# 並列 Lisp による操作系 SOKO における実験環境†

寺田 実††

並列 Lisp による OS "SOKO" の設計のひとつのねらいとして，OS に関する実験を容易に行える環境の構築がある．新しい装置を計算機に付加する場合など，それをサポートする OS レベルでの実験が不可欠であるが，従来そういった実験は容易ではなかった．これまで OS 実験を困難にしていた問題点としては，スタンドアロンの OS としての動作が必要；適当な記述言語がない；測定/解析ツールの不足；実験のターンアラウンドが長い，などがあった．これに対しソフトウェアの世界では，実験環境として Lisp に代表される開発支援システムの有効性（高度な対話性や自動記憶管理機構など）が広く知られている．この方式を OS 実験に適用したのが本研究であり，OS 実験の具体例として SOKO のディスク制御部をとりあげ，効率をあげるための種々の方式の比較実験を通して，SOKO の実験環境としての評価を行った．この実験は OS を動作させたままの状態で行えたため能率がよく，また実験を支援するツール群も容易に作成することができた．

## 1. はじめに

筆者らは，並行動作の記述可能な Lisp 言語 mUtilisp[1] を用いてワークステーション上にオペレーティングシステム（以下 OS と略す）SOKO[2]を開発してきた．SOKO の開発は，記述言語の評価や装置管理方式の研究などいろいろな目的をもっている．そのひとつに，OS に関する実験を容易に行えるような環境を構成することがあった．本論文は，この SOKO の実験環境としての適性を，実例を通して評価しようとするものである．

近年計算機の使いやすさを高めるためさまざまな周辺装置を付加することが多くなってきた．このような傾向は個人用計算機（ワークステーション）の一般化によってさらに顕著になってきている．それらの装置を活用するためにはソフトウェア（たとえばウィンドゥシステムやネットワークファイルシステムなど）が重要であり，そのために OS レベルでの管理方式の研究や実験が必要である．

ところが，従来は OS 実験は以下のような問題点をもち，容易ではなかった：

1）スタンドアロンの OS として動作するためには，各種デバイスの制御などの些細な（そして実験にとっては非本質的な）部分もすべて作る必要がある；

2）記述のための適当な言語がない．デバイスを制御するために必要な低レベルの記述が可能でなければならないが，実験を容易にするためには並行プロセスの動的生成・データタイプなどの高級な機能も必要；

3）実験結果の測定や解析のための道具を作ることが，それ自身として大きな作業になってしまう；

4）修正・実行のターンアラウンドが長い．コンパイル・リンクの後，現在のシステムをいったん停止させ，新しいシステムで起動する．エラーが発生した場合は，メモリダンプの解析が必要になることもある．

このような実験的開発システムの使いやすさについては，ソフトウェアを対象とするものに先例がみられる．たとえば言語処理系の試作や，人工知能のような実験的プログラム作成のためには，以前から Lisp に代表される対話的システムが活用されてきた．その意味で，本研究はソフトウェアの世界で培われてきた対話的システムを OS 実験に適用するものといえる．

高級言語によって OS を記述する試みは，これまでにもいろいろ行われてきた．古くは Concurrent Pascal による Solo[3] があり，新しいところでは Smalltalk-80 システム[4]がある．これらのシステムでは，高級言語によって高いレベルでの記述を行い，可読性，機械独立性を高めることを目的としているため，実際のデバイスの扱いなどは言語処理系の内部に隠されている．たとえば Smalltalk-80 システムでは，ウィンドゥを含む大きなシステムが Smalltalk-80 で記述してあるが，マウスのセンスやディスクの入出力などの低レベルの部分はプリミティブメソッドとしてインタプリタ内部で処理される．

これらと比較すると，本研究においては可能な限り低いレベルまで高級言語で記述することによって，より柔軟な実験環境を実現しようとするものである．

† Experiment Environment in Multi-Processing Lisp OS "SOKO" by MINORU TERADA (Department of Mathematical Engineering and Information Physics, Faculty of Engineering, University of Tokyo).

†† 東京大学工学部計数工学科

また，Cにより記述されている Unix[5] と比較すると，SOKO は，解釈実行型言語による記述；並行動作を言語レベルでサポート；OS の構成要素をプロセスとして分割しており，構造が理解しやすい；自動記憶管理機能，などの点で，より OS 実験に適しているといえる．

SOKO において OS 実験をする場合，SOKO は二重の役割を果たす—実験材料としての役割と，実験支援環境としての役割である．

実験材料としての役割とは，SOKO を構成するプロセス（システムプロセス）の一部を変更することによって実験を行うという意味で，変更された SOKO を動作させてデータなどを採取する．

実験支援環境としての役割とは，システムプロセスの変更を支援し，実験時の測定や動作解析のための道具を提供することである．

本論文の構成は，第2章で（実験材料としての）SOKO の紹介を行い，実験例の理解に必要な内部構造などを述べる．第3章ではディスク制御方式の実験例について，方法や結果などについて述べる．第4章では，実験支援環境としての SOKO を，ソフトウェア実験と OS 実験の共通点や相違点をみながら評価する．

## 2. SOKO の構成

SOKO は sun-1 ワークステーション上に，mUtilisp によって記述された OS である．

mUtilisp は，陽な並行プロセスをサポートする Lisp 処理系である．mUtilisp におけるプロセスは，動的に生成され，名前によって識別される．すべてのプロセスはヒープを共有するが，プロセスごとに別の名前空間を用いているため，変数の競合などは起こらない．プロセス間通信は，send/receive 関数を用いて，メッセージにより行う．

本章では，まず SOKO における OS 実験の基礎となるシステムプロセスの交換について述べ，つぎに次章でとりあげるディスク制御部について必要な解説を行う．

### 2.1 システムプロセスの交換

SOKO は，複数のプロセスから構成されている．それらプロセスは通常のユーザプロセスであるが，それが果たす役割からシステムプロセスと呼ぶ．システムプロセスは，SOKO の起動時に初期プロセスから fork して生まれ，SOKO が停止するまで動作を続ける．

システムプロセスは細かい単位に分割され，それぞれ特定の名前をもつ．ユーザからのサービス要求や，システム内部での通信などは，最終的にはこの特定の名前のプロセスあてのメッセージとなる．いいかえれば，名前がシステムプロセスを識別するわけである．したがって，特定の名前をもち，サービス要求のメッセージに正しく応じることさえできれば，システムプロセスをユーザの作った別のものに交換することが可能である．

mUtilisp におけるプロセス操作は以下のとおり：

- 生成
- 名前の変更
- 内部状態（実行中/中断中/終了）の変更
- 外部からの式の強制評価（signal という）

これらを利用して，以下のような手順でシステムプロセスの交換を行う：

1）新プロセスのためのプログラムを用意する；
2）現在のプロセスを中断させる；
3）現在のプロセスの名前を変更する；
4）新プロセスをその名前で起動する；
5）必要な登録処理をする．

ここで，2と3は旧プロセスを残しておくために必要な操作で，実験終了時にその旧プロセスを復帰させるためである．

5の登録処理について述べる．SOKO では，システムプロセスは名前によって識別されると書いたが，厳密には登録が必要なものがある．これは主として性能上の理由から，プロセス実体（これは mUtilisp のオブジェクトである）そのものを保持することによって，名前からプロセス実体を検索するオーバヘッドを回避している部分があるためである（一種のキャシュ）．その場合，新プロセスを生成した後キャシュされた実体を変更する必要があるが，これが登録処理である．

### 2.2 ディスクプロセスの概要

SOKO はハードディスク上に木構造のファイルシステムを構成している．ファイルはリスト形式のパス名をもち，内容には構造はなく，単なる文字の列である．mUtilisp からのアクセス方式は，パス名を指定してストリームオブジェクトを生成し，それを open し，関数 read や print の引数として利用する．

ファイルシステムを管理するためのシステムプロセスとして，*filsys* がある．*filsys* はディレクトリや空

ページを管理し，ファイルの生成/削除と，ストリームのオープン/クローズなどを扱い，ディレクトリや空ページの排他的アクセスを保証する．*filsys* は物理的な記憶媒体（を管理するプロセス）とは分離され，pageread/pagewrite のメッセージによって通信する．

物理的記憶媒体としてのハードディスクを管理するのがシステムプロセス *disk* である．*disk* の動作は，上位のプロセス（*filsys* や，一般のユーザプロセス）から pageread/pagewrite の要求（引数としてページ番号とバッファ（固定長ストリング）が付随する）を受け，IO 動作を行い返事をする．実際のハードウェアとのインタフェースは以下のとおりである：

（1） ヒープ外バッファ

SOKO の内部ではファイルバッファとしてヒープ中に Lisp オブジェクトとして確保した固定長ストリングを使用しているが，ディスクの入出力のためにはヒープの外の固定番地にバッファが必要になる．（その理由は，ガーベジコレクタのコンパクションによってヒープ内バッファは移動してしまうことと，そもそもハードウェアの制約によって，ヒープ内バッファはディスクコントローラボードからアクセスできないため．）これをヒープ外バッファと呼ぶ．現在のインプリメンテーションでは，512バイトのものを16本用意した．ヒープ内バッファとの転送のために，専用の組込関数（drstr, dwstr）を用意した．

（2） ディスクコントローラに対するコマンド

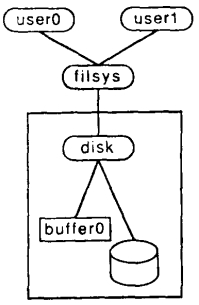
コマンドの発行のためにも専用の組込関数（drcom, dwcom）を用意した．この関数は，ディスクコントローラに指定されたコマンドを送り，完了を待つことなくリターンする．

（3） ディスク動作の完了

ディスク動作の完了は，本来は割込みとなって OS に伝わる．SOKO では，このような外部事象を仮想的なメッセージとして該当プロセス（ここでは *disk*）に送っている．すなわち，ディスク管理プロセスはコマンドを発行した後 receive によって（封鎖状態で）完了を待てばよい．実際の仮想 send の処理は，システムプロセス *scheduler* が担当する．したがって，ディスク動作の完了のみを選択的に待ちたい場合は *scheduler* を相手として receive をかければよい．

ここで，ディスクまわりのプロセス構成と，最も単純なディスク管理プロセスの構成を図1に示す．

次章ではいくつかのディスク管理プロセスの構成を述べるが，その名前は




丸枠は mUtilisp のプロセスを表す．大枠の中がディスクプロセスである．

**図 1** ディスク関連のプロセス構成
Fig. 1 Process organization around disk.

〈バッファ数〉/〈プロセス数〉[/〈付属情報〉]

という形式に従う．図1に示した構成は，以降 1/1 と呼ぶことにする．

## 3. 実験例—ディスクプロセスの高速化

これまでに述べたような SOKO の OS 実験への適性を実証するため，SOKO のディスク入出力プロセスを高速化する実験を行った．以下では，まず試みた3種の高速化技法について述べ，ついでその結果を述べる．

### 3.1 高速化技法

#### 3.1.1 多重バッファ化

第一の実験はヒープ外バッファを複数利用するもので，以下の2通りのプログラムを作成した．

2/1 ヒープ外バッファを2個使って，ディスク動作とバッファの転送を並行して行う（**図2**）．

16/1 16 個のヒープ外バッファを使用する（**図3**）．それぞれのバッファには現在保持しているデ

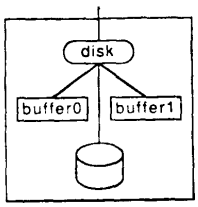



**図 2** 2/1 の構成
Fig. 2 Organization of 2/1.

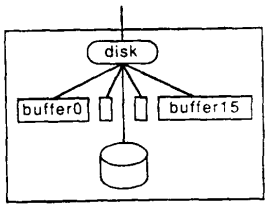



**図 3** 16/1 の構成
Fig. 3 Organization of 16/1.

ィスクページの番号が記録されており，ライトスルーのキャシュとして働く．これにより，同一ページに対する読出しが頻発する場合に高速化がはかられる．

### 3.1.2 多プロセス化

第二の実験では，多重バッファをそれぞれ独立のプロセス (*buf 0-buf 15*) で管理し，全体を管理するプロセス (*disk*)，ディスクコントローラの相互排除のためのプロセス (*diskc*) とあわせて，18 プロセスの構成とした (**図 4**)．これを 16/18 と呼んでおく．上位プロセスからのリクエストは適当なバッファプロセスに転送され，その完了を待たずに次のリクエストが処理される．

これによって，リクエストは必ずしも到着順に完了するわけではなくなり，キャシュがヒットしたリクエストがそうでないものを追いこすことがある．(もちろん同一ページに対するリクエストの順は厳密に保たれる．)

### 3.1.3 ページの先読み

第三の実験では，ファイルのあるページを読んだとき次ページが存在していればそれを読むコマンドを発行しておく．これによってシーケンシャルなファイルアクセスに対して高速化が期待できる．(ただし，この方式は厳密にいえば SOKO の方針を逸脱している．本来ページの内部構造に関する知識はファイルシステムプロセスが管理すべきものであるから，ディスクプロセスが次ページといった論理的なリンクを知っていてはならないはずである．)

1/1/UP 1/1 に先読みを追加したもの．後続リクエストの有無にかかわらず先読みをする．

1/1/P 1/1/UP と異なり，後続にリクエストがない場合に限り先読みをする．

16/1/P 16/1 に先読みを追加したもの．後続リクエストがない場合のみ先読みをする．

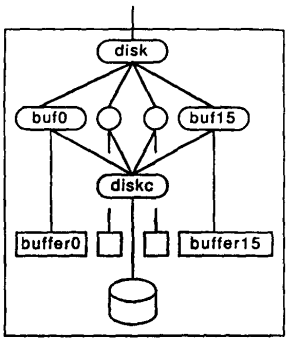



**図 4** 16/18 の構成
Fig. 4 Organization of 16/18.

```
(setq source (contents "buf2.1")          ;(1)
(unmount)                                 ;(2)
(process-rename "disk" "olddisk")         ;(3)
(mapcar source 'eval)                     ;(4)
(mount)                                   ;(5)
.. testing ..                             ;(6)
(unmount)                                 ;(7)
(send "disk" 'terminate)                  ;(8)
(process-rename "olddisk" "disk")         ;(9)
(mount)                                   ;(10)
```

**図 5** 実験の手順
Fig. 5 Procedure of experiment.

16/18/P 16/18 に先読みを追加したもの．

### 3.2 実験の手順とテストプログラム

実験は**図 5** のような手順で進行する．(1)現在のディスクプロセスを利用して，新しいディスクプロセスのためのソースをリストの形でファイルからもってくる．(2)次にファイルシステムプロセスをディスクプロセスからいったん切りはなし，(3)現在のディスクプロセスは改名して保存しておき，(4)新しいディスクプロセスを起動する．

この段階で *disk* という名前の新しいディスクプロセスが生成されるので，(5)ファイルシステムプロセスをそれに接続する．これによって，SOKO のファイルシステムは前と同様に機能し始めるので，さまざまなテストを行って，性能を測定する．

実験終了時には，新しいディスクプロセスを停止させ (メッセージによる(8)かあるいは関数 kill-process を使ってもよい)，保存しておいたディスクプロセスを復旧する．

テストプログラムとして，以下の 3 種を用いた：

test-n n 個のユーザプロセスが一斉に特定ファイルのページを順次アクセスする．実際にはファイルオープン時の相互排除によって，各ユーザプロセスには一定の進行差が与えられ，そのアクセスページには一定の差が生じる．

open 特定のファイルのアウトオープンとクローズを 100 回繰り返す．同一ページに対するアクセスが重なる．

cp ファイルのコピー．現実的な利用形態に最も近い．

### 3.3 結果と考察

実験の結果を**表 1** に示す．以下で，3 通りの高速化技法について評価をしてみる．

〈多重バッファ化〉

2/1 における二重バッファの効果は，test-1 で認められるが，それほど大きくはない．原因はバッファ

表 1 実験結果 (単位: msec)
Table 1 Result of experiment.

| | test-1 | test-2 | test-3 | test-4 | cp | open |
|---|---|---|---|---|---|---|
| 1/1 | 3526 | 6696 | 10960 | 12498 | 40582 | 54978 |
| 2/1 | 3370 | 6628 | 11555 | 15356 | 40108 | 51416 |
| 16/1 | 3446 | 6248 | 10306 | 11927 | 40366 | 23312 |
| 16/18 | 6268 | 6411 | 9412 | 9639 | 45382 | 30928 |
| 1/1/UP | 3254 | 12813 | 18063 | 24388 | 43890 | 54958 |
| 1/1/P | 3250 | 6693 | 10936 | 12408 | 43194 | 54986 |
| 16/1/P | 3202 | 6212 | 10254 | 11885 | 33284 | 24564 |
| 16/18/P | 6206 | 6308 | 6610 | 9817 | 33914 | 30744 |

コピーがディスク動作に対して十分高速なためである．また，test-2 以降ではディスク動作完了待の時間がほかのプロセスによって有効利用されるため，二重バッファの効果がないと思われる．

16/1 においては，test-n でも一定の高速化をはたしているが，open に対して 2 倍以上の効果をもっている．これは open においては少数ページにアクセスが集中する結果としてキャシュのヒット率が向上するためである．

〈多プロセス化〉

16/18 は test-2, test-3, test-4 と open において 1/1 にまさっている*．いずれも原因はバッファのキャシュとしての利用にあり，特に test-3 と test-4 を比較するとプロセスが増加しても所要時間がほとんど変化せず，キャシュの効果を示している．さらに，同じく 16 個のバッファを利用する 16/1 と比較しても test-3, 4 で優位に立っている．

その理由は，並列に動作する 3 ないし 4 プロセスのうちの後発のものが，先頭が読みこんだバッファをキャシュとして利用することで先頭との差を縮めていくことにある．その結果キャシュが有効利用され，後発のものが 16/1 におけるよりもはやく完了するのである．

〈先読み〉

1/1/UP は test-1 の場合のみ，1/1 にくらべて改善がみられる．これは test-1 ではディスクアクセスが完全にシーケンシャルであることと，ディスク動作完了待の時間が完全にアイドルになるためである．だが test-2 以降のように並列な要求がきた場合，先読みの結果が全く無駄になってしまい，ほぼ 2 倍の時間がかかってしまう．

1/1/P では，test-1 の場合の改善は同様であるが，test-2 以降は事実上先読みをしないため，1/1 とほとんど同じ結果が出ている．

テストプログラム cp の場合は一見先読みが有効に思えるが，単一バッファであることがわざわいして先読みした結果を書出しのために破壊してしまい高速化に役立っていないことを結果が示している．

一方多重バッファに先読みを追加したものはどちらも効果を示している．cp の例では，16/18/P が 25%，16/1/P が 15% 程度の高速化になっている．

以上の実験結果から，多重バッファをもち先読みを行うものが最良ということがいえる．現在まで実際に稼働していたディスクプロセスは 2/1 であったが，16/1/P に変更することがのぞましいわけである．

## 4. 評　価

この章では，実験支援環境としての SOKO を，ソフトウェア実験と OS 実験の共通点や相違点をみながら，評価を行う．

### 4.1 実験支援環境としての利点

まず，ソフトウェア実験との共通性がもたらす SOKO の利点について述べる．

はじめに，Lisp 言語の長所に由来するものとして，対話的利用と自動記憶管理があげられる．

システムプロセスを OS 動作中に対話的に交換でき，さらには動作中のプロセスの関数を外部から変更できるので，実験のターンアラウンドを非常に高速化できる．また実験中にプロセスの状態（内部の変数の値や，スタックのトレースなど）を簡単に得ることができる．実行中にエラーが生じた場合，そのプロセスがコンソールに対して read-eval-print ループを行うため，エラー原因の調査なども容易である．ポストモーテムダンプに対してデバッガを起動し，まったく異なるコマンド体系を使う必要がない．

自動記憶管理機能によって，実験の際必要となる作業領域などを，とりあえず簡単に確保できる．性能の向上が必要な場面では，個別に記憶管理のコードを作成すればよいのである．

次に，支援ツールの作成が容易であることも特徴である．以下のような支援ツール群を作成した．

第一に，資源の利用状況を測定するため，CPU 時間，経過時間，セル消費をプロセスごとに表示する関数を作った．組込関数を利用して，2 ms の単位で測定

* test-1 の結果は，2 グループにはっきりと分かれている．これはディスクの媒体の回転周期に原因があり，媒体が 1 周するあいだに次のコマンドを発行できなかった場合に，もう 1 周待たされるためほぼ 2 倍の差が生じるのである．

できる.

第二に，メッセージ通信のログの採取を可能にした．これは組込関数 send/receive の再定義によるもので，時刻，発信・受信プロセスを記録した後，本来の動作をするように修正した（この修正はインタプリタ本体の修正を要しない．Lisp レベルのかきかえで実現できる）．この方式の利点は，着目したプロセスのみの関数定義を変更することによって，そのプロセス関連のメッセージのみ選択的に記録可能であることで，関係のないプロセス間のメッセージを記録から排除できる.

第三に，プロセススイッチのログの採取も行った．これはシステムプロセスであるスケジューラプロセスを修正して記録を残すようにしたものである.

マルチプロセスのシステムのデバッグのためには，それぞれのプロセスがどのように動いているかをモニタするのが第一歩である．それにはメッセージとプロセススイッチのログが有用である．これら2種類の生のデータをもとに，プロセスの動作を図式的に表示するプログラムも作成した.

### 4.2 問題点とその解決

実験環境としての SOKO の問題点には，ソフトウェア実験と OS 実験の相違に起因するものが考えられる．OS 実験の特徴は，以下のようなものがある：

（1） ハードウェアに由来する並列性

ハードディスクを例にとれば，ディスクコントローラにコマンドを与えた後，CPU は別の仕事を行うことができる．通常のソフトウェアでは，このような並列性はないが，OS ではこれをできるだけ自然に記述する必要がある.

（2） 実時間処理の必要性

シリアル回線やネットワークにおいては，外部事象の発生から一定の時間内に対応が必要とされる．また，ヒューマンインタフェースの点からも実時間性は重要である.

（3） 支援環境の不安定性

ファイルシステムなど複雑な内部状態をもつものを扱うため，誤動作によって回復不能な事故が起きやすく，実験の続行に支障をきたすことが多い.

これらに対し，本研究では以下のような対応を行った：

（1） 並列性は記述言語の能力を利用する.

この場合，問題になるのは高級言語で記述したことによるオーバヘッドである．単純なメッセージセンドによるプロセススイッチの所要時間は約 3 ms であった．これに対しディスクの媒体の1回転には約 18 ms かかる．これは決して無視できるものではないが，本研究においてはそれでも意味のあるデータを得ることができた.

（2） 実時間処理に必要な部分は，言語処理系内部で処理する.

実時間処理のための最大の問題は，記述言語である mUtilisp 処理系のガーベジコレクション (GC) による実行の中断である．中断の期間は 10 秒程度であり，頻度はプログラムの振舞いによって大きく異なる.

これにともなう問題点は，OS としての計算機管理に対する影響と，対話的システムとしてユーザに対する影響のふたつがある．前者への対策として，外部入力（キーボード・RS 232 C・ディスク動作完了など）のバッファリングを行っている．GC 中に起こったイベントをとりあえずバッファしておき，GC 完了後に処理するようにした．しかし，RS 232 C で XON/XOFF などのフロー制御を無視して大量のデータを送られると，バッファがオーバフローしてしまう欠点がある.

後者については特別な対策を行えなかった．特に，キーボード入力のエコーバック処理も OS が行っているため，GC 中はそれも停止してしまいユーザに心理的負担をかけた.

しかし，OS 実験だけに関していえば，作業領域などを使い捨てにせず，自分で記憶管理をすることでかなり GC の頻度を減らせることが確かめられた．（たとえばファイルシステムプロセスで用いるバッファなど.）

（3） デバッグ支援については，高級言語記述の利点をある程度利用できた.

たとえば異常な状態のプロセスの内部変数などを対話的に調査し，さらには対話的に復旧処理を行うこともできた．ファイルシステムの一貫性のチェック・修復プログラムは当初から作成していたが，実際にはそれでは処理できないような事故もあり，ディスクページを直接修復することができ，対話性が大変役立った.

## 5. おわりに

ソフトウェア開発のために用いられている方法を OS 実験に流用し，対話性や自動記憶管理機構などの適性を示した．このような方法は，OS 実験に限らず計算機上でのさまざまな分野での実験に有効であろう.

**謝辞** 本研究を行う機会を与えてくださった和田英一教授と，言語 mUtilisp の開発者であり SOKO の共同開発者である岩崎英哉助手に感謝いたします．

## 参考文献

1) 岩崎英哉：Lisp における並列動作の記述と実現，情報処理学会論文誌，Vol. 28, No. 5, pp. 465-470 (1987).
2) 寺田 実，岩崎英哉：Lisp ベースの操作系 SOKO—内部構造と実現，第 35 回情報処理学会全国大会論文集，pp. 257-258 (1987).
3) Hansen, P. B.: The Solo Operating System: Processes, Monitors and Classes, *Information Science*, California Institute of Technology (1975).
4) Goldberg, A. and Robson, D.: *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, MA (1983).
5) Bach, M. J.: *The Design of the Unix Operating System*, Prentice-Hall Inc., Englewood Cliffs, NJ (1986).

(昭和 63 年 5 月 20 日受付)
(平成元年 1 月 17 日採録)

**寺田 実**（正会員）

1959 年生．1981 年東京大学工学部計数工学科卒業．1983 年同大学院工学系研究科情報工学専門課程修士課程修了．同年より同大学工学部計数工学科助手．主な研究分野は，Lisp に代表されるプログラム言語やオブジェクト指向言語で，言語処理系の作成やオペレーティングシステムへの応用などに関心をもつ．最近は Lisp による並行動作記述について研究している．ACM 会員．