中であるかどうかを判断するために，フラグを導入しているがそのフラグの加減算のためにシステムコールを追加しているため，ソースコードの変更が必要である．

また，JavaのリフレクションAPIを用いて動的なインタフェイスおよび実装の更新を実現しているものもある[5]．しかし，この方法はオーバヘッドが非常に大きく，通常の処理に対してバイトコード実行時で約5～10倍もの処理時間を要する．

2.2 問題点

機能の拡張や差し換えが可能なシステムに関するこれまでの研究では、状態を持たないタスクやモジュールを差し替える際に状態を持たないことを前提としているものが多い．また，動的な差し替えが可能であったとしても，その実現をソースコードの変更に頼るものがほとんどで，ユーザがプログラムを作成する際の負担が大きい．

システムコールを使用してプログラムの実行状態を監視する手法は，状態を監視するためのコードが他の処理に対して大きなオーバヘッドとなる可能性がある．例えば，Intel PentiumIII 500MHz，メモリ256MBを搭載したPC/AT互換機上にFreeBSD-3.4をインストールし，関数呼び出しとシステムコールをそれぞれ1億回行い平均をとった時，表1に示すような差が生じた．この差が生じる理由として，システムコール実行時の保護レベル切り替えに要する時間が考えられる．

本稿で提案する手法では，ユーザに対してコー

表 1: 関数呼び出しとシステムコールの処理時間

	実行時間 (ns)	比
関数呼び出し	21.8	1
システムコール	826.1	37.89

ドの変更を一切要求せず，モジュールの状態を管理するための情報を自動生成する．これにより，プログラム作成時の負担を軽減できる．また，これらの情報はタスク実行時にそれぞれのタスク内で保持され，OS は必要な時に情報を取得できるため処理時間に対するオーバヘッドを軽減できる．さらに，1 つのモジュールをインタフェイス部分と処理部分に分割し，後にインタフェイスの変更が起こる場合に対応する手段を提供する．

3 Portal

関数の使用状況を明確にしたり，呼び出しに対して制御を行うためにプログラム内部に *Portal* の導入を行なう．

3.1 Portal の構造

Portal の構造を図 1に示す． *Portal* はモジュールの利用状況を把握するために，1 つのカウンタとアクセスフラグを保持する．カウンタは関数を呼び出す際に 1 インクリメントされ，関数の処理が終了しリターンする前に 1 デクリメントされる．カウンタの値が 0 てあるということは，その *Portal* によって管理されている関数の処理部分がその時点で使用されていないことを示す．

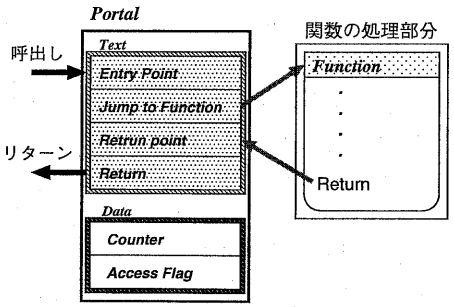

図 1: *Portal* の構造

アクセスフラグは， *Portal* に対して呼び出しがあると 1 にセットされる．これを利用してモジュール差し替え時に状態再現を再実行する必要があるか否かを判断できる．状態を再現する前にアクセスフラグを 0 にリセットしておき，状態再現が終了した時にもう一度アクセスフラグを参照する．この時アクセスフラグが 1 にセットされているということは，状態再現中にモジュールの対して何らかのアクセスが行なわれたことを示し，状態再現を再実行する必要があると判断できる．

また，カウンタとアクセスフラグを利用して，OS 内部に *Portal* を管理する機構を作成すると，以下に示す 4 つの状態の監視を行うことができる．

1. **初期状態**：関数が，まだどこからも呼び出されていない状態
2. **待機状態**：関数を 1 回以上実行し，その実行が終了した後，再度呼び出されるのを待っている状態
3. **実行状態**：関数を実行しているが，他の関数を呼び出していない状態
4. **一時中断状態**：関数内部から他の関数を呼び出して，その終了を待っている状態

3.2 インタフェイスと処理の分離

通常は 1 つのソースファイルから 1 つのモジュールが生成されるが， *Portal* の導入により 2 つのモジュールが生成されるようになる (図 2)．1 つは *Portal* が提供されるモジュールで，これを以降インタフェイスモジュールと呼ぶ．もう 1 つは，関数の処理部分を提供するモジュールで，これを処

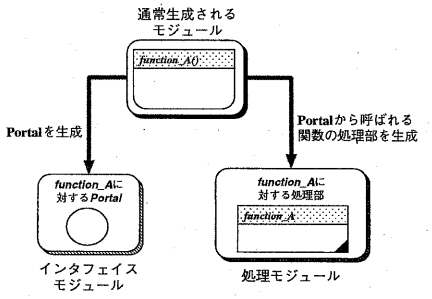

図 2: 2 つに分割されるモジュール

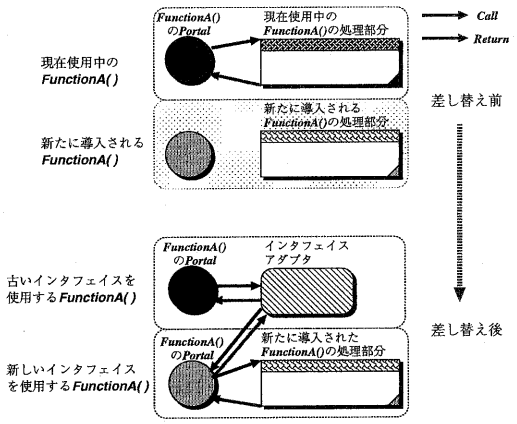

図 3: 新しいインタフェイスモジュールの導入

理モジュールと呼ぶ．*Portal*と関数の処理部分は1対1で対応するので，インタフェイスモジュールと処理モジュールも1対1で対応する．

インタフェイスモジュールと処理モジュールのような分割を行なった理由は，後に関数呼び出しに対する引数の型の変更など，インタフェイスの仕様に変更があった場合に対応するためである．新たに導入される処理モジュールが，既存のインタフェイスモジュールに対応する処理モジュール内部の関数の処理に対して共通のインタフェイスでアクセスできる時，既存のインタフェイスモジュールはそのまま使用することが可能である．しかし，新たに導入される処理モジュールにおいて，呼び出しのインタフェースが変更されている場合，新しいインタフェイスモジュールの導入も一緒に行なわねばならない．

古いインタフェイスモジュールに対しては，図3に示すように，呼び出しのインタフェイスの変換を行うインタフェイスアダプタを提供する必要がある．インタフェイスアダプタでは新しく導入されるインタフェイスモジュールに対して，互換性が保たれてることを保証せねばならない．

3.3 Portal を介した関数呼び出し

*Portal*を介した関数呼び出しをする様子を図4に示す．これは，2つのソースコードAとBをコンパイルして実行されているタスクである．

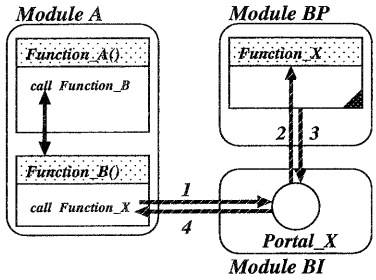

図 4: Portal を介した関数呼び出し

ここで，*Portal*はソースコードBに含まれる関数に対してのみ生成されており，処理モジュール(Module BP)と *Portal*を提供しているインタフェイスモジュール(Module BI)で実現されている．

インタフェイスモジュールが生成されていることにより，Module BP内の関数の処理部分は全てインタフェイスモジュール内の*Portal*を介して呼び出される．しかし，Module Aについては，関数は互いに直接呼び合うことが可能である．図4には示されていないが，インタフェイスモジュールが生成されているモジュールでは同一モジュール内の関数を呼び出す場合も*Portal*を介して呼びされる．

4 Portal Creator (*PoC*)

*Portal*を手作業でソースコードに追加する方法だと，どの部分を監視するか細かく設定できるという点では優れているが，開発する上での負担が増す上，プログラム作成時のミスも問題となる．そこで，*Portal*を自動生成する Portal creator(*PoC*) を開発した．

*PoC*は，当初 gcc が生成するアセンブラコードに対して働くフィルタとして実装された．現在は gcc の wrapper として働き，内部から gcc を呼び出し，インタフェイスモジュールと 処理モジュールの両方を生成する．*PoC*の機能には以下のものがある．

1. *Portal*の生成
2. モジュール内部のシンボル情報の生成

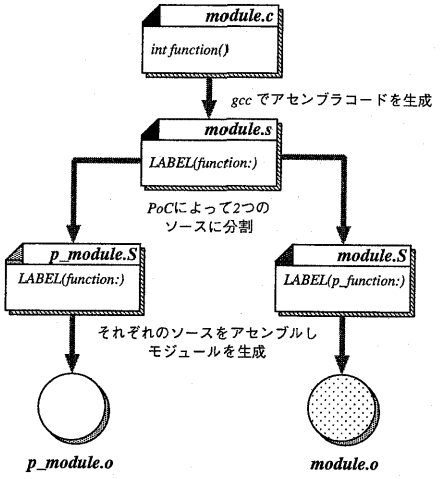

図 5: *Portal* 生成の手順

```
 1: .data
 2: .globl _pv_func1
 3: .globl _ac_func1
 4: _pv_func1:   .long   0
 5: _ac_func1:   .long   0
 6:
 7: .text
 8: .globl func1
 9: .globl retp_func1
10: func1:
11:     jmp     1f
12: 1:  movl    $1, _ac_func1
13:     incl    _pv_func1
14:     jmp     entp_func1
15: retp_func1:
16:     decl    _pv_func1
17:     ret
```

図 6: func1() に対する *Portal*

3. モジュール内部から参照される関数やデータへのポインタの生成（オプション）

4.1 Portal の自動生成

どの関数に *Portal* を生成するかに関しては以下の 2 つの方針をユーザが選択できる．

- 全ての関数：モジュール内部でのみ使用される private 関数を含む全ての関数に対して生成
  モジュールに含まれる関数を単位として差し換えを行いたい場合に有効である．
- Public 関数：モジュールの外部に公開している public 関数に対してのみ生成
  モジュール単位で差し換えを行う場合に有効である．また，*Portal* の数が少なくなることで性能面でも有利であると考えられる．

また，生成は以下の手順で行なわれる (図 5)．

1. ソースコードをコンパイルする時，直接オブジェクトを生成せずに，アセンブラコードを生成する．
2. 生成されたアセンブラコードを基に，*PoC* は関数のエントリおよび，シンボル情報を全てとりだす．その後，処理モジュール上に存在する関数のエントリをインタフェイスモジュールからの呼び出しに対応するように変更するとともに，処理モジュールにある `ret` 命令をインタフェイスモジュールへの `jmp` 命令に変換する．
3. 関数のエントリを基に *Portal* を生成し，さらに全てのシンボルを連結できるように，関数とデータそれぞれのシンボルへのポインタを生成する．
4. 最後に，生成された 2 本のアセンブラコードをアセンブルし，インタフェイスモジュールと処理モジュールをそれぞれ生成する．

表 2: 使用言語による *Portal* 処理時間の違い

使用言語	処理時間 (ns)	比
アセンブラ	38.1	1
C 言語	70.4	1.84

C 言語の `func1()` という名前の関数に対し，Intel Pentium プロセッサ用に生成される *Portal* のアセンブラコードを図 6に示す[1]．

アセンブラを用いて *Portal* を作成した理由としては，性能によるところが大きい．C 言語を用いて *Portal* に相当するものを作成し，それぞれを 1 億回呼び出してその実行時間の平均をとると，表 2に示すように処理時間が約 1.8 倍にもなる．この差が生じる理由としては，C 言語で作成した

[1] 簡単のため，データの整列などに関する部分は省略してある．

場合 *Portal* を呼び出すためにスタックを操作し，もう一度処理部分を呼び出すためにスタックの操作を行なうということが考えられる．

4.2 Portal のロックおよび解除

プログラム部品を動的に差し替えるにあたり，プログラム実行の流れを制御する必要がある．そこで *Portal* には，関数の呼び出しを一時的に禁止したり条件付きで許可するための部分があり，ここをロックすることが可能である．ロックの実行は，*Portal* のみで提供されるものではなく，OS 内部に *Portal* 管理機構を実装することによって実現される．具体的には，図6に示したアセンブラコードの 11 行目に相当する部分を書き換えることによって実現されている．

これと同様の方法で，*Portal* を削除することも可能である．*Portal* を削除することにより，それ以降差し換えを行うことは不可能になるが，性能を向上させることが可能となる．

4.3 シンボル情報の生成

PoC は *Portal* を生成するだけでなく，モジュール内部に含まれている *Portal* や関数の処理部分，データの１つ１つに対し，これらを管理するための情報を生成する．この情報をシンボル情報と呼ぶ．シンボル情報は実行時には全てユーザタスクのデータ領域に置かれ，カーネルは必要なときに情報を参照することができる．

また，カーネルがシンボル情報を参照できるように *PoC* は１つの実行イメージに対しシンボル情報位置を登録するための関数を生成し実行可能イメージに組み込む．この関数はタスク開始時に１度だけ呼ばれる．

4.4 Portal の生成に関する問題点

Portal は呼ばれた関数は必ず戻る事が保証されているという前提で作成されている．これは，プログラミングをする上での制約となる．例えばC言語では `setjump()` と `longjump()` を使用した大域脱出が利用できなくなる．プログラム作成者は常に呼ばれた関数が戻るか，もしくはタスクが終了するかを考えながらプログラムを作成しなければならない．

また，現在の実装ではC言語で `static` 定義された関数へ正しい対応はできない．通常C言語で `static` 定義された関数は，リンク時に外部のモジュールに対して名前を公開せず，その存在を private なものとして扱う．もしこのことを意識せずに *Portal* を生成しインタフェイスモジュールと処理モジュールに分割してしまうと，インタフェイスモジュールと処理モジュール間で正常にリンクを行なうことができなくなる．

そこで，この問題を回避するために，*Portal* 生成時に private な関数名をモジュール外部から参照可能な public な名前に変換するという方法をとっている．ただし，システムが名前を管理する上では変換前の名前を使用する．これにより，リンクは正常に行なわれるようになるが関数名の衝突が起きる可能性がある．しかし，これ以上のサポートをするよりも関数名の衝突が起きないように工夫する方が現実的であるという楽観的な方法をとった．

5 評価および考察

Portal 付加によるオーバヘッドについて調査するため，実験を行ない性能を評価した．評価環境は Intel PentiumIII 500MHz，メモリ 256MB で FreeBSD-3.4 がインストールされている PC/AT 互換機を用いた．

まず，図7に示す構成のプログラムで実測した．測定は，メインループでのループ回数を１億回とし，１回の呼び出しにかかる平均時間を求めた．また，メインループから呼ばれる Function1 には，引数も戻り値も持たず何も行なわない関数 (`nullfunc()`) と，与えられた引数を入れ換える関数 (`swap()`) の２種類を作成し測定した (表3)．

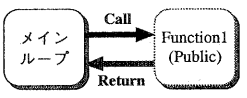

図 7: 測定に用いたプログラムの構造 (1)

表 3: *Portal* 生成による処理時間の比較 (1)

関数	*Portal* 有無	処理時間 (*ns*)	比
`nullfunc()`	なし	22.4	1
	あり	38.1	1.70
`swap()`	なし	43.5	1
	あり	53.1	1.22

表 3では，`nullfunc()` に対して *Portal* を生成した場合，70% ものオーバヘッドが生じている．しかし，処理に何も記述がされない関数によって構成されるプログラムは現実的にはほとんど存在せず関数内部には何らかの処理が書かれるものである．`swap()` 関数では，22% にまでオーバヘッドが減少している．

次に，図 8に示す構成のプログラムを用いて実測した．Function1 では Function2 への関数呼び出しが付加されるものの，それ以外の関数の処理内容は，Function1 と Function2 で統一した．関数には図 7の実験で用いた，`nullfunc()` と `swap()` の 2 種類を使用し，このそれぞれに対し，*Portal* 生成を以下に示つの 3 通りの方針にしたがって行い測定した．この結果を表 4に示す．

1. *Portal* の生成を行わない（これを基準とする）
2. 全ての関数に対して *Portal* を生成
3. public 関数についてのみ *Portal* を生成

表 4の `nullfunc()` では，全ての関数に *Portal* を生成した場合，約 150% のオーバヘッドがある．しかし，`swap()` では内部の処理があることで，オーバヘッドが相対的に 14% にまで抑えられている．また，`swap()` を実行するプログラムで private 関数に対して *Portal* の生成を行わない場合は，全ての関数に *Portal* を生成した場合と比べて 7% しか速くならず，大きな性能の向上があったとはいえない．しかし，関数内部で処理する内容

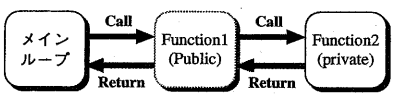

図 8: 測定に用いたプログラムの構造 (2)

表 4: *Portal* 生成による処理時間の比較 (2)

関数	方針	処理時間 (*ns*)	比
`nullfunc()`	1	34.8	1
	2	86.4	2.48
	3	63.3	1.81
`swap()`	1	89.0	1
	2	101.7	1.14
	3	95.1	1.07

表 5: *Portal* の実行にかかるクロック数

命令	クロック数	*Portal* 内の命令数
`jmp`	1	2
`movl`	1	1
`incl`	3	1
`decl`	3	1
合計 (clocks)		9

によって，この実験における 7% がどのように影響するかは未知であるため，今後も検討が必要である．

Intel 486 プロセッサのマニュアル [2] によると，*Portal* の内部で用いているそれぞれの命令に要するクロック数は表 5のとおりである．ただし，`ret` 命令に関しては，処理モジュール内部に書かれていたものをインタフェイスモジュールに移したものであるため計算には入れず，代りに処理モジュールからインタフェイスモジュールへの移行に利用される `jmp` 命令を *Portal* によって追加される処理と考える．

よって，*Portal* の追加により *Portal* 1 つあたり 9 クロック増える．関数内部の処理にかかるクロック数を c とすると，関数の処理に対するオーバヘッドの理論値は $overhead = 9/c$ となる．*Portal* を付加しないで `nullfunc()` を呼び出す場合でも最低 13 クロックを要することから，C 言語で作成されたプログラムに *Portal* を付加したときのオーバヘッドは 1 つの関数あたり最高で 69% になると考えられ，関数の処理にかかるクロック数を横軸とすると，オーバヘッドは図 9のように

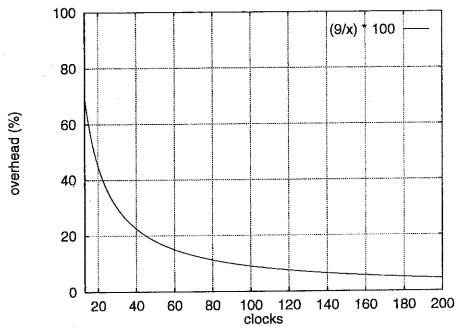

図 9: *Portal*追加によるオーバヘッド

表 6: Emacs の make 時のシステム CPU 時間

Portal 生成の方針	時間 (s)	比
Portal なし	4.14	1
全ての関数に生成	5.23	1.26
public な関数のみ生成	4.99	1.21

なる.

*Portal*によるオーバヘッドは，プログラムの書き方に完全に依存する．プログラムする際に関数の粒度を細かくすればするほど，それだけ 1 関数あたりの命令数は減少するため，処理時間に対するオーバヘッドは増加する．よって，一般的な平均をとることは難しい．しかし以上の実験の結果より *Portal*付加によるオーバヘッドは，最近の CPU の性能の向上とシステム構成への柔軟化という利点を考えた時には，十分に実用に耐え得る範囲だと考えられる．

最後に，参考までに FreeBSD のカーネルに対し *Portal*の生成を 3 つの方針に基づいて行い，Emacs-20.5 の make を行なった時の CPU 使用時間を測定した．この時のシステム CPU 使用時間を表 6に示す．ユーザ CPU 時間については，どの場合も約 67.7(s) になった．また，*Portal*を含むカーネルを生成する時に変更した部分は，Makefile のみでありカーネルのソースコードには変更を加えていない．

6 まとめ

本稿では，関数呼び出しの際に *Portal*を導入することによって関数の使用状況を判断し，後のプログラム変更要求に応える手法を示した．また，処理とインタフェイスの分離を行なうことで，関数呼び出しのインタフェイスに変更があった場合にも対応可能となることを示した．

*Portal*はコンパイル時に自動生成され，プログラマの負担を軽減することが可能となった．*Portal*生成は，全ての関数に対して生成する方法と，public 関数に対してのみ生成する方法の 2 つを選ぶことができる．

*Portal*を追加したプログラムの評価を行ない，*Portal*によるシステムへのオーバヘッドは，実用に耐える程度であることが示された．

7 最後に

我々は，周囲の状態が変化にユーザの要求に対じる形で動的適応可能な OS「Aya」を開発中である．*PoC*はその開発にあたり「ソースコードに面倒なものを持ち込みたくない」という要求から生まれたものである．

参考文献

[1] Bryan Ford, Kevin Van Maren, Jay Lepreau, Stephen Clawson, Bart Robinson, and Jeff Turner. The Flux OS Toolkit: Reusable Components for OS Implementation. In *Proceedings of the Sixth IEEE Workshop on Hot Topics in Operating Systems*, May 1997.

[2] Intel. Intel486^{TM} *Microprocessor Famiry Programmer's Reference Manual*, 1992.

[3] A. C. Veitch and N. C. Hutchinson. Kea – A Dynamically Extensible and Configurable Operating System Kernel. In *Proceedings of the Third Conference on Configurable Distributed Systems*, 1996.

[4] 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. 電子情報通信学会論文誌, Vol. J78-D-I, No. 5, pp. 492–499, May 1995.

[5] 森大志, 前川守. 実行時の状態再現を伴うソフトウェアモジュールの動的差し替え機構. 第 55 回 (平成 9 年度後期) 全国大会講演論文集 (1), pp. 224–225. 情報処理学会, September 1997.

[6] 盛合敏, 徳田英幸. 次世代 OS のためのマイクロカーネルトレイアーキテクチャ. 情報処理学会研究報告, No. 97-OS-75, pp. 1–6, 1997.