システムソフトウェアと
オペレーティング・システム 65−15
（１９９４．７．２１）

オブジェクトの堆積モデルに基づく間接オブジェクトの実現

島袋 浩二 <ko@ocean.ie.u-ryukyu.ac.jp>
新城 靖 <yas@ie.u-ryukyu.ac.jp>
翁長 健治 <onaga@ie.u-ryukyu.ac.jp>

琉球大学　情報工学科
〒903-01　沖縄県西原町千原1番地
電話：098-895-2221　内線3266
Ｆａｘ：098-895-2688

概要　オブジェクトの堆積(object-stacking)とは、object-basedシステムを構造化するためのモデルである。オブジェクトの堆積では、オブジェクトの層を作ることで、オブジェクトの機能を組み合わせることができる。この論文では、オブジェクトの堆積に基づき、高度な機能を提供する間接オブジェクトの実現方式について述べている。間接オブジェクトとは、受け付けたメッセージの内容を変更することなく、下位層オブジェクトに送るオブジェクトである。今回実現した間接オブジェクトは、オブジェクトの移動、不可分操作を実現するために用いることができる。この論文は、間接オブジェクトの機能として、凍結・解凍・下位層オブジェクトの変更の実現について述べる。間接オブジェクトの実現において、軽量プロセス間の同期プリミティブの１つであるロックと、永続構造体ライブラリが利用されている。

The Implementation of Indirection Objects based on the Object-Stacking Model

Koji Shimabukuro <ko@ocean.ie.u-ryukyu.ac.jp>
Yasushi Shinjo <yas@ie.u-ryukyu.ac.jp>
Kenji Onaga <onaga@ie.u-ryukyu.ac.jp>

Department of Information Engineering
University of the Ryukyus
Nishihara, Okinawa 903-01, Japan
Phone: +81 98 895 2221 Ext. 3266
Fax: +81 98 895 2688

Abstract　Object-stacking is a model for structuring object-based systems. In object-stacking, by creating layers of objects, their functions are integrated. This paper presents the implementation method of indirection objects that provide high-level functions based on the object-stacking model. An indirection object is an object which receives a message from a client and sends the message to the lower object without changing the contents of the message. An indirection object can use to realize object migration and atomic action. This paper presents the implementations of procedures of indirection objects: freezing, unfreezing, changing lower object. In the implementation of indirection objects, a lock synchronization primitive among lightweight processes the persistent structure library are used.

1 はじめに

オブジェクトの堆積（object-stacking）は、object-based システムを構造化するためのモデルである。オブジェクトの堆積では、個々のサーバは、単純な機能を持つオブジェクトを提供する。それらのオブジェクトは、オブジェクト識別子と遠隔手続き呼出しにより結合され、それらの機能を合せ持つようなオブジェクトを形成する。この論文では、間接オブジェクトの実現、すなわち、間接オブジェクトを提供するサーバの実現について述べる。

文献 [6] では、オブジェクトの堆積の基本的な考え方について述べた。文献 [3] では、フィルタ機能をもつファイル・オブジェクトの実現について述べた。この論文では、間接オブジェクトの実現について述べる。本論文で述べる間接オブジェクトは、オブジェクトの移動（object migration）、不可分操作などに利用することができる。その特徴は、様々な型のオブジェクト（ファイル型、ディレクトリ型、プロセス型など）に対応している点にある。その間接オブジェクトは、凍結、解凍、下位層オブジェクトの変更という機能がある。

間接オブジェクトの実現の特徴は、軽量プロセスと永続構造体ライブラリを用いている点にある。軽量プロセスは、遠隔手続き呼び出しの要求に応じて生成され、要求間の同期を実現するために利用される。永続構造体ライブラリは、C言語の構造体に加えて、タグ付き共用体、可変長配列などをファイルに保存する機能を持っている。そして、永続部分と揮発部分を合わせてキャッシングする機能を持つ。

2 オブジェクトの堆積

オブジェクトの堆積の特徴は、一様なインタフェースを持つオブジェクトを積み重ねることである。オブジェクトとは、データとそれに関する手続きをカプセル化したものである。オブジェクトの内部のデータの操作は、公開された手続きによってのみ許されている。オブジェクトは、オブジェクト識別子(OID: object identifier)により参照される。オブジェクトに対する手続きをオブジェクト手続き(object procedure)という。オブジェクトを積み重ねる(堆積させる)とは、上位層のオブジェクトが下位層のオブジェクトの識別子を持ち、かつ、上位層オブジェクトが、その機能を実現するために下位層オブジェクトを呼び出すことである。

オブジェクトの堆積では、積み重ねるオブジェクトのインタフェースを一様にすることが重要である。インタフェースとは、オブジェクトが受け付けることのできる手続きの集合である。インタフェースが一様であることにより、オブジェクトを積み重ねる順番を自由に変えることができる。インタフェースが一様でない場合、積み重ねる順番と層の構成が制限されてしまう。

図1に3つのオブジェクトが堆積されている様子を示す。上の2つのオブジェクトは、**堆積可能オブジェクト**(stackable object)である。堆積可能オブジェクトとは、下位層オブジェクトの識別子を持つオブジェクトのことをいう。最下層オブジェクトは、**基底オブジェクト**(bottom object)で、他のオブジェクトの識別子を保持しない。

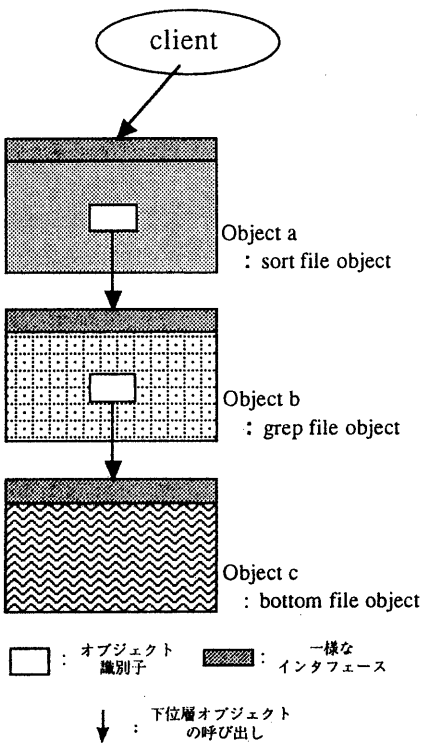

図1　オブジェクトの堆積による
オブジェクトの機能の統合

図1において、各オブジェクトは、次のような機能をもつ。

Object a: sortファイル・オブジェクト
　文字列を並び替える機能を持つファイル・オブジェクト

Object b: grepファイル・オブジェクト
　文字列のパタン検索を行なうファイル・オブジェクト

Object c: 基底ファイル・オブジェクト
　データを貯蔵するファイル・オブジェクト

このように、それぞれ1つの機能しか持たないオブジェクトを積み重ねると、文字列の並べ替え、パ

タン検索を行ない、そのデータを貯蔵する１つのオブジェクトが作られる。このように、オブジェクトを積み重ねることは、オブジェクトの機能を組み合わせることを意味する。また、このように組み合わされてできたオブジェクトを堆積オブジェクト(stack object)という。

sort ファイル・オブジェクトや grep ファイル・オブジェクトのようなオブジェクトは、フィルタ・オブジェクト(filter object)と呼ばれる [3]。フィルタ・オブジェクトは、受け付けたメッセージに何らかの処理を行ない、その結果を下位層オブジェクトに送る。あるいは、下位層オブジェクトから受けとった結果を加工してクライアントに返す。

3 間接オブジェクト

間接オブジェクトとは、受け付けたメッセージの内容を変更することなく、下位層オブジェクトにそのメッセージを送るオブジェクトである。間接オブジェクトは、総称的（generic）である。すなわち、間接オブジェクトでは、いかなる型のオブジェクト(ファイル型、ディレクトリ型など)を取り扱うことができる。例えば、ファイル型のオブジェクトでもディレクトリ型のオブジェクトでもオブジェクトの移動を行なうことができる。

この章では、まず間接オブジェクトの利用について述べる。主な利用方法として、オブジェクトの移動と不可分操作が挙げられる。次に、間接オブジェクトの手続きの実現について述べる。普通の手続きにおいて、間接オブジェクトは、受け付けたメッセージを自分自身で解釈せず、下位層オブジェクトに送る。これに対し、下位層オブジェクトの変更(change_lower())・凍結(freeze())・解凍(unfreeze())手続きは、間接オブジェクト自身が解釈し、実行する手続きである。

3.1 オブジェクトの移動

オブジェクトの移動(object migration)とは、あるサイト(計算機)にあるオブジェクトを別のサイトへ移すことである。オブジェクトの移動は、下位層オブジェクトの変更(change_lower())・凍結(freeze())・解凍(unfreeze())という手続きを備えた間接オブジェクトにより実現される。凍結手続きが実行されると、解凍手続きが実行されるまで、その他の手続きの実行が一時停止される。ただし、下位層オブジェクトの変更手続きは、凍結中でも実行される。下位層オブジェクトの変更手続きは、下位層オブジェクトの識別子を置き換える手続きである。

オブジェクトの移動は、管理者(Initiator)により行なわれる。以下に、オブジェクトの移動の処理手順について述べる（図２）。

1)オブジェクトの移動を行なう管理者は、間接オブジェクトの凍結手続きを呼び出す。それ以降のクライアントから呼び出された手続きの実行が一時停止される。

2)管理者は、新しいオブジェクト(New lower object)を生成して、下位層オブジェクトの内容をそのオブジェクトにコピーする。

3)管理者は、2)で生成されたオブジェクトの識別子を引数として、下位層オブジェクトの変更手続きを呼び出す。間接オブジェクトは、その下位層オブジェクトの識別子を新しい識別子に変える。

4)管理者は、間接オブジェクトの解凍手続きを呼び出す。この時、凍結手続きにより一時停止されていた手続きの実行が再開される。

凍結手続きは、次のような場合が考えられるのでオブジェクトの移動時に必要となる。

・手順2)を実行中に他のクライアンが、移動前のオブジェクトの内容を変更する可能性がある。

この場合、元のオブジェクトと移動先のオブジェクトの内容が同一でなくなる。このような問題を防ぐためにオブジェクトの移動において間接オブジェクトは、凍結される必要がある。

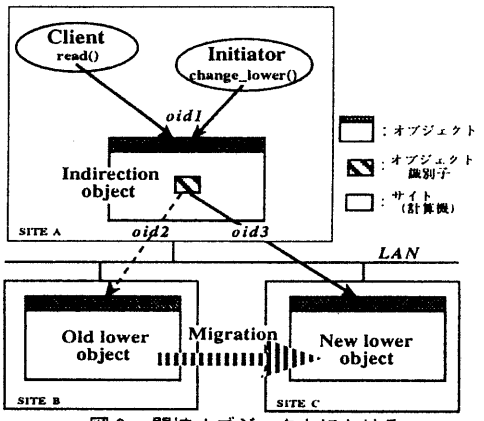

図２　間接オブジェクトにおける下位層オブジェクトの変更

3.2 不可分操作

間接オブジェクトを利用することで、不可分操作が、実現される。不可分操作(atomic action) とは、その操作の集合をすべて行ない、途中に他の操作が入ってこないような操作のことである。不可分操作は、データベースにおけるデータの更新などで必要とされる。

UNIX では、システム・コール単位で不可分操作が実現されている。ファイル単位の不可分操作を実現するためには、１回のシステム・コールでファイ

ル全体を読み書きすればよい。

オブジェクトの堆積では、オブジェクト手続きの呼び出しに遠隔手続き呼び出し(RPC)が利用されることが多い。1度に扱えるメッセージの大きさに限界がある遠隔手続き呼び出しを利用する場合、ファイル単位の不可分操作ができない。しかし、間接オブジェクトを利用することでファイル単位の不可分操作を実現することができる。具体的には、不可分操作を行いたいオブジェクトの上に間接オブジェクトを積み重ねる。オブジェクトの更新の時には、新しいオブジェクトを生成し、一貫性のある内容を書き込む。その後、下位層オブジェクトの識別子を、生成したオブジェクトの識別子に置き換える。

3.3 間接オブジェクトの普通の手続き

この節では、間接オブジェクトの普通の手続きについて述べる。普通の手続きの例として、ファイル・オブジェクトの読み出しを行う手続き file_read() を用いる。図3の1～14行に、その骨格を示す。間接オブジェクトは、サーバ内では、iobj_t という構造体で実現されている。iobj_fetch() は、構造体のデータをファイルから読み出す手続きである。詳しくは、6章で述べる。

file_read()の処理手順を以下に示す。

- iobj_fetch()により、オブジェクトの構造体のデータを読み出す。
- iobj_ensure_unfreezed() により、そのオブジェクトが、凍結されてないことを確認している。iobj_ensure_unfreezed() については、4.4節で述べる。
- 引数中のオブジェクト識別子を、下位層のオブジェクトの識別子に置き換える。
- 対応する下位層オブジェクトへ遠隔手続き呼び出しを用いて自分自身と同じ手続きを呼び出す。

間接オブジェクトの普通の手続きであるfile_read()では、クライアントから受け取ったメッセージの内容をそのまま下位層オブジェクトに渡している。これは、間接オブジェクトは、メッセージを解釈していないことを意味している。

3.4 下位層オブジェクトの変更手続き

この節では、下位層オブジェクトの変更手続き(change_lower())の実現方法について述べる。凍結、解凍については、4章で述べる。下位層オブジェクトの変更手続きの骨格を図3の16～27行に示す。以下にchange_lower()の処理手順を示す。

- iobj_fetch()により、オブジェクトの構造体のデータを読み出す。
- 下位層オブジェクトの識別子（io->io_lower）を新しいオブジェクト識別子に置き換える。
- iobj_dfix_update_mark()により、ファイルに保存されるようにマークする。

iobj_dfix_update_mark() については、6.3節で詳しく述べる。下位層オブジェクトの識別子の置き換え中に他のクライアントが、このオブジェクトを参照する可能性がある。よって、オブジェクト識別子を置き換える前に、enter_critical_section()を用いてロックを行なっている。enter_critical_section()、leave_critical_section()については、4.2節で述べる。

3.3節で述べた普通の手続きに対して、change_lower()では、クライアントから与えられたメッセージの内容（arg->lower）を利用している。これは、間接オブジェクトが自らメッセージを解釈して実行していることを意味している。

```
 1: file_read( arg, res )
 2:   file_read_arg_t *arg ;
 3:   file_read_result_t *res ;
 4: {
 5:   iobj_t  io ;
 6:   io = iobj_fetch( &arg->oid );
 7:   iobj_ensure_unfreezed( io );
 8:   arg->oid = io->io_lower ;
 9:   rpc_call_oid( &io->io_lower, FILE_READ,
10:   xdr_file_read_arg_t, arg,
11:   xdr_file_read_result_t, res,
12:   TIMEOUT );
13:   iobj_free( io );
14: }
15:
16: change_lower( arg, res )
17:   change_lower_arg_t *arg ;
18:   void_result_t *res ;
19: {
20:   iobj_t io;
21:   io = iobj_fetch( arg->oid );
22:   enter_critical_section(&io->iobj_lock);
23:   io->io_lower = arg->lower ;
24:   leave_critical_section(&io->iobj_lock);
25:   iobj_dfix_update_mark( io );
26:   iobj_free( io );
27: }
```

図3　普通のブジェクト手続きと
下位層オブジェクト変更手続きの骨格

4 軽量プロセス間の同期による凍結の実現

この章では、軽量プロセス間の同期による凍結手続きの実現について述べる。サーバの外から見た間接オブジェクトは、サーバの中では、構造体として実現されている。これを、間接構造体と呼ぶことにする。この章では、メモリ中の構造体についての操作について述べる。ファイルから間接構造体のデータを読み出すことについては、6章において述べる。

4.1 軽量プロセスによるサーバの構成

オブジェクトの堆積におけるサーバでは、クライアントから遠隔手続き呼び出しの要求を受け付ける

ごとに軽量プロセスが作られる [7]。軽量プロセスとは、1つのプログラム内部の並列処理の単位としてプロセスである。遠隔手続き呼び出しのサーバにおいて、要求を受け付けてから応答を行うまでをセッションと呼ぶことにする。サーバ内には、セッションに対応した軽量プロセス以外にも、定期的な作業を行うための軽量プロセスが存在する。

4.2 軽量プロセス間 advisory lock 同期プリミティブ

ここで用いた軽量プロセス・ライブラリは、様々なレベルにおいて軽量プロセス間の同期・通信プリミティブが利用可能になっている [8]。今回は、最も高いレベルにおいて定義されている advisory lock[*1] を用いて凍結を実現した。このロックは、UNIX における flock() システム・コールと同様に、ロック手続きを呼び出した軽量プロセスの間でのみ有効になるものである。

その軽量プロセス・ライブラリの advisory lock には、次のような手続きがある [7]。

(1) enter_critical_section() : 短期間の排他ロックを行う。
(2) leave_critical_section(): (1) に対してアンロックを行う。
(3) shared_lock() : 長期間の共有ロックを行う。
(4) exclusive_lock() : 長期間の排他ロックを行う。
(5) unlock() : (3), (4) に対してアンロックを行う。

ここで、(3), (4) は、それぞれＵＮＩＸの flock() システム・コールと同様の共有ロック（shared lock）と排他ロック（exclusive lock）を行う。(1), (2) は、短期間のロックやオブジェクトの状態を参照し長期のロックの方法を選択する時に用いられる。これらの手続きの利用については、4.4節において述べる。

4.3 間接構造体の状態

間接オブジェクトに対応する構造体（間接構造体）には、属性、所有者、下位層のオブジェクトの識別子を保持するフィールドがある。1つの間接オブジェクトに対応する間接構造体は、それを参照する複数の遠隔手続き呼び出しのセッションにおいて共有される。したがって、それらの操作においては、相互排除が必要になる。

間接構造体には、表1に示すような状態がある。（第1列のロックの状態については、4.4節で述べる。）状態遷移を、図4に示す。

[*1] これに対して、ロック手続きを呼び出すと、それ以外の操作も不可能になるものは、mandatory lock と呼ばれている。間接オブジェクトに対する手続き freeze(), unfreeze() は、mandatory lock の一種である。

間接構造体は、オブジェクトが生成されたり既存のオブジェクトがアクセスされた時にメモリ上に割り当てられる。割り当てられた構造体は、通常、活動中状態にある。既存のオブジェクトがアクセスされた時には、6章で述べるライブラリを用いてファイルから読み出される。この処理の間は、読み込み中状態になる。オブジェクト手続き freeze() が呼ばれると、凍結中状態になる。オブジェクト手続き unfreeze() が呼ばれると、もとの活動中状態になる。オブジェクト手続き kill() が呼ばれると削除状態になる。そして、どのセッションからも参照されなくなった時に、構造体が占めているメモリが解放される。一度参照されたオブジェクトは、しばらくの間キャッシュとしてメモリ中に留まる（キャッシュ状態）。そして、一定時間アクセスされなかった場合、そのメモリが解放される。

表1　間接構造体の状態とロックの状態の対応

ロックの状態	間接構造体の状態	意味
なし (none)	キャッシュ (cached)	どのセッションからも利用されていない。
共有 (shared)	活動中 (active)	どれかのセッションから利用されている。
排他 (exclusive)	読み込み中 (fetching)	構造体の永続部分読み込み操作を行っている。
排他 (exclusive)	凍結中 (freezed)	凍結されている。
-	削除(killed)	削除要求が受け付けられ、利用終了を待っている。

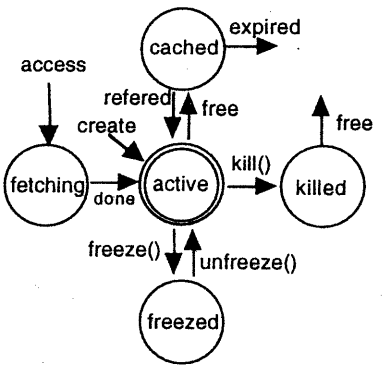

図4　間接構造体の状態遷移

4.4 間接構造体の状態とロックの状態

4.3節で述べたように、間接オブジェクトには5つの状態がある。この5つの状態を遷移するように、応用固有の軽量プロセス間同期通信プリミティブを

作ることも可能である。今回は、4.2節で述べたadvisory lock同期プリミティブを用いて状態遷移を実現した。

今回の実現の基本的な考え方は、間接構造体の状態をロックの状態に縮退させることである。これを表1に示す。このように、構造体の内容を変更する時や他のセッションを待たせる時に排他ロックを行っている。構造体が存在していることを保証するために、共有ロックを行っている。手続きfreeze()の実現には、単に排他ロックを行えばよい。ファイルからの読み込み操作においては、他のセッションを待たせるために排他ロックを行っている。

ロックの操作を容易にするために、次のような手続きを用いている。

iobj_ensure_unfreezed()：この手続きは、指定された構造体が凍結されていないことを保証する。すなわち、オブジェクトが活動状態になるまで待つ。構造体の内容を変更しない普通の手続き（3.3節）において用いられる。内部では、共有ロック（4.2節(3)）を行っている。

iobj_freeze()：この手続きは、構造体を凍結中状態に変える。既に凍結中状態であった場合には、凍結が解除されるまで待つ。オブジェクト手続きfreeze()に対応している（4.5節）。内部では、排他ロック（4.2節(4)）を行っている。

iobj_unfreeze()：この手続きは、iobj_freeze()で凍結状態になった構造体を活動中状態に変える。オブジェクト手続きunfreeze()に対応している（4.5節）。内部では、アンロック（4.2節(5)）を行っている。

4.5 凍結と解凍の実現

4.3節で述べた機能を利用して、間接オブジェクトに対する凍結手続きと解凍手続きを実現した。その骨格を図5に示す。図5では、エラー処理、メモリの解放処理が省略されている。

オブジェクト手続きfreeze()では、4.4節で述べたiobj_freeze()、オブジェクト手続きunfreeze()では、iobj_unfreeze()が呼び出されている。iobj_freeze()を呼び出した軽量プロセスは、排他ロックが確保できるまで待たされることがある。図3に示した普通のオブジェクト手続き（ファイルの読み込みfile_read()）では、iobj_ensure_unfreezed()が呼び出されている。この手続きを呼び出した軽量プロセスは、共有ロックが確保できるまで待たされることがある。

```
 1: freeze( arg, res )
 2:   oid_array_t   *arg ;
 3:   void_result_t *res ;
 4: {
 5:   iobj_t  io ;
 6:   io = iobj_fetch( arg->oid );
 7:   if( iobj_freeze( io ) == FAILD )
 8:       res->status = ERROR_FREEZED ;
 9:   iobj_free( io );
10: }
11:
12: unfreeze( arg, res )
13:   oid_array_t   *arg ;
14:   void_result_t *res ;
15: {
16:   iobj_t io ;
17:   io = iobj_fetch( arg->oid );
18:   iobj_unfreeze( io );
19:   iobj_free( io );
20: }
```

図5　凍結・解凍手続きの骨格

5 安定と永続

永続的（persistent）なデータとは、プロセスの寿命よりも長い寿命を持つようなデータである。プログラミング言語におけるデータ（変数）の寿命は、主に手続きの実行中とプログラムの実行中に分類される。C言語においてauto変数やレジスタ変数は、前者、大域変数、静的変数、ヒープ上の変数(malloc()で記憶領域を割り当てたもの)は、後者である。永続的なデータとは、プロセス（実行中のプログラム）の寿命よりもさらに長い寿命を持つデータである。このようなデータを支援するプログラミング言語が研究されている[1]。

永続と類似の言葉に、安定（stable）がある。Argusシステムでは、安定なデータとは、システムがクラッシュして再起動した後も値が保存されるようなデータという意味で使われている[4]。Argusでは、システムは、ガーディアン（guardian）と呼ばれるオブジェクトの集合として記述される。ガーディアンの変数は、安定なものと揮発的（volatile）なものに分類される。システムがクラッシュすると、揮発的なものは全て失われるが、安定なものは、失われない。システムが再起動したときには、安定な変数から揮発的な変数が回復され、ガーディアンが再生される。

Argusでいうシステムは、決して終了することはない。オペレーティング・システムのカーネルやサーバ・プロセスも、終了しないプログラムである。よって、上記の永続の定義が成り立たない。しかしながら、安定と永続は、実現する上で共通の技術が用いられる。6章で述べる永続構造体は、一般の終了するプログラムで用いれば、構造体に永続性を与えるものである。オペレーティング・システムのサーバ・プロセスに対して用いることにより、安定な

構造体が実現される。

6 永続構造体ライブラリ

この章では、構造体に永続性を与えるためのライブラリについて述べる。ここで扱う構造体は、C言語の構造体よりもクラスが大きく、可変長の配列やタグ付きの共用体を含む。

オブジェクト指向データベースでは、しばしば永続的なデータが扱えるC++言語処理系を持っている[2]。そのようなC++言語処理系と比較して、本ライブラリの特徴は、第1に1つの構造体（オブジェクト）全体ではなく、その一部分に永続性を与えるというモデルに基づいている点にある。第2に、自立しているシステムの記述に適している点にある。本ライブラリが必要とする機能は、ファイル入出力だけである。ここでファイルには、ディスク装置のように線形にアドレス付けされたデバイスを含んでいる。第3の特徴は、内部的に遠隔手続き呼び出しの技術を用いていることである。

6.1 構造体のフィールドの性質

本ライブラリでは、構造体のフィールドを次の3種類に分類して管理する（図6）。

(1) **揮発部分**（volatile part）

(2) **固定長永続部分**（fixed length persistent part）

(3) **可変長永続部分**（variable length persistent part）

(1)は、メモリ中にのみ置かれる。(2)と(3)は、ファイルに格納され、サーバがクラッシュした場合にも保持される。本ライブラリは、(2)と(3)を支援するものである。(1)については、利用者に任される。

本ライブラリでは、永続的な可変長のデータを支援する。具体的には、可変長の配列、タグ付き共用体、ポインタによる木構造（ループや共有がないもの）を扱うことができる。本ライブラリは、直接的には揮発的な可変長データを支援しない。それは、ポインタ等を用いて利用者により記述される。

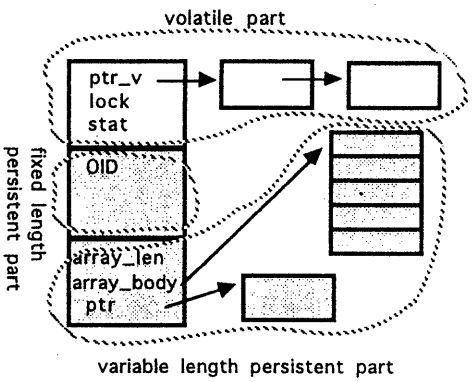

図6 構造体のフィールドの性質

6.2 ライブラリの構成

本ライブラリは、次の3つの部分から構成される。

(1) **キャッシュ表とメモリ管理軽量プロセス**：構造体は、揮発部分も永続部分も、アクセスされるたびにメモリが割り当てられるのではない。一度メモリが割り当てられた構造体は、全体がキャッシュとしてメモリ中に置かれる。このモジュールは、そのキャッシュを管理する。このモジュールには、軽量プロセスが張り付いている。それは、アクセスされなくなったキャッシュを定期的に消去する。

(2) **固定長永続部分モジュール**：構造体の固定長部分の保存・回復を行う。

(3) **可変長永続部分モジュール**：構造体の可変長部分の保存・回復を行う。

このように、構造体の永続的な部分を、固定長部分と可変長部分に分けている。そして、それぞれを別のファイルに異なる方法で保存している。固定長部分は、UNIXのアイノードと同様に、単純にファイルを永続的な配列として利用している。よってこの部分は、配列の添字で識別される。可変長部分は、1つの構造体が連続領域に配置されるように保存される。よって、この部分は、ファイルの先頭からのオフセットと大きさで識別される。

(2)、(3)において、メモリ中のデータをファイルに格納するにあたり、XDRを用いてコード化している。XDRは、SunRPCにおいて異機種間の通信を可能にするために標準的なデータ表現形式を定めたものである。XDRを利用することにより、(3)の可変長部分を容易にファイルの連続領域に格納することが可能となっている。

XDRに必要とされる手続きは、利用者によりSunRPCのスタブ生成器であるrpcgenコマンドの入力として記述される。ただし、記述されるのは、構造体の定義の部分だけで十分であり、プログラム（手続き）の部分は不要である。

6.3 ファイルへの保存

構造体は、図6に示されるような形式でメモリ中に格納されている。永続部分は、変更された場合にのみファイルへ保存される。構造体の揮発部分には、変更されたことを示すフラグが用意されている。永続部分を変更するオブジェクト手続きは、そのフラグを立てる手続きを呼び出す[*1]。遠隔手続き呼び出しの処理の最後には、それらのフラグが参照される。もし変更されていれば、永続部分は、ファイルに書き出される。

[*1] 3.4節で述べた手続き iobj_dfix_update_mark() は固定長部分を更新したことを示すフラグを立てるのである。

固定長永続部分は、そのまま同一の場所に保存される。可変長部分は、ファイル上に新たにそれを保存するために十分な連続領域領域が割り当てられ、そこに保存される。そして、今まで利用されていた領域は、解放される。

6.4 ファイルからの読み込み

構造体に対する最も基本的な操作は、クライアントから与えられたオブジェクト識別子からメモリ中に構造体を割り当てることである。間接オブジェクトにおいて、これを行っている手続きが、3.3節、3.4節、および、4.5節で述べた iobj_fetch() である。

iobj_fetch() は、次のような手順でメモリ中に構造体を割り当てる。

(1) キャッシュ表を検索する。見つかれば、それを返す。

(2) 構造体の揮発部分、固定長永続部分、および、可変長永続部分を差し示す部分（配列の大きさと先頭番地、ポインタなど）を格納するためのメモリを一括して確保する。これは、図6において、左側のデータの塊の部分に相当する。

(3) オブジェクト識別子の部分のみを設定した構造体を読み込み中状態（排他ロック）にして、キャッシュ表に登録する。以後同じオブジェクト識別子に対する構造体の検索は、成功する。しかしその構造体には、排他ロックが設定されているため、それを操作することはできない（読み込みが完了するまで iobj_ensure_unfreezed() の呼び出しから復帰しない）。

(4) 構造体の固定長永続部分を読み出す。

(5) 構造体の可変長永続部分を読み出す。このとき、配列やポインタにより差し示された構造体のメモリは、ＸＤＲの働きにより自動的に割り当てられる。

(6) その他、揮発部分の初期化を行う。

(7) 構造体を活動中状態にする。(3) で設定した排他ロックを解放する。

6.5 オブジェクト識別子

オブジェクトの堆積におけるオブジェクト識別子には、システムが解釈する部分とサーバごとに設定される部分がある。後者は、普通シリアル番号、ノード番号、チェックを含んでいる。シリアル番号は、オブジェクト識別子の一意性を実現するためのものである。オブジェクトを生成するたびに増やされる。ノード番号は、構造体の固定長永続部分を読み込む時に使われる（6.2節）。すなわち、ＵＮＩＸのアイノード番号と同様に、固定長永続部分を保存するファイルにおいて配列の添え字として利用される。最後のチェックは、オブジェクト識別子の改変と偽造を検査するためのものである。具体的には、ここに乱数を入れておく。そして、構造体の固定長永続部分に格納された識別子と比較する。

7 まとめ

本論文では、オブジェクトの堆積における間接オブジェクトの実現について述べた。オブジェクトの堆積において間接オブジェクトは、他の全てのオブジェクトの基本となるものである。今回実現した間接オブジェクトは、下位層オブジェクトの変更、凍結、および、解凍という手続きを持っている。凍結と解凍の実現には、軽量プロセス間の同期プリミティブである advisory lock が利用されている。間接オブジェクトのデータを保持する構造体は、永続構造体ライブラリを利用してファイルに保存されている。構造体の永続部分は、固定長と可変長の部分に分割され、ＸＤＲの機能を利用してファイルに格納される。そのライブラリには、揮発性の部分を含めて構造体全体のキャッシングを行う機能がある。

今後は、実現したサーバの性能評価を行う。さらに、ファイルの複製を行うオブジェクトや高度なディレクトリ操作を行うオブジェクトを実現していきたい。永続構造体ライブラリでは、可変長部分でフラグメンテーションを起こす。今後、その影響を調べ、その問題に適した解決方法を開発していきたい。

参考文献

[1] M.P.Atkinson and O.P.Buneman: "Types and Persistence in Database Programming Languages", ACM Computing Surveys, Vol.19, No.2, pp.105-190 (1987).

[2] R.G.G.Cattell (Eds.): "Next-Generation Database Systems", Communications of the ACM, Vol.34, No.10, pp.30-120 (1991).

[3] 苅部, 新城, 清木: "オブジェクト堆積モデルに基づくファイル・サーバの実現", 情報処理学会研究会報告93-OS-58, Vol.93, No.27, pp.9-16 (1993年3月)

[4] B.Liskov: "Distributed Programming in ARGUS", Communications of the ACM, Vol.31, No.3, pp.300-312 (1988).

[5] "Network Programming", Sun Microsystems,Inc. (1990).

[6] Y.Shinjo and Y.Kiyoki: "The Object-Stacking Model for Structuring Object-Based Systems", Proc. 2nd International Workshop on Object Orientation in Operating Systems (I-WOOOS'92), pp.328-340 (1992).

[7] Y.Shinjo, K.Onaga and Y.Kiyoki: "RPC and Rendezvous library with lightweight processes", Proc. 1994 Joint Technical Conference on Circuits/Systems, Computer and Communications (JTC-CSCC'94), B7-7 (July 1994). (To appear).

[8] 新城, 清木: "並列プログラムを対象とした軽量プロセス実現方式", 情報処理学会論文誌, Vol.33, No.1, pp.64-73 (1992).