Vol.2010-OS-113 No.4
2010/1/27

OCamlによるOSの実装

井 上 翔 大[†1]　大 山 恵 弘[†1]

現在，Java，C#，ML のような高い生産性での安全なプログラミングを支援する現代的なプログラミング言語が様々な目的で利用されている．しかし，OS や組み込み系ソフトウェアなどの低レベルのシステムソフトウェアの分野では，未だに C 言語が広く用いられており，低い生産性や致命的なバグの原因となっている．そこで，本研究では，現代的な関数型言語 OCaml で OS を開発し，低レベルのシステムソフトウェアの記述における関数型言語の有効性を評価する．その OS は極めて小さいが，カーネル空間とユーザ空間の分離，マルチタスク，割り込み処理，初歩的な入出力デバイス管理などのいくつかの基本 OS 機構を備えている．その OS の大部分は OCaml で記述され，残りは C 言語とアセンブリ言語で記述されている．この論文では，その OS の設計と実装について述べるとともに，開発を通じて得た，関数型言語を使用することの利害得失などに関する知見を示す．

Implementation of an Operating System in OCaml

Shouta Inoue[†1] and Yoshihiro Oyama[†1]

Modern programming languages such as Java, C#, and ML support secure and high productivity programming, and are currently used on various purposes. Unfortunately, in the field of low-level system software including operating systems and embedded software, the C language is still widely used and causes low productivity and critical bugs. In this work, we develop an operating system in a modern functional language OCaml, and evaluate the effectiveness of functional languages in writing low-level system software. Though the operating system is extremely small, it provides several basic operating system mechanisms including user-kernel separation, multitasking, interrupt handling, and primitive I/O device handling. Most of the operating system is described in OCaml, and the rest is described in C and assembly. This paper decribes the design and implementation of the operating system, and presents several insights we obtained through the development (e.g., pros and cons of using functional languages).

1. はじめに

現在，多くの分野で現代的な言語が普及している．例えば Java，C#，ML，Haskell などの言語が登場し，広く使われるに至っている．しかし，OS や組み込み系ソフトウェアなど低レベルなシステムソフトウェアの分野では，未だに C や C++の様な安全性の低い言語が主流となっている．これは，主にハードウェアアクセスやメモリアクセスなど低レベル処理の記述し易さが理由であるが，メモリリークやバッファオーバフローなどバグの温床となっているという問題がある．また安全性の問題に加えて，システムソフトウェアの高度化，大規模化が進み，内部ロジックが複雑化しているという問題もある．そのため，現代的な言語が持つオブジェクト指向やモジュールなど設計を容易にする機能へのニーズが高まっている．

そこで本研究では，実際に現代的な言語を使って OS を実装することで，その利点及び，問題点を明らかにする．具体的には現代的な言語を使用することで，記述し易くなる処理，記述しにくくなる処理を明らかにする．また，現代的な言語を使用したことによる OS への影響を併せて調査する．本研究では，言語として OCaml を使用する．また，対象とするハードウェアは，x86 系 CPU の PC/AT 互換機とする．本研究で実装する OS を OCOS と呼ぶ．OCOS は，基本的なデバイスが扱え，ユーザプロセスをマルチタスクに実行できるある程度実用的な OS を目指す．これにより，より現実的な知見が得られることが期待できる．

本論文は以下の様な構成になっている．第 2 章では，OCaml を使用した理由を述べる．第 3 章，第 4 章では，OCOS の設計と実装についてそれぞれ説明する．第 5 章では，OCaml を使用したことによる利点と欠点についての評価結果を報告する．第 6 章では，関連研究を示し本研究との違いを述べる．第 7 章ではまとめと今後の課題を述べる．

2. OCamlを使用した理由

OCaml とは，関数型言語 ML の方言であり，オブジェクト指向などの有用な機能を取り入れた言語である．本研究では，以下の理由から OCaml を使用した．

- 安全性の高さ

†1 電気通信大学
The University of Electro-Communications

OCaml はコンパイル時に厳しい型検査を行い，この検査を通ったプログラムでは型に関する安全性が保証される．これにより不正なメモリアクセスができないようになっている．またガーベージコレクション (以下 GC) により，利用されていないメモリ領域は自動的に解放され，メモリリークが起こらない．

- メンテナンス性の良さ
  OCaml では，オブジェクト指向の機能やモジュールといったプログラムの設計を容易にする機能を持っている．また，多相性により関数やデータ構造の抽象性を高めることができ，型毎に似たような処理やデータ構造をいくつも用意する必要がない．これらの機能によってコードの再利用性や可読性，さらには安全性をも高めることができる．
- 現実的な言語設計
  OCaml は関数型言語であるが，柔軟な言語設計になっており破壊的操作も簡単にでき，ループ文も用意されている．また，型推論や多相性により柔軟性の高い型定義を許すなど，プログラムの実装を簡単にする工夫がされている．また，OCaml コンパイラはバイトコードだけでなくネイティブコードも生成することができる．そのため，OCaml は OS の様なハードウェア上で直接動作するようなプログラムの記述に適している．

3. OCOS の設計

本研究で開発した OS は**図 1** の様な構成になっている．ハードウェア上では，ランタイムシステム (以下 RTS) と I/O アクセス層が動作し，OCOS はそれらを利用して動作する．

OCaml のコードを動作させるためには，GC やメモリへのオブジェクト割り当てなどの機能を提供する RTS が必要となる．OCaml のコードが通常利用する RTS は，OS 上で動作することを前提として作られている．しかし，本研究ではハードウェア上で直接 OCaml のコードを動作させる必要があるため，Linux 用の RTS に改造を加えて使用している．この RTS は C 言語とアセンブリによって記述されている．

また，ポートへのアクセスや，番地を指定したメモリアクセスなど OCaml では記述できない処理がある．そのため，ハードウェアアクセスなど低レベル処理を行うための C とアセンブリで記述された I/O アクセス層を用意した．OCOS は，この I/O アクセス層を通してハードウェアアクセスを行う．

OCOS は以下の機能を提供する．

- 割り込み処理
  タイマやキーボードなど外部の機器からの割り込みを処理する部分である．各割り込みに対応するハンドラが呼び出され，割り込みの処理を行う．
- プロセス管理
  マルチタスクを実現するためにユーザプロセスの管理を行う部分である．プロセスの生成や中断，再開処理，次に実行するプロセスを選ぶスケジューリングなどを行う．
- メモリ管理 (一部未実装)
  ユーザプロセスのためのメモリ領域の確保や仮想アドレス空間と物理アドレス空間とのマッピング等を行う．
- ファイルシステム (未実装)
  ディスク上のブロックを管理し，ファイルへのアクセス手段を提供する．

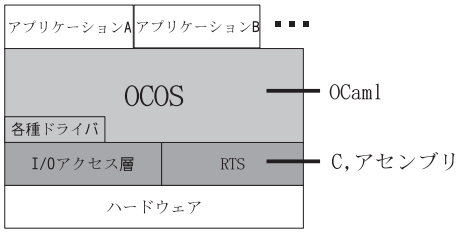

図 1 OCOS の概要図

4. OCOS の実装

4.1 OCaml のランタイムシステムの移植

OCaml のコードをハードウェア上で直接動作させるために，Linux 用の RTS を以下の様に書き換えた．

まず，RTS で使用している C の標準ライブラリを OS に依存しないように書き換えた．具体的には，標準ライブラリのコードを置き換えたり，標準ライブラリの呼び出し自体を削除した．更に置き換えたライブラリを OS イメージに静的にリンクするようにした．例えば，ヒープの確保をするために使われている `malloc` については，RTS が予め用意しておいた領域を割り当てるだけの関数で置き換えた．OCOS の実装において必要としない処理を行う関数，例えば，ファイルをオープンする `fopen`，スレッドにシグナルを送る `raise` などは，RTS からその処理ごと取り除いた．

次に RTS を動作させるために (OCaml を動作させるために) どうしても必要になるハードウェアの初期化処理を加えた．一つは，Global Descriptor Table (GDT) の初期化である．ブートローダが OS イメージ (OCOS のイメージと RTS のイメージ) を展開するメモリ領域と，OS イメージを作る際に指定したアドレスとは異なっている．例えば，OS イメージに含まれるコードは，アドレス 0x00000000 に置かれることを前提として作られている．しかし，実際に展開されるのは 0x00280000 から始まるメモリ領域である．そのため，RTS を動作させるためには，このアドレスのずれを吸収する必要がある．本研究では，このずれの吸収にセグメントの機構を使っているため，GDT の初期化が必要となる．

もう一つは，割り込み処理のために，Interrupt Descriptor Table (IDT) の初期化を行い，RTS の初期ハンドラを割り込みハンドラとして登録した．初期ハンドラとは，RTS で割り込みを一旦受け付けるための割り込みハンドラである．後に述べるように OCOS では，割り込み処理を RTS と協調して処理する．そのため，OCaml のコードを動作させる前に RTS 側で割り込みに関する最低限の初期化を行う必要がある．そのために RTS で IDT の初期化を行っている．

4.2 I/O アクセス層

I/O アクセス層では，OCaml では記述できない処理を行う．具体的には，ポートへの入出力，アドレスを指定したメモリへの直接アクセス，`cli`，`sti` などの特殊な命令の発行，レジスタへのアクセスなどの手段を提供する．I/O アクセス層では，それらの処理が C 言語とアセンブリの関数で記述されており，OCOS からそれらの関数を呼び出すことによって，対応する処理を行う．

図 2 は，I/O アクセス層の関数の例である．この関数では，OCOS からポート番号と値を受け取り，対応するポートへ値を出力している．OCOS から渡された，`port` と `val` の値は OCaml のデータ表現になっており，そのままでは C の関数で使用することができない．そこで最初にそれらの値を C のデータ表現に変換し，それから `out` 命令を発行することによって，ポートへの出力を実現している．この関数では，返り値は unit 型になっているが，返り値が必要な処理 (例えばポートから入力を読む) では引数の時とは逆に，返り値を C のデータ表現から OCaml のデータ表現に変換し OCOS に返している．

4.3 割り込み処理

割り込みは，RTS と OCOS が協調して処理する．RTS の協力が必要となるのは，GC とメモリへのオブジェクト割り当ての問題のためである．例えば，OCOS で GC が動作中であったとする．GC 中にはオブジェクトの移動や参照ポインタの書き換えが起こっている．

```
CAMLprim value caml_io_out8(value port, value val)
{
  CAMLparam2(port, val);
  /* OCaml の内部表現から変換 */
  int c_val = (Long_val(val)),
    c_port =  (Long_val(port));

  /* outb でポートに出力 */
  __asm__ volatile("movl %0, %%edx"::"m"(c_port));
  __asm__ volatile("movb %0, %%al"::"m"(c_val));
  __asm__ volatile("outb %al, %dx");

  CAMLreturn (Val_unit);
}
```

図 2 I/O アクセス層の関数の例

この時に，OCaml で記述された割り込みハンドラが呼び出されると，ヒープの状態が不正であるため，OCaml のコードを正しく実行することができない．また，割り込みハンドラの中でオブジェクト割り当てが起こると，ヒープの構造自体が破壊されてしまう．そのため，非同期に起こる割り込みを OCaml のコードにとって都合のよいタイミングで処理させる必要がある．そこで，本研究では一旦 RTS で割り込みを受け付けて，タイミングを見計らって OCaml で記述された割り込みハンドラを実行する．

図 3 が割り込み処理の概要である．割り込みを保留するためのデータ構造として，割り込み記録フラグを導入した．割り込み記録フラグは，割り込みの種類と同じ長さを持つ配列であり，要素は 0 か 1 である．0 は対応する割り込みを RTS が受け付けていないことを示し，1 は受け付けたことを示す．割り込みが起こると，RTS の初期ハンドラが呼び出される．その後の処理は，割り込みが起こった時の OS の状態によって分かれる．

- OS が実行中の場合
  OS が実行中の場合は，割り込みへの応答性を向上させるため割り込みは保留されることなく OCOS によって処理される．すなわち，RTS の初期ハンドラが直接 OCOS の割り込みハンドラを呼び出す．OS が実行中で，GC を行っておらず，メモリへのオブジェクト割り当ても行っていなければ，OCaml のコードを呼び出しても問題ない．
- GC 実行中または，オブジェクト割り当て中の場合
  GC が実行中であったり，OCaml がメモリへのオブジェクト割り当てを行っている時

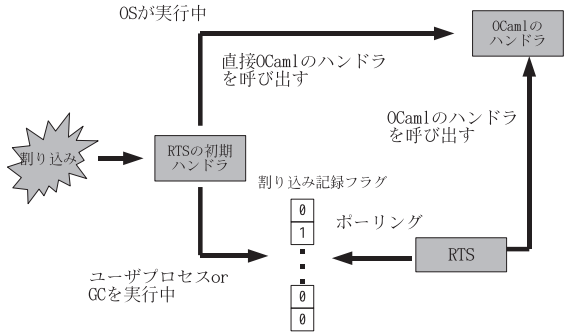

図 3 割り込み処理の概要

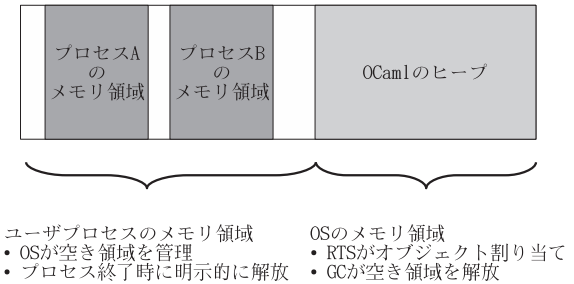

図 4 メモリ空間

に，OCOS の割り込みハンドラを呼び出すと，前述した様にヒープを汚染する可能性がある．そのため，この時は割り込みがあったことだけを RTS 内部の割り込み記録フラグに記憶しておく．GC やオブジェクト割り当てが終わって OCaml のコードを実行できる状態になってから，RTS が割り込み記録フラグをポーリングし，割り込みに対応する OCOS のハンドラを呼び出す．

- ユーザプロセスが実行中の場合
  ユーザプロセス実行時は OS が実行中であった場合と同じく，割り込みは保留されず OCOS のハンドラが直接呼び出される．ただし，その直後にユーザプロセスから OS へのコンテキスト切替えを行う．これは OCOS の割り込みハンドラは初期ハンドリングしか行わないため，割り込みに対応した処理の大部分 (例えば，画面上のマウスポインタを書き換える) は，割り込みハンドラ以外の OCOS 本体で行う必要があるからである．ユーザプロセスの実行の再開は，この処理が終わった後に必要に応じて行われる．

4.4 メモリ管理

OCOS は，仮想メモリをサポートしない．そのため，全てのプロセス (OS も含む) のメモリ使用量の上限は物理メモリのサイズと同じである．OCOS は，メモリ領域を**図 4** の様に 2 つに分けて利用する．一つは，OS 自身が動作するためのメモリ領域，もう一つはユーザプロセスが利用するメモリ領域である．OS 自身が動作するためのメモリ領域は，OCaml のヒープとして使用される．この領域は RTS が管理し，オブジェクトの確保，解放は自動的に行われる．OCOS がこれらを意識する必要はない．一方のユーザプロセスが利用するメモリ領域は，OCOS が明示的に空き領域をフリーリストで管理する．OCOS はユーザプロセスの要求に応じてメモリ領域を確保し，そのプロセスが終了すると，その領域を解放する．

OS がユーザメモリ領域を確保する処理について説明する．ユーザプロセスの開始時やシステムコールを通じた要求によって，メモリ領域が必要になると，OCOS は空き領域を管理しているフリーリストを辿り，必要な量の物理メモリページを確保して，それをリストの形にまとめる．次に，OCOS はユーザプロセスの仮想アドレス空間にこのページをマッピングするために，物理アドレスと仮想アドレスの対応を決定する．最後に，I/O アクセス層を利用してプロセス毎に用意されたページテーブルにこの対応付けを書き込む．このページングの処理は未実装であり，現在は複数に分かれた物理メモリ領域を一つの連続したメモリ領域として割り当てることはできない．

4.5 プロセス管理

OCOS では，ユーザ空間でプロセスをマルチタスクに実行することができる．すなわち，x86 の特権レベル 3 で複数のプログラムを並行に動作させることができる．プロセスのスケジューリングは 3 段階の優先度付きラウンドロビンで行われる．ユーザプロセスの生成は OS が**図 5** のコンテキスト (レジスタ群) を表現するレコードの初期値を設定し，それをスケジューラに登録することで行われる．逆にユーザプロセスの終了は，スケジューラから対応するコンテキストのレコードを削除することで行われる．

OCOS のプロセス実行の様子を**図 6** に示す．スケジューラによって次に実行されるべき

```
(*レジスタの集合 *)
type regs = {
  cr3: int32;  eip: int32;  eflags: int32;   eax: int32;  ecx: int32;  edx: int32;
  ebx: int32;  esp: int32;  ebp: int32;      esi: int32;  edi: int32;
  es: int;     cs: int;     ss: int;         ds: int;     fs: int;     gs: int;
  iomap: int32;
}
```

図 5 コンテキストを管理するためのレコード

プロセスが決定すると，RTS 中の `exec_u_proc` というユーザプロセスの実行を行う関数を呼び出す．この関数で，各レジスタへの値の設定を行い，ユーザプロセスの実行を開始する．ユーザプロセスを実行中に割り込みや例外が起こると，割り込みのハンドリングを行った後で，OCOS へのタスクスイッチを行う．RTS で，ユーザプロセスのコンテキストを保存し，OCOS のコンテキストを復帰する．最後に，ユーザプロセスのコンテキストを図 5 の OCaml のレコードに変換し `exec_u_proc` 関数の返り値とし，OCOS にリターンする．

4.6 ユーザプログラム

OCOS では，ユーザプログラムの実効ファイルとして a.out 形式のものが利用でき，記述する言語については制限しない．ユーザプロセスは，システムコールを通じて OS の機能を利用する．このシステムコールは Linux と同じく，ユーザプロセスが 0x80 の割り込みを発行することで実現されている．ユーザプロセスは，レジスタかメモリ上の特定のデータ構造に引数を格納し，OS に渡す．システムコールの実現方法は Linux と同じであるが，その機能やシステムコール番号が Linux と異なるため，libc の様な既存のライブラリを利用することはできない．ユーザプログラムは OCOS 用の libc を使用する．

ユーザプロセスの生成は，exec システムコールを通じて行われる．exec システムコールでは，実行するプログラムのファイルパスが引数として渡される．OCOS では，最初，新しく実行するプロセスが使うデータ用，コード用，スタック用のメモリ領域を確保する．それから，実効形式のファイルを解析し，コードのコピーや初期化の必要なデータの配置等を行う．最後に図 5 のコンテキストを表現するレコードの初期値を設定し，スケジューラに新しいプロセスを登録する．

現在は，ファイルシステムと a.out 形式の解釈部が未実装であるため，ユーザプロセスからシステムコールを通してプロセスを実行することはできない．現状ではユーザプログラムのコードを静的に OS イメージにリンクし，それを実行している．

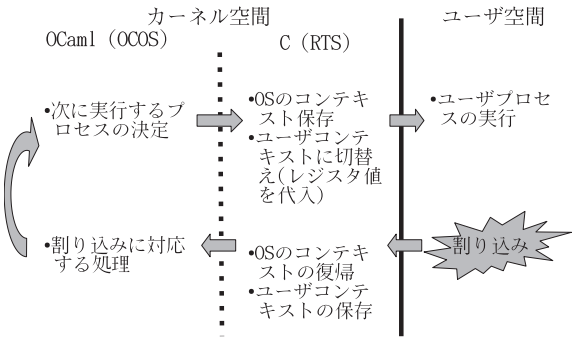

図 6 ユーザプロセス実行の概要

4.7 実装の現状

現在の実装についてまとめる．RTS の移植が完了し，ハードウェア上で OCaml のコードを直接動作させることに成功した．また，割り込み処理 (例外とシステムコールも含む)，プロセス管理を行う部分の実装が完了し，マウスやキーボード，タイマ等の基本的なデバイスが扱えるようになった．ユーザプロセスの実行もできるようになり，フィボナッチ数列の計算や 1 文字表示のシステムコールを繰り返すプログラムを複数同時に動かすことができる．メモリ管理については，物理メモリの確保，解放を行う部分は実装が完了したが，ページングの処理が未実装で，仮想アドレスを使うことはできず，物理アドレスを直接使っている．そのため現在は，OS とユーザプログラムが同一のメモリ空間で動作しており，メモリ保護は行われていない．

5. 評　　価

5.1 OCaml を使用したことによる利点

OCaml を使用したことの利点としては，記述性と安全性の向上が挙げられる．特に以下の 4 つの特徴が記述性，安全性の向上に寄与した．

- ヴァリアント型と強力なパターンマッチ
  ヴァリアント型とは複数のデータ型を一つにまとめて扱うデータ型である．一方パターンマッチとは，データの値，型によって処理を分岐させるための構文である．OCOS 中のヴァリアント型とパターンマッチの使用の例を**図 7** に示す．`cell` は，OCOS にお

```
(*割り込みキューのセル*)
type cell = CNil | TIMER | KEY_Data of int
            | MOUSE_Data of int | GPF
            | PUT_Char of int32

----------------------------------------

let a = dequeue int_queue  in
  match a with
      CNil -> ()
    | KEY_Data data ->    (* キーボード *)
        keybord_event data
    | MOUSE_Data data ->  (* マウス *)
        mouse_event data;
    ......
    |_ ->
       (*エラー処理*)
```

図 7 ヴァリアント型とパターンマッチを用いたコードの例

いて割り込みや例外を管理するキューのセルのデータの型である．このデータ型では，タイマ割り込みの発生を表すデータ，キーボード割り込みが起こったことを表すデータ等々を一つのヴァリアント型にまとめている．その下のコードは，パターンマッチの例である．この例では，`cell` のデータの種類によって，処理を分岐させ割り込みに対応した処理を行っている．

ヴァリアント型とパターンマッチを使うことで，データの区別やそれに応じて処理を分岐させることが容易になる．C 言語で同様の処理を記述する場合には，セルを表現する構造体にデータを区別するための int 型のフィールドを持たせ，1 ならタイマ割り込み，2 ならキーボード割り込み，···，−1 ならエラーといったようにプログラマが定義する必要がある．そして，そのフィールドの値に応じて if 文や case 文によって処理を分岐させることになる．この処理の記述では，常に if 文の処理洩れやデータを区別する値とデータの対応間違いなどの危険性が付きまとう．しかし，OCOS では，ヴァリアント型を使うことで，言語レベルでデータを区別することができる．分岐洩れをコンパイル時にチェックでき，安全性や記述性を高めることが可能になった．

- モジュールシステム
  モジュールは OCaml におけるプログラム分割の単位となっている．モジュールにより，外部に公開する関数を制限できるのはもちろん，抽象データ型によって，モジュールのインタフェースを経由しないデータの操作も禁止することができる．OCOS では，このモジュールの機能を使い，プログラム同士の結合を緩やかにし，メンテナンス性や安全性を向上させることができた．
- 自動メモリ管理
  OCOS 自身が使用するメモリ領域 (ヒープ領域) の管理は，GC に任せることができた．また，OCaml が持つ GC の機能により，開発では，メモリ領域を明示的に確保，解放する煩雑さからも解放された．メモリリークやダングリングポインタなどの問題が生じることもなかった．
- 強力な型システム
  OCaml はコンパイル時に厳しい型チェックを行う．この型チェックにより，多くのバグを未然に防ぐことができた．そのため，OS の実装中に型の不一致に起因するようなバグが発生することはなく，実行時に型の不整合によるエラーが生じることもなかった．

5.2 OCaml を使用したことによる欠点

欠点としては，ハードウェアアクセス部分のコードが複雑になることが挙げられる．OCOS ではポートへのアクセスやアドレスを指定したメモリへの直接アクセスなどの処理は全て I/O アクセス層を経由して行う．I/O アクセス層を経由するため，OS の処理を C と OCaml を跨いで記述することになる．そのため，記述が繁雑になり，可読性も低下する．加えて，I/O アクセス層の経由に伴う C 言語と OCaml の間でのデータ表現の変換などによりオーバヘッドや安全性の問題が生じる．

安全性の問題の例を挙げる．図 2 は I/O アクセス層でポートへの 8 ビットの出力を行う関数である．OCaml の型チェックはコンパイル時に行われるため，当然ながらこの関数の内部の処理や返り値はチェックされない．また，この関数をコンパイルする gcc でも，OCaml で記述された呼び出し側のコードはチェックされない．例えば，この関数が OCaml のデータ表現を無視した値を返したり，OCaml で 2 引数のこの関数に 5 つの引数を与える様なコードを記述しても，gcc，OCaml コンパイラのどちらもチェックをしない．この様に I/O アクセス層のインターフェイスは，C 言語，OCaml のコンパイラのどちらの検査も受けず，安全性が低下してしまう．

オーバヘッドの問題は，OCaml と C のデータ構造を変換する際のものである．図 2 に示したコードでも，ポート番号 `port` や，出力する値 `val` を OCaml のデータ表現から C のデータ表現に変換している．逆に C の関数から OCaml の関数へと値を返す時は，OCaml

表 1 使用した言語のコード量 (単位:行)

	OCaml	C	アセンブリ
RTS	-	10703	916
OS	2353	83	190

のデータ表現に変換する必要がある．この操作は，ヒープへのオブジェクト割り当てを伴うため，オーバヘッドが大きい．図 5 に示した様な大きなデータ構造を頻繁に返す処理では，この問題が無視できなくなる可能性がある．

もう一つオーバヘッドの例として OCaml の int 型の問題がある．OCaml では，整数とアドレスを区別するため，int 型の 1 ビットをフラグとして使用している．つまり，OCaml の int は 32 ビットでなく 31 ビットである．そのため OCOS 内ではメモリアドレスを int 型でなく int32 型で表現している．OCaml では，int32 型のデータはヘッダやフィールドを持つヒープ上のオブジェクトとして扱われる．このため，int32 型は int 型に比べて，オブジェクトの割り当てやオブジェクト同士の演算で大きなコストがかかる．OCOS でのアドレスの処理はこのようなオーバヘッドが生じている．ただし，アドレスの処理を行う部分は多くないので，このオーバヘッドはそれ程大きくならないと予想している．

5.3 コード量

OS のコードがどの程度 OCaml で記述できたのかを検証するため，使用した言語のコード量を計測した．結果が**表 1** である．OCOS では，割り込み処理や一部のハードウェアの初期化のように OS が本来行う処理を RTS が代行している．そのため，RTS の行数には OS の処理を行っているコードも含まれている．それを考慮しても，RTS が動作し OCaml を実行できれば，OS の大部分を OCaml によって記述できることが分かる．また，本研究では RTS の移植に大きな労力がかかり OCOS 本体の実装は不十分な所も多い．そのため，これから OS 本体の実装が進めば，OCaml のコードの割合が更に大きく増加していくと予想される．

表 2 は元の Linux 用の OCamlRTS と本研究で改造した RTS の行数の比較である．ハードウェアへのアクセスや割り込みハンドラのコードが加わったためアセンブラのコード量は増えている．しかし，前述した様にファイル操作やシグナルなどハードウェア上で動作させるのに不要な処理は RTS から取り除いたため，全体ではコード量は少なくなっている．この様に RTS は十分に小さく，カーネル空間に置いてもメモリ領域を圧迫することはない．

表 2 RTS のコード量の変化 (単位:行)

	C	アセンブリ	全体
オリジナル	15071	345	15416
改造後	10703	916	11619

6. 関連研究

メモリ安全性を保証する高級言語を用いた OS の記述については，現在までに多くの研究が行われてきた．たとえば Java で記述された OS として JavaOS[10]，Jx[6]，JNode[9] が，C#で記述された OS として Singularity[8] や CooS[13] が提案されている．Perl で記述された PerlOS[12] も存在する．古くは，Pascal で書かれた UCSD p-System や，Modula-3 で書かれた SPIN[1] が存在する．以下では特に関数型言語で記述された OS を取り上げ，本研究との関連について述べる．

House[4),7] は本研究の OS と同じく，現代的な関数型言語である Haskell によって記述された OS である．Haskell には，遅延評価が行われることや，副作用を伴う処理の記述が複雑になるなどの，OS のコードを複雑化させる特徴がある．本研究では，副作用を伴う処理を記述しやすく，評価順序が明確である OCaml を使用することで，簡潔で自然なコードを記述することを目指している．そして，その結果として，C や C++と関数型言語の間での記述力の違いをより見通しよく理解できるようになることを狙っている．

Hello OS[5] は Standard ML (SML) によって記述された OS である．同じく SML で記述されている TCP/IP のプロトコルスタック実装である FoxNet[2] をアプリケーションプログラムとして動作させることができる．Hello OS は非常に小さい OS であり，サポートされている OS 機能は極めて限られている．たとえば，ユーザ空間とカーネル空間の分離やマルチタスク処理はサポートされていない．本研究では，それらを含む多くの OS 機能を実装することにより，記述性をより詳細に評価することを目指している．また，Hello OS は OS のブートやメモリ管理などの部分で Linux のコード（C，アセンブラ）を利用しているのに対して，本研究ではメモリ管理を含む OS のほぼ全部を OCaml で記述している．

OCaml で記述された OS である Desert Spring-Time (DST) についての断片的な情報がウェブ上に存在する[3]．我々の知る限り，この OS に関する論文は発表されておらず，OS の実装に関する文書も OS のソースコードも現在入手できない．

Genera[11] は Lisp で記述され，Lisp マシン上で動作する OS である．本研究は，現在広く普及している汎用プロセッサである IA-32 を対象としている点や，OCaml という現代的

な関数型言語を対象としている点で，この研究とは異なる．

7. まとめと今後の課題

本研究では，現代的な関数型言語により低レベルのシステムソフトウェアを記述する利点及び問題点を明らかにするために OCaml で OS を記述した．一部に C 言語を使用したが，OCOS の大部分を OCaml で記述し，OS の様な低レベルのシステムソフトウェアも OCaml によって十分開発可能であることを示した．また，開発においてはヴァリアント型やモジュールなどの OCaml が持つ機能が OS の安全性，メンテナンス性を高めるのに有用であった．

今後の課題としては，まずファイルシステムやページングの処理を実装し OCOS の完成度を向上させることが挙げられる．また，OCOS の性能についても評価を行い，OCaml の利用が性能に与える影響についても明らかにしていきたい．

謝辞 本研究の一部は科研費 (19700024) の助成を受けたものである．

参 考 文 献

1) Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers, and Susan Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, pp. 267–284, Copper Mountain Resort, Colorado, December 1995.
2) Edoardo Biagioni, Robert Harper, and Peter Lee. A Network Protocol Stack in Standard ML. *Higher-Order and Symbolic Computation*, Vol.14, No.4, pp. 309–356, 2001.
3) Desert Spring-Time. http://alan.petitepomme.net/cwn/2005.05.31.html#11.
4) Iavor Diatchki, Thomas Hallgren, Mark Jones, Rebekah Leslie, and Andrew Tolmach. Writing Systems Software in a Functional Language. In *Proceedings of the 4th Workshop on Programming Languages and Operating Systems (PLOS 2007)*, 2007.
5) Guangrui Fu. Design and Implementation of an Operating System in Standard ML. Master's thesis, The University of Hawaii, 1999.
6) Michael Golm, Meik Felser, Christian Wawersich, and Jürgen Kleinöder. The JX Operating System. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pp. 45–58, June 2002.
7) Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew P. Tolmach. A Principled Approach to Operating System Construction in Haskell. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming (ICFP 2005)*, pp. 116–128, 2005.
8) Galen C. Hunt and James R. Larus. Singularity: Rethinking the Software Stack. *ACM SIGOPS Operating Systems Review*, Vol.41, No.2, pp. 37–49, April 2007.
9) JNode. http://www.jnode.org/.
10) James G. Mitchell. JavaOS: Back to the Future (abstract). In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, p.1, 1996.
11) Janet H. Walker, David A. Moon, Daniel L. Weinreb, and Mike McMahon. The Symbolics Genera Programming Environment. *IEEE Software*, Vol. 4, No. 6, pp. 36–45, 1987.
12) 浅野一成, 並木美太郎. PerlOS の試作と評価. 情報処理学会研究報告システムソフトウェアとオペレーティング・システム 2007-OS-106, pp. 87–94, 2007.
13) 高橋明生. CLI で実装する次世代オペレーティングシステム. Master's thesis, 武蔵工業大学, 2006.