リフレクションを用いた
拡張可能なオペレーティングシステムの構成法

岡坂 史紀[†]　　上野 球[†]

アプリケーション，サーバー，カーネルの開発に統一のプログラミング環境を提供することによって容易に拡張可能なコンポーネントオペレーティングシステムを構築する手法として，システムコールおよびアップコールを C++言語の純粋仮想関数呼出しに基づいて定義し，それらをオペレーティングシステムのカーネルがリフレクションを使って処理する手法を提案する．また，この手法によって，ファイルサブシステムや TCP/IP プロトコルスタックを特定のオペレーティングシステムに依存しないアプリケーションプログラムとして開発した事例について報告する．

An Extensible Operating System Architecture Using Reflection

Shiki Okasaka[†]　　Kyu Ueno[†]

We propose an extensible component operating system architecture in which an operating system kernel uses reflection to process C++ pure virtual function based system calls and upcalls to provide a unified programming environment for application, server, and kernel development. We found that we could even develop file subsystems and a TCP/IP protocol stack on an existing operating system based on this architecture.

1. はじめに

ソフトウェア IC という概念 [9]が提案されてから 20 年が経過して，今日ではアプリケーションソフトウェアをさまざまな部品(コンポーネントソフトウェア)を組み合わせて構築するという手法がオフィススイーツからサーバーサイドまで分野を問わず広く利用されるようになっている [2, 21]．オペレーティングシステムの構成においても，モノリシックカーネルをマイクロカーネルとサーバープロセスとして実装したコンポーネントに分離して構成するという研究が詳細に行われてきた [5, 19] ．

しかしながら，ソフトウェアの部品化に関する取り組みは各々の領域で別々に発展してきたため，あるコンポーネントフレームワーク用に開発したコンポーネントソフトウェアを別のコンポーネントフレームワークでは利用できないと言った不都合も生まれた．そこで全領域のコンポーネントソフトウェアを統一的にオペレーティングシステムのレベルで管理可能にするコンポーネントオペレーティングシステムの開発の可能性について提案がなされている [18]．

筆者らは最新の C++言語仕様に基づいた処理系を用いてオープンソースベースで es コンポーネントオペレーティングシステム [30]の開発を進めている．es オペレーティングシステムでは，C++言語レベルで見た場合，すべてのシステムコールがインターフェイスを表現した抽象クラスの純粋仮想関数呼出しとして実現されている．Unix で図 1 のように記述していたプログラムは，es では図 2 のようなプログラムになる．

```
int fd;
char buf[SIZE];
write(fd, buf, sizeof buf);
```

図 1　Unix のシステムコール

† 任天堂株式会社
Nintendo Co., Ltd.

```
IStream* stream;
char buf[SIZE];
stream->write(buf, sizeof buf);
```

図 2　es のシステムコール

es オペレーティングシステムのカーネルは，インターフェイスのメタ情報を登録する機能を備えており，アプリケーションプログラムから利用可能なシステムコールインターフェイスを動的に追加，拡張することを可能にしている．追加したインターフェイスに対応したソフトウェアコンポーネントを実行することでアプリケーションプログラムからは利用可能なシステムコールインターフェイス自体が動的に追加，拡張されていくように見える．es カーネルは，システムコール，アップコール [7]，遠隔手続き呼出し(RPC)を登録されたメタ情報を基にリフレクションを用いて解釈しながら処理を進めることで，サーバーやアプリケーションにインターフェイスごとに別々のスタブやランタイムをリンクする必要をなくし，柔軟なコンポーネントソフトウェアの開発とコンフィギュレーションを可能にしている．

本稿では，近年の C++言語の発展を踏まえて，コンポーネントソフトウェアアーキテクチャを改めて見直し，新しいコンポーネントオペレーティングシステムのカーネルが備えるべき機能について提案を行う．2 章では，コンポーネントオペレーティングシステムに関する関連研究についてまとめる．3 章では，es オペレーティングシステムのプログラミング環境について述べる．4 章では，es カーネルのシステムコール，アップコール，ローカルな RPC の処理について述べる．5 章では，現在の es オペレーティングシステムの実装について述べる．最後にまとめと今後の課題を述べる．

2. 関連研究

ソフトウェアコンポーネントを組み合わせてアプリケーションプログラムを構築する手法が一般化していく中で，Mendelsohn はオペレーティングシステムレベルのソフトウェアコンポーネントの支援が十分ではないことを示し，全領域のコンポーネントソフトウェアを統一的にオペレーティングシステムのレベルで管理する新しいコンポーネントオペレーティングシステムの開発の可能性について提案した [18]．

コンポーネントオペレーティングシステムでは，新しいコンポーネントをオペレーティングシステムに追加する際に，予めオペレーティングシステムによって規定されているファイルやディレクトリといった限定的なアブストラクションに強引に実装を合わせなければならなかったり，あるいは `ioctl` システムコールのような汎用的なシステムコールを経由してオペレーティングシステムのアブストラクションとはおよそ無関係の機能を実装し，それらをラップしたより扱いやすいユーザーライブラリを別途実装しなければならなかったりするようなことは回避できていなければならない．アプリケーション，サーバー，カーネルの開発に統一のプログラミング環境を提供するという Multics[8]の時代からオペレーティングシステムに繰り返し求められている目標 [10, 22]の実現も，コンポーネントオペレーティングシステムに求められている機能のひとつである．

Spin[5]や Spring[19]など従来の多くのマイクロカーネルベースのオペレーティングシステムが非常に複雑化したモノリシックカーネルをマイクロカーネルとユーザー空間の複数のサーバープロセスに分割して再構築するという閉じたコンテキストの中で研究されてきたのに対して，コンポーネントオペレーティングシステムでは全領域のアプリケーションのソフトウェアコンポーネントをカーネルレベルで支援することが念頭におかれている．そのため，コンポーネントオペレーティングシステムには，ソフトウェアのパッケージング，コンフィギュレーションに関してより柔軟で堅牢なオペレーティングシステムレベルの支援が求められている [14]．

また，従来の多くのマイクロカーネルベースのシステムでは，RPC を実現するためにカーネルはメッセージパッシングをサポートするだけでメッセージの内容についてはケイパビリティの受渡し程度しか関与せず，引数の受渡しについてはクライアントおよびサーバーにリンクしたスタブが行うという方法が取られてきた．そのため，カーネル空間にコンポーネントを配置した場合でも，コンポーネントを呼び出すためにいちいちスタブを経由させなければならないと言うようなコストが発生する場合があった．

es カーネルでは，システムコール，アップコール，RPC に関して，プロセッサの制御がユーザーモードか

らカーネルモードに変わるとき(システムコール)，カーネルモードからユーザーモードに変わるとき(アップコール)といった横断的な側面 [17]に着目して処理を集約し，インストールされたメタデータを基にカーネルがリフレクションを使って引数の受渡しや範囲チェックなどをメタレベルで行うようにしている．そのため，それぞれのインターフェイスを実装するクラスでは，それがカーネル用のプログラムであるのかユーザーレベル用のプログラムであるのかを区別する必要がない．またリフレクションの導入によって，すべてのコンポーネント間の動作のロギングを行ったり，あるいはコンポーネント間のアクティビティに割り込んで横断的な処理を追加したりといったことが容易に実現できるようになっている．

コンポーネントソフトウェアのアーキテクチャとしては CORBA，JavaBeans [27]などが開発されてきているが，そのような中でコンポーネントオブジェクトモデル(COM) [6]は C++言語において非静的なデータメンバを持たない仮想基底クラスのランタイムオブジェクトモデルは単純な関数のテーブルで表現できる [16] という特性に基づいた概念的には非常にシンプルな設計がなされている．

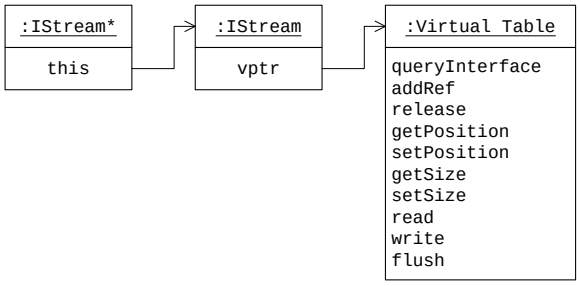

図 3　インターフェイスポインタの
オブジェクトモデル

図 3 に COM におけるインターフェイスポインタのランタイムオブジェクトモデルを示す．インターフェイスポインタはインターフェイスのすべてのメンバ関数のポインタからなるテーブル(仮想表)へのポインタ(vptr)へのポインタとして表現される．C++言語での *this*->write(*buf*, *len*);という記述は，C 言語での (**this*)->write(*this*, *buf*, *len*);という記述と等価である．重要なのは，インターフェイスポインタの呼出し側から透過的に仮想表のメンバ関数をフックして RPC に置き換えるといったことが行える点で，コンポーネントソフトウェア間のバイナリインターフェイスを設計する上で扱いやすいオブジェクトモデルになっている．

このようにシンプルな COM のフレームワークはオペレーティングシステムの構成法にも影響を与え，Flux OSKit [12]では既存のオペレーティングシステムのコードを比較的容易に COM コンポーネント化できることが示された．さらに，Rialto カーネル [10]では，相互排除ロックと条件変数に関するシステムコールを除いたすべてのシステムコールと RPC を COM に基づいて定義し直し，アプリケーションとオペレーティングシステムに共通のコンポーネントソフトウェアフレームワークを適用できることを示した．

しかし，C++言語との相性の良さを期待された COM も OSKit や MMLite [13]など初期の COM コンポーネントオペレーティングシステムでは適切な C++言語の処理系およびカーネルレベルで利用可能なランタイムがなく，実際には C 言語で関数テーブルを直接記述しており生産性という面では明らかに非効率であった．また COM は C++言語の発展途中に生まれた規格のため，C++言語レベルの例外を統合できていないといった不完全な面も残っている．

現在では，ISO による C++言語の標準化 [29]，C++ランタイムの実装に必要な C++アプリケーションバイナリインターフェイス標準(C++ ABI) [23]の策定，gcc 4.0 [25]などこれらの標準に準拠したコンパイラの完成といった C++言語によるオペレーティングシステムの記述に必要な用件が十分に整ってきている．このような C++言語の発展もあって，es オペレーティングシステムではシステム記述言語として標準 C++を使用し，例外やスレッドローカルストレージ(TLS) [11]といった言語機能も含めてアプリケーションと同じ環境でシステムを記述することができるようになっている．

3. esプログラミング環境

es のインターフェイスは COM と同様に C++言語の非静的なデータメンバを持たない仮想基底クラスに基づいて定義している．C++言語から見たときのインターフェイスの例を図 4 に示す．es では，メンバ関数の戻り値がエラーコードを示す整数型でなければいけな

いと言う COM の制限を排除し，C++言語の例外をインターフェイスのメンバ関数呼出しでも使用できるようにしている．そのため getter/setter メソッドなどを自然に記述することが可能になっている．

```
const Guid IID_IAudioFormat =
{
   0x839febda, 0x25dc, 0x11db,
   { 0x9c, 0x02, 0x00, 0x09, 0xbf, 0x00, 0x00, 0x01 }
};

class IAudioFormat : public IInterface
{
public:
  virtual unsigned char getBitsPerSample() = 0;
  virtual unsigned char getChannels() = 0;
  virtual unsigned short getSamplingRate() = 0;
  virtual void setBitsPerSample(unsigned char bits) = 0;
  virtual void setChannels(unsigned char channels) = 0;
  virtual void setSamplingRate(unsigned short rate) = 0;
  static const Guid& interfaceID()
  {
    return IID_IAudioFormat;
  }
};
```

図 4　インターフェイスの例

煩雑になりがちであった COM のプログラミングはスマートポインタによって簡略化できるようになってきている．スマートポインタを使用していない図 5 に示したプログラムは，スマートポインタを使用すると図 6 のように書き直すことができる．

```
void setChannels(IStream* stream, int channels)
{
  IAudioFormat* dsp;
  stream->queryInterface(IID_IAudioFormat,
    reinterpret_cast<void**>(&dsp));
  dsp->setChannels(channels);
  dsp->release();  // この行がないとメモリリークする
}
```

図 5　スマートポインタを使えない場合

```
void setChannels(Handle<IStream> stream, int channels)
{
  Handle<IAudioFormat> dsp = stream;
  dsp->setChannels(channels);
}
```

図 6　スマートポインタを使った場合

es の `Handle` スマートポインタクラスは，参照数の管理だけでなく，あるインターフェイスポインタから別のタイプのインターフェイスポインタを取得するための `queryInterface` の呼出しもコンストラクタの中で管理している．

図 7 に `Handle` スマートポインタの実装の一部を示す．`Handle` スマートポインタで使用しているメンバテンプレートは，C++の言語仕様に取り込まれたのが 1994 年，C++処理系が対応しはじめたのはさらに数年あとのことで，COM のプログラムではその間，図 6 のようなスタイルを利用することができなかった．

```
template<class I>
class Handle
{
  template<class J>
  Handle(const Handle<J>& comptr)
  {
    IInterface* u = comptr.get();
    if (!u)
    {
      object = 0;
    }
    else
    {
      void* ptr;
      if (!u->queryInterface(I::interfaceID(), &ptr))
      {
        ptr = 0;
      }
      object = static_cast<I*>(ptr);
    }
  }
}
```

図 7　スマートポインタクラスの実装(抜粋)

さて，es ではインターフェイスをカーネルがリフレクションを使用して処理するためのメタ情報を生成できるように，実際にはインターフェイス定義言語(esidl)によってインターフェイスを定義し，esidl コンパイラによって C++のヘッダーファイルとインターフェイスのバイナリ形式のメタデータを生成するようにしている．メタデータには，インターフェイスが提供するメソッド数，メソッドの引数と戻り値の型や属性などが含まれている．esidl の言語仕様は COM の msidl と同様に DCE IDL[24]の文法を拡張したものとなっている．図 8 に es のインターフェイスの定義例を示す．

```
[object, uuid(0328e552-25db-11db-9c02-0009bf000001)]
interface IProcess : IInterface
{
   void kill();
   void start();
   void start(IFile* file);
   void start(IFile* file, [in] char* arguments);
   int wait();
   int getExitValue();
   boolean hasExited();
   void setRoot(IContext* root);
   void setIn(IStream* input);
   void setOut(IStream* output);
   void setError(IStream* error);
}
```

図 8　インターフェイス定義例

4. esカーネルの構成

4.1. システムコールインターフェイス

es オペレーティングシステムでは，C++言語レベルで見た場合，すべてのシステムコールがインターフェイスを表現した抽象クラスの純粋仮想関数呼出しとして実現されている．すなわち，図 3 で示した仮想表から参照しているメンバ関数がカーネルへのトラップと結びついている．システムコールインターフェイスはカーネルへのトラップを受けて，どのカーネルオブジェクトに対してどのメンバ関数が呼び出されたのかを調べ，カーネル内の対応するメンバ関数を呼び出す．

es カーネル内部ではカーネルがユーザープロセスに提示しているカーネルオブジェクトを表にまとめて管理している．この表の各項目にはカーネルオブジェクトへのインターフェイスポインタ，オブジェクトのメタ情報へのポインタなどが含まれている．

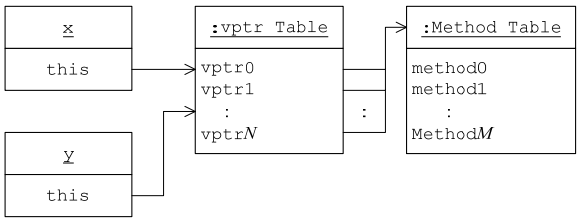

図 9　ユーザー空間のシステムオブジェクトモデル

ユーザー空間では，カーネルオブジェクトを間接的に呼び出すインターフェイスポインタからカーネルオブジェクト表へのインデックスを高速に取得できるように図 9 のように `vptr` 表と仮想表(Method Table)を配置している．すなわち，カーネルオブジェクトを間接的に呼び出すユーザー空間のインターフェイスポインタはユーザー空間内にただ 1 つだけ存在する `vptr` 表のどれか 1 つの項目を指している．そして `vptr` 表の各項目はすべて共通の仮想表を指している．

この構成によって，ユーザー空間のインターフェイスポインタからカーネルオブジェクト表へのインデックスは *this* - `vptr0` によって得ることができる．またインターフェイスのどのメソッドが呼び出されたかは，仮想表の何番目のメソッドが実際に呼び出されたかによって分かる．

```
methodm(void* this, …)
{
  int n = this – vptr0;
  va_list param;
  va_start(this, param);
  trap(n, m, param);
  va_end(param);
  if (trap が例外コードを返している)
    throw  システム例外;
}
```

図 10　`method`*m* の実装

図 10 に `method`*m* の実装の概略を示す．`method`*m* は *n* 番目のインターフェイスポインタから *m* 番目のメソッドをどのような引数で呼び出した，ということをトラップによってカーネルに通知する．またトラップから復帰したときに例外コードが返されていれば，`method`*m* は対応するシステム例外を `throw` する．

```
typedef long long (*Method)(void* self, ...);

long long Process::systemCall(int n, int m, va_list
param)
{
  SyscallProxy* proxy = syscallTable[n];
  Interface* interface = getInterface(proxy->iid);
  Function method = interface->getMethod(m);
  for (i = 0; i < method.getParameterCount(); ++i)
  {
    Parameter parameter = method.getParameter(i);
    // parameter の型に応じて必要な処理をする
  }
  Method** object = proxy->getObject();
  try
  {
    // メソッド呼出し
    result = (*object)[m](object, ...);
  }
  catch (...)
  {
    // 例外コードを保存
  }
  for (i = 0; i < method.getParameterCount(); ++i)
  {
    Parameter parameter = method.getParameter(i));
    // parameter の型に応じて必要な処理をする
  }
  Type returnType = method.getReturnType();
  // returnType に応じて result に関して必要な処理をする
}
```

図 11　システムコールインターフェイス

図 11 にカーネルのシステムコールインターフェイスの概略を示す．システムコールインターフェイスはまずパラメータの有効性のチェックなどをインターフェイスのメタデータを参照しながら行う．たとえば，ポインタ参照範囲がユーザー空間内かどうかを調べたり，引数がインターフェイスポインタなら，カーネルオブジェクト表中のインターフェイスポインタに置換したり，あるいはアップコール用のインターフェイスポインタを用意したりする．メソッドの呼出し中にカーネル空間で発生した例外はシステムコールインター

フェイスが一括して受け取り，例外コードとして保存する．

ユーザー空間へ戻るときに新しいインターフェイスポインタをユーザーに返す場合には，カーネルオブジェクト表に新しい項目を追加し，ユーザーにはそれに対応する vptr 表の項目へのポインタを返す．例外が発生していれば例外コードをユーザーレベルに返すことで method*m* が例外を throw できるように準備する．

4.2. アップコールインターフェイス

es カーネルはアップコールをサポートすることによって，カーネル空間に配置したソフトウェアコンポーネントからユーザー空間のソフトウェアコンポーネントを呼び出せるようにしている．

アップコールの実装においてはカーネル空間内に図 9 と同様な 1 組の vptr 表と仮想表を用意し，仮想表の各メソッドが実際のアップコールを行うアップコールインターフェイスを呼び出すように構成している．カーネル空間内のアップコール用のインターフェイスポインタは，アップコール用の vptr 表の項目へのポインタとなっている．カーネルは，この vptr 表の項目と 1 対 1 に対応するアップコールオブジェクト表を管理している．アップコールオブジェクト表の項目にはユーザー空間内のオブジェクトへのインターフェイスポインタ，オブジェクトのメタ情報へのポインタ，さらにオブジェクトを保持しているユーザープロセスオブジェクトへのポインタが保持されている．

```
long long hall()
{
  Method** object;
  int method;

  trap(&object, &method);
  (*object)[method]();
}

void* focus(void*)
{
  try
  {
    for (;;)
    {
      hall();
    }
  }
  catch (...)
  {
    // 例外コードを保存する
  }
  return 例外コード
}
```

図 12　アップコール用ランタイム

アップコールインターフェイスは，システムコールインターフェイスと同様にリフレクションを使用して引数や戻り値の受渡しを処理し，ユーザー空間内のオブジェクトのメンバ関数を呼び出す．es のユーザーランタイムライブラリはアップコールを受け付けられるようにカーネルに対して予めアップコールのエントリーポイントとなる focus 関数を登録している．図 12 に focus 関数の概要を示す．

アップコールの手順は以下のとおりである．アップコールインターフェイスは，アップコールを実行するスレッドのカーネルスタックベースアドレスを現カーネルスタックアドレスに再設定し，アップコール処理完了後にカーネルスタックを元に戻してアップコールの続きから処理を再開できるよう準備する．続いて，現スレッドのユーザーアドレス空間を対象となるサーバープロセスのアドレス空間に切り換え，スレッドがユーザー空間で標準のスレッドスタートアップルーチンを経由して focus 関数を呼び出すように新しいユーザースタックを構築する．focus 関数を実行したスレッドは hall 関数を呼び出してカーネルにトラップし，制御を一度カーネルに戻す．カーネルはここでユーザースタックにアップコールの引数を積み，ユーザー空間内のオブジェクトへのインターフェイスポインタおよびメソッド番号をレジスタにセットして制御を再びユーザー空間に移す．トラップから戻った hall 関数は指定されたオブジェクトのメソッドを呼び出したあと，focus 関数に戻ることでアップコールのためにカーネルが操作したユーザースタックポインタを巻き戻す．focus 関数は再び hall 関数を呼び出してカーネルにトラップする．アップコールインターフェイスはこのときのユーザーレベルのレジスタコンテキストを保存してアップコール処理を完了する．同じサーバープロセスに対して再びアップコールを掛ける場合には，最後のトラップで保存したユーザーレベルのレジスタコンテキストを使ってアップコールの引数をユーザースタックに積むことで，トラップの次の行から直接ユーザーレベルで処理を開始し，ユーザースタックの割当てやスレッドスタートアップルーチンのオーバヘッドを回避するようになっている．

アップコールで呼び出した関数が例外を throw した場合には focus 関数がその例外を catch し，スレ

ッドの実行を一旦終了する．カーネルはアップコール中のスレッドが終了しようとしたときは，ユーザースタックを解放したあと，スレッドを終了するかわりに戻り値を例外コードとして扱い，カーネル内で例外を `throw` し直す．

4.3. ローカルRPC = システムコール + アップコール

es カーネルでは，カーネル空間内のアップコール用のインターフェイスポインタをクライアントプロセスのカーネルオブジェクト表に登録し，クライアントプロセスがそのカーネルオブジェクトに対してシステムコールを発行すれば，カーネルは与えられた引数を使ってサーバーに対してアップコールを掛ける．これはローカルな RPC の実現に他ならない．es カーネルにはローカルな RPC を実現するための専用のコードはなく，システムコールとアップコールの処理によってローカルな RPC は自動的に実現されている．es カーネルのシステムコールインターフェイスは最終的に呼び出すメソッドがアップコールなのかカーネル内部のオブジェクトのメソッド呼出しなのか区別していないし，アップコールインターフェイスは，アップコールを要請したのがユーザープロセスなのかカーネルオブジェクトなのかを区別していない．

この実装においては，サーバーで処理を行うカーネルスレッドはクライアントを実行していたカーネルスレッドのままなので LRPC[4]と同様にスケジューリングオーバーヘッドのない効率のよい RPC となっている．

5. 実装と評価

es オペレーティングシステムは開発開始後 2 年余りが経過したところであるが，その間にオープンソース化し，製品品質までは作りこまれていないものの SMP 対応のスレッド，プロセス，FAT ファイルサブシステム，ISO9660 ファイルサブシステム，TCP/IP プロトコルスタックといった基本的なコンポーネントをリリースしている．現在のビルド環境には，gcc-4.1.1[25]および binutils-2.16.1[26]を，ランタイム環境については newlib-1.14.0[28]を es 用に一部変更して使用している．

es オペレーティングシステムはこれまで qemu[3]上でエミュレートされている PC 上で動作検証を行っている．実装の検証用にはオープンソースの Smalltalk の実装である Squeak[15]をポーティングし利用している．ファイルシステムやデバイスドライバの動作検証などがごく簡単なサポートコードを記述するだけで Squeak から行えるようになるため，es のように完全に新規のオペレーティングシステムの開発において Squeak はよいテスト環境となっている．

es オペレーティングシステムの開発に当たっては，es のカーネルレベルで提供するインターフェイスと共通のインターフェイスを提供する Linux 用のランタイムライブラリを準備することで，ファイルサブシステムや TCP/IP ネットワークスタックなどを Linux 上で開発できるようしている．es オペレーティングシステムではカーネルコードを記述するための特別なサブルーチンや規則を設けていないため，Linux 上で開発したコンポーネントソフトウェアは一切のコード修正なしにそのまま es カーネル上でも動作する．このことは，レガシーなオペレーティングシステムからコンポーネントオペレーティングシステムに段階的に移行していくことも原理的には可能であることを示している．

6. まとめ

本稿ではコンポーネントオペレーティングシステムに期待されている課題をまとめ，筆者らが開発を進めている es オペレーティングシステムを例に，コンポーネントソフトウェアの統一な開発環境を構築するため，オペレーティングシステムのカーネルがリフレクションを用いてシステムコールおよびアップコールを処理する方法を提案した．今後は既存のプラグインなどに対応したアプリケーションを es のコンポーネントフレームワークに移植する作業などを進め，コンポーネントオペレーティングシステムに求められる機能の検証を進めていく予定である．

近年，Xen[1]をはじめとした仮想化技術が PC でも利用できるようになってきたことで，今後はレガシーなオペレーティングシステムから新しいシステムに移行していくことも現実的な選択肢となっていくことが期待される．Web や新しい標準の登場によってオペレーティングシステムの新規開発は非常に困難になってきているものの[19, 20]，誰でもが自由にソフトウェアコンポーネント開発して，それらを拡張したり置き換えた

りできるようなオペレーティングシステム環境の構築は今後の重要な課題であろう.

参考文献

1) Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pp. 164-177, 2003.
2) Barrand, G., Belyaev, I., Binko, P., Cattaneo, M., Chytracek, R., Corti, G., Frank, M., Gracia, G., Harvey, J., Herwijnen, E., Maley, P., Mato, P., Probst, S., Ranjard, F. GAUDI — A Software Architecture and Framework for Building HEP Data Processing Applications, *Computer Physics Communications 140*, pp. 45-55, 2001.
3) Bellard, F. QEMU Open Source Processor Emulator. http://fabrice.bellard.free.fr/qemu/ .
4) Bershad, B. N., Anderson, T. E., Lazowska, E. D., Levy, H. M. Lightweight Remote Procedure Call. In *Proceedings of the 12th Symposium on Operating Systems Principles*, pp. 102-113, December 1989.
5) Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Chambers, C., Eggers, S. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pp. 267–284, December, 1995.
6) Brockschmidt, K. Inside OLE, 2nd ed., Microsoft Press, 1995.
7) Clark, D. D. The Structuring of Systems using Upcalls. In *Proceedings of the 10th Symposium on Operating Systems Principles*, pp. 171-180, December 1985.
8) Corbato, F. J., Saltzer, J. H., and Clingen, C. T. Multics—The First Seven Years. In *Proceedings of the American Federation of Information Processing Societies Spring Joint Computer Conference*, pp. 571–583, 1972.
9) Cox, B. J. Object-Oriented Programming: An Evolutionary Approach, Addison-Wesley, 1986.
10) Draves, R. P., Cutshall, S. M. Unifying the User and Kernel Environments. Technical Report MSR-TR-97-10, Microsoft Research, March 1997.
11) Drepper, U. ELF Handling for Thread-Local Storage, Version 0.20, Red Hat, December 2005.
12) Ford, B., Back, G., Benson, G., Lepreau, J., Lin, A., Shivers, O. The Flux OSKit: A Substrate for Kernel and Language Research. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pp. 38-51, October 1997.
13) Helander, J., Forin, A. MMLite: A Highly Componentized System Architecture. In *Proceedings of the Eighth ACM SIGOPS European Workshop*, pp. 96-103, September 1998.
14) Hunt, G., Larus, J., Tarditi, D., Wobber, T. Broad New OS Research: Challenges and Opportunities. In *Proceedings of the 10th Workshop on Hot Topics in Operation Systems*, June 2005.
15) Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., Kay, A. Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself. In *Proceedings of OOPSLA'97 Conference*, pp. 318-326, 1997.
16) Lippman, B. S. Inside the C++ Object Model. Addison Wesley, 1996.
17) Lopes, C. V. Aspect-Oriented Programming: An Historical Perspective (What's in a Name?). Technical report, Institute for Software Research, University of California, December 2002.
18) Mendelsohn, N. Operating Systems for Component Software Environments. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, pp. 49-54, May 1997.
19) Mitchell, J. An Overview of the Spring System (In IEEE COMPCOM '94) - Introduction by Jim Mitchell. In *Sun Microsystems Laboratories The First Ten Years: 1991-2001*, October 2001. http://research.sun.com/features/tenyears/volcd/papers/mitchell.htm .
20) Pike, R. Systems Software Research is Irrelevant. *Invited talk*, University of Utah, February 2000.
21) Szyperski, C., Gruntz, D., Murer, S. Component Software – Beyond Object-Oriented Programming. 2nd ed., Addison-Wesley, 2002.
22) Yokote, Y., Teraoka, F., Tokoro, M. A Reflective Architecture for an Object-Oriented Distributed Operating System. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, pp. 89-106, July 1989.
23) C++ ABI Summary, CodeSourcery, March 2001. http://www.codesourcery.com/cxx-abi/ .
24) DCE: Remote Procedure Call, Open Group Technical Standard Document Number C309, August 1994.
25) GCC, the GNU Compiler Collection, Free Software Foundation. http://gcc.gnu.org/ .
26) GNU Binutils, Free Software Foundation. http://www.gnu.org/software/binutils/ .
27) JavaBeans™ Specification 1.01, Sun Microsystems, August 1997.
28) newlib, Red Hat Inc. http://sources.redhat.com/newlib/ .
29) Programming Languages — C++. 2nd ed., ISO/IEC 14882, 2003.
30) The es Operating System, Nintendo. http://nes.sourceforge.jp/ .