DA
1112
1992
H
寄贈
新城 靖 氏
平成 年 月 日

並列分散応用プログラムを対象とした
オペレーティング・システムの研究

１９９２年

新城　靖

95003848

目次

第１章　序論・・・1

１．１　並列／分散応用プログラムとオペレーティング・システム・・・・・・・・・・・・・1

１．２　資源割当てモジュール・・3

１．３　競合と調和・・・5

１．４　主な研究成果・・・8

１．５　関連した研究・・10

１．６　論文の構成・・・13

第２章　システムの構成とカーネルの機能・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・15

２．１　システムの概観・・15

２．２　対象とするハードウェアの構成・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・16

２．２．１　結合モデルとサーバ・モデル・・・・・・・・・・・・・・・・・・・・・・・・・・・・16

２．２．２　ネットワークと機密性・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・18

２．３　対象とする応用プログラム・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・18

２．３．１　並列／分散応用プログラムと逐次応用プログラム・・・・・・・・18

２．３．２　利用者と応用プログラム・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・18

２．３．３　並列応用プログラムと分散応用プログラム・・・・・・・・・・・・・19

２．４　カーネル・・19

２．５　オブジェクトとサーバ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・21

２．６　オブジェクト識別子・・・21

２．６．１　オブジェクト識別子の構造・・・・・・・・・・・・・・・・・・・・・・・・・・・・21

２．６．２　サイト識別子・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・22

２．６．３　位置情報を含むオブジェクト識別子に関する議論・・・・・・・・24

２．７　遠隔手続き呼出し・・25

２．７．１　位置一様性とサーバ型一様性・・・・・・・・・・・・・・・・・・・・・・・・・25

２．７．２　分散システムの多重視点・・・・・・・・・・・・・・・・・・・・・・・・・・・・・26

２．７．３　遠隔手続き呼出しを行うカーネル・コール・・・・・・・・・・・・・27

２．８　認証・・・28

２．８．１　利用者識別子・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・28

２．８．２　遠隔手続き呼出しとアクセス権の確認・・・・・・・・・・・・・・・・28

２．９　名前サーバによる位置独立な名前付け・・・・・・・・・・・・・・・・・・・・・・・・29

２．１０　カーネル・サーバ群……30
２．１０．１　プロセス・サーバ……30
２．１０．２　メモリ・サーバ……31
２．１０．３　名前サーバ……31
２．１０．４　ファイル・サーバ……31
２．１０．５　タイム・サーバ……31
２．１１　関連した研究……31
２．１１．１　カーネル化されたカーネル……31
２．１１．２　マイクロカーネル……32
２．１２　まとめ……33

第３章　軽量プロセス……35
３．１　軽量プロセスの概念……35
３．２　軽量プロセス機能の実現目標……36
３．３　代表的な軽量プロセス実現方式……37
３．４　マイクロプロセスと仮想プロセッサ……38
３．４．１　他の実現方式との比較……40
３．４．２　プロセッサのスケジューリング……42
３．５　マイクロプロセスを実現するライブラリの機能と構造……43
３．５．１　sleep/wakeup モデル……44
３．５．２　他の実現方式と比較……45
３．５．３　実行の軌跡の記録と解析……49
３．６　仮想プロセッサを実現するカーネルの機能……52
３．６．１　カーネル・コール……52
３．６．２　カーネル・コールの利用例……52
３．６．３　仮想プロセッサの固有領域……55
３．６．４　仮想プロセッサの固有領域の利用……56
３．６．５　固有領域を利用しない方法との比較……57
３．７　マイクロプロセスと仮想プロセッサの利用……58
３．７．１　共有メモリ型マルチプロセッサにおける並列処理……58
３．７．２　入出力の重ね合わせ……59
３．７．３　非同期通信の実現……61
３．７．４　動的負荷分散……62
３．８　応用固有の同期・通信プリミティブの開発……63
３．８．１　層構造を持つライブラリの利用……63
３．８．２　コルーチンの性質の利用……64
３．８．３　応用固有の同期・通信プリミティブの開発例……65

３．９　応用固有のスケジューラの開発……………………………70
３．９．１　層構造を持つライブラリの利用………………………70
３．９．２　応用固有の軽量プロセス・スケジューラの開発例………70
３．１０　仮想プロセッサを実現するカーネルの構成法…………………72
３．１０．１　仮想プロセッサの固有領域の内部実現……………72
３．１０．２　実プロセッサの固有領域………………………………75
３．１０．３　仮想プロセッサとプロセスのスケジューラ…………76
３．１１　実現…………………………………………………………………79
３．１１．１　仮想プロセッサを実現するカーネルの実現…………79
３．１１．２　マイクロプロセス・ライブラリの実現………………81
３．１１．３　ＵＮＩＸにおける仮想プロセッサのエミュレーション‥84
３．１２　性能…………………………………………………………………87
３．１２．１　実験環境…………………………………………………87
３．１２．２　比較対象…………………………………………………89
３．１２．３　プロセスの生成・消滅…………………………………90
３．１２．４　コンテキスト切替えを伴わないプロセス間の同期……93
３．１２．５　コンテキスト切替えを伴うプロセス間の同期…………95
３．１２．６　仮想プロセッサの性能…………………………………98
３．１２．７　応用固有のスケジューラの性能………………………98
３．１３　関連した研究………………………………………………………101
３．１３．１　他の軽量プロセス実現方式との比較…………………101
３．１３．２　他の仮想プロセッサ方式を用いる方式との比較………102
３．１３．３　スピンロックとプロセッサの横取りの問題……………103
３．１３．４　共有ｆｏｒｋとの比較……………………………………104
３．１３．５　軽量プロセスの集合によるカーネルの構築……………105
３．１４　まとめ………………………………………………………………105

第４章　マッピング・コントローラ……………………………………107
４．１　マッピング・コントローラの概念………………………………107
４．２　マッピング・コントローラに対するＲｅＳＣシステムの方針………107
４．３　応用固有のマッピング・コントローラの支援……………………108
４．４　並列シェル…………………………………………………………109
４．４．１　並列シェルの構文と機能…………………………………109
４．４．２　プロトタイプの実現………………………………………111
４．４．３　実験…………………………………………………………112
４．５　関連した研究………………………………………………………113
４．５．１　従来の分散型オペレーティング・システムとの比較……113

４．５．２　負荷均衡の方針の研究……………………………114
４．５．３　負荷均衡の仕組みの研究…………………………115
４．６　まとめ………………………………………………115

第５章　オブジェクトの堆積……………………………………117
５．１　目標と背景…………………………………………117
５．２　基本概念……………………………………………119
５．２．１　ＳｔｄＦＳ、ＺＦＳ、ＣＦＺ…………………………122
５．２．２　サーバの堆積とオブジェクトの堆積………………124
５．２．３　層化プログラミングとオブジェクトの堆積…………125
５．２．４　ＵＮＩＸのパイプラインとオブジェクトの堆積………125
５．２．５　抽象データ型、部分型とオブジェクトの堆積………126
５．２．６　Ｃ＋＋によるオブジェクトの堆積の例………………126
５．２．７　継承とオブジェクトの堆積……………………………130
５．２．８　委譲とオブジェクトの堆積……………………………131
５．２．９　メタ・オブジェクトとオブジェクトの堆積……………131
５．３　システムに対する要求…………………………………133
５．４　オブジェクトの堆積による
分散型オペレーティング・システムの構築………………………135
５．４．１　望まれる性質…………………………………………136
５．４．２　堆積を生成する手続き………………………………137
５．４．３　ひな形オブジェクト・ベース…………………………138
５．４．４　書込み時コピー………………………………………139
５．５　種々の堆積可能オブジェクトとサーバ…………………139
５．５．１　フィルタ・サーバ………………………………………141
５．５．２　間接サーバ……………………………………………142
５．５．３　グループ・サーバ………………………………………143
５．６　分散型堆積可能オブジェクトとサーバ…………………144
５．６．１　複製ファイル・サーバ…………………………………144
５．６．２　間接オブジェクトによるオブジェクトの移動…………145
５．６．３　分散型堆積可能オブジェクト間の競合………………147
５．７　多重視点………………………………………………147
５．７．１　堆積の上下を参照することによる多重視点…………148
５．７．２　多重視点とキャッシュ…………………………………149
５．８　遠隔手続き呼出しスタブ生成器とインタフェース記述言語……150
５．８．１　インタフェース記述言語の構文と意味………………150
５．８．２　ＳｕｎＲＰＣのインタフェース記述言語との比較……152

５．９　堆積可能サーバの実現……154
５．９．１　典型的な堆積可能サーバの構成要素……154
５．９．２　キャッシングを利用した
融合ディレクトリ・サーバの実現……157
５．９．３　外部のオブジェクトへのキャッシュの保存……158
５．９．４　オブジェクトの堆積を実現する環境……158
５．１０　関連した研究……159
５．１０．１　Ｐｌａｎ９……160
５．１０．２　Ｉｎｔｅｒｐｏｓｉｔｉｏｎ……160
５．１０．３　多重継承……160
５．１０．４　手続きの堆積……161
５．１１　まとめ……162

第６章　結論……165

謝辞……171

参考文献……173

付録……181

図の目次

図１－１　並列／分散ハードウェア環境……………………………………1
図１－２　従来のシステムにおける２つの競合………………………………6
図１－３　ＲｅＳＣシステムの構成……………………………………………8

図２－１　対象とするハードウェアと応用プログラム………………………17
図２－２　カーネル・サーバと外部サーバ……………………………………20
図２－３　局所サイト型オブジェクト識別子による
　　　　　位置独立のオブジェクトの識別………………………………………23
図２－４　放送型、マルチキャスト型オブジェクト識別子による
　　　　　位置独立のオブジェクトの識別………………………………………23
図２－５　分散型システムの多重視点…………………………………………27
　　　　　図２－５（ａ）　分散型システムとして見る…………………………27
　　　　　図２－５（ｂ）　集中型システムとして見る…………………………27

図３－１　プロセス、マイクロプロセス、仮想プロセッサの関係……………38
図３－２　軽量プロセス実現方式の比較………………………………………41
　　　　　図３－２（ａ）　マイクロプロセス／仮想プロセッサ方式…………41
　　　　　図３－２（ｂ）　カーネル制御方式……………………………………41
　　　　　図３－２（ｃ）　コルーチン方式………………………………………41
図３－３　マイクロプロセス・ライブラリの構造……………………………43
図３－４　２進セマフォの実現…………………………………………………46
　　　　　図３－４（ａ）　マイクロプロセス・ライブラリによる……………46
　　　　　図３－４（ｂ）　カーネル制御方式による……………………………47
　　　　　図３－４（ｃ）　コルーチン方式による………………………………48
図３－５　実行の軌跡の記録より得られる統計情報…………………………50
図３－６　実行の軌跡の記録を視覚化したもの………………………………51
図３－７　vp_allocate() カーネル・コールによる仮想プロセッサの生成……53
図３－８　vp_sleep() カーネル・コールによる実プロセッサの開放…………54
図３－９　vp_switch() カーネル・コールによる
　　　　　他の仮想プロセッサへの制御の移動…………………………………54
図３－１０　２つの仮想プロセッサの論理アドレス空間と物理メモリの対応……55

図３－１１　配列と仮想プロセッサ識別子による　………………………57
図３－１２　ループ分割する前の逐次プログラム………………………59
図３－１３　マイクロプロセスによるループ分割………………………60
図３－１４　マイクロプロセスによる入出力の重ね合わせ………………61
図３－１５　プロセスの移動………………………………………………62
　図３－１５（ａ）　重量プロセスの移動………………………………62
　図３－１５（ｂ）　軽量プロセスの移動………………………………62
図３－１６　共有メモリ型マルチプロセッサにおける
　マイクロプロセスを利用したＳＭＡＳＨの実現………………………68
図３－１７　ネットワーク環境における
　マイクロプロセスを利用したＳＭＡＳＨの実現………………………69
図３－１８　プロセス変換表と仮想プロセッサ変換表…………………73
図３－１９　２段階のレディ・キュー……………………………………78
図３－２０　実験に用いた関係データベースへの問い合せの構造………99

図４－１　ストリームで結合されたコマンドのマッピング……………111
　図４－１（ａ）　並列シェルによるもの………………………………111
　図４－１（ｂ）　簡単な負荷分散を行う
　　マッピング・コントローラによるもの………… 111

図５－１　オブジェクト、サーバ、クライアントの関係………………119
図５－２　サーバに管理されているオブジェクトと堆積オブジェクト………121
図５－３　圧縮、暗号、標準ファイル・サーバ…………………………123
図５－４　サーバの堆積と層化プログラミング…………………………124
　図５－４（ａ）　オブジェクトの概念があるサーバの堆積…………124
　図５－４（ｂ）　オブジェクトの概念がある層化プログラミング…124
図５－５　Ｃ＋＋言語によるオブジェクトの堆積の実現………………127
　図５－５（ａ）　抽象データ型の定義…………………………………127
　図５－５（ｂ）　サーバのインタフェースの定義……………………127
　図５－５（ｃ）　サーバの実現…………………………………………128
　図５－５（ｄ）　クライアント…………………………………………128
図５－６　ｍａｉｎプログラムの実行結果………………………………129
図５－７　さまざまな堆積可能サーバ……………………………………141
図５－８　複製ファイル・サーバ…………………………………………145
図５－９　間接サーバによるオブジェクトの移動………………………146
図５－１０　３つの上位層のオブジェクトによる３つの視点の実現…………148

図５－１１　堆積オブジェクトによる３つの視点の実現……………………148
図５－１２　同期型キャッシュ・サーバ…………………………………149
図５－１３　簡単なファイル・サーバのインタフェース記述………………151
図５－１４　ＳｕｎＲＰＣにおける
簡単なファイル・サーバのインタフェース記述………………153
図５－１５　典型的な堆積可能サーバの構造……………………………154
図５－１６　オブジェクト識別子とファイル構造体………………………156
図５－１７　外部のオブジェクトへキャッシュを保存するＺＦＳの実現……158

表の目次

表１－１　２つの調和と本論文の構成・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・13

表３－１　プロセスの生成・消滅の実行時間の比較・・・・・・・・・・・・・・・・・・・・・・・・・91
　表３－１（ａ）　１回目・・91
　表３－１（ｂ）　２回目以降・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・92
表３－２　コンテキスト切替えを伴わないプロセス間の同期の実行時間の比較・・94
表３－３　コンテキスト切替えを伴うプロセス間の同期の実行時間の比較・・・・・・96
表３－４　仮想プロセッサの生成・消滅の性能・・・・・・・・・・・・・・・・・・・・・・・・・・・・98
表３－５　仮想プロセッサのコンテキスト切替えの性能・・・・・・・・・・・・・・・・・・・・・・98
表３－６　データベースの並列処理の実行結果・・・・・・・・・・・・・・・・・・・・・・・・・・・100

表４－１　実験に用いたスクリプトの性質・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・112
表４－２　スクリプトの実行時間・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・113

第１章　序論

この論文では、並列応用プログラム、および、分散応用プログラムを対象としたオペレーティング・システムの設計と実現について述べる。この章では、はじめに、本研究の背景となった分散型オペレーティング・システム、共有メモリ型マルチプロセッサ用のオペレーティング・システム、および、並列／分散応用プログラムについて考察する。次に、本研究の目的を、調和という概念を用いて述べる。そして、本研究の主な研究成果、および、関連した研究について概略を述べる。最後に、本論文の構成について述べる。

１．１　並列／分散応用プログラムとオペレーティング・システム

共有メモリ型マルチプロセッサ、および、単一プロセッサが高速ＬＡＮ (Local Area Network) によって結合された並列／分散ハードウェア環境が広く普及してきた（図１－１）。このようなハードウェア環境の利用は、エンジニアリングの分野に始った。近年では、事務処理の分野においても、１台の大型計算機を共有する形態から、複数の個人用ワークステーションをネットワークで結合して利用する形態へと変化してきている。これは、大型計算機を複数人で共有するよりも、マイクロプロセッサを利用した個人用ワークステーションをネットワークで結合して利用する方が価格性能比がよくなってきたからである。共有メモリ型マルチプロセッサも、もはや珍しいも

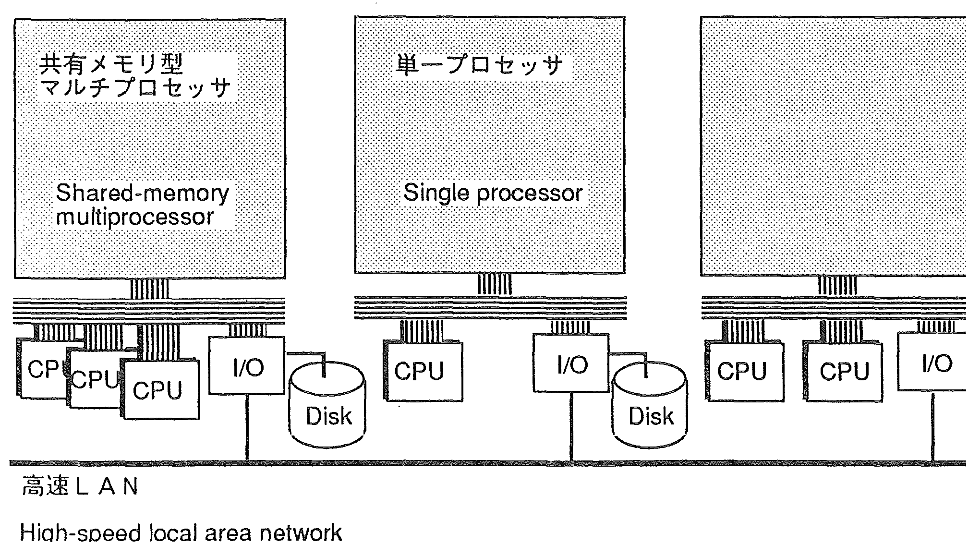


図１－１　並列／分散ハードウェア環境

Figure 1-1: A parallel/distributed hardware environment.

のではなく、しばしばワークステーションの上位モデルとして提供されている。

図１－１に示したような並列／分散ハードウェア環境の利用法を、次の２種類に分類することができる。

（１）単一プログラミング・システム

（２）多重プログラミング・システム

（１）では、そのハードウェア環境に含まれる計算機資源を１つの応用プログラムが専有し、その応用プログラムについて高い性能（performance）、信頼性（reliability）、および、可用性（availability）を実現することを目標とする。ここで計算機資源とは、ＣＰＵ、メモリ、ファイル等である。今までに数多くの並列／分散応用プログラムが、単一プログラミング・システムとして開発されてきた。このようなシステムの例としては、ＳＯＤＡ［３７］、Ａｍｂｅｒ［１１］があげられる。

単一プログラミング・システムは、しばしば単一のプログラミング言語によって記述される［５］。このような並列／分散処理記述言語の例として、ＡＲＧＵＳ［４９］、ＣＯＮＩＣ［５１］があげられる。これらのシステムでは、プログラミング言語のレベルにおいて、明示的に並列処理や分散処理が記述される。このほかに、関数型言語、論理型言語、オブジェクト指向言語等によりプログラムを記述し、言語処理系により並列性を抽出する方法も研究されている。

単一プログラミング・システムでは、プロセッサやメモリなどの計算機資源は、全て言語処理系の管理下におかれる。言語処理系とは、コンパイラ、あるいは、実行時ルーチンのことである。コンパイラや実行時ルーチンは、並列／分散処理環境を最大限に活用するように、静的、あるいは、動的に資源割当ての最適化を行う。この場合、その応用の特性を利用することにより、資源割当ての最適化を容易に行うことができる。したがって、この方法は、１つの応用プログラムについてのみ高い性能と信頼性を実現する必要がある場合には、有効である。しかしながら、並列／分散ハードウェア環境を複数の応用プログラムにより共有する必要がある場合には、保護（protection）の問題が生じるため、この方法を用いることはできない。さらに、他の応用プログラムと接続する必要がある場合にも、特別な工夫が必要となる。こうした要求に答えるためには、プログラムの保護や応用プログラム間のデータのやりとりを行うオペレーティング・システムが必要となる。

ネットワークにより結合された計算機群を、上記（２）の方法により利用するとは、分散型オペレーティング・システム（distributed operating system）を構築し、その上で複数の応用プログラムを走らせることである。代表的な分散型オペレーティング・システムの研究の例としては、Ａｍｏｅｂａ［５４］、Ｌｏｃｕｓ［６５］、Ｖシステム［１２］があげられる。分散型オペレーティング・システムに関する重要な研究課題としては、ネットワーク上に分散したファイルやプリンタなど計算機資源に対するアクセスの実現、高速なプロセス間通信の実現、負荷分散、フォールト・トレラントなファイル・システムの構築があげられる。

従来の分散型オペレーティング・システムの主要な研究課題は、分散透明性（network transparency）の実現であった。分散透明性とは、オペレーティング・システムが応用プログラムに対して仮想的な１台の集中型システムというモデルを提供し、応用プログラムがネットワークの構造を意識することなく、ネットワークに接続された計算機資源を利用することを可能にすることである。分散透明性が実現されたシステムでは、集中型システムにおいて開発された逐次応用プログラムを、全く変更することなく、並列／分散ハードウェア環境において利用することが可能となる。

ＬＡＮにより結合された計算機の利用の研究が進む一方、ＬＡＮ上の個々の計算機のマルチプロセッサ化が進んでいる。商用のマルチプロセッサとして最も普及しているものは、共有メモリ型マルチプロセッサである［６］［３２］［１００］。これは、複数のプロセッサ、１つの共有メモリ、および、入出力装置が、１つのバスで結合された構成をとる（図１－１）。プロセッサには、メモリへのアクセス速度を向上し、共有バスの負荷を抑えるために、キャッシュ・メモリが付加される。

初期の共有メモリ型マルチプロセッサは、主に従来の単一プロセッサのスループットを向上させるために利用された。従来の単一プロセッサ用の逐次応用プログラムを複数同時に走らせることで、単位時間当たりの処理可能な仕事の数を増やすことが可能となる。並列処理を行う場合、最初システムでは、従来のプロセスが用いられた。しかしながらこの方法には、プロセスの操作、すなわち、プロセスの生成・消滅やプロセス間の同期・通信のオーバヘッドが大きいという問題があった。このため、しばしば並列処理の効果が打ち消されることがあった［１０３］。

この問題を解決するために、並列処理のための効率的なプロセスの研究が盛んに行われるようになった。このようなプロセスは、軽量プロセスと呼ばれる。軽量プロセス（lightweight process）とは、従来の、多重プログラミング・システムにおける資源割当てと保護の単位としてのプロセスとは異なる、並列処理の単位としてのプロセスである。軽量プロセスは、多重プログラミング・システムの言葉であり、単一プログラミング・システムにおける並列処理の単位としてのプロセスと対応する。単一プログラミング・システムでは、保護の単位としてのプロセスは、存在しない。

単一プログラミング・システムにおいて、並列処理の単位としてのプロセスは、その応用プログラムにより管理され、スケジュールされる。この場合、応用固有の情報を利用して、効率的なスケジューリングを行うことが可能となる。一方、多重プログラミング・システムにおいても、最近では、効率的な並列処理を行うためには、軽量プロセスのスケジューリングが重要であるとの認識が深まってきた。これを支援する方法が、重要な研究課題となっている。

１．２　資源割当てモジュール

資源割当て（resource allocation）とは、ＣＰＵ、メモリ、ファイル等の計算機

資源を、ある目的をもっていくつかの主体に分配することである。目的としては、次のようなことがあげられる。

- 高速処理の実現
- フォールト・トレランスの実現
- システム全体のスループットの改善
- 利用者の間の公平の実現

1つのプログラム（1つのオペレーティング・システムを含む）の中で、資源割当ての最適化を行う部分を資源割当てモジュールと呼ぶことにする。資源割当てモジュールには、次のようなものがある。

（1）スケジューラ：
実行可能なプロセスの中から次に実行するプロセスを選択する。
（本論文では、スケジューリングという言葉とマッピングという言葉を区別して用いる。スケジューリングとは、1プロセッサ内部においてプロセスにＣＰＵを割り当てることを意味するものとする。マッピングとは、ネットワークに結合された複数のプロセッサに対して、プロセスやデータを割り当てることを意味するものとする。）

（2）マッピング・コントローラ（分散オプティマイザ）：
プロセスやデータ・ファイルをネットワーク上のどのプロセッサに割り当てるかを決定する。

（3）ページャ：
ページングに基づく仮想記憶システムにおいて、主記憶が不足した場合、どのページを2次記憶に追い出すかを決定する。

（4）ディスク・バッファ管理モジュール：
ディスク・ブロックのバッファリング（キャッシング）を行う場合、複数の入出力バッファの中から犠牲（victim）となるものを選択する。

（5）ディスク・ヘッドのスケジューラ：
複数の入出力要求の中で、ヘッドの位置をもとに、次に入出力を行うディスク・ブロックを決定する。

このような資源割当てモジュールは、旧来からオペレーティング・システムの重要な構成要素であった。現在でも、ハードウェア環境が大きく変化したにも関わらず、資源割当てモジュールがオペレーティング・システムの重要な構成要素であることには変りがない。ところが、最近の並列／分散応用プログラムの出現により状況が変化してきた。すなわち、資源割当てモジュールを内部に含むような応用プログラムが数多く開発されてきたのである。

逐次応用プログラムは、内部に処理単位を1つしか含んでいない。このため、資源割当てモジュールを持つ必要がない。これに対して、並列／分散応用プログラムは、内部に複数の処理単位を含んでいる。このため、資源割当てモジュールが必要となる。並列／分散応用プログラムの第1の目的は、高速処理や高信頼性の実現である。この

目的のためには、資源割当てが非常に重要となる。したがって、逐次応用プログラムと比較して、並列／分散応用プログラム特徴は、内部に資源割当てモジュールを含んでいる点にあるといえる。特に並列／分散応用プログラムの場合には、上記の資源割当てモジュールの中で、（１）スケジューラと（２）マッピング・コントローラが重要である。

１．３　競合と調和

従来のオペレーティング・システムは、主に逐次応用プログラムを想定して機能が設計されていた。このため、並列／分散応用プログラムという、内部に資源割当てモジュールを含む応用プログラムに対する支援が不十分であった。従来の分散型オペレーティング・システムの主要な研究課題は、分散透明性の実現であった。分散透明性が実現されたシステムの上で並列／分散応用プログラムを走らせようとすると、様々な問題が生じる。並列／分散応用プログラムは、並列処理や分散処理のために複数のプロセッサを必要とするが、従来の分散型オペレーティング・システムは、定義上、１つの仮想的な集中型システムというモデルしか提供しない。ネットワーク上の資源の位置も、並列／分散応用プログラムにとっては、資源割当てを行う際の重要な情報となるが、従来の分散型オペレーティング・システムは、この情報を覆い隠してしまう。

多重プログラミングの共有メモリ型マルチプロセッサにおいて、内部にスケジューラという資源割当てモジュールを含む並列応用プログラムを走らせる場合においても、従来の分散型オペレーティング・システムと似た問題に遭遇する。オペレーティング・システムのスケジューラは、複数の利用者や応用プログラムの間の公平なＣＰＵ資源の利用を目的として、スケジューリングを行う。この方針が、しばしば１つの応用プログラムの内部の、並列処理の単位としてのプロセス（軽量プロセス）に対しても適用される。すなわち、１つの応用プログラム内部の軽量プロセスが公平に実行される。これに対して、並列応用プログラムのレベルでは、公平なスケジューリングではなく、むしろ特定の軽量プロセスに多くのＣＰＵ資源を割り当て、その応用プログラムの全体の処理時間を短縮することを試みる。

分散型オペレーティング・システムでは、システムと並列／分散応用プログラムの間でマッピングの方針に食い違いが生じる。同様に、共有メモリ型マルチプロセッサのオペレーティング・システムにおいても、システムと並列応用プログラムの間でスケジューリングの方針に食い違いが生じる。このような食い違いを、オペレーティング・システムと並列／分散応用プログラムの間の競合（conflict）と呼ぶことにする。これを、図１－２の Conflict(1) に示す。

図１－２において、逐次応用プログラム（sequential application）は、オペレーティング・システムから分散透明性やマッピングという機能が提供されることを望ん

でいる。これに対して、並列／分散応用プログラム（parallel/distribute application）は、そのようなシステムの機能が不用であるばかりでなく、自分自身の資源割当てを妨害するものであると考える。このように、オペレーティング・システムが提供するある機能が、逐次応用プログラムにとっては必要であり、並列／分散応用プログラムにとっては、不用であることがある。このように、逐次応用プログラムと並列／分散応用プログラムの間で、システムに対する要求が対立することを、**逐次応用プログラムと並列／分散応用プログラムの間の競合**とよぶ。これを図１－２の Conflict(2) に示す。

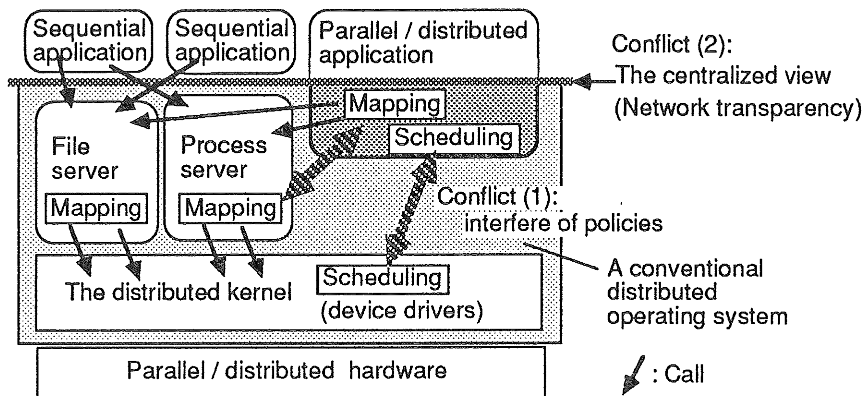


図１－２　従来のシステムにおける２つの競合

Figure 1-2: Conflicts in a conventional operating system.

本研究では、逐次応用プログラムに加えて、並列／分散応用プログラムという、内部に資源割当てモジュールを含むプログラムを対象とした分散型オペレーティング・システムにの設計と実現を行った［７６～８８］。このシステムをＲｅＳＣと名付けた。ＲｅＳＣが対象とするハードウェア環境は、共有メモリ型マルチプロセッサ、および、単一プロセッサが高速ネットワークにより結合されたものである（図１－１）。ＲｅＳＣが対象とする応用プログラムは、並列／分散応用プログラム、および、逐次応用プログラムである。ここで、並列／分散応用プログラムとは、内部に複数の処理単位と資源割当てモジュールを含むものとする。逐次応用プログラムは、内部に１つの処理単位しかなく、資源割当てモジュールを含んでいないものとする。

ここで導入した並列／分散応用プログラムと逐次応用プログラムの分類は、本システムの機能を説明するためのものである。したがって、一般的な意味における並列／分散応用プログラムが本システムの逐次応用プログラム用の機能を利用する場合も、その逆の場合も在り得る。また、システムの支援の観点では、並列応用プログラムと分散応用プログラムの差異は、小さい。なお、プログラマにより逐次的に記述されたプログラムが言語処理系により複数の処理単位に分割されたものは、本システムでは、並列／分散応用プログラムに分類される。

本研究の目的は、次の２種類の調和（harmonization）を実現することに集約される［７６］［７９］。

（１）オペレーティング・システムと並列／分散応用プログラムの間の調和：
システムは、並列／分散応用プログラムと競合することなく、そのような応用プログラムに対して、高速処理や高信頼性を実現するためにスケジューリングやマッピングを制御する機能を提供する。

（２）逐次応用プログラムと並列／分散応用プログラムの間の調和：
１つの並列／分散ハードウェア環境の中で、並列／分散応用プログラムと逐次応用プログラムを同時に利用することを可能にする。逐次応用プログラムが、システムを集中型システムとして見ることを可能する。これにより、集中型システムで開発された逐次応用プログラムを一切変更することなく、ネットワークで結合された計算機群で実行することが可能となる。同時に、並列／分散応用プログラムがシステムを並列／分散システムとして見ることを可能にする。これにより、並列／分散応用プログラムは、並列／分散ハードウェア環境をほとんどそのままの形で利用することができる。

上記（２）の目標の、逐次応用プログラムに関する部分は、従来の分散型オペレーティング・システムの研究の目標と一致する。ただし、応用プログラムの実行効率に関しては、内部に資源割当てモジュールを含む並列／分散応用プログラムを優先することにする。この結果、逐次応用プログラムは、従来の分散型オペレーティング・システムの方が効率的に動作することが予想される。しかしながら、システムの資源を大量に消費する並列／分散応用プログラムの実行効率を改善することが、結果としてシステム全体の処理能力の改善につながると思われる。

上記の目標を実現するために、分散型オペレーティング・システムを、オペレーティング・システム・カーネル上で動作する並列／分散応用プログラムの集合として設計した。これを、図１－３に示す。システムは、カーネル（kernel）、外部サーバ群（external servers）、並列シェル（parallel shell）から構成される。カーネルは、プロセス間通信等の基本的なサービスと、並列／分散応用プログラムのための効率的なサービスを提供する。外部サーバ群は、ファイルやディレクトリのキャッシング、複製、および、移動といった高度な機能を提供する。並列シェルは、ネットワーク上の負荷が小さいプロセッサを探し、コマンド群を並列に実行する。

システムの構成要素である外部サーバ群と並列シェルは、並列／分散応用プログラムと同じ立場でカーネルが管理している計算機資源を利用する。これにより、上記の調和（１）が実現される。すなわち、並列／分散応用プログラムは、システムの要素である外部サーバ群や並列シェルに含まれている資源割当てモジュールと競合することなく、カーネルが提供するシステムの計算機資源を利用することができる。

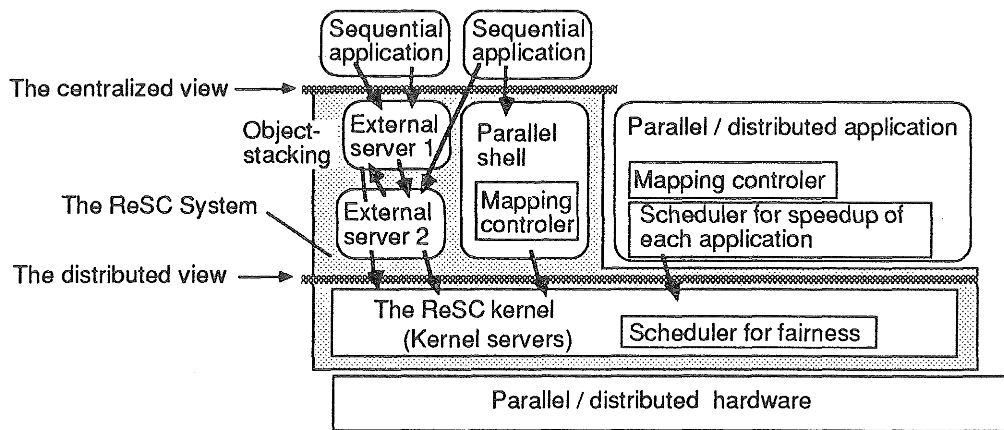


図１－３　ＲｅＳＣシステムの構成

Figure 1-3: The organization of the ReSC system.

上記の調和（２）は、並列シェルと外部サーバ群により実現される。逐次応用プログラムは、並列シェルと外部サーバ群によって提供される、従来の分散型オペレーティング・システムと同じ環境を利用することができる。同時に、並列／分散応用プログラムは、これらの機能を利用することなく、カーネルを直接利用することで、並列／分散ハードウェア環境を（多重プログラミングの範囲内で）ほとんどそのままの形で利用することができる。

本研究は、ＲＩＳＣ（reduced instruction set computer）アーキテクチャの影響を受けている。本システムのＲｅＳＣという名前は、a distributed operating system with Reduced System Calls の略で、本来は、オペレーティング・システム全体ではなくカーネルの名前であった。（現在では、「ＲｅＳＣシステム」として、オペレーティング・システム全体を指すためにも用いる。）ＲＩＳＣでは、機械語命令のレベルが内部のＡＬＵのレベルに設定されており、コンパイラがパイプラインや遅延スロットを考慮した最適なコードを生成することを期待している。これに対して、ＲｅＳＣでは、システム・コールのレベルが下位層の並列／分散ハードウェアのレベルに設定されており、並列／分散応用プログラムがマッピングやスケジューリングの最適化を行うことを期待している。

１．４　主な研究成果

本研究では、並列／分散応用プログラム、および、逐次応用プログラムを対象としたオペレーティング・システムに関連して、幅広い範囲で研究を行った。その結果得られた成果を以下にまとめる。

（１）システムの全体構成の提案

本研究では、１．３節で述べた２つの調和を実現するという研究の目的に適したオペレーティング・システムの全体構成について提案を行った［７６］［７９］。オペレーティング・システムは、全体では様々な機能を提供するモジュールの巨大な集合である。したがって、それらのモジュールを整理する方法、ならびに、応用プログラムに対するインタフェースは、重要な研究課題となっている。具体的には、次のような構成要素に対する機能分割の方法とインタフェースが課題となっている。

・カーネル
・カーネル外のサーバ
・ライブラリ
・シェル
・コマンド群
・公のファイル

本研究では、これらの構成要素に対する機能の分割の方法を提案した。

（２）カーネル機能の提案

システムの構成要素の中で特に重要となるカーネル機能の提案を行った［８２］［８７］。カーネルは、他の構成要素とは異なり、オペレーティング・システムの中で唯一実行時に利用者定義のものに取り替えられない部分である。ゆえに、どのような機能をカーネルに持たせるかが重要な研究課題となっている。本研究では、カーネルを、全ての応用プログラムに対して機能を提供するモジュールであり、多重プログラミング・システムとしての資源割当てを行う部分として位置付けた。一方、並列／分散応用プログラムにとっては、カーネルがその動作環境全体となることを考慮して機能を設計した。さらに、応用プログラムの性質に応じて、システムが集中型システムとしても分散型システムとしても利用可能になるように、システム・レベルのオブジェクト識別子とプロセス間通信機能を設計した。

（３）軽量プロセス実現方式の提案

応用固有の同期・通信プリミティブとスケジューラの開発を支援する軽量プロセス実現方式の提案を行った［８１］［８２］［８４］［８７］［８８］。具体的には、利用者レベルの軽量プロセスであるマイクロプロセスとカーネル・レベルの実体である仮想プロセッサにより軽量プロセス機能を提供する方法を提案した。マイクロプロセスは、利用者レベルで効率的に実現される。さらに、マイクロプロセスは、層構造を持つライブラリにより提供されるため、応用固有の同期・通信プリミティブやスケジューラが容易に実現される。

（４）仮想プロセッサを提供するカーネルの構成法の提案

仮想プロセッサを提供するオペレーティング・システムのカーネルを、カーネル上

で動作する並列応用プログラムと同じ手法を用いて構成する方法を提案した［７７］。具体的には、利用者に対する抽象である仮想プロセッサをカーネル内の軽量プロセスとして実現し、その軽量プロセスを実プロセッサで実行する方法を提案した。その結果、カーネルが応用プログラムと相似形になり、さまざまな利点が生じる。また、この構成法は、他の外部仕様を持つ仮想プロセッサやカーネル・レベルの軽量プロセスを実現する際にも利用可能である。

（５）オブジェクトの堆積モデルの提案

object-basedシステムにおける複数のオブジェクトの機能を統合して利用するためのモデルとして、オブジェクトの堆積（object-stacking）を提案した［７８］［８０］［８３］［８５］。そして、このモデルを利用するためのシステムの条件を明らかにした。さらに、分散型オペレーティング・システムにおける有用なサーバを提示した。

１．５　関連した研究

この節では、本研究の動機となった主な研究について述べる。具体的な他の研究との比較については、個々の章で述べる。

（１）従来の分散型オペレーティング・システム

文献［９８］において、Tanenbaum は、次のように分散型オペレーティング・システムを定義している。

> 「利用者には、通常の集中型のオペレーティング・システムのように見えるが、複数の独立したＣＰＵ（ネットワーク上のプロセッサ）の上で実行されるオペレーティング・システムである。・・・利用者は、システムを異なった計算機としてではなく、“仮想的な１台のプロセッサ”として認識する。・・・」

その他の多くの文献においても、この文献が引用され、さらに、ネットワーク・オペレーティング・システムと対比することで、類似の定義を行っている［２２］［４３］。

１．３節で述べたように、この定義に従った分散型オペレーティング・システム上では、並列／分散応用プログラムが動作しない。すなわち、“仮想的な１台のプロセッサ”では、並列処理を行うことも、分散処理を行うことも不可能である。実際のシステムでは、現実的な選択として、特殊な方法によりネットワークの位置の情報を提供している。

本研究では、この分散型オペレーティング・システムの定義、すなわち、分散型オペレーティング・システムの研究の目標を新たに設定し直した。研究の目標は、

１．３節で述べた２つの調和を実現することである。本研究における分散型オペレーティング・システムの定義を、以下に示す。

> 分散型オペレーティング・システムとは、ネットワークにより結合された計算機群を統一的に管理するオペレーティング・システムである。個々の計算機で独立したオペレーティング・システムが動作するのではなく、全体で１つのオペレーティング・システムが動作する。１つの分散型オペレーティング・システム上では、共通のオブジェクト識別子や利用者識別子が利用可能である。そして、ネットワークに接続された全ての計算機において、統一的なアクセス権のチェックが行われる。また、ネットワーク上の異なるプロセッサに配置されたプロセスの間の通信も、同一のプロセッサに配置されたプロセス間の通信と全く同一の方法で実行することができる。

ネットワーク通信機能を持ちながら、個々の計算機で独立したオペレーティング・システムが動作するものは、ネットワーク・オペレーティング・システム（network operating system）とよばれている［９８］［２２］［４３］。この定義は、本研究においても有効である。

（２）並列／分散応用プログラムを対象とした単一プログラミング・システム

並列／分散応用プログラムを対象とした代表的な単一プログラミング・システムとしては、ＳＯＤＡ［３７］とＡｍｂｅｒ［１１］があげられる。ＳＯＤＡ（a Simplified Operating system for Distributed Applications）は、本研究と同様に、ＲＩＳＣアーキテクチャの影響を受けている。ＳＯＤＡでは、多重プログラミング・システムの複雑さを排除することにより、分散応用プログラムに適したオペレーティング・システムを構築することを目指している。本研究では、従来の分散型オペレーティング・システムと同様に、多重プログラミングのシステムを構築することを目標とする。

Ａｍｂｅｒは、ワシントン大学で開発された、高い性能を得ることを目標とする応用プログラムのためのプログラミング・システムである［１１］。Ａｍｂｅｒは、ネットワークで結合された計算機群において、移動可能オブジェクト（mobile object）とスレッドからなる、object-basedプログラミング・モデルを提供する。Ａｍｂｅｒのプログラマは、オブジェクトの移動（object migration）を使ってネットワーク上のデータとプロセスの位置を制御する。

Ａｍｂｅｒでは、応用プログラムがネットワークの構成を知り、その応用プログラムの最大限の高速化のためにそれを利用する。すなわち、ネットワーク上のデータの配置は、実行時ルーチン（Ａｍｂｅｒシステム）ではなく、プログラマにより制御されるべきであると主張している。ＲｅＳＣシステムも、Ａｍｂｅｒと全く同じ立場である。相違点としては、Ａｍｂｅｒは、単一プログラミング・システムであるのに対して、ＲｅＳＣは、多重プログラミング・システムを目指していることがあげられる。

Ａｍｂｅｒでは、Ｃ＋＋を制限したオブジェクト指向言語を提供しているのに対して、ＲｅＳＣでは、特にプログラミング言語を規定していない。

（３）軽量プロセスに関する研究

軽量プロセスが普及する以前、共有メモリ型マルチプロセッサにおける並列処理では、従来の重たいプロセスが利用されていた。この段階においても、システムの負荷に適応（adapt）させるために、実行時にプロセスの数を変更することか試みられている［１０３］。一方、単一プロセッサでは、コルーチンによる軽量プロセスの実現が数多く行われた［９３］［１０６］。コルーチンの場合、並列処理を行うことができないという大きな問題点がある。

共有メモリ型マルチプロセッサにおける軽量プロセスの研究で最初に大きな注目を集めたシステムは、カーネギー・メロン大学で開発されたＭａｃｈシステムである［２］［６７］。Ｍａｃｈでは、従来のプロセスに相当するタスクに加えて、スレッドとよばれる軽量プロセスを提供している。

Ｍａｃｈでは、カーネルにおいて軽量プロセスの実現を行っている。これに対して、利用者レベルにおいて軽量プロセスを実現し、それをカーネル・レベルの仮想プロセッサにより実行する方式が、本研究［８８］［８１］をはじめ、数多く提案された［３］［２３］［３０］［５３］。これらの方式の特徴は、カーネル・レベルの軽量プロセスと比較してよりオーバヘッドが小さいこと、および、利用者レベルの情報が得やすいこと等があげられる。これらのシステムとの詳細な比較は、３．１３節において行う。

（４）オブジェクト指向システムとオブジェクト指向プログラミング言語

最近の分散型オペレーティング・システムの多くは、object-basedシステムとして構成されている［９８］［２２］［４３］。object-basedシステムとは、継承（inheritance）、あるいは、委譲（delegation）の機能が支援されている必要がないオブジェクト指向システムである。オペレーティング・システムのレベルで継承、あるいは、委譲を支援している例は、非常に稀である［４３］。したがって、オブジェクト指向オペレーティング・システムとよばれているシステムの多くは、object-basedシステムに分類される［１］［１５］［１０９］。

本研究では、object-basedシステムにおいて、複数のサーバを統合して利用するためのモデル化の手法として、オブジェクトの堆積を提案する。ＵＮＩＸのパイプは、単純なコマンドを結合し、複雑なコマンドを作るために使われる［４］［４６］。これに対してオブジェクトの堆積は、単純なオブジェクトを結合し、複雑なオブジェクトを作るために使われる。層化プログラミングは、再利用可能性が高いモジュールを構築するために有力な方法の１つである［１６］。同様に、オブジェクトの堆積も、再利用可能性が高いモジュールをサーバとして開発するための有効な方法である。

１．６　論文の構成

１．３節で述べた２つの調和と本論文の構成の関係を、表１－１にまとめる。この表においては、逐次応用プログラムと並列／分散応用プログラムの間の調和として、逐次応用プログラムに対して提供する機能を示している。正確には、この調和は、逐次応用プログラムがここであげた機能を利用し、並列／分散応用プログラムがカーネルを直接利用することで実現される。

表１－１　２つの調和と本論文の構成

調和／章	システムと並列／分散応用プログラムの間の調和	逐次応用プログラムと並列／分散応用プログラムの間の調和（逐次応用プログラムに対する機能）
２章 カーネル	・カーネル・サーバ群 ・位置情報を含むオブジェクト識別子	・名前サーバ ・位置独立のＲＰＣ発行
３章 軽量プロセス	・利用者レベルの軽量プロセス	・仮想プロセッサの自動生成
４章 マッピングコントローラ	・応用固有のマッピングコントローラの支援	・並列シェル
５章 オブジェクトの堆積		・外部サーバ群による高度な機能

本論文は、以下のように構成されている。

第２章では、システムの構成とカーネルの機能について述べる。まず、本研究の対象となるハードウェア、および、応用プログラムについて深く考察する。次にカーネルの機能とカーネル・レベルのオブジェクト識別子、および、カーネル内部のサーバについて述べる。

第３章では、軽量プロセスについて述べる。まず、軽量プロセスの概念を明確に規定し、本システムにおける軽量プロセス実現方式であるマイクロプロセス／仮想プロ

セッサ方式について述べる。次に、これを利用した応用固有の同期・通信プリミティブとスケジューラの開発について述べる。続いて、仮想プロセッサを提供するカーネルの構成法について述べる。そして、実現した軽量プロセス機能の性能を評価し、他のシステムとの比較を行う。

第４章では、マッピング・コントローラについて述べる。始めに、マッピング・コントローラの概念を明確にする。続いて、ＲｅＳＣシステムにおけるマッピング・コントローラの支援について述べる。最後に、逐次応用プログラムのためのマッピング・コントローラである並列シェルの機能とそのプロトタイプの性能について述べる。

第５章では、オブジェクトの堆積について述べる。まず、オブジェクトの堆積のモデルとそれを利用するためのシステムへの要求を整理する。次に、分散型オペレーティング・システムにおけるオブジェクトの堆積の利用について述べる。

第６章では、本論文のまとめと今後の研究の展開について述べる。

第２章　システムの構成とカーネルの機能

この章では、ＲｅＳＣシステムの構成とカーネルの機能について述べる。ＲｅＳＣは、多重プログラミングの object-based 分散型オペレーティング・システムである。ＲｅＳＣは、並列／分散応用プログラム、および、逐次応用プログラムの両方を支援する。システムの機能は、カーネル、カーネル外のサーバ、ライブラリ、シェルにより提供される。ＲｅＳＣの特徴は、カーネルの機能選択、および、システム・レベルのオブジェクト識別子に現れる。

本章では、まず本研究の対象としたハードウェア、および、応用プログラムについて述べる。次にカーネルの機能とシステム・レベルのオブジェクト識別子、および、カーネル内部のサーバについて述べる。最後に、関連した研究との比較を行う。

本システムが提供する機能の中で、軽量プロセスに関する機能については、第３章において述べる。マッピング・コントローラを支援する機能、および、シェルにより提供する機能については、第４章において述べる。第５章で述べるオブジェクトの堆積の実現では、本章で述べるカーネルの機能が重要な役割を果たす。オブジェクトの堆積で必要とされる機能については、第５章において詳しく述べる。

２．１　システムの概観

本ＲｅＳＣシステムは、共有メモリ型マルチプロセッサ、および、単一プロセッサがＬＡＮにより結合されたハードウェアを管理するオペレーティング・システムである。従来の分散型オペレーティング・システムと同様に、本システムは、マルチプログラミング、かつ、マルチユーザのシステムである。本システムでは、複数の逐次応用プログラム、および、複数の並列／分散応用プログラムが同時に動作する。そして、複数の利用者が同時にログインし、システムの機能を利用する。

システムは、カーネル、外部サーバ、ライブラリ、並列シェルから構成される。カーネルは、特権モードで動作し、下位層の並列／分散ハードウェアを管理する。そして、応用プログラムに対してハードウェアの能力を提供する。

並列応用プログラムに対しては、カーネルは、下位層の並列／分散ハードウェアをほとんどそのままの形で提供する。このような応用プログラムからは、並列／分散ハードウェアを単独で使用しているように見えるかもしれない。しかしながら、本システムは、多重プログラミング・システムであり、プロセッサ、メモリ、ネットワーク等の資源は、全ての応用プログラムにより共有されている。本システムのカーネルは、これらの資源を管理し、公平な資源の割当てを行う。

逐次応用プログラムに対しては、従来の分散型オペレーティング・システムと同様に、分散透明な資源参照を可能にする。そのために、ＲｅＳＣシステムは、ネットワーク上の位置に依らない遠隔手続き呼出しの発行、および、名前サーバによる位置に依らないオブジェクトの名前付けの機能を提供する。

本システムでは、様々なサービスを提供するプロセスをサーバとよぶ。カーネルは、クライアント（client）とサーバの間の通信機能を提供する。サーバは、カーネル内のサーバとカーネル外のサーバに分類される。前者を、カーネル・サーバ（kernel server）、または、内部サーバ（internal server）とよぶ。後者を、外部サーバ（external server）とよぶ。カーネル・サーバは、プロセス管理、メモリ管理、ファイル入出力、ネットワーク通信等の基本的な機能を提供する。カーネル・サーバは、プロセスではないが、クライアント・プロセスからは、遠隔手続き呼出しという、プロセスと同じインタフェースによりアクセスすることが可能になっている。より高度な機能は、外部サーバにより提供される。並列／分散応用プログラムは、主にカーネル・サーバを利用し、逐次応用プログラムは、主に外部サーバを利用する。外部サーバも、１つのクライアント・プロセスとして、カーネル・サーバや別の外部サーバの機能を利用する。

２．２　対象とするハードウェアの構成

図２－１に、ＲｅＳＣが対象とするハードウェアを示す。これは、ＬＡＮで結合された複数の計算機から構成されている。各計算機を、サイト（site）とよぶことにする。各サイトは、共有メモリ型マルチプロセッサ、または、単一プロセッサである。各サイトの構成、すなわち、プロセッサの数、メモリの容量、ディスクの有無は、それぞれ異なっていてもよい。

２．２．１　結合モデルとサーバ・モデル

文献［６４］において、Popek らは、システムの構成法を次の２種類に分類している。

（１）結合モデル（integrated model）
　　ほぼ同質の計算機を対等に結合する。

（２）サーバ・モデル（server model）
　　ファイル・サーバ、プリント・サーバ、広域ネットワーク・サーバなど、専用機能を持つサーバを用いる。

この分類法に従うと、本システムが対象とするハードウェア構成は、（１）に分類される。Ａｍｏｅｂａ［５４］やＶシステム［１２］のハードウェア構成は、（２）である。

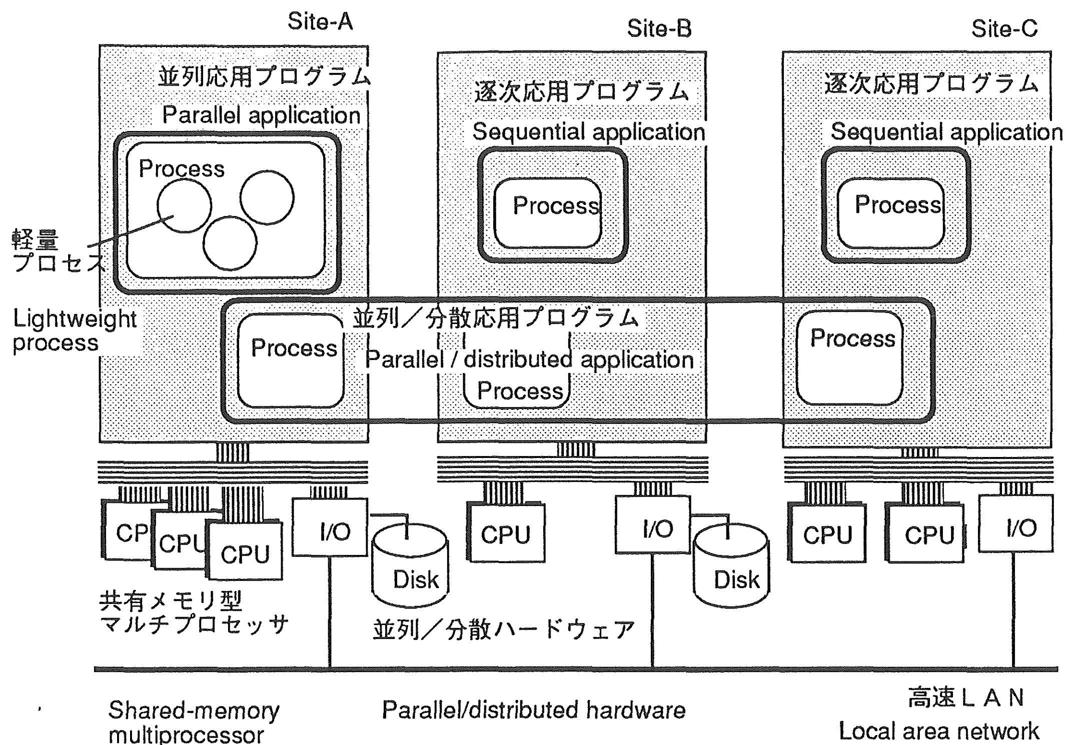


図２－１　対象とするハードウェアと応用プログラム

Figure 2-1: The hardware environment and applications of ReSC.

このようなハードウェア構成を仮定した理由を、以下に示す。

（１）入出力の高速化を図ることができる。近年、プロセッサの高速化とメモリの大容量化は、目をみはるものがある。これに対して、ディスク装置の速度は、それほど改善されていない。図２－１のように、複数の計算機にディスクを接続することにより、複数の入出力を並列に実行することが可能となり、入出力の高速化を図ることができる。また、ディスク・アレイ［６２］と同様に、複数のサイトのディスクにデータを重複して保存することにより、ディスクの故障やサイトの故障に対応することが可能となる。

（２）計算機の入れ替えが容易である。古い計算機を取り外したり、新しい計算機を組み入れることが容易である。大容量のディスクを供えた集中化されたファイル・サーバ専用計算機は、導入した瞬間には、すばらしい構成に思える。しかしながら、それを導入した翌年には、個人用のワークステーションがそのファイル・サーバよりも高速のプロセッサを供えることになる。さらに、そのファイル・サーバよりも大容量のディスクを備える可能性もある。

本システムでは、共有メモリ型マルチプロセッサの各プロセッサを、互に等価なも

のとして扱う。したがって、応用プログラムは、特定のプロセッサを指定してプロセスを生成することはできない。これは、キャッシュ・メモリの効果を無視すれば、各プロセッサから共有メモリへのアクセス時間が等しくなるからである。ただし、各応用プログラムに対して割り当てられた（複数の）プロセッサは、その応用プログラムの内部で自由に利用される。

２．２．２　ネットワークと機密性

本研究が対象とするネットワークは、ＬＡＮ（Local Area Network）とする。ＷＡＮ（Wide Area Network）との接続は、ゲートウエーを介して行うこととする。したがって、本研究では、ネットワークの機密性が保たれているものと仮定する。

２．３　対象とする応用プログラム

２．３．１　並列／分散応用プログラムと逐次応用プログラム

ＲｅＳＣでは、対象とする応用プログラムを次の２つに分類する（図２－１）。

（１）並列／分散応用プログラム（parallel/distributed application programs）
　　内部に複数の処理単位と資源割当てモジュールを含んでいる。

（２）逐次応用プログラム（sequential programs）
　　内部に１つの処理単位しか含んでいない。

この分類方法は、システムの支援の違いを議論するときに有用である。すなわち、本システムでは、前者に対しては、プログラムの実行効率を重視した支援を行い、後者に対しては、プログラムの記述性を重視した支援を行う。ただし、この分類は、完全に排他的ではない。ある並列／分散応用プログラムが、効率に大きな影響を与えない部分では、逐次応用プログラムに対する機能を利用することもある。

２．３．２　利用者と応用プログラム

ＲｅＳＣでは、利用者（user）と応用プログラム（application program）をはっきり区別して考える。利用者は、応用プログラムを利用して仕事をする人間である。応用プログラムは、利用者とは異なる人間（プログラマ）により記述されるものと仮定する。この論文で述べる応用プログラムに対する支援とは、応用プログラムのプログラマに対して提供するシステムのモデル、並びに、そのプログラマが記述した実行時ルーチンに対して提供するシステムの機能を意味する。

２．３．３　並列応用プログラムと分散応用プログラム

システムによる支援の視点からは、並列応用プログラムと分散応用プログラムの差異は、小さい。したがって、本論文では、並列応用プログラムと分散応用プログラムをまとめて扱うことが多い。あえてこれらの２つを分類すると、次のようになる。

（１）並列応用プログラム：

高性能（high performance）を得ることを主目的とする応用プログラム。そのためには、共有メモリ型マルチプロセッサを利用したり、あるいは、ネットワークにより結合された計算機群を利用する。

（２）分散応用プログラム：

高い信頼性（reliability）や可用性（availability）を得ることを主目的とする応用プログラム。そのためには、ネットワークに結合された複数の計算機を利用する。１台の共有メモリ型マルチプロセッサだけを利用することはない。

２．４　カーネル

ＲｅＳＣシステムは、カーネル、外部サーバ、ライブラリ、並列シェルから構成される（図２－２）。この中でカーネルは、最も重要な構成要素である。それは、全ての計算機上にコピーが存在し、全ての応用プログラムから利用されるからである。カーネルは、特権モードで動作する。外部サーバや並列シェル等の他の構成要素とは異なり、カーネルを利用者定義のものに取り替えることはできない。

カーネルは、以下のような機能を提供する。これ以外の機能は、外部サーバ、並列シェル、または、ライブラリの形で提供される。

（１）共有資源の管理と公平な配分。
　（メモリ管理、ＣＰＵのスケジューリング）
（２）プロセス間通信と保護。
（３）並列／分散応用プログラムのための、単純で効率的な形の下位層のハードウェアが持つ能力。
　（ファイル・サービス、プロセス・サービス、ネットワーク・サービスなど）
（４）標準的なインタフェース

上記（１）と（２）は、多重プログラミング・システムとしての機能である。上記（３）に本システムの特徴がある。ＲｅＳＣシステムは、他の分散型オペレーティング・システムと同様に、「カーネルを可能な限り小さくする」という方針に基づいて設計されている。相違点は、ＲｅＳＣでは、カーネルがデバイス・サーバではなく、完全なサーバを含んでいる点にある。たとえば、Ｖシステムでは、カーネル中のデバ

イス・サーバは、それぞれの物理的なディスク装置について生のブロック型デバイス（a raw block device）としてのアクセスを提供する。そして、カーネル外のファイル・サーバは、そのデバイス・サーバを利用して、ファイルという抽象を実現する。ＲｅＳＣでは、カーネル中に完全なファイル・サーバが存在する。この設計により、カーネルが大きくなってしまうが、効率は、良くなる。なぜならば、保護の壁を横切るオーバヘッドが減るからである。Ｖシステムとの違い関しては、２．１１．１項においてさらに詳しく述べる。

上記（４）は、オブジェクトの堆積を用いてサーバを統合する時に必要である。オブジェクトの堆積の詳細については、第５章において述べる。

カーネル内のサーバを、カーネル・サーバ（kernel server）、または、内部サーバ（internal server）とよぶ（図２－２）。一方、カーネル外にあるサーバを外部サーバ（external server）よぶ。外部サーバは、プロセスまたは、プロセスの集合である。カーネル・サーバについては、２．１０節において述べる。外部サーバについては、５章において述べる。

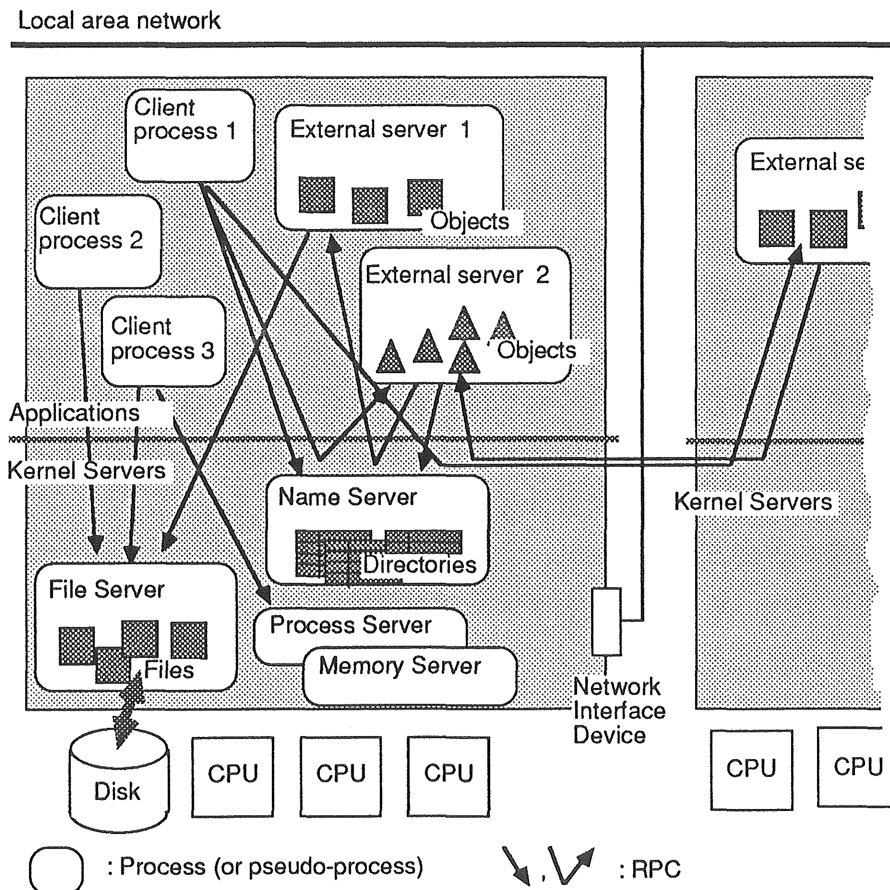


図２－２　カーネル・サーバと外部サーバ

Figure 2-1: Kernel servers and external servers.

２．５　オブジェクトとサーバ

ＲｅＳＣシステムは、object-basedシステムである。たとえば、ファイル、ディレクトリ、プロセスのようなソフトウェア的な資源は、オブジェクト（object）として統一的に扱われる。システムの機能は、そのようなオブジェクトを通じて提供される。

オブジェクトは、それぞれのサーバにより管理される。サーバとは、クライアント・プロセスに対してサービスを提供するプロセス、プロセスの集合、あるいは、カーネル中の疑似プロセスである。疑似プロセス（pseudo-process）とは、カーネル中に在り、プロセスではないが、クライアント・プロセスから普通のプロセスと同じように見えるものである（図２－２）。

サーバは、オブジェクトを管理するオブジェクト、すなわち、オブジェクトのメタ・オブジェクト（meta-object）に相当する。オブジェクトの操作は、そのオブジェクトのサーバに対してメッセージを送ることにより行われる。サーバ自身は、プロセス・オブジェクトであり、カーネル内のプロセス・サーバにより管理される。またカーネルは、プロセス間通信機能、すなわち、オブジェクトに対してメッセージを送る機能を提供する。プロセス間通信については、２．７節において述べる。

カーネルは、サーバのサーバ、すなわち、メタ・オブジェクトのメタ・オブジェクトになっている。そしてカーネル自身は、ハードウェアにのみ依存し、ソフトウェア的には自立（self-support）、および、自律（autonomic）している。カーネルは、カーネル外のオブジェクトやサーバに依存していない。この性質により、カーネルの動作が安定し、カーネルの開発が容易になる。

２．６　オブジェクト識別子

本システムでは、システム・レベルのオブジェクトの統一的な扱いを可能にするために、一様なオブジェクト識別子を導入した。このオブジェクト識別子の特徴は、位置依存のオブジェクトと独立独立のオブジェクトの両方を統一的に識別可能である点にある。

２．６．１　オブジェクト識別子の構造

本システムのオブジェクト識別子は、次の３つ組から構成される。

＜サイト識別子、サーバ識別子、オブジェクト番号＞

サイト識別子（site-id）は、そのオブジェクトを管理するサーバが存在するサイトを示す数である。（サイト識別子については、２．６．２項において詳しく述べる。）
サーバ識別子（server-id）は、そのオブジェクトを管理するサーバに割り当てられ

た、遠隔手続き呼出しのポート番号である。ネットワーク上に分散したプロセスの集合が、1つのサービスを提供する場合、それらのプロセスは、1つのサーバ識別子を共有する。**オブジェクト番号**(object-number)は、各サーバにより管理される番号である。

2．6．2　サイト識別子

ＲｅＳＣでは、位置依存のオブジェクトと位置独立のオブジェクトの両方を支援する。これを実現するために、次の3種類のサイト識別子を導入した。

(1) **サイト一意型サイト識別子**(a site-uniq site-id)
これは、各サイトに一意になるように割り当てた数である。インターネット［１０１］におけるホスト・アドレスに対応する。

(2) **局所サイト型サイト識別子**(the local site site-id)
これは、局所サイトを示す数である。インターネットにおける、ソフトウェア・ループバック・ネットワーク・インタフェース(software loopback network interface)のアドレスに相当する。多くのインターネットのホストにおいて、このアドレスは、ホスト名"localhost"により参照されるアドレス、すなわち、127.0.0.1 になっている。

(3) **放送型、あるいは、マルチキャスト型サイト識別子**
(a broadcast or multicast site-id)
これは、放送(broadcast)、あるいは、マルチキャスト(multicast)を示す数である。インターネットにおける放送、または、マルチキャストを表すアドレスに相当する。

ここで、(1)を含むオブジェクト識別子を**サイト一意型オブジェクト識別子**(a site-uniq object-id)とよぶことにする。同様に、(2)含むオブジェクト識別子を**局所サイト型オブジェクト識別子**(a local-site object-id)、(3)を含むオブジェクト識別子を、**放送型オブジェクト識別子**(a broadcast object-id)、あるいは、**マルチキャスト型オブジェクト識別子**(a multicast object-id)とよぶことにする。

サイト一意型オブジェクト識別子を利用することにより、位置依存のオブジェクトを識別することができる。したがって、サイト一意型オブジェクト識別子を、**位置依存型オブジェクト識別子**(a location-dependent object id)とよぶ。以下で述べるように、局所サイト型、放送型、あるいは、マルチキャスト型のオブジェクト識別子を用いることで、位置独立なオブジェクトを識別することができる。このようなオブジェクト識別子を、**位置独立型オブジェクト識別子**(a location-independent

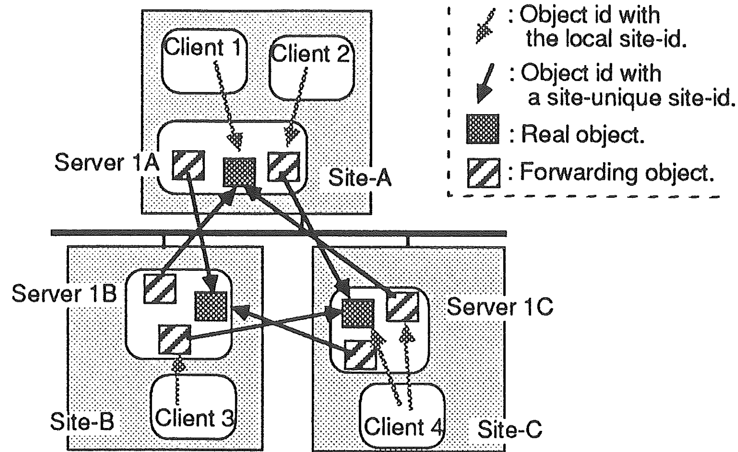


図２－３　局所サイト型オブジェクト識別子による
位置独立のオブジェクトの識別

Figure 2-3: Identifying location independent objects
by the object-id with the local site-id.

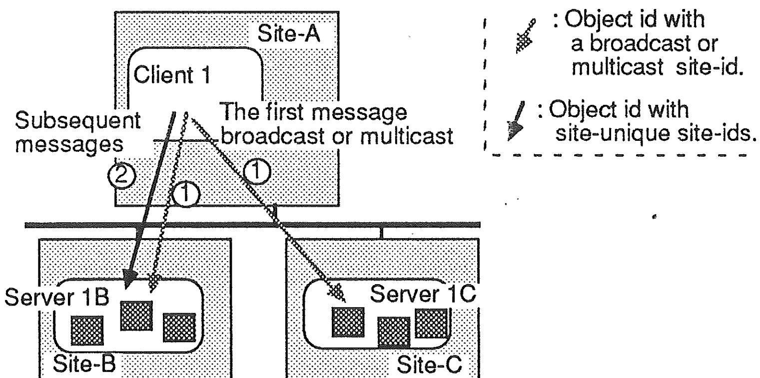


図２－４　放送型、マルチキャスト型オブジェクト識別子による
位置独立のオブジェクトの識別

Figure 2-4: Identifying a location independent object
by the object id with a broadcast or multicast site-id

object-id）とよぶことにする。

局所サイト型オブジェクト識別子により指し示されたオブジェクトを利用する場合、クライアントは、自分自身と同一のサイト上のサーバに要求を送る（図２－３）。そのサーバは、操作対象となるオブジェクトの位置を知っている。自分自身がそのオブ

ジェクトを管理している場合、サーバは、その要求に応答する。そうでない場合、別のサイト上に在り、実際にそのオブジェクトを管理しているサーバに、その要求メッセージを転送（forward）する。

放送型、あるいは、マルチキャスト型オブジェクト識別子により指し示されたオブジェクトを利用する場合、クライアントは、放送、または、マルチキャストを使って要求を送る（図２－４）。以後、応答を返したサーバと通信する。これは、Ｖシステムや Ａｍｏｅｂａシステムなど他の分散型オペレーティング・システムで用いられている方法とよく似ている。相違点は、本システムの場合、オブジェクト識別子中のサーバ識別子を見ただけでは、サーバの位置を特定する事ができないこと、および、複数のサーバが応答する可能性があることである。

２．６．３　位置情報を含むオブジェクト識別子に関する議論

オブジェクト識別子に位置情報を含めることについては、議論がある。多くの分散型オペレーティング・システムでは、位置情報を含まないオブジェクト識別子が好んで使われている。その理由は、システムのレベルでオブジェクトの複製やオブジェクトの移動を実現することが可能になるからである。

ＲｅＳＣでは、以下の理由によりオブジェクト識別子に位置情報を含めることにした。

（１）分散型システムには、位置独立のオブジェクトに加えて、位置依存のオブジェクトが存在する。位置依存のオブジェクトの例としては、物理ディスク装置、ディスプレイ、キーボード、フロッピ・ディスク装置があげられる。

（２）並列／分散応用プログラムは、位置依存のオブジェクトを利用する。たとえば、分散応用プログラムの１つとして、複製を行うファイル・サーバのプログラムを書くことを考える。この時、複製ファイル・サーバは、オブジェクトの位置に注意を払う必要がある。もし１つのファイルについて、全てのコピーが１つのサイトに置かれたならば、複製は働かない。（オブジェクトの堆積による複製ファイル・サーバについては、５．６．１項で述べる。）

（３）位置独立のオブジェクトは、２．６．２項で述べた位置独立型オブジェクト識別子（局所サイト、放送、あるいは、マルチキャストを意味するサイト識別子を含むオブジェクト識別子）を使って識別することができる。

文献［４７］では、オブジェクト（文献［４７］では、ファイルのみ）の名前付け（名前には、識別子を含む）とオブジェクトの位置に関して、次のような概念を規定している。

・位置透明性（location transparency）

オブジェクトの名前には、それが物理的にどこに置かれているかを示す一切の示唆が含まれない。

・位置独立性（location independence）

オブジェクトの物理的な位置が変っても、オブジェクトの名前を変更する必要がない。

ここで、位置独立性の方が強い概念である。すなわち、位置独立性を実現するためには、位置透明性に加えて、オブジェクトの位置が変化した前後（オブジェクトの移動の前後）で、アクセスの透明性を保つために、動的にオブジェクトの位置を特定する機能が必要になるからである。

ＲｅＳＣのオブジェクト識別子に対しては、この分類方法は、無効である。すなわち、２．６．２項で述べた位置依存型オブジェクト識別子（サイト一意型オブジェクト識別子）は、位置透明性も位置独立性も満たしていない。しかしながら、位置独立型オブジェクト識別子（局所サイト型、放送型、マルチキャスト型）は、位置独立性を満たしている。このように、ＲｅＳＣのオブジェクト識別子は、性質が異なるオブジェクト識別子が同一のドメインに含まれている点に特徴がある。

２．７　遠隔手続き呼出し

カーネルは、プロセス間通信プリミティブとして遠隔手続き呼出し（RPC, Remote Procedure Call）を提供する。遠隔手続き呼出しは、同期式の通信を実現する。したがって、カーネルにおいてパラメタのバッファリングを行う必要がなく、データのコピーのオーバヘッドを軽減し、効率を改善することができる。また、遠隔手続き呼出しは、通常の手続き呼出しと同じ様に扱うことができるので、使いやすい［９８］。また、非同期式の通信が必要となる局面では、第３章で述べる軽量プロセスを用いることにより、それを実現することも可能である。さらに、エラーを考えると、ほとんど全てのサーバの呼出しは、必ず結果を返すので、単方向の通信プリミティブと比較して双方向の通信プリミティブである遠隔手続き呼出しが適している。

遠隔手続き呼出しは、Ａｍｏｅｂａ、Ｖシステムをはじめとして、他の分散型オペレーティング・システムにおいて広く採用されている［９８］［１２］［５４］。

２．７．１　位置一様性とサーバ型一様性

ＲｅＳＣの遠隔手続き呼出しは、次の２つの一様性を備えている。

（１）位置一様性：

クライアントは、呼び出すサーバが局所サイト上にあるか、遠隔サイト上にあるか、サーバの位置によらず、同一の方法で呼び出すことができる。このことは、ある種の分散透明性を実現していることになる。ゆえに、ＲｅＳＣは、ネットワーク・オペレーティング・システムではなく、分散型オペレーティング・システムの範疇に分類される。

（２）サーバ型一様性：

クライアントは、呼び出すサーバが外部サーバであるか、カーネル・サーバであるか、サーバの型によらず、同一の方法で呼び出すことができる。本システムでは、カーネルの内部にプロセス・サーバやファイル・サーバを含んでいる。そして、カーネルに対するサービスの要求も、他のカーネル外のサーバの呼出しと同一のインタフェースにより行うことが可能になっている。この性質を実現することにより、システムの拡張性が高められる。その理由は、カーネル・サーバもカーネル外のサーバもクライアントから区別ができないため、高い機能を持つサーバをカーネル外に構築することが容易になるからである。

２．７．２　分散システムの多重視点

位置一様な遠隔手続き呼出しと位置が見えるオブジェクト識別子により、分散システムの多重視点（multiple view）が実現される。並列／分散応用プログラムは、システムを分散システムとして見て、遠隔サイト上のサーバに遠隔手続き呼出しを行い、遠隔サイトにファイルやプロセスを生成することができる。同時に、逐次応用プログラムは、システムを集中型システムとして見て、遠隔サイト上のファイルを局所サイト上のファイルと同じ方法でアクセスすることができる。

図２－５（ａ）に、システムを分散型システムとして見る例を示す。この図において、Process 1 が Site-B 上のプロセス・サーバを遠隔手続き呼出しにより呼び出し、Site-B 上に Process 2 を生成している。Process 2 は、局所サイト上のファイル・サーバを呼び出し、ファイルを読み込んでいる。そして、フィルタ操作を行い、ヒットしたデータのみを Site-A 上の Process 1 に送っている。この結果、ネットワーク上を転送されるデータ量が削減され、Process 1 と Process 2 の間で並列処理が実現されている。

図２－５（ｂ）に、システムを集中型システムとして見る例を示す。この図において、 Process 1 は、局所サイト Site-A 上のファイルと全く同じ方法で遠隔サイト Site-B 上のファイルをアクセスしている。

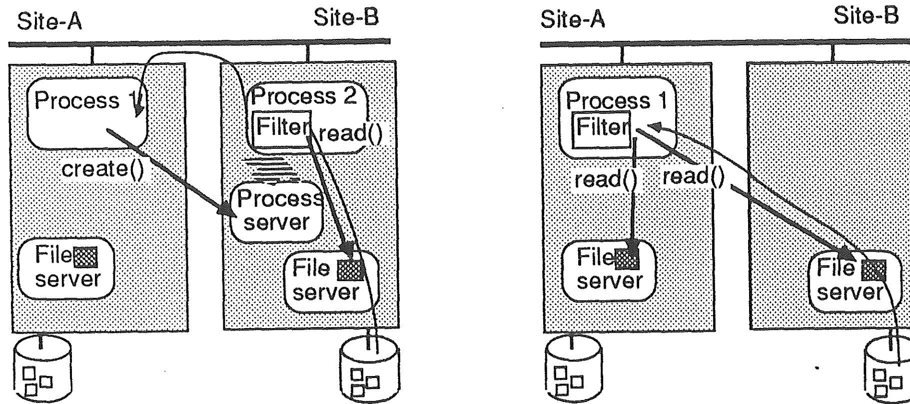


図２－５(a)　分散型システムとして見る
Figure 2-5(a): Viewing as a distributed system.

図２－５(b)　集中型システムとして見る
Figure 2-5(b): Viewing as a centralized system.

図２－５　分散型システムの多重視点
Figure 2-5: Multiple views of the distributed system.

２．７．３　遠隔手続き呼出しを行うカーネル・コール

遠隔手続き呼出しの機能は、次の５つのカーネル・コールにより提供される。（１）は、クライアントにより、残りは、サーバにより利用される。

（１）rpc_send()

これは、クライアントがサーバを呼び出すカーネル・コールである。クライアントは、サーバからの応答が到着するまでブロックされる。

（２）rpc_receive()

これは、サーバが、クライアントから要求を受け付けるカーネル・コールである。サーバは、要求が到着するまで、ブロックされる。

（３）rpc_reply()

これは、サーバが、クライアントへ応答を返すカーネル・コールである。

（４）rpc_forward()

これは、サーバが、他のサーバへ要求を転送するカーネル・コールである。

（５）rpc_send_with_uid()

これは、サーバが、別の利用者識別子を伴い他のサーバを呼び出すカーネル・コールである。

付録Ａに、これらのカーネル・コールの詳細と、それらを利用したサーバとクライアントの例を示す。

サーバでは、普通、１つの遠隔手続き呼出しのセッションについて１つの軽量プロセスが生成される。その軽量プロセスは、クライアントからの要求を処理し、クライアントへ応答を行い、終了する。軽量プロセスを用いたサーバの構築については、５．９．１項において述べる。これに対して、軽量プロセスを生成することなく要求に対する処理を行い、応答を返す方法がある。この場合、Ａｄａ［６８］におけるランデブ（rendezvous）と同じ動きになる。

２．８　認証

本システムは、マルチユーザのシステムである。したがって、一般の分散型オペレーティング・システムと同様に、利用者の認証（authentication）を行う必要がある。本システムでは、ＬＡＮという機密性が保たれた環境を利用して、集中型システムにおいて用いられている方法と同じように、暗号化の技術を用いない単純な方法により利用者の認証を行う。

２．８．１　利用者識別子

本システムでは、オブジェクトへのアクセス権の確認、セキュリティのチェックのために、統一的な利用者識別子を導入する。これは、次の組から構成される。

＜利用者番号、マスク＞

利用者番号は、利用者に対して一意になるように割り当てた整数である。マスクとは、対応する整数の有効範囲を現すものである。マスクのビットが１であることは、対応する整数のそのビットが有効であることを意味する。アクセス権の確認は、マスクのビットが１の部分について行われる。一般の利用者のマスクとしては、全ビット１（３２ビットの整数として、0xFFFFFFFF）の整数を用いる。サーバ・プロセスを起動したり、システム管理を行うための特権利用者（ＵＮＩＸにおけるスーパーユーザ）を示す利用者識別子としては、マスクが０である利用者識別子を用いる。これにより、特権利用者は、アクセス権のチェックをまったく受けることなく、全てのオブジェクトを操作することが可能になる。マスクを適当に設定することにより、インターネット［１０１］におけるサブネットと同様に、グループを設定することも可能である。付録Ａ－３と付録Ａ－４に、この利用者識別子の利用例を示す。

２．８．２　遠隔手続き呼出しとアクセス権の確認

カーネルは、個々の遠隔手続き呼出しにおいて、クライアント・プロセスに対応している利用者の識別子をサーバに渡す。サーバは、これを利用して、オブジェクトに対するアクセス権を確認する。

アクセス権を確認するために、プロセスの利用者の情報を付加することは、カーネルにおいて行う必要がある。たとえば、ＳｕｎＲＰＣ［６９］では、ライブラリにお

いて（ＵＮＩＸの）利用者識別子を遠隔手続き呼出しのヘッダに付加している。このため、容易に利用者識別子を偽ることが可能になっている。たとえば、次のような関数をリンクするだけで、利用者識別子が 0 の利用者（ＵＮＩＸでは、スーパーユーザ）になりすますことができる。

```
getuid()
{
        return( 0 );
}
```

これは、ＳｕｎＲＰＣのライブラリの内部の、遠隔手続き呼出しのヘッダを作成する部分で、getuid() という関数を呼び出しているからである。getuid() は、本来、トラップ命令を含むシステム・コールである。この例では、利用者によりシステム・コールがオーバーライド（override）されている。その結果、アクセス権の確認が無効になっている。

最近のＳｕｎＲＰＣでは、特権利用者の確認を、遠隔手続き呼出しのヘッダ内の利用者識別子ではなく、インターネットの特権ポートを利用しているかどうかにより判定している。これにより、上記の問題は、解決されている。しかしながら、特権利用者以外の一般利用者については、問題が残されている。ＲｅＳＣシステムでは、カーネルにおいて利用者の情報を付加するので、このような問題は生じない。

２．９　名前サーバによる位置独立な名前付け

ＲｅＳＣでは、オブジェクトの文字列の名前は、そのオブジェクトのサーバではなく、名前サーバ（name server）により管理される。名前サーバは、ディレクトリ（directory）、すなわち、オブジェクトの文字列の名前とオブジェクト識別子の組のリストを管理する。これにより、名前サーバは、ディレクトリ・サーバ（directory server）ともよばれる。名前サーバは、クライアントから検索（lookup）要求を受け付けると、渡された文字列が指定されたディレクトリに含まれているかどうかを検索する。その結果、含まれていた場合、対応するオブジェクト識別子をクライアントに返す。

２．６節において述べたように、オブジェクト識別子の中には、位置情報が含まれている。それにもかかわらず、文字列による名前付けでは、２．６．３項で述べた位置透明性と位置独立性が実現されている。すなわち、オブジェクトの文字列の名前は、オブジェクトの位置が変化したとしても、変更する必要はない。このため、利用者は、局所サイト上のオブジェクトと同じ方法で遠隔サイト上のオブジェクトを扱うことができる。このような位置透明性のことを、文献［６０］では、名前透明性（name tra

nsparency）とよんでいる。

ＲｅＳＣの名前サーバ（ディレクトリ・サーバ）は、Ａｍｏｅｂａのディレクトリ・サーバと類似している。Ａｍｏｅｂａでは、ディレクトリ・サーバを利用して、書き換えがない（immutable）ファイルに対してアトミック・トランザクションを実現している［５４］。ＲｅＳＣのディレクトリ・サーバも、同じ目的に利用することが可能である。これに加えて、ＲｅＳＣでは、書き換えがないオブジェクトに対して、オブジェクトの移動（object migration）を実現することができる。オブジェクトの移動は、その操作の実行後でオブジェクトの位置が変るようなアトミック・トランザクションにほかならない。５．６．２項では、オブジェクトの堆積による書き換えがある（mutable）オブジェクトに対するオブジェクトの移動について述べる。

２．１０　カーネル・サーバ群

この節では、カーネル・サーバの機能の概観を示す。カーネル・サーバには、プロセス・サーバ、名前サーバ、メモリ・サーバ、ファイル・サーバ等がある。カーネル・サーバは、下位層の並列／分散ハードウェアの機能を、過度に仮想化することなく、ほとんどそのままの形で提供する。そして、キャッシング、マッピングといった、並列／分散応用プログラムが好んで制御するような方針を、並列／分散応用プログラムに任せる。逐次応用プログラムに対する高度な機能は、外部サーバにおいて提供する。外部サーバについては、第5章で詳しく述べる。

カーネル・サーバは、プロセスではなく、カーネル内のモジュールである。それらのモジュールは、互に依存関係にある。しかしながら、カーネルは、カーネル外のモジュールに依存することはない。すなわち、カーネルは、自立して動作することが可能である。これにより、カーネルの動作が安定し、かつ、セキュリティを高めることが可能となる。

主なカーネル・サーバの機能を以下に示す。これらのカーネル・サーバのインタフェースを、付録Bに示す。

２．１０．１　プロセス・サーバ

プロセス・サーバは、次の様な機能を提供する。

（１）プロセスの生成・消滅

（２）仮想プロセッサの生成・消滅

（３）プロセッサのスケジューリング

（４）プロセッサの利用状況の提供

仮想プロセッサに関する機能の詳細は、第3章において述べる。プロセッサのスケジューリングは、通常の時分割システム　（TSS, Time Sharing System）と同様な方針により行われる。（４）の機能は、第4章で述べるマッピング・コントローラにより

利用される。

２．１０．２　メモリ・サーバ

メモリ・サーバは、主記憶の管理を行う。主な機能を、以下に示す。

（１）プロセスに対してメモリ・オブジェクトを割り当てる。

（２）プロセスのメモリ・オブジェクトのマッピングを変更する。

（３）カーネル内で用いるメモリを管理する。

２．１０．３　名前サーバ

名前サーバ（ネーム・サーバ、ディレクトリ・サーバ）は、２．９節で述べたように、与えられた文字列を対応するオブジェクト識別子に変換するサーバである。カーネル内の名前サーバは、基本的な機能を提供する。キャッシングや複製といった高度な機能を提供する名前サーバは、カーネルの外に外部サーバとして構築される。

２．１０．４　ファイル・サーバ

カーネル・ファイル・サーバは、標準ファイルに対するブロックごとの入出力機能を提供する。これは、ＮＦＳサーバ［５９］の、ファイル・サーバとしての機能と同等の機能を提供する。（ＮＦＳサーバは、ディレクトリ・サーバとしての機能も持っている。）提供するファイルのモデルは、ランダム・アクセス・ファイルである。１回の遠隔手続き呼出しごとに入出力を開始する点と長さを指定する。ファイルを開く操作は、存在せず、クライアントとの間に結合（connection）は作られない。ディスク・ブロックのキャッシングは、行われない。

カーネル・ファイル・サーバは、外部サーバにより提供可能な機能や、並列／分散応用プログラムから利用されない機能は、提供しない。たとえば、キャッシングやマッピングは、外部サーバにおいて提供することが可能である。さらに、並列／分散応用プログラムが、自分自身でキャッシングやマッピングを制御することで性能改善を目指す可能性がある。したがって、カーネル・ファイル・サーバは、キャッシングとマッピングの機能を提供しない。

２．１０．５　タイム・サーバ

タイム・サーバは、日付と時刻の情報を管理する。これは、応用プログラムが実行の統計を取る支援を行う。

２．１１　関連した研究

２．１１．１　カーネル化されたカーネル

オペレーティング・システムの構成法の１つに、カーネル化されたカーネル

（kernelized kernel）を使う方法がある［４３］。この方法では、カーネルは、プロセス間通信と物理入出力だけを提供する。オペレーティング・システムの機能の大部分は、カーネル外のサーバ・プロセスにより提供される。これに対して、カーネル外のサーバを用いることなく、１つの巨大なカーネルを利用する方法がある。このようなシステムは、単層システム（monolithic system）、そのカーネルは、単層カーネル（monolithic kernel）とよばれる。

カーネル化されたカーネルは、普通、内部にデバイスを操作する部分だけを含んでいる。この部分は、カーネル外に実現するサーバによってのみ利用され、利用者から直接利用されることはない。利用者は、カーネル外のサーバを通じて間接的にカーネル内のデバイスを操作する部分を利用することになる。

カーネル化の目的は、オペレーティング・システムのできるだけ多くの部分をカーネル外のサーバとして利用者プログラムと同じように開発することを可能にすることである。その結果、機能の拡張と変更が容易になるという利点が得られる。また、オペレーティング・システム内部の並列性が、複数のサーバの並列実行という形で、容易に抽出される。さらに、単層カーネルを用いる方法と比較して、モジュール化が明確になるという利点もある。カーネル化の問題点としては、プロセス間の保護の壁を交差するオーバヘッドが大きくなるため、効率が低下することである。

カーネル化されたカーネルは、しばしば分散カーネルとして実装される。分散カーネル（distributed kernel）とは、カーネルがネットワーク通信機能を提供し、かつ、ネットワーク上の全ての計算機においてカーネルのコピーが動作するようなものである。カーネル化された分散カーネルを利用して分散型オペレーティング・システムを構築している例としては、Ａｍｏｅｂａシステム［５４］、Ｖシステム［１２］、Ｃｈｏｒｕｓシステム［１］があげられる。

本システムのカーネルの特徴は、並列／分散応用プログラムには単層カーネル、逐次応用プログラムには、カーネル化されたカーネルになっている点にある（図１－３）。他のカーネルと異なり、本システムのカーネルは、内部に完全なサーバ（応用プログラムが直接呼び出すことができるサーバ）を含んでいる。これにより、並列／分散応用プログラムには、単層カーネルと同等の効率のよいサービスを提供することが可能となっている。

一方、逐次応用プログラムは、高度な機能を提供するカーネル外のサーバ（外部サーバ）を利用する。このことは、他のカーネル化されたカーネルを利用するシステムとほとんど同じである。５章では、オブジェクトの堆積というモデル化の手法を用いて、カーネル外の複数のサーバを統合して利用する方法について述べる。

２．１１．２　マイクロカーネル

マイクロカーネル（micro-kernel）という言葉は、２．１１．１項で述べたカーネル化されたカーネルを意味する言葉としても利用される。その他に、オペレーティング・システムのオペレーティング・システム、すなわち、メタ・オペレーティング・

システムを意味する言葉としても利用される［１］［２］［２６］。メタ・オペレーティング・システムとは、仮想計算機（ＶＭ、virtual machine）と同様に、オペレーティング・システムを実現すための環境を提供するものである。仮想計算機の場合、提供するモデルは、計算機のハードウェアである。そして、１台の物理的な計算機上に、複数の仮想計算機が動作し、それぞれの仮想計算機において独立したオペレーティング・システムが動作する。これに対して、マイクロカーネルの場合、提供するモデルが計算機のハードウェアではなく、ソフトウェア・アーキテクチャである。

メタ・オペレーティング・システムとしてのマイクロカーネルを利用した例としては、Ｍａｃｈシステムがあげられる［２］。Ｍａｃｈでは、マイクロカーネルにおいて、タスク、スレッド、ポート、メッセージ、メモリ・オブジェクトという５つの抽象を提供する。そして、マイクロカーネル上に、サーバとしてオペレーティング・システムが構築される。現在、ＵＮＩＸ、ＭＳ－ＤＯＳ、ＭａｃｈｉｎｔｏｓｈＯＳ等がマイクロカーネル上のサーバとして動作している。

本システムのカーネルは、メタ・オペレーティング・システムとしてのマイクロカーネルではなく、それ自身として自律したオペレーティング・システムである。したがって、Ｍａｃｈマイクロカーネル上で、本システムのカーネルを動作させることも考えられる。しかしながら、本システムのカーネルの機能は、Ｍａｃｈマイクロカーネルの機能よりも低レベルの部分があるため、それを行うことは困難である。たとえば、３章で述べる仮想プロセッサの機能を、Ｍａｃｈマイクロカーネルのスレッドやメモリ・オブジェクトの機能を利用して実現することは、困難である。

２．１２　まとめ

この章では、本ＲｅＳＣシステムのハードウェア構成、対象とする応用プログラム、全体構成、および、カーネルの機能について述べた。本システムの特徴は、カーネルの機能の設定にある。カーネルは、カーネル化されたカーネルの機能の他に、多重プログラミング・システムとしての資源割当ての方針を決定する部分や、並列／分散応用プログラムのための効率的なサービスを提供するサーバを含んでいる。また、システム・レベルのオブジェクト識別子は、位置依存のオブジェクトと位置独立のオブジェクトを同一のドメインで扱うことができるようになっている。位置依存のオブジェクト識別子と位置独立の遠隔手続き呼出しにより、システムは、分散型システムとも集中型システムとも見ることができるようになっている。

第３章　軽量プロセス

この章では、本ＲｅＳＣシステムにおける軽量プロセスの機能と実現方式について述べる。ＲｅＳＣでは、マイクロプロセスと仮想プロセッサという概念を用いて軽量プロセスを実現する。ここでは、まず軽量プロセスの概念について整理を行い、従来の代表的な軽量プロセスの実現方式であるカーネル制御方式とコルーチン方式について説明する。次に、本マイクロプロセス／仮想プロセッサ方式について述べ、本方式の特徴である、応用固有の軽量プロセス間の同期・通信プリミティブとスケジューラの開発の支援について述べる。続いて、仮想プロセッサを提供するカーネルの構成方式、裸の共有メモリ型マルチプロセッサ上への実現、既存のシステム上でのカーネル機能のエミュレートについて述べる。最後に、実現したシステムの性能について述べ、関連した研究との比較を行う。なお、この章では、軽量プロセスを利用して並列応用プログラムを記述するプログラマに対して提供するシステムの機能について述べる。したがって、この章で利用者とは、並列応用プログラムのプログラマを、利用者レベルとは、カーネルではなく応用プログラムのレベルを意味するものとする。

３．１　軽量プロセスの概念

多重プログラミングのオペレーティング・システムにおいて、プロセスという言葉は、資源割当て（resource allocation）、および、保護（protection）の単位を指すものとして用いられる。オペレーティング・システムは、プロセスごとに資源の管理表のエントリを設け、メモリ資源やＣＰＵ資源の公平な分配を行う。さらに、各プロセスに別々のアドレス空間を割り当てることで、他のプロセスの誤動作、あるいは、悪意をもつ利用者による攻撃からプロセスを保護する。一方、並列処理の分野では、プロセスという言葉は、並列処理の単位を指すものとして用いられる。この場合、プロセスは、ＣＰＵ割当ての単位であり、ＣＰＵ以外の資源割当てや保護の単位ではない。

軽量プロセス（lightweight process）は、多重プログラミング・システムにおいて従来のプロセスとは異なる、１応用プログラム内部の並列処理の単位としてのプロセスとして定義することができる。これに対して、従来のプロセスを重量プロセス（heavyweight process）とよぶ。たとえば、共有メモリ型マルチプロセッサにおいて並列処理を行う場合、軽量プロセスを並列処理の単位として用いることにより、ＣＰＵ処理とＣＰＵ処理の重ね合せを、従来のプロセスを用いる方法よりも効率よく実現することが可能となる。また、軽量プロセスは、ウインドウ・サーバやファイル・サーバなど、複数のクライアントを同時に扱う必要があるサーバを実現する際に有

効であることが知られている［９７］。この場合、クライアントごとに軽量プロセスを生成することにより、複数のサーバ・クライアント間の通信処理、入出力処理、および、ＣＰＵ処理の重ね合せが実現される。

軽量プロセスは、その名が示すように、従来の資源割当てと保護の単位としてのプロセスと比較して、効率よく実現されることが求められる。具体的には、次のような操作の効率がよいことが求められる。

（１）プロセスの生成、および、消滅
（２）プロセス間の同期、および、通信
（３）コンテキスト切替え

３．２　軽量プロセス機能の実現目標

軽量プロセス機能の実現における目標は、次のようにまとめられる。

（１）実行時の効率がよい軽量プロセスを実現する

プロセスの生成・消滅、プロセス間の同期・通信、コンテキスト切替えのオーバヘッドを小さくする。

（２）並列処理を実現する

ＣＰＵ処理とＣＰＵ処理の重ね合せ、および、複数の通信処理、複数の入出力処理とＣＰＵ処理の重ね合せを実現する。

（３）応用固有の同期・通信プリミティブの実現を支援する。

並列応用プログラムでは、様々な種類の軽量プロセス間の同期・通信プリミティブが使われる。その同期・通信プリミティブとしては、セマフォやモニタ等、一般的なものが使われることもあるが、各応用プログラムに適した固有のものが使われることが非常に多い。このため、並列応用プログラムの開発においては、応用固有の同期・通信プリミティブを作ることが重要な作業となる。したがって、システムとしてこれをいかに支援するかが重要となる。

（４）応用固有のスケジューリングを実現する環境を提供する。

共有メモリ型マルチプロセッサにおいて並列処理を行う場合、応用固有のスケジューリングを行うことにより、著しい性能向上が可能な応用プログラムが存在する。たとえば、実行に先立ってスケジューリングを行うことにより、同期処理の回数を減らしたり、あるいは、並列性が高くなるように優先順位を決定することができる。このような応用プログラムに対しては、応用固有のスケジューリングを実現する環境を

提供することが重要となる。

（５）軽量プロセス機能を実現するライブラリとカーネルの構成法を開発する。

オペレーティング・システムのカーネルは、カーネル上で動作する利用者プログラムと比較して、開発とデバッグが困難である。そのため、カーネルの構成法は、非常に重要である。２．１１．１項で述べたように、オペレーティング・システムの機能の大部分をカーネルの外で動作するサーバ・プロセスにより提供することで、カーネルを小さくすることも試みられている。しかしながら、軽量プロセスを実現する上で、カーネルの外に移動することができない部分が存在する。したがって、そのようなカーネルを構成する方法が重要な研究課題となっている。

３．３　代表的な軽量プロセス実現方式

軽量プロセスを実現する基本的な方法は、重量プロセスから保護の機能を省略することである。具体的には、アドレス空間を共有することである。こうすることにより、軽量プロセス間の通信を、軽量プロセスの共有変数を介して実現することが可能となり、効率が改善される。重量プロセスの場合、プロセス間通信によりプロセスが破壊されないようにオペレーティング・システムの保護機能が働く。

代表的な軽量プロセスの実現方式は、次の２種類に分類される。

（１）カーネル制御方式

この方式の基本的な考え方は、重量プロセスより保護の機能を省略することで、軽量プロセスを実現することである。軽量プロセス間の同期と通信は、同一のアドレス空間にあることを利用して、軽量プロセス間の共有変数を利用して実現される。軽量プロセスの生成・消滅、軽量プロセスのコンテキスト切替えは、カーネルにより制御される。この方式では、ＣＰＵ処理の並列処理が可能である。この方式は、Ｍａｃｈシステム［２］［６７］で用いられている。

（２）コルーチン方式

これは、軽量プロセスをコルーチンとして実現する方式である。この方式では、軽量プロセスの生成・消滅、軽量プロセス間の同期・通信など全ての操作は、利用者レベルにおいて実現される。この方式では、共有メモリ型マルチプロセッサにおいてＣＰＵ処理を並列に実行することができないが、ＣＰＵ処理と入出力処理の重ね合わせについては、非同期入出力とソフトウェア割込みを用いて実現することは、可能である［２７］。この方式は、ＳｕｎＯＳ［１０８］で用いられている。

以下では、マイクロプロセスと仮想プロセッサという概念を用いた軽量プロセスの

実現方式について述べる。本方式は、カーネル制御方式と比較して、効率的な軽量プロセスの実現が可能である。また、コルーチン方式と異なり、共有メモリ型マルチプロセッサにおいてＣＰＵ処理を並列に実行することができる。

３．４　マイクロプロセスと仮想プロセッサ

本システムでは、マイクロプロセスと仮想プロセッサという概念を用いて軽量プロセスを実現する。

マイクロプロセス（microprocess）は、並列処理の単位としてのプロセス、すなわち、軽量プロセスである。各マイクロプロセスは、重量プロセスの中にあり、独立のプログラム・カウンタとスタックを備えている。プロセスとマイクロプロセスの関係を図３－１に示す。マイクロプロセスの生成・消滅、マイクロプロセス間の同期・通信など、全ての制御は、利用者レベルにおいて行われる。オペレーティング・システムのカーネルは、一切介在しない。したがって、それらの制御が高速に実行される。

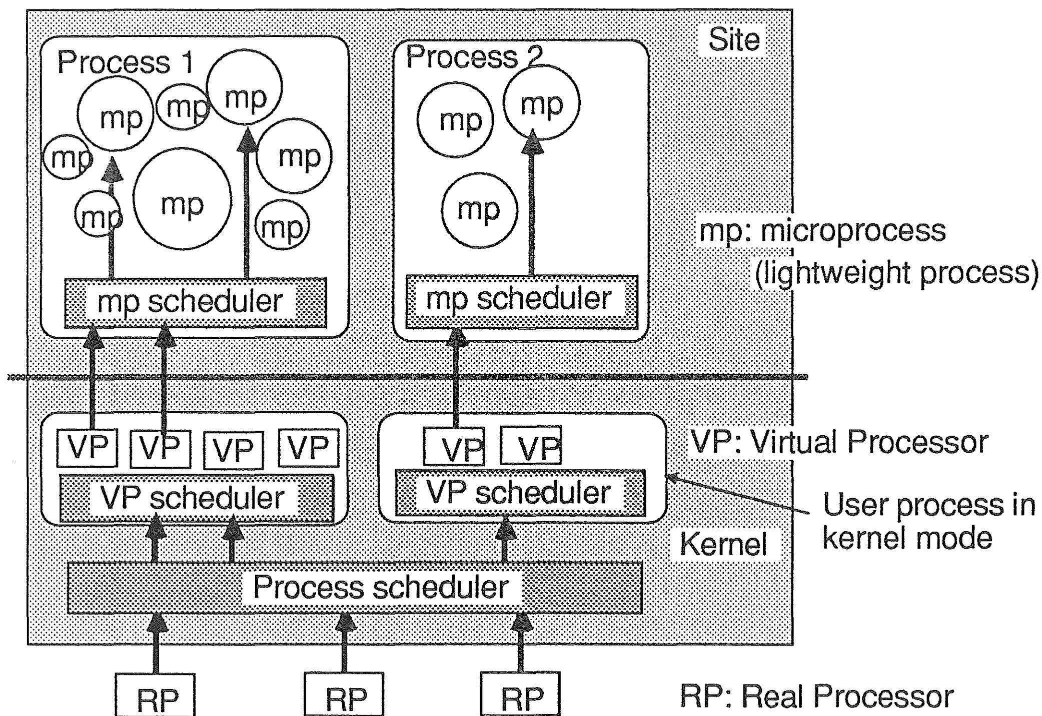


図３－１　プロセス、マイクロプロセス、仮想プロセッサの関係

Figure 3-1: Processes, microprocesses and virtual processsors.

仮想プロセッサ（virtual processor）は、カーネルが利用者プロセスに対して実プロセッサ（real processor、ハードウェアのプロセッサ）を割り当てるためのエントリである。図３－１に、プロセスと仮想プロセッサの関係を示す。各仮想プロセッ

サは、１つの利用者プロセスと対応しており、その利用者プロセスに割り当てられた資源、利用者識別子、アクセス権、プロセスの優先順位などを共有している。

仮想プロセッサは、利用者レベルの物理的な並列処理を行うための仕組みである。共有メモリ型マルチプロセッサでは、カーネルは、１つのプロセスに複数の実プロセッサを割り当てる。各実プロセッサは、そのプロセスの仮想プロセッサの中から、実行可能なものを選択し、実行する。その結果、制御が利用者空間へ移動する。そして、その複数の仮想プロセッサにより複数のマイクロプロセスが並列に実行される。単一プロセッサでは、ＣＰＵ処理を並列に実行することはできないが、複数の仮想プロセッサを生成することで、１つのＣＰＵ処理、複数の入出力処理、および、複数の通信処理を並列に実行することが可能となる。たとえば、利用者プロセスが、あるマイクロプロセスにおいて入出力処理を行うカーネル・コールを発行した場合、制御は、カーネル空間へ移動する。カーネルは、カーネル・コールを発行した仮想プロセッサを、入出力完了待ち状態に変え、同一プロセスの別の実行可能な仮想プロセッサを選択し、コンテキストを切り替える。この結果、制御は、再び利用者空間へ移動し、別の実行可能なマイクロプロセスが実行される。

プロセスが生成される時には、自動的に１個の仮想プロセッサが生成され、割り当てられる。これにより、逐次応用プログラムと並列応用プログラムの調和が図られる。逐次応用プログラムは、そのまま単一仮想プロセッサにより処理を進める。これにより、従来の逐次応用プログラムを仮想プロセッサに関して全く変更することなく共有メモリ型マルチプロセッサ上で利用可能となる。ゆえに、従来の単一プロセッサ・システムからの移行は、容易である。

一方、並列応用プログラムは、カーネル・コールを発行し、複数の仮想プロセッサを得る。並列応用プログラムからは、専用の共有メモリ型マルチプロセッサを使っているように見えるかもしれない。しかしながら、本システムは、多重プログラミング・システムであり、実プロセッサのスケジューリングは、カーネルにより行われる。すなわち、カーネル内のスケジューラが、各プロセスに割り当てる実プロセッサの数を調整する。このスケジューラをプロセス・スケジューラ（process scheduler）、あるいは、（１サイト内の）大域スケジューラ（global scheduler）とよぶ（図３－１）。そのサイトの負荷が低く、他に実プロセッサを必要としているプロセスが存在しない場合、全ての実プロセッサが１つのプロセスに割り当てられる。その結果、そのプロセスのプログラムは、資源の専有が可能な単一プログラミング・システムにおける実行と同等の速度で実行される。しかしながら、システムの負荷が高く、他に実プロセッサを要求しているプロセスが存在するならば、１つのプロセスに対して割り当てられる実プロセッサの数が減らされる。

プロセス、マイクロプロセス、仮想プロセッサについてまとめると次のようになる。

（１）プロセス

プロセスは、資源割当ての単位である。資源としては、メモリ、アドレス空

間、ファイルなどがある。実プロセッサ資源の割当ての単位もプロセスである。カーネル内のプロセス・スケジューラは、各プロセスに割り当てる実プロセッサの個数を決定する。

（２）マイクロプロセス

マイクロプロセスは、１応用プログラム内の並列処理の単位である。マイクロプロセスは、独立したスタックとプログラム・カウンタを持つ。マイクロプロセスの生成・消去、コンテキスト切替え、マイクロプロセス間の通信など全ての制御は、応用プログラムのレベルで行われる。この結果、並列処理を行う際にオペレーティング・システムが介在する場面を極力減らすことが可能となり、効率が改善される。

（３）仮想プロセッサ

仮想プロセッサは、カーネルがプロセスに実プロセッサを割り当てるためのエントリである。共有メモリ型マルチプロセッサでは、物理的並列処理単位となる。逐次プロセッサにおいても、複数の仮想プロセッサを生成することにより、１つのＣＰＵ処理、複数の入出力処理、および、複数のプロセス間通信処理を同時に行うことが可能となる。

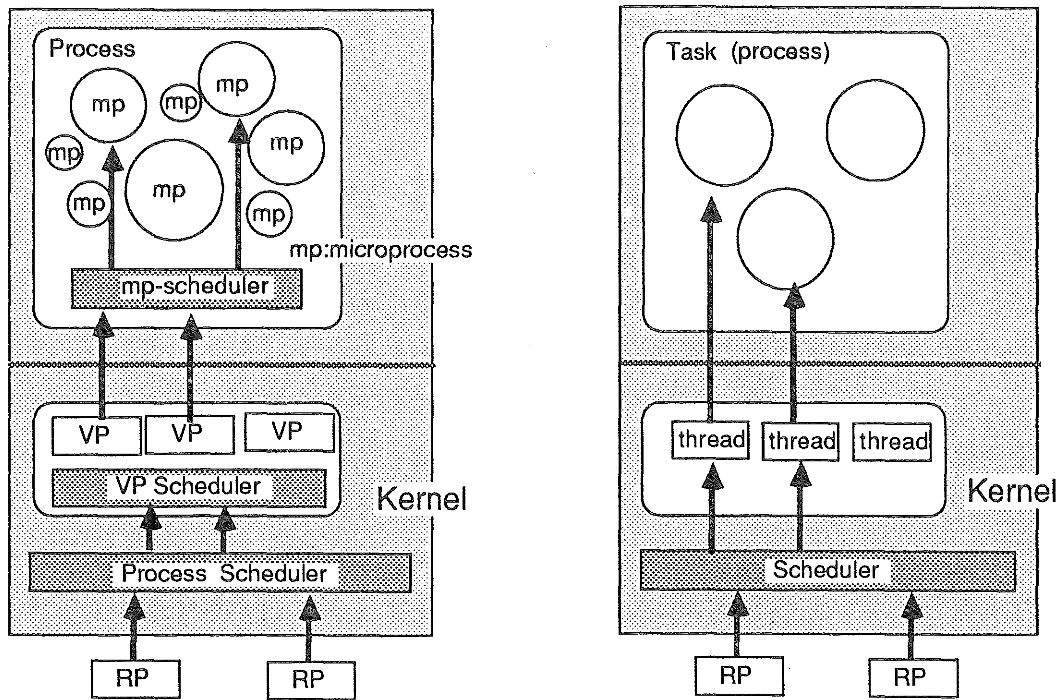
３．４．１　他の実現方式との比較

本方式、カーネル制御方式、および、コルーチン方式よる軽量プロセスの実現の様子を図３－２に示す。

■カーネル制御方式との比較

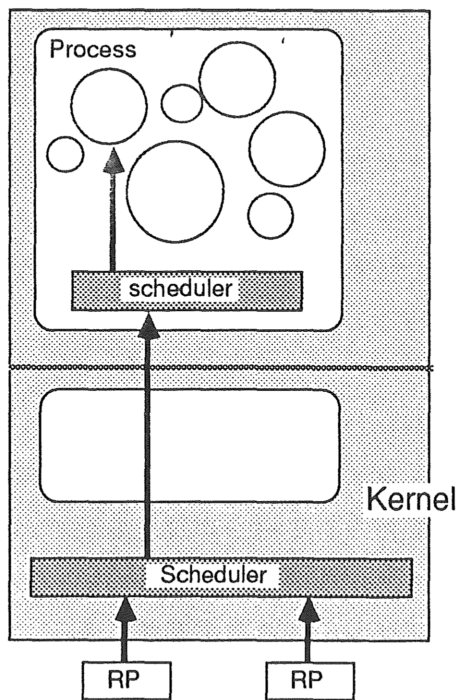
図３－２（ａ）に、本方式、図３－２（ｂ）に、Ｍａｃｈシステムにおけるカーネル制御方式による軽量プロセスを示す。Ｍａｃｈでは、軽量プロセス（スレッド，thread）をカーネルが直接制御している。これに対して、本方式では、軽量プロセス（マイクロプロセス，mp）を応用プログラム・レベルのスケジューラが制御している。

本方式における仮想プロセッサは、Ｍａｃｈシステムにおけるスレッドと対応することがわかる。しかしながら、本方式の仮想プロセッサの場合、利用者レベルで軽量プロセスが実現されていることを前提として機能が設計されている点が、大きく異なる。たとえば、本システムの仮想プロセッサは、１度生成されるとプロセス全体が終了するまで存在し続ける。また、本方式の仮想プロセッサには、仮想プロセッサごとの固有領域が存在し、利用者レベルの軽量プロセス（マイクロプロセス）を実現するライブラリにおいて利用される。仮想プロセッサの固有領域については、３．６．３項において詳しく述べる。



(a) マイクロプロセス／
仮想プロセッサ方式

(b) カーネル制御方式 (Mach)



(c) コルーチン方式

VP:Virtual Processor

RP:Real Processor

◯ : Lightweight process
軽量プロセス

図３－２　軽量プロセス実現方式の比較

Figure 3-3: Comparision of lightweight processs implementation methods.

■コルーチン方式との比較

図３－２（ａ）に示した本方式と図３－２（ｃ）に示したコルーチン方式を比較すると、本方式におけるマイクロプロセスとコルーチンが非常に類似していることがわかる。マイクロプロセスとコルーチンの相違点は、コルーチンの場合、１度に１ルーチンしか実行されないが、マイクロプロセスの場合、複数の仮想プロセッサにより並列に実行されることである。そのため、共有メモリ型マルチプロセッサにおいて並列処理が可能となる。単一プロセッサにおいても、１つのＣＰＵ処理、複数の入出力処理、および、複数のプロセス間通信が並列に実行される。

マイクロプロセスの場合、軽量プロセス間の共有変数を参照する時にロックなどにより相互排除を行う必要がある。コルーチンの場合、その必要はない。また、コルーチンでは、あるルーチンでカーネル・コールを発行するとプロセス全体の実行が停止してしまう。マイクロプロセスでは、カーネル・コールを発行したマイクロプロセスとそれを実行していた仮想プロセッサは、一時停止するが、他の仮想プロセッサにより、プロセスの実行は継続される。

３．４．２　プロセッサのスケジューリング

本方式では、マイクロプロセス、プロセス、仮想プロセッサの３つのレベルにおいてプロセッサのスケジューリングを行う（図３－１）。それぞれ，次の様な機能をもつ。

（１）マイクロプロセス・スケジューラ

利用者プロセス中にあり、各応用プログラム固有の情報を利用して、応用固有のスケジューリングを実現する。コンテキスト切替えは、マイクロプロセスが同期プリミティブを発行した時に限り行われる。（横取りを行わない（non-preemptive）スケジューリングを行う。）

（２）プロセス・スケジューラ（process scheduler）

カーネル中にあり、多重プログラミング・システムにおける複数の利用者プロセス間の公平なプロセッサ資源の分配を行う。これを実現するために、量子時間（time quantum）ごとに、プロセッサの横取り（preemption）を行い、各利用者プロセスに対して割り当てる実プロセッサの数を調整する。

（３）仮想プロセッサ・スケジューラ

カーネル中にあり、同一プロセス内の別の仮想プロセッサにコンテキストを切り替える。このコンテキスト切替えは、ある仮想プロセッサが入出力待ち、または、通信待ちのために実行を継続できなくなった時に限り行われる。（横取りを行わないスケジューリングを行う。）

このように機能を分割することにより、それぞれのスケジューラが単純化される。一方、スケジューラが多段になり、コンテキスト切替えのオーバヘッドが大きくなることが予想される。しかしながら、スケジューラを実行する頻度がレベル間で異なり、圧倒的にマイクロプロセス・スケジューラの頻度が大きいので、仮想プロセッサ・スケジューラとプロセス・スケジューラは、性能に大きな影響を与えない。

本方式では、仮想プロセッサ・スケジューラにおいてプロセッサの横取りを行わない。この理由は、次の２点にある。

（１）マイクロプロセス・スケジューラを活用する。

仮想プロセッサ・スケジューラにおいて、プロセッサの横取りを行うならば、１つの並列応用プログラムの実行において、２つのスケジューラが同時に動作することになる。その場合には、利用者空間内のマイクロプロセス・スケジューラが機能しなくなる。

（２）無駄なコンテキスト切替えを避ける。

本来、プロセッサの横取りは、複数のプロセス間において、プロセッサ資源の公平な利用を実現するための機能である。本方式では、資源割当ての単位は、プロセスであるので、同一プロセス内において資源の公平な利用を実現する必要がない。

３．５　マイクロプロセスを実現するライブラリの機能と構造

本方式では、マイクロプロセスは、利用者空間内に実現されるので、カーネル・コールではなくライブラリにより支援される。このライブラリは、図３－３に示すように、層構造になっている。このため、各並列応用プログラムの要求に応じた部分的な利用が容易になっている。

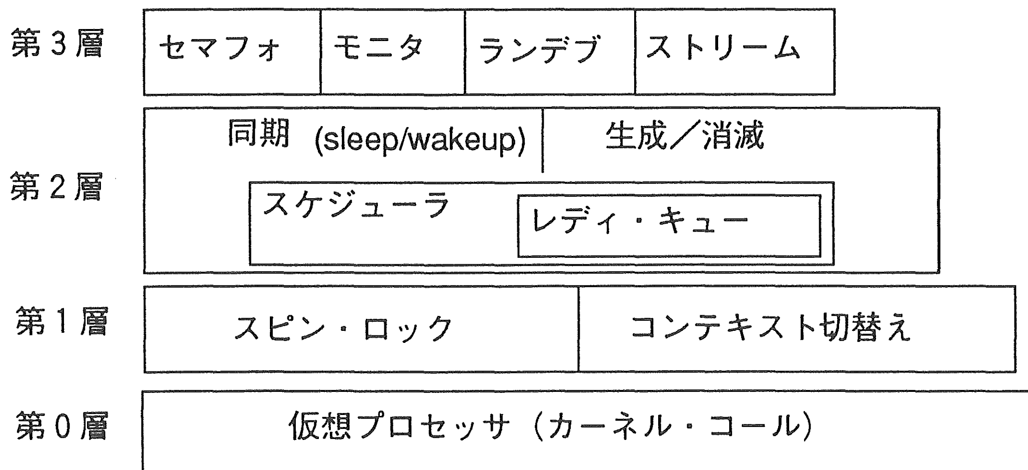


図３－３　マイクロプロセス・ライブラリの構造

Figure 3-3: The structure of the microprocess library.

第０層では、カーネルによって実現される仮想プロセッサに関連したカーネル・コールが提供される。この層を利用する並列応用プログラムは、仮想プロセッサだけを利用して、完全に独自の方式により軽量プロセスを実現することになる。仮想プロセッサに関連したカーネル・コールについては、３．６．１項で述べる。

第１層では、各プロセッサ・アーキテクチャやバス・アーキテクチャに独立なコンテキスト切替えとスピン・ロックを実現する機能が提供される。この層では、それらのアーキテクチャに独立したプログラムを記述することが可能となる。スピンロックは、test-and-set 命令や、swap命令のような、原始的な（atomic）命令を用いて実現される。コンテキスト切替えは、主にレジスタの退避と回復により実現される。

第２層では、基本的なマイクロプロセスの生成・消滅機能と、sleep/wakeup モデルに基づく簡単な同期機能が提供される。このモデルの詳細については、３．５．１項において述べる。また、第２層は、スケジューラとレディ・キューを含んおり、これらは、応用固有のスケジューリングを行うために利用される。応用固有のスケジューリングについては、３．９節において述べる。

第３層では、セマフォ、モニタ、ランデブ、バイト・ストリーム（ＵＮＩＸのパイプと同等の機能）等の同期・通信プリミティブが提供される。これらのプリミティブのソース・コードは、各並列応用プログラム固有の同期・通信プリミティブを構築するプログラマにプロトタイプを与える。

並列応用プログラムは、通常、第２層の機能を用いて構築される。これは、セマフォやモニタを利用するよりも、低いレベルの機能を利用する方が各並列応用プログラム固有の同期・通信プリミティブを構築しやすいからである。

マイクロプロセス・ライブラリの主要な手続きの仕様を、付録Ｃに示す。

３．５．１　sleep/wakeup モデル

sleep/wakeup モデルでは、主に次の２つの手続きを用いて同期・通信プリミティブを構築する。

（１）mp_sleep():
実行中状態（running state）の自分自身（マイクロプロセス）を、待ち状態（blocked state）に移す。

（２）mp_wakeup():
待ち状態にある（他の）マイクロプロセスを、実行可能状態（runnable state、ready state）に移す。既に実行可能、あるいは、実行中の場合には、何もしない。

この他に、自分自身を実行可能状態にしたまま、他の実行可能なマイクロプロセスに制御を移すための手続き、mp_yield()がある。各プロセッサは、実行可能状態にあるマイクロプロセスを選び、それを実行中状態に変え、そのマイクロプロセスへ制御を

移す。このモデルの特徴は、他の実行中のマイクロプロセスを待ち状態、あるいは、実行可能状態に移すプリミティブが存在しない点にある。

ここでは、典型的な同期プリミティブとして、２進セマフォを取り上げ、sleep/wakeup モデルについて述べる。図３－４（ａ）に、このモデルに基づく２進セマフォの実現例を示す。この図において、lock()、unlock() は、第１層の手続きであり、それぞれ、スピンロックによるロック／アンロックを行うものである。このプログラムでは、これらの手続きを用いて、際どい部分（critical section）の相互排除を行っている。

Ｐ命令の実現を図３－４（ａ）第１１行に示す。第１６行において、他のマイクロプロセスがそのセマフォを保持している（sem->inuse が TRUE）の時には、自分自身（current）を、待ち行列に保存し、セマフォの資源を開放する。そして、第１９行において、mp_sleep() を呼び出し、現在実行中のマイクロプロセスを待ち状態にする。セマフォが利用されていない時（sem->inuse が FALSE）には、自分自身が利用していることをマーク（第２２行）している。

Ｖ命令の実現を、図３－４（ａ）第２８行に示す。Ｐ命令の中で mp_sleep() により実行を中断したマイクロプロセスは、第３７行の mp_wakeup() により、実行可能状態に戻される。

第１５行において、現在実行中のマイクロプロセス（current）に掛けられたロックは、mp_sleep() の内部で解除される。この解除を、この部分で先に行うことはできない。それは、sem->lock が開放された瞬間に、第３６において、mp_wakeup() が呼び出される可能性があるからである。実行可能中のマイクロプロセスに対して mp_wakeup() が適用された場合、何も行われないため、そのシグナルが落ちる。

３．５．２　他の実現方式と比較

この項では、２進セマフォ同期プリミティブの実現を通じて、カーネル制御方式、および、コルーチン方式との比較を行う。

■カーネル制御方式との比較

図３－４（ｂ）に、カーネル制御方式の軽量プロセスにおけるセマフォの実現を示す。カーネル制御方式の例として、Ｍａｃｈシステムのスレッドを取り上た。ここで、３．５．１項と同様に、スピン・ロックによりロック機構を実現するライブラリ関数 lock()、unlock()、および、ＦＩＦＯのキューを実現するライブラリ関数 enqueue()、dequeue() を利用する。

第２８行、および、第４７行に現れる mach_msg() がＭａｃｈシステムのカーネル・コールである。この実現では、メッセージ・パッシングを用いてスレッドの一時停止／実行再開を実現している。すなわち、スレッドの一時停止を行う時には、メッセージを受け取るカーネル・コール（mach_msg() の第２引数が MACH_RCV_MSG (Mach receive message)）を発行している。逆に、一時停止したスレッドの実行再開を要求

```
 1: extern private struct mpcb *current ;
 2:
 3: struct sem
 4: {
 5:         int             inuse ;
 6:         lock_t          lock ;
 7:         struct mpcb    *queue ;
 8: };
 9: typedef struct sem sem_t ;
10:
11: P( sem )
12:     sem_t *sem ;
13: {
14: loop:   lock( &sem->lock );
15:         lock( &current->lock );
16:         if( sem->inuse )
17:         {
18:             enqueue( current,&sem->queue );
19:             unlock( &sem->lock );
20:             mp_sleep( 0, 0 );
21:             goto loop;
22:         }
23:         sem->inuse = TRUE ;
24:         unlock( &sem->lock );
25:         unlock( &current->lock );
26: }
27:
28: V( sem )
29:     sem_t *sem ;
30: {
31:     struct mpcb *next ;
32:     struct mpcb *dequeue();
33:
34:         lock( &sem->lock );
35:         sem->inuse = FASLE ;
36:         next = dequeue( &sem->queue );
37:         if( next != MP_NULL )
38:             mp_wakeup( next );
39:         unlock( &sem->lock );
40: }
```

図３－４（ａ） マイクロプロセス・ライブラリによる２進セマフォの実現

図３－４　２進セマフォの実現

```
1: struct proc
2: {
3:         struct proc      *link ;
4:         mach_port_t       port ;
5:         mach_msg_header_t msg ;
6: };
7: typedef struct proc *proc_t ;
8: extern  proc_t proc_self();
9:
10: struct sem
11: {
12:         int               inuse ;
13:         lock_t            lock ;
14:         proc_t           *queue ;
15: };
16: typedef struct sem sem_t ;
17:
18: P( sem )
19:     sem_t *sem ;
20: {
21: loop:   lock( &sem->lock );
22:         if( sem->inuse )
23:         {
24:             mach_msg_header_t msg;
25:             proc_t current = proc_self();
26:             enqueue( current,&sem->queue );
27:             unlock( &sem->lock );
28:             mach_msg( &msg, MACH_RCV_MSG, 0, sizeof(msg),
29:                  current->port, MACH_MSG_TIMEOUT_NONE, MACH_PORT_NULL );
30:             goto loop;
31:         }
32:         sem->inuse = TRUE ;
33:         unlock( &sem->lock );
34: }
35:
36: V( sem )
37:     sem_t *sem ;
38: {
39:     proc_t next ;
40:     extern proc_t dequeue();
41:         lock( &sem->lock );
42:         sem->inuse = FALSE ;
43:         next = dequeue( &sem->queue );
44:         unlock( &sem->lock );
45:         if( next != MACH_PORT_NULL )
46:         {
47:             mach_msg( &next->msg, MACH_SEND_MSG, sizeof(next->msg), 0,
48:                  MACH_PORT_NULL, MACH_MSG_TIMEOUT_NONE, MACH_PORT_NULL);
49:         }
50: }
```

図３－４（ｂ）　カーネル制御方式における２進セマフォの実現

図３－４　２進セマフォの実現

```
 1: extern struct coroutine *current ;
 2:
 3: struct sem
 4: {
 5:         int             inuse ;
 6:         struct coroutine *queue ;
 7: };
 8: typedef struct sem sem_t ;
 9:
10:
11: P( sem )
12:     sem_t *sem ;
13: {
14: loop:   if( sem->inuse )
15:         {
16:             enqueue( current,&sem->queue );
17:             c_sleep();
18:             goto loop;
19:         }
20:         sem->inuse = FALSE ;
21: }
22:
23: V( sem )
24:     sem_t *sem ;
25: {
26:     struct coroutine *next ;
27:     struct coroutine *dequeue();
28:         sem->inuse = FASLE ;
29:         next = dequeue( &sem->queue );
30:         if( next != C_NULL )
31:             c_wakeup( next );
32: }
```

図３－４（ｃ）　コルーチン方式による２進セマフォの実現

図３－４　２進セマフォの実現

する場合には、メッセージを送信するカーネル・コール（mach_msg() の第２引数が MACH_SEND_MSG (Mach send message)）を発行している。この方法は、Ｍａｃｈ３．０のＣ－Ｔｈｒｅａｄｓライブラリにおいて用いられている。

Ｍａｃｈのスレッドを用いた実現（図３－４（ｂ））と、本システムにおける実現（図３－４（ａ））を比較すると、Ｍａｃｈの場合に、カーネル・コールになっている部分が、本システムでは、（カーネル・コールを利用しない）ライブラリにより実現されている。また、コンテキスト切り替えの必要がない場合においては、カーネル制御方式の場合においても、カーネル・コールを発行しないことがわかる。

■コルーチン方式との比較

図３－４（ｃ）に、コルーチン方式の軽量プロセスにおけるバイナリ・セマフォの実現を示す。 c_sleep()、c_wakeup() は、３．５．１項において述べた mp_sleep(),

mp_wakeup() と同等なライブラリ関数である。

コルーチン方式におけるセマフォの実現は、本方式における実現と非常に類似している。これは、本方式における実現からロックの制御を削除したものに相当する。

３．５．３　実行の軌跡の記録と解析

本マイクロプロセス・ライブラリには、マイクロプロセスの生成・消滅や、コンテキスト切替えが起きたことをイベントとしてその時刻を記録する機能がある。この機能は、並列応用プログラムを開発する上で、並列性の抽出の様子を確認するために利用される。

記録されたイベントを用いて、各マイクロプロセス、および、仮想プロセッサごとのＣＰＵ利用時間、入出力時間、コンテキスト切り替えの回数といったの統計情報を得ることができる。その例を、図３－５に示す。この例では、マイクロプロセスが９個（０番～８番）、仮想プロセッサが３個生成されている。マイクロプロセス関連の情報において、マイクロプロセス０番は、最初のマイクロプロセスを生成するスケジューラである。１番から８番は、普通のマイクロプロセスである。この部分の列は、それぞれ、次のような意味を持つ。

（１）マイクロプロセスの番号

（２）そのマイクロプロセスが利用したＣＰＵ時間の合計とその割合

（３）ＣＰＵを利用した回数（ＣＰＵバーストの回数）

（４）そのマイクロプロセスが行った入出力の時間の合計とその割合

（５）入出力回数

（６）ＣＰＵ時間と入出力時間の合計とその割合

（７）コンテキスト切替えの回数

この例では、マイクロプロセス１番が全体の１９％の実行時間を占めていること、特に入出力が重たくなっていること等が読み取れる。

仮想プロセッサ関連の部分では、次のような情報が得られる。

（１）仮想プロセッサ番号

（２）ＣＰＵ稼働時間（この仮想プロセッサがマイクロプロセスを実行していた時間）とその割合。

（３）入出力時間（この仮想プロセッサにより入出力行っていた時間）とその割合

（４）その仮想プロセッサがアイドル状態にあった時間とその割合

（５）この軌跡をファイルに保存するために要した時間と割合

（６）その他の時間（時刻の入手、コンテキスト切替えなど）と割合

（７）合計とその割合

最終行は、全仮想プロセッサの平均である。図３－５の例では、プロセッサのアイドル時間が１５％であったことがわかる。

```
## microprocess information
No  cputime  % / #ex   iotime  % / #io    total  % / #++
 0   0.000   0 /   1   0.000   0 /   0   0.000   0 /   1
 1   0.140   8 /  67   0.370  36 /  62   0.510  19 / 129
 2   0.370  22 /  20   0.000   0 /   0   0.370  14 /  20
 3   0.100   6 /  66   0.180  18 /  65   0.280  10 / 131
 4   0.360  21 /  18   0.000   0 /   0   0.360  13 /  18
 5   0.110   6 /  66   0.180  18 /  65   0.290  11 / 131
 6   0.350  21 /  13   0.000   0 /   0   0.350  13 /  13
 7   0.130   8 /  66   0.160  16 /  65   0.290  11 / 131
 8   0.140   8 /  72   0.130  13 /  65   0.270  10 / 137
##   1.700 100 / 389   1.020 100 / 322   2.720 100 / 711

## virtual processor information
No     cpu   %|    io   %|   idle   %|  trace   %| kernel   %|  total   %|
 0   0.600  54| 0.240  21| 0.230  21| 0.000   0|  0.050   4| 1.120 100|
 1   0.560  50| 0.370  33| 0.160  14| 0.000   0|  0.030   3| 1.120 100|
 2   0.540  48| 0.410  37| 0.130  12| 0.000   0|  0.040   4| 1.120 100|
##   1.700  51| 1.020  30| 0.520  15| 0.000   0|  0.120   4| 3.360 100|
```

図３－５　実行の軌跡の記録より得られる統計情報

Figure 3-5: Execution statistics from microprocess traces.

記録されたイベントを、実行の軌跡として図３－６に示すように視覚化することも可能である。１番から８番において、黒い部分がマイクロプロセスの実行を表す。i0, i1, i2は、それぞれ仮想プロセッサの０番、１番、２番が、他に実行するマイクロプロセスが存在しないため、アイドル状態になったことを表す。並列応用プログラムの開発者は、このような情報を利用して、並列性の抽出の様子を観察することが可能である。

マイクロプロセス・ライブラリでは、具体的には、次のようなイベントを記録している。

（１）生成（created）

（２）終了（exit）

（３）中断（sleep）

（４）実行可能化（waked）

（５）プロセッサの譲渡（yield）

（６）実行開始（scheduled）

（７）入出力開始（io start）

（８）入出力終了（io stop）

（９）プロセス全体の終了（process exit）

（１０）仮想プロセッサの一時停止（vp suspend）

（１１）仮想プロセッサの実行再開（vp resume）

（１２）軌跡の記録のための入出力開始（trace file io start）

（１３）軌跡の記録のための入出力終了（trace file io stop）

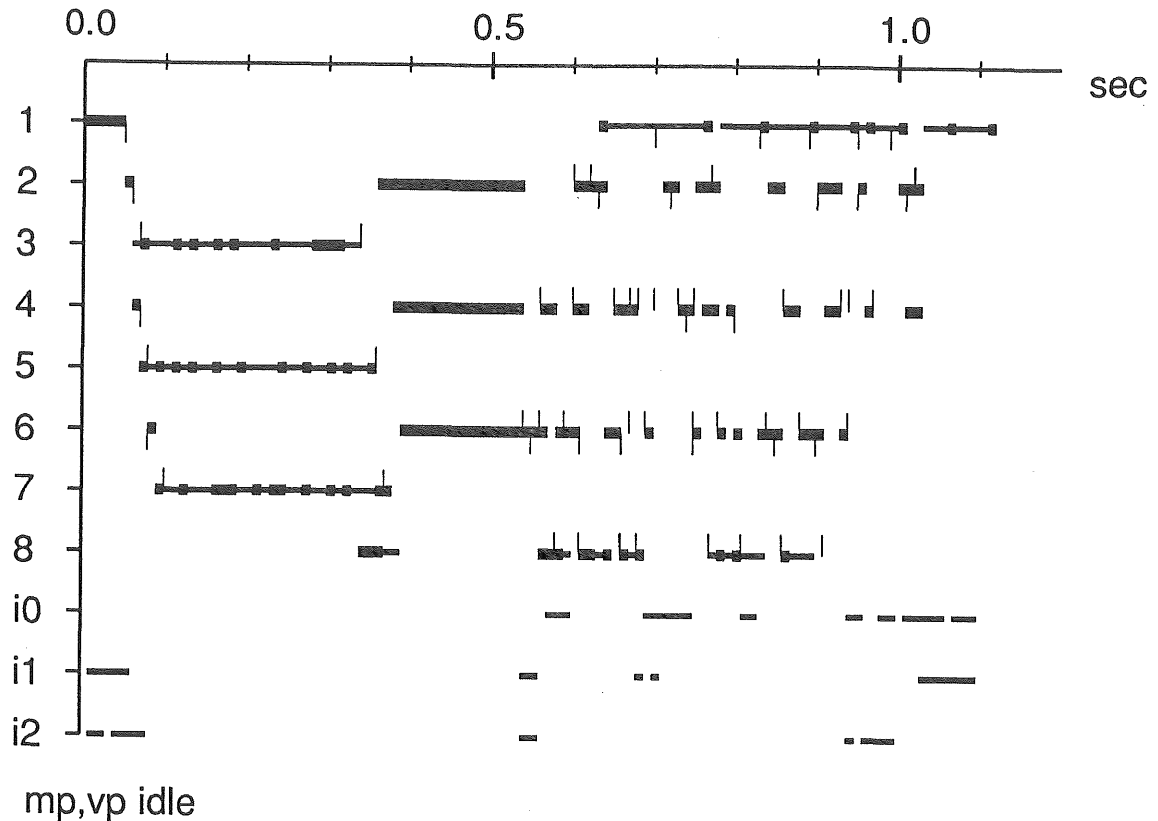


図３－６　実行の軌跡の記録を視覚化したもの

Figure 3-6: Visualized event traces of microprocess excution.

最初の４つは、それぞれ第２層の mp_create()、 mp_exit()、mp_sleep()、mp_wakeup()、 mp_yield() に関係している。ただし、mp_wakeup() において、既に実行可能になっているマイクロプロセスを再び実行可能にすることを試みた場合は、記録されない。（６）は、スケジューラにおいて、実行可能のマイクロプロセスに対してプロセッサが割り当てられたことを示すイベントである。（７）、（８）は、マイクロプロセスのＣＰＵ処理と入出力処理を区別して記録する必要がある場合に用いられる。（９）、（１０）は、仮想プロセッサの動作、（１１）は、プロセス全体の動作の示すイベントである。（１２）、（１３）は、軌跡を記録するためのファイル出力を表すイベントである。このようなイベントは、仮想プロセッサごとに記録される。

探り針効果

実行の軌跡を記録する場合、注意すべきことは、**探り針効果**（probe effect）である。探り針効果とは、イベントを記録するために、その記録という行為が応用プログラムの実行に影響を与えてしまうことである。このマイクロプロセス・ライブラリでは、特に、イベントが発生した時刻を得るためのカーネル・コールのオーバヘッドが大きい。これを改善するために、カーネル・コールによらない時刻の取得方法が求められる。

３．６　仮想プロセッサを実現するカーネルの機能

仮想プロセッサは、カーネルにおいて実現される。利用者は、カーネル・コールを通じて、仮想プロセッサの機能を利用する。

３．６．１　カーネル・コール

仮想プロセッサに関連したカーネル・コールには、次の４つがある。

（１）vpid=vp_allocate(n)

これは、新たにｎ－１個の仮想プロセッサを生成することを要求するカーネル・コールである。このカーネル・コールの発行後、合計ｎ個の仮想プロセッサによる並列処理が開始される。返り値は、仮想プロセッサの識別子である。

（２）vp_switch(vpid)

これは、同一プロセス内の他の仮想プロセッサへ制御を切り替えるためのカーネル・コールである。スピンロックを保持したまま実プロセッサを奪われた仮想プロセッサに制御を移すために用いられる。vpidは、次に実行すべき仮想プロセッサのヒントである。

（３）vp_sleep(t)

これは、その仮想プロセッサに対応している、一時的に不用になった実プロセッサをカーネルに返すためのカーネル・コールである。ｔは、実プロセッサを放棄する時間のヒントである。

（４）vp_wakeup(n)

これは、vp_switch()、 あるいは、vp_sleep() で実行を停止している仮想プロセッサの実行再開を要求するカーネル・コールである。ただし、このカーネル・コールを用いなかったとしても、vp_sleep() を発行した仮想プロセッサは、指定された時間が経過した後には、実行を再開する。

仮想プロセッサを削除するカーネル・コールは、存在しない。それは、仮想プロセッサの数は、実プロセッサの数と同様に、プロセスの実行の途中で変化しないからである。

３．６．２　カーネル・コールの利用例

図３－７、図３－８、図３－９に、Ｃ言語による上記のカーネル・コールの利用例を示す。（図３－７の１行目、２行目のキーワード private、 図３－８の第１行目のキーワード shared については、次項（３．６．３項）において説明する。）

```
 1: private int vpid ;        /* Virtual Processor ID */
 2: private struct mpcb *mp_current ; /* Microprocess Control Block */
 3:
 4: main()
 5: {
 6:         init_single();   /* initialize in a sigle VP. */
 7:         vpid = vp_allocate( 2 ); /* create a new VP. */
 8:         init_multi();    /* initialize in multiple VPs. */
 9:         mp_scheduler();  /* jump into mp_scheduler(). */
10:         /* Not reached. */
11: }
```

図３－７　vp_allocate() カーネル・コールによる仮想プロセッサの生成
Figure 3-7: Creatiting a virtual processor by invoking the vp_allocate() kernel call.

図３－７第４行の main() は、このプログラムの実行が開始される場所を示している。プロセスには、起動時に１個の仮想プロセッサが自動的に割り当てられる。この仮想プロセッサにより、main() が実行される。まず、単一仮想プロセッサにより、さまざまな初期化を行う（第６行）。図３－７第７行では、 vp_allocate() カーネル・コールにより、新たに仮想プロセッサが１個作られ、これ以後、合計２個の仮想プロセッサが活動することになる。その後、複数の仮想プロセッサによる初期化を行い、制御をマイクロプロセス・スケジューラ（mp_scheduler()）に移す。その後、制御が再び戻って来ることはない。

図３－８は、簡単なマイクロプロセス・スケジューラの例である。この例では、第８行において、実行可能なマイクロプロセスを１個取り出し、それを mp_run() により実行している。この時、実行可能なマイクロプロセスが存在しない場合、next_mp が 0 になる。第１１行において、vp_sleep() カーネル・コールにより、現在この仮想プロセッサに対応している実プロセッサを開放している。VP_SLEEP_TIME は、実行を中断する時間のヒントである。

図３－９は、簡単なスピンロック（spin lock）の実現の例である。この例では、SPIN_THRESHOLD （第５行）に定義された回数だけ test_and_set() （第８行）を繰り返し実行する。ここで、test_and_set() は、test-and-set 命令のような、原始にテストとセットを行う機械語命令で実現されるものとする。SPIN_THRESHOLD に定義された数だけ回繰り返してもロックが保持できない場合、ループを抜けて、第１３行目で vp_switch() カーネル・コールを発行している。カーネルは、プロセッサの横取りを行った仮想プロセッサが存在しないかを調べ、それが見つかった時には、その仮想プロセッサへ制御を移す。

```
 1: shared lock_t mp_ready_lock ;
 2:
 3: mp_scheduler()
 4: {
 5:     struct mpcb *next_mp ;
 6:
 7: loop:   lock( &mp_ready_lock );
 8:             next_mp = mp_ready_dequeue();
 9:         unlock( &mp_ready_lock );
10:
11:         if( next_mp == 0 )
12:         {
13:             vp_sleep( VP_SLEEP_TIME );
14:             goto loop;
15:         }
16:         mp_run( next_mp );
17:         /* Not reached. */
18: }
```

図３－８ vp_sleep() カーネル・コールによる実プロセッサの開放

Figure 3-8: Releasing a real processor by invoking the vp_sleep() kernel call.

```
 1: lock( lock_t *v )
 2: {
 3:     int count ;
 4: loop:
 5:         count = SPIN_THRESHOLD ;
 6:         do
 7:         {
 8:             if( test_and_set( v ) == 0 )
 9:                 return;
10:         }
11:         while( --count > 0 );
12:
13:         vp_switch( VP_NONE );
14:         goto loop;
15: }
16:
17: unlock( lock_t *v )
18: {
19:         *v = 0 ;
20: }
```

図３－９ vp_switch() カーネル・コールによる他の仮想プロセッサへの制御の移動

Figure 3-9: Forcing context switch to another virtual processor by invoking the vp_switch() kernel call.

３．６．３　仮想プロセッサの固有領域

本方式では、仮想プロセッサごとに固有のメモリ領域がある。すなわち、各仮想プロセッサには、アドレス空間の一部に、仮想プロセッサごとに異なる物理メモリが参照される領域が存在する。これを仮想プロセッサの固有領域（private area）とよぶ。これに対して、固有領域以外の領域を共有領域（shared area）とよぶことにする。固有領域は、マイクロプロセスを実現するライブラリにおいて、仮想プロセッサごとの情報を格納するために利用される。この領域は、実プロセッサのレジスタが拡張されたものと考えることができる。

図３－１０に、１つのプロセスに属する２つの仮想プロセッサの論理アドレス空間と物理メモリの対応を示す。これは、図３－７に示したプログラムの実行の結果、構築される対応である。図３－７第７行の vp_allocate() カーネル・コールにより、新たに仮想プロセッサが１個作られ、合計２つの仮想プロセッサが活動している。

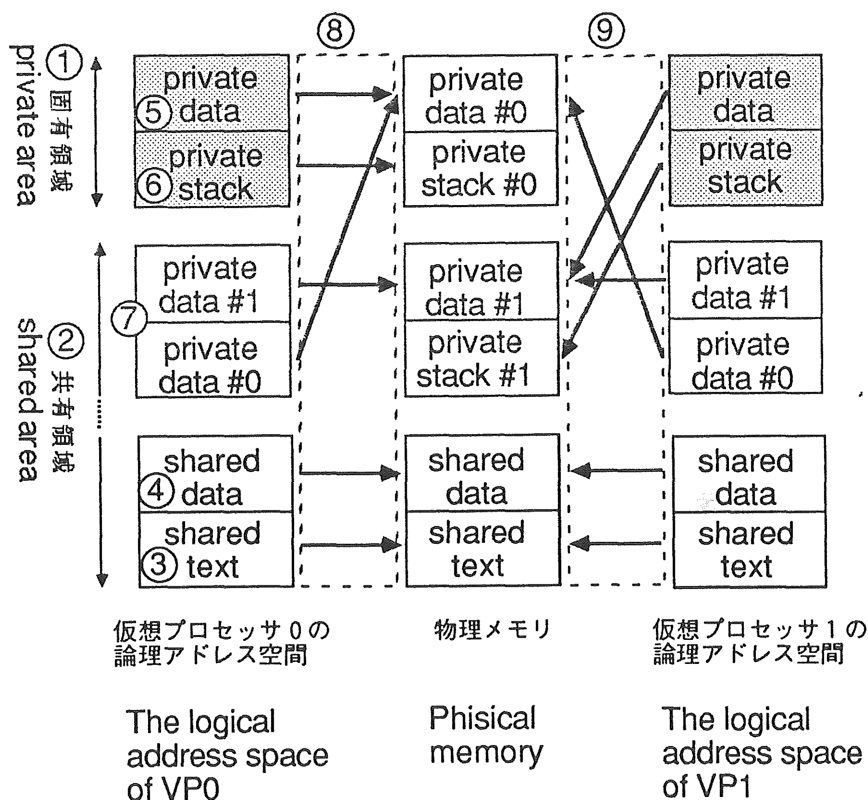


図３－１０　２つの仮想プロセッサの論理アドレス空間と物理メモリの対応

Figure 3-10: Mapping the logical address spaces of two virtual processors to phisical memory.

図３－１０では、アドレス空間の上部（アドレスが大きい方、図でも上）に固有領域（①）、アドレス空間の下部に共有領域（②）が割り付けられている。共有領域は、

テキスト・セグメント（text segment）（③）、データ・セグメント（data segment）（④）に分けられる。前者は、機械語命令を格納する部分であり、読込み専用になっている。後者は、データを格納する部分であり、読み書き可能になっている。固有領域は、固有データ・セグメント（private data segment）（⑤）と固有スタック・セグメント（private stack segment）（⑥）に分けられ、両方とも読み書き可能である。前者は、仮想プロセッサごとに固有の変数を割り付けるために利用される。後者は、マイクロプロセス・スケジューラ（図３－７第９行のmp_scheduler()）のスタックとして用いられる。各仮想プロセッサの固有データ・セグメントは、他の仮想プロセッサによる参照を可能にするために、アドレス空間の一部に割り付けられている（⑦）。

図３－７において、キーワード private（第１行、第２行）は、この変数を固有領域に割り付けるように指示するものである。これとは逆に、変数を共有領域に割り付けるように指示するキーワードは、図３－８第１行に現れている shared である。このような機能は、Ｄｙｎｉｘシステム［６］におけるいくつかの言語処理系において既に実現されている。Ｄｙｎｉｘでは、ＵＮＩＸのオブジェクト・コード形式（a.out形式）を拡張して、共有と固有の２つのデータ・セグメントを提供している。

３．６．４　仮想プロセッサの固有領域の利用

仮想プロセッサの固有領域は、次の４種類のデータを格納するために用いられる。下記（１）、（２）、（３）は、固有データ・セグメントに、（４）は、固有スタックセグメントに割り付けられる。

（１）仮想プロセッサの識別子（図３－７第１行）。

（２）仮想プロセッサごとに必要となるデータ。

利用者空間において軽量プロセスを実現する場合、このようなデータが必ず必要になる。たとえば、カーネル・コールのエラーを保持する変数 errno や現在実行しているマイクロプロセスの制御ブロック（mpcb、 microprocess control block）へのポインタが仮想プロセッサごとに必要となる（図３－７第２行）。

（３）共有する必要がない静的変数。

Ｃ言語のライブラリ関数の中には、静的変数（static variable）に結果を代入しそのポインタを返すものや、静的変数に中間状態を格納するものが多数存在する。前者の例として、時刻を文字列に変換する ctime()、各種検索用ライブラリ gethostbyname()、getpwent()、後者の例としては、引数を解析するルーチン getopt() があげられる。このような静的変数を固有領域に割り付けるにより、逐次プログラム用に開発されたライブラリ関数を並列プログラムにおいて利用することが可能となる。

（４）マイクロプロセス・スケジューラ用のスタック。
これは、（２）の特殊なものである。（どの仮想プロセッサも任意のマイクロプロセスを実行することを可能にするために、マイクロプロセスのスタックは、共有領域に置かれる。）

固有スタック・セグメントの存在により、図３－７に示したようなプログラムをスタック・ポインタの切替えなしに動作させることが可能になっている。たとえば、図３－７の第７行において、カーネル・コール vp_allocate()を発行する仮想プロセッサ（仮想プロセッサ０）も、新たに生成される仮想プロセッサ（仮想プロセッサ１）も、同一のアドレスに復帰する。そして、どの仮想プロセッサも同一のアドレスにあるスタックを利用することが可能となる。

３．６．５　固有領域を利用しない方法との比較

仮想プロセッサごとの固有領域を利用しない方法として、仮想プロセッサの識別子を実プロセッサのレジスタに格納する方法が考えられる。この方法では、３．６．４項で述べた（２）、（３）、（４）を、共有領域にある配列として実現しなければならない。たとえば、現在実行中のマイクロプロセス制御ブロック（struct mpcb）へのポインタ mp_current は、図３－１１に示すように共有領域にある配列を用いて実現することができる。ここで、MAX_NVP（図３－１１第１行）は、仮想プロセッサ数の最大値である。get_vpid()（図３－１１第３行）は、実プロセッサに格納された仮想プロセッサの識別子を参照するための関数である。このプログラムでは、マクロを用いて配列を単純な変数として扱えるようにしている。

```
1: #define MAX_NVP 32
2: struct mpcb *mp_current_array[MAX_NVP]; /* shared */
3: #define mp_current mp_current_array[get_vpid()]
4: ....
5: {
6:         mp_lock( mp_current );
7:         ....
8:         mp_unlock( mp_current );
9: }
```

図３－１１　配列と仮想プロセッサ識別子による
仮想プロセッサごとの固有領域の実現
Figure 3-11: A per-virtual-processor variable by using
virtual processor identifiers and an array

配列と実プロセッサのレジスタに格納された仮想プロセッサの識別子を用いる方法には、次のような問題点がある。

（１）配列のアドレス計算のオーバヘッドにより実行速度が低下する。

（２）Ｃ言語のように静的に配列の大きさを決定する言語では、最大の仮想プロセッサ数分のメモリが常に必要になり、メモリの無駄が生じる。メモリの無駄を避ける方法として、必要なメモリをヒープ上に動的に確保する方法が考えられる（Ｃ言語では、malloc()ライブラリ関数を用いる）。しかしながら、この方法では、メモリへのアクセス回数が増加し、実行速度の低下を招く。

（３）実プロセッサのレジスタを１個専有することになるため、応用プログラムの利用可能レジスタ数が減る。

（４）大域的なレジスタ変数機能を提供している言語処理系が一般的に利用可能にはなっていない。

仮想プロセッサの固有領域を用いることにより、これらの問題を解決することができる。

仮想プロセッサの識別子を実プロセッサのレジスタに格納する代りに、必要になった時にカーネル・コールを発行する方法も考えられる。しかしながら、利用者空間内において軽量プロセスを実現しているという利点が失われてしまうので、この方法を採用することはできない。

３．７　マイクロプロセスと仮想プロセッサの利用

この節では、マイクロプロセスと仮想プロセッサの利用法を示す。最も重要な利用は、共有メモリ型マルチプロセッサにおける並列処理である。そのほかに、ファイルの先読み、入出力処理とＣＰＵ処理の重ね合わせなどにも利用される。

３．７．１　共有メモリ型マルチプロセッサにおける並列処理

ここでは、簡単な例として、ループ分割（loop splitting）をマイクロプロセスを利用して実現する。図３－１２に示した、配列の要素の和を求める逐次プログラムを、マイクロプロセスを用いてループ分割したものを、図３－１３に示す。ここでは、同期プリミティブとして計数セマフォ（counting semaphore）を用いている。

図３－１３で、nmprocs は、マイクロプロセスの数である。ＣＰＵ処理を主とする（CPU-intensive）プログラムの場合、実プロセッサ数と等しい数の仮想プロセッサを生成する。そして、実プロセッサ数の数倍のマイクロプロセスを生成して、並列処理を行う。ここで、実プロセッサ数よりも多くのマイクロプロセスを生成する点に、多重プログラミング・システムの特徴が現れている。

```
1: array_sum( a,n )
2:     int a[];
3:     int n ;
4: {
5:     int i,sum ;
6:         sum = 0 ;
7:         for( i=0 ; i<n ; i++ )
8:         {
9:             sum += a[i] ;
10:        }
11:        return( sum );
12: }
```

図３－１２　ループ分割する前の逐次プログラム

Figure 3-13: A sequential program to be loop-splited.

そのサイトの負荷が低い場合、このプログラムは、単一プログラミング・システムとほぼ同等の速度で実行される。そのサイトの負荷が大きく、割り当てられる実プロセッサの数が少なくなったとしても、このプログラムは、動作する。さらに、並列処理の途中で、割り当てられる実プロセッサの数が変動したとしても、まったく問題なく動作する。

単一プログラミング・システム用の並列プログラムの場合、プロセッサ数に等しいプロセス（並列処理の単位）が生成される。すなわち、nmprocs を実プロセッサ数に合せる。このようなプログラムをそのまま多重プログラミング・システムに移植しても、効率よく動作しない。それは、全ての実プロセッサがそのプログラムに割り当てられるとは限らないからである。上のプログラムにおいて、nmprocs を実プロセッサ数にした時、プロセッサが１個欠け、nmprocs-1 個しか割り当てられなかったとすると、そのプログラムの実行時間は、nmprocs 個のプロセッサが割当てられた時の２倍になる。さらに、同期処理において、ビジー・ウエイトが用いられている場合、大量のＣＰＵ時間が浪費されてしまう。この浪費は、多重プログラミング・システムでは許されない。

３．７．２　入出力の重ね合わせ

複数の入出力の並列実行や、ディスク・ブロックの先読みを実現するためにも、マイクロプロセスと仮想プロセッサを利用することができる。たとばＵＮＩＸでは、ファイルをシーケンシャルにアクセスする場合に限り、１ブロックの先読みを行う。データベース処理では、ファイルをランダムにアクセスする場合でも、次に必要となるディスク・ブロックを予め知ることができる場合が非常に多い。このような場合、以下に述べるような方法により、先読みを実現することができる。

```
 1: array_sum( a, n, nmprocs )
 2:     int a[];    /* int a[0..n-1]; */
 3:     int n ;
 4:     int nmprocs ; /* number of microprocesses */
 5: {
 6:     int i, sum, bite ;
 7:     sem_t mutex ;
 8:     sem_t done ; /* private semaphore */
 9:
10:         sem_init( &done, -nmprocs+1 );
11:         sem_init( &mutex, 1 );
12:         bite = n / nmprocs ;
13:         for( i=0 ; i<n ; i+=bite )
14:         {
15:             mp_create( array_sum_task, ARRAY_SUM_TASK_SSIZE, MP_READY,
16:                        &a[i], bite, &mutex, &done, &sum );
17:         }
18:         sem_P( &done );
19:         return( sum );
20: }
21:
22: array_sum_task( a, n, mutexp, donep, sump )
23:     int a[];
24:     int n ;
25:     sem_t *mutexp ;
26:     sem_t *donep ;
27:     int *sump ;
28: {
29:     int i, sum ;
30:         sum = 0 ;
31:         for( i=0 ; i<n ; i++ )
32:         {
33:             sum += a[i] ;
34:         }
35:         sem_P( mutexp );
36:             *sump += sum ;
37:         sem_V( mutexp ); /* mutual exclusion */
38:         sem_V( donep );  /* wakeup parent */
39:         mp_exit();
40: }
```

図３－１３　マイクロプロセスによるループ分割

Figure 3-13: Loop-splitting by using microprocesses.

図３－１４に示すように、データ処理を行うマイクロプロセスの他に、データ入力を行うマイクロプロセスとバッファを２個用意する。データ入力を行うマイクロプロセスは、データ処理を行うマイクロプロセスが一方のバッファについて処理を行っている間に、別のバッファに次に必要となるブロックの読込みを行う。この時、データ入力を行うマイクロプロセスを実行していた仮想プロセッサは、入出力の完了待ち状態になる。しかしながら、そのプロセスには、他に実行可能な仮想プロセッサが存在

するので、その仮想プロセッサにコンテキストを切り替えることで、即座に応用プログラムの実行に戻ることが可能である。

従来、ファイルの先読みは、非同期入出力機能を利用して実現されることが多かった。しかしながら、非同期入出力は、同期型入出力に比較してプログラミングが困難であるという問題がある。マイクロプロセス（軽量プロセス）を利用する場合、論理的に並列に実行したい部分をマイクロプロセスとして記述するだけで、自然に並列性が抽出される。

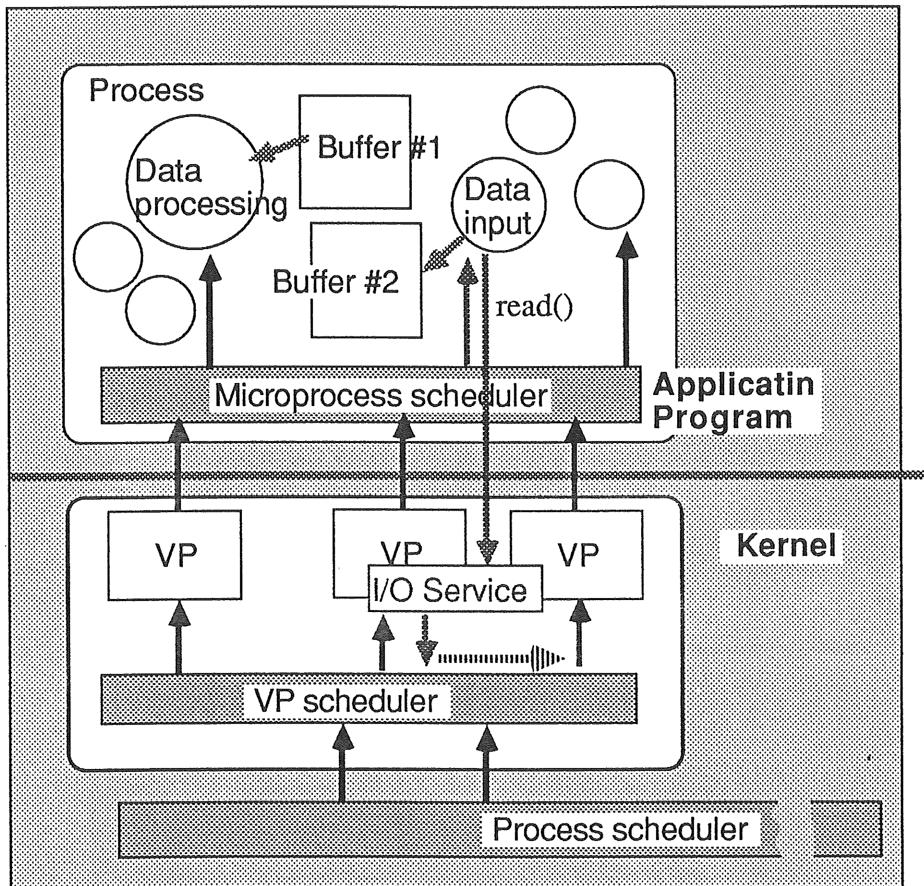


図３－１４　マイクロプロセスによる入出力の重ね合わせ

Figure 3-14: Overlapping I/O and CPU processing by using microprocesses.

３．７．３　非同期通信の実現

本システムでは、プロセス間通信のプリミティブとして、同期式のもの（遠隔手続き呼出し）を採用している。応用プログラムにおいて、非同期式のものが必要になった場合は、マイクロプロセスと仮想プロセッサを用いて簡単に実現することができる。

非同期処理を行いたい部分で、新たにマイクロプロセスを生成する。新たに生成されたマイクロプロセスは、他の仮想プロセッサにより実行される。その後、必要になった時点で、先ほどのマイクロプロセスの実行完了を待つ。

3．7．4　動的負荷分散

分散型オペレーティング・システムの研究分野において、動的負荷分散は、重要な課題となっている。その手段の１つとしてプロセスの移動（process migration）があげられる。プロセスの移動とは、実行中のプロセスを別のサイトに移動し、プロセスの実行を継続することである（図３－１５（ａ））。

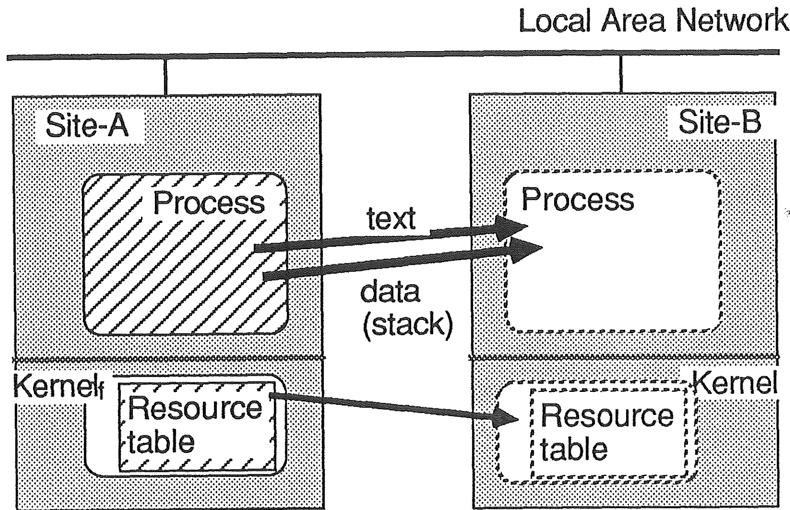


図３－１５（ａ）　重量プロセスの移動

Figure 3-15(a): Migration of a heavyweight process.

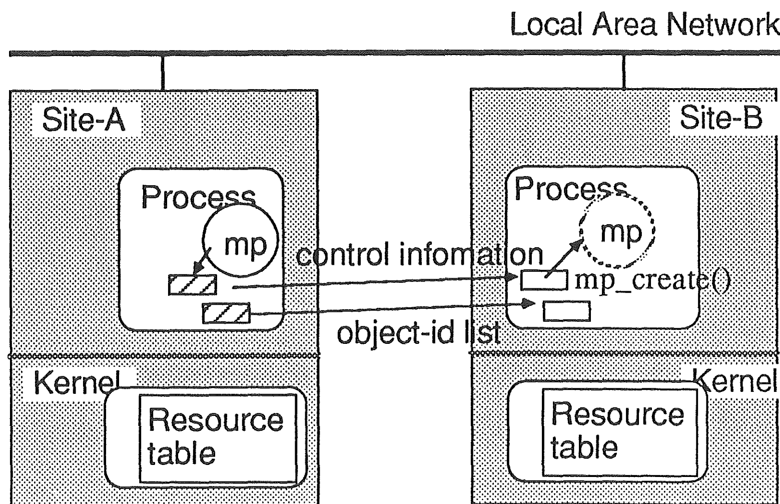


図３－１５（ｂ）　軽量プロセスの移動

Figure 3-15(b): Migration of a lightweight process.

図３－１５　プロセスの移動

Figure 3-15: Process migration.

プロセスの移動において、重量プロセス（資源割当てと保護の単位）ではなく、軽量プロセスを用いる方法が考えられる（図３－１５（ｂ））。重量プロセスの場合、プロセスの移動そのものに要するコストが大きく、かつ、時間もかかる。これに対して、軽量プロセスの場合、移動に必要となるデータ量を削減することが可能で、処理効率が改善されると思われる。その理由は、第１に、複数のサイトを利用して１つの応用プログラムを実行する場合、各サイトで実行されるプログラムのロード・モジュールが同一であることが多いからである。この場合、移動先のプロセスに同一のテキストが存在するので、テキストを転送する必要がない。

第２に、マイクロプロセスの性質によっては、非常に小さな制御情報からマイクロプロセスを再構成することが可能な場合がある。この場合、その制御情報だけを送ることにより、マイクロプロセスの固有データ領域全体を転送する必要がなくなる。このように、応用プログラムの性質を利用することにより、応用プログラムのレベルにおいてマイクロプロセスの移動を行い、動的負荷分散を実現することも可能である。

小さな制御情報からマイクロプロセスを再生することは、ＡＲＧＵＳにおけるガーディアン（gurdian） の再生と類似している［４９］。ガーディアンとは、一種のプロセスであり、内部に変数を持ち、ハンドラ（handler）と呼ばれる手続きを通じてのみ内部の変数へのアクセスが許される。内部の変数は、通常のメモリ中にのみ記録され、障害が発生すると失われる揮発性の変数と、ディスクなどに保存され、障害が発生しても失われない安定な（stable）変数に分類される。障害が発生すると、ＡＲＧＵＳでは、この小さな安定な変数から、通常の変数の状態を回復し、ガーディアンを再生する。このような機構は、マイクロプロセスの移動の目的も活用することができる。

応用プログラムのレベルにおいてマイクロプロセスの移動を行う場合、カーネル中のプロセスの資源管理表の移動がしばしば問題となる。本システムでは、この問題は生じない。それは、結合を作らない（connection-less）の通信プリミティブである遠隔手続き呼出しと、利用者レベルで受け渡し可能なオブジェクト識別子を利用しているからである。現在参照しているオブジェクトのオブジェクト識別子を移動先に転送することにより、プロセスの資源表を転送せずに元のプロセスと同一のオブジェクトを参照することが可能である。

３．８　応用固有の同期・通信プリミティブの開発

３．８．１　層構造を持つライブラリの利用

３．５節で述べたように、本システムのマイクロプロセス・ライブラリ（利用者レベルの軽量プロセスを実現するライブラリ）は、層構造を持っている。このライブラリを利用する場合、各並列プログラムのプログラマは、応用固有の同期・通信プリミティブを構築するために、たとえば、以下で示すようなレベルで機能を利用すること

ができる。高いレベルの機能を利用するほど、細かい記述から開放される。一方、低いレベルの機能を利用するほど、高い性能を得ることができる。

（レベル４）第３層の同期・通信プリミティブから選択して利用する。この場合、複数のプリミティブを混在させて利用することも可能である。

（レベル３）第３層の同期・通信プリミティブを利用して、独自の同期・通信プリミティブを開発する。

（レベル２）第２層の sleep/wakeup モデルに基づく同期・通信プリミティブを利用して、独自のプリミティブを開発する。

（レベル１）第１層のスピンロック・ライブラリとコンテキストの保存・回復の機能を用いて、独自のプリミティブを開発する。このレベルでは、アーキテクチャから独立したプログラムを開発することができる。

（レベル０）第０層のカーネル・コールだけを利用して同期・通信プリミティブを開発する。

３．８．２　コルーチンの性質の利用

マイクロプロセス間の同期・通信プリミティブは、応用プログラムごとに開発される。本方式では、コルーチンの性質を活用して、固有の同期・通信プリミティブを効率的に開発することができる。

３．４．１項、および、３．５．２項で述べたように、マイクロプロセスは、コルーチンと非常に類似している。相違点は、マイクロプロセスの場合、並列処理の対象となる点にある。そのためマイクロプロセスの場合、マイクロプロセスの共有変数の相互排除を行う必要がある。逆に、マイクロプロセスにおいて、共有変数の相互排除操作を削除することにより、完全にコルーチンに還元することが可能である。

並列プログラムのデバッグにおいて、実行の再現性がないことが大きな障害となっている。対象的に、コルーチンの場合、プログラムの実行に再現性がある。これは、プログラムが逐次的に実行されるからである。この再現性があるという性質は、プログラムのデバッグにおいて、非常に都合がよい。本システムでは、１個の仮想プロセッサを用いることにより、容易にコルーチンと同じ動作を行わせることができる。すなわち、カーネル・コール vp_allocate() の引数として１を指定するか、または、そのカーネル・コールを発行しなければよい。

以上のことより、本マイクロプロセス方式において、新たにマイクロプロセス間の同期・通信プリミティブを開発する場合、次の様な手順で効率的に行うことができる。

（１）相互排除操作を含まないマイクロプロセスにより、同期・通信プリミティブを構築する。これは、コルーチンと完全に等価である。

（２）マイクロプロセス間の共有変数に対する相互排除の操作を付加する。そして

これを、１つの仮想プロセッサを用いて実行する。この時に、スピン・ロックの解除を忘れる、あるいは、１つのマイクロプロセスで同じ変数に２回ロックを試みるなどのバグが取除かれる。この時点で、実行のタイミングに関するバグを除いて、ほとんどのバグが取除かれる。

（３）最後に、複数の仮想プロセッサを用いて実行し、タイミングに関するバグを取る。

ステップ（２）において、スピン・ロックを実現するライブラリ手続きとして、デバッグ用の特殊なものを用意している。それは、ロック操作において、既にロックされている場合に、メッセージを表示して直ちに処理を中断するものである。１つの仮想プロセッサにおいては、ロックが重なることは、デッドロックを意味する。

３．８．３　応用固有の同期・通信プリミティブの開発例

この節では、データベースの並列処理システムとして当研究室で開発しているＳＭＡＳＨを取上げ、本システムで提案した軽量プロセス機能の有効性について論じる。ＳＭＡＳＨは、データベース、および、知識ベースを対象とした並列処理システムである［３８］［３９］［４０］［５０］。ＳＭＡＳＨが対象とするハードウェアは、本システムと同様、図１－１に示すような共有メモリ型マルチプロセッサと単一プロセッサが高速ネットワークにより結合された環境である。

ＳＭＡＳＨの特徴は、関数型プログラミングの概念をデータベース処理に適応している点にある。関数の評価方式としては、要求駆動型評価（demand driven evaluation）を用いている。ＳＭＡＳＨでは、関数間のデータの受け渡しにストリームを用いている。ＳＭＡＳＨでは、任意のデータベース演算を関数として記述し、システム内に組み込むことを可能にしている。このため、関数間に存在する並列性を抽出することを可能になり、データベースの多様な応用分野に柔軟に対応することが可能となっている［３８］［５７］［７３］。

３．８．３．１　ＳＭＡＳＨのプリミティブ・セット

ＳＭＡＳＨは、１章において述べた単一プログラミング・システムに相当する。ＳＭＡＳＨは、図１－１のような並列／分散処理ハードウェア環境を、専有して使用する。

ＳＭＡＳＨでは、このような環境のもとで、関数型計算を実行するためのプリミティブ・セットを設定している。これは、抽象計算機の命令セットであり、一種の中間言語である。そして、各プリミティブに対応する実行時ルーチンが存在する。関数型言語で記述されたプログラムは、このプリミティブを呼び出すコードを含むオブジェクト・コードにコンパイルされる。したがって、このプリミティブ・セットを実現することが、ＳＭＡＳＨシステムを移植することになる。ここでは、ＳＭＡＳＨのプリ

ミティブ・セットを本システムが提供する機能を利用して実現することを考える。

ＳＭＡＳＨのプリミティブの主要な機能は、次の３つである。

（１）関数のインスタンスの生成・消去

（２）ストリーム・データの通信路であるチャネルの生成・消去

（３）チャネルを通じたストリーム・データの受け渡し

ＳＭＡＳＨにおけるフロー制御（flow control）は、要求駆動型評価に基づいておこなわれる。すなわち、ストリーム・データの生産者側の実行は、消費者側からの要求（demand）により制御される。これは、ＴＣＰ／ＩＰにおけるウインドウ制御と類似している［４６］［１０１］。ＳＭＡＳＨでは、高いレベルにおいてフロー制御を行っている点に特徴がある。

主なプリミティブの名称と機能を、以下に示す。

new(func,site,arg1,arg2,...)

指定された関数 func のインスタンス（これを関数インスタンスとよぶ）をサイト site 上に生成する。arg1,arg2,... は、関数インスタンスへ引数として渡される。引数として、以下で述べるチャネルの識別子を渡すことができる。この時、自動的にコネクションが形成される。

channel_t channel(type, gran , prod_site, cons_sites)

チャネルと呼ばれるストリーム通信路を設定する。type は、 ストリーム要素の型、gran は、 チャネル用のバッファの大きさを指定する。prod_site は、ストリームの生産者となる関数インスタンスが存在するサイト、 cons_sites は、消費者となる関数インスタンスが存在するサイトを指定する。消費者として、複数のサイトを記述することができる。

get(p,data)
receive(p,data)
put(p,data)
send(p,data)
mark_eos(p)

get(), receve() は、ストリームからデータを入力するプリミティブ、put(), send() は、ストリームヘデータを出力するプリミティブである。p は、ポートと呼ばれるコネクションが形成されたチャネルである。get() と put() は、バッファが、あるいは、データが未到着で入出力に失敗した際、そのまま待ち続ける。これに対して、receive() と send() は、失敗を返し、ブロックされることはない。mark_eos() は、ストリームの終了の目印をつける。

select(p1,p2,...)

enable(p)
disable(p)

select は、入出力可能なストリームの非決定的な選択を行う。ＵＮＩＸ４．３ＢＳＤの select システム・コールと同様な機能を提供する［４６］。
enable()/disable()は、指定されたポートを選択の対象に入れる／対象から外す操作を行う。

３．８．３．２　１サイト内におけるＳＭＡＳＨのプリミティブの実現

前項で述べたプリミティブ・セットを、本システムが提供する軽量プロセス機能を利用して実現する方法について述べる。ここでは、共有メモリ型マルチプロセッサにおける実現方式を述べ、ネットワーク環境における実現は次の３．８．２．３で述べる。ここで、単一プロセッサは、プロセッサ数が１の共有メモリ型マルチプロセッサと考える。

基本的な実現方式は、ＳＭＡＳＨの関数インスタンスをマイクロプロセスとして実現することである。関数インスタンスの生成 new() プリミティブは、マイクロプロセスを生成するライブラリ関数 mp_create() を用いて実現した。

関数インスタンス間の通信プリミティブを、本マイクロプロセス・ライブラリの第２層の機能を利用して第３層に実現した。ＳＭＡＳＨのチャネルは、図３－１６に示すように、マイクロプロセス間の共有データ、マイクロプロセスの固有データ、実際にストリームデータを格納するバッファにより実現される。それそれ、チャネル制御表、ポート制御表、ページとよぶ。チャネル制御表は、複数のマイクロプロセスから同時にアクセスされる可能性があるので、ロックにより相互排除が行われる。ポート制御表には、１つのマイクロプロセスに固有の情報が格納される。したがって、相互排除を行う必要がない。１つのページついて処理を行っている間、マイクロプロセスは、ポート制御表だけを参照し、チャネル制御表を参照することはない。このようにして、マイクロプロセス間の同期制御のオーバヘッドを軽減している。

図３－１６では、ストリームの通信路であるチャネル１個に対して、２個のページが割り当てられている。この結果、ストリームの生産者となるマイクロプロセスと消費者となるマイクロプロセスを並列に実行することが可能となっている。

ＳＭＡＳＨプリミティブの実装を、Ｃ言語により行った。ソース・プログラムのお大きさは、全体で約２５００行である。

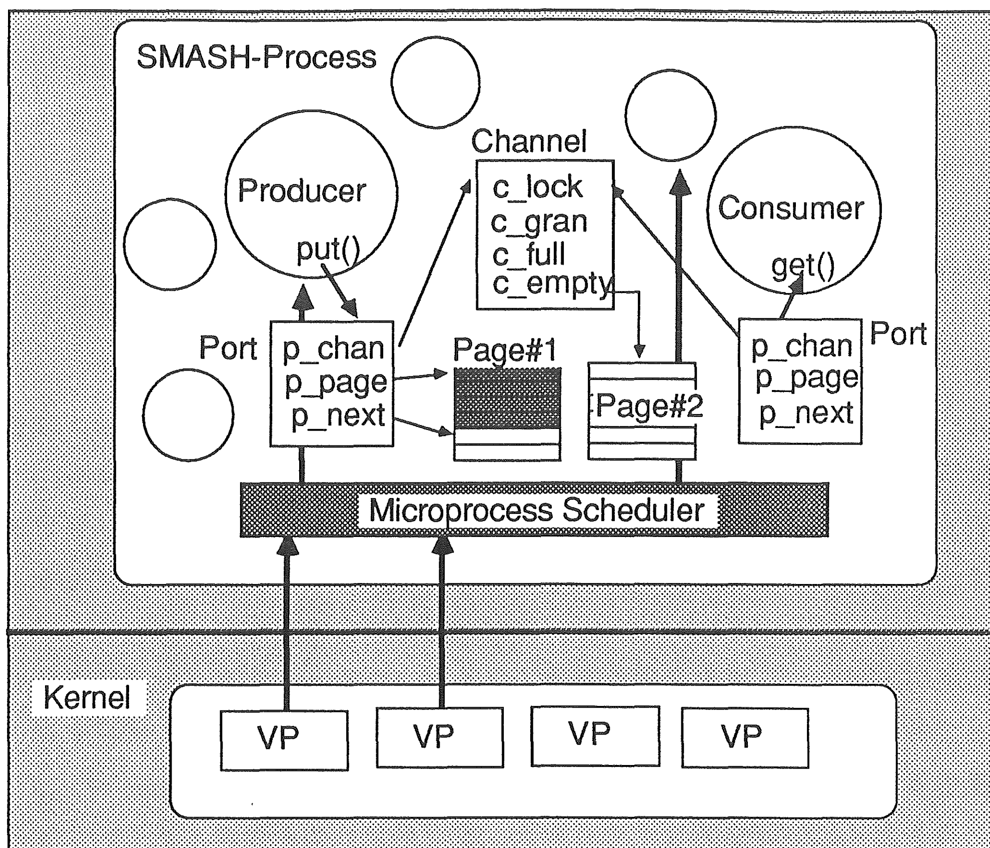


図３－１６　共有メモリ型マルチプロセッサにおける
マイクロプロセスを利用したＳＭＡＳＨの実現

Figure 3-16: The implementation of SMASH primitives on a shared-memory multiprocessor by using microprocesses.

３．８．３．３　ネットワーク環境におけるＳＭＡＳＨのプリミティブの実現

ネットワーク環境におけるＳＭＡＳＨのプリミティブ・セットの実現を、図３－１７に示す［７２］。ＳＭＡＳＨのインスタンスは、１サイト内と同様に、マイクロプロセスとして実現される。マイクロプロセスの生成に関しては、完全に応用プログラムの制御にまかされており、本システムのカーネルは、一切関与しない。したがって、遠隔サイト上に関数インスタンスを生成する場合、ネットワーク通信機能を用いて、遠隔サイト上のＳＭＡＳＨプロセスに対してメッセージを送る。メッセージを受取ったプロセスは、自分自身の中にマイクロプロセスを生成する。

この実現においては、関数インスタンスの他に、ネットワーク通信を実現するためにマイクロプロセスが利用されている。具体的には、ネットワークからのメッセージを受信するために、マイクロプロセスが利用されている。マイクロプロセス（軽量プロセス）の利用により、並列性の記述が自然に行われ、プログラムの論理的な構造が整理されている。

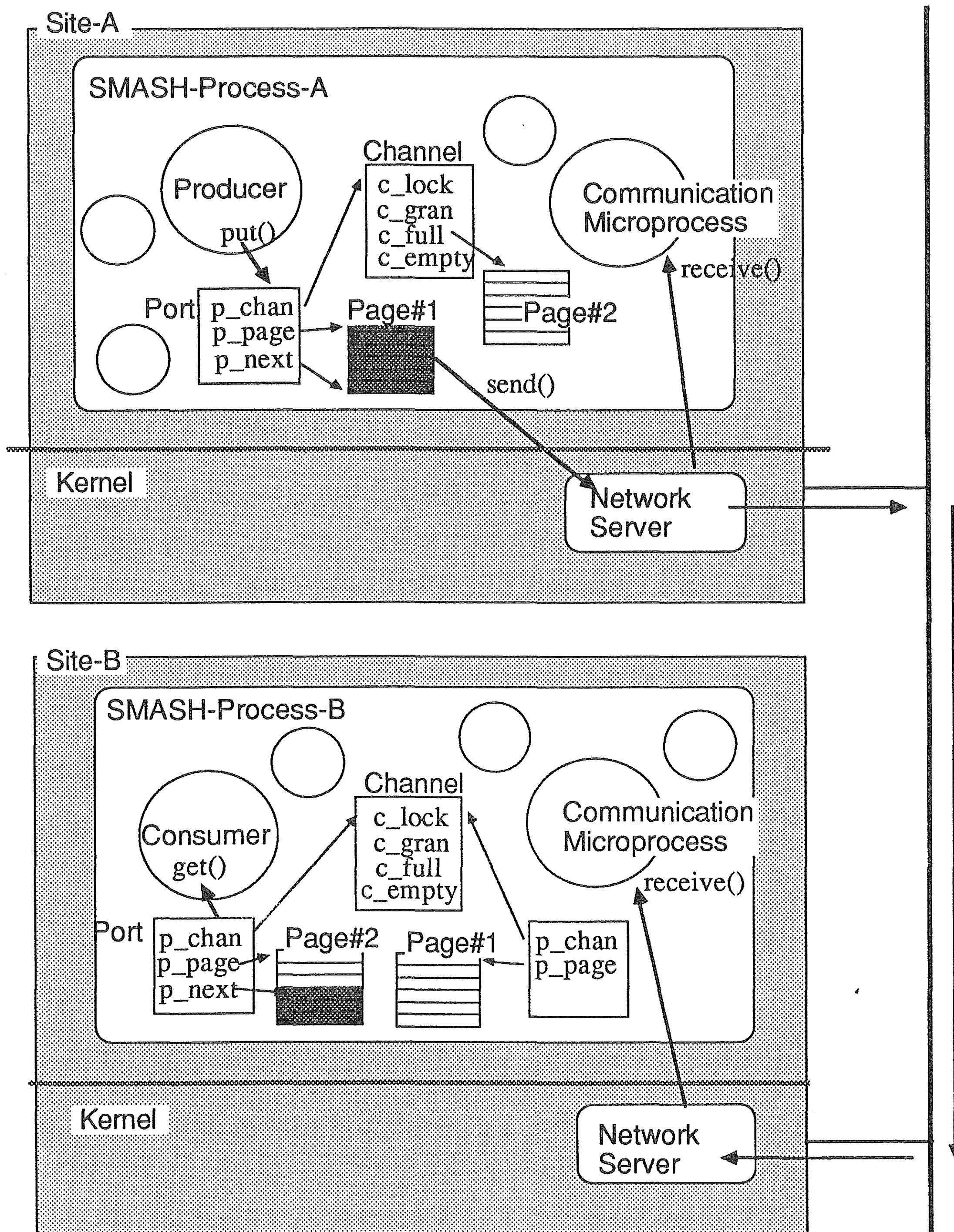


図３－１７　ネットワーク環境における
マイクロプロセスを利用したＳＭＡＳＨの実現

Figure 3-17: The implementation of SMASH primitives on networked processors by using microprocesses.

３．９　応用固有のスケジューラの開発

本方式では、応用固有のスケジューラを、マイクロプロセス・スケジューラとして、利用者空間内に構築する。この方法の利点は、以下の通りである。

（１）応用固有の情報の入手が容易である。スケジューラは、プロセス間の保護の壁を越えることなく、実行の状況を調べることができる。

（２）カーネル中にスケジューラを設ける方法と比較して、スケジューラを取り替えることが容易である。

（３）同時に動作する複数の並列応用プログラムのスケジューラが互に干渉しあうことがない。

３．９．１　層構造を持つライブラリの利用

３．５節において述べたマイクロプロセス・ライブラリを利用する場合、各並列プログラムのプログラマは、応用固有のマイクロプロセス・スケジューラを構築するために、たとえば、次のようなレベルで機能を利用することができる。高いレベルの機能を利用するほど、細かいスケジューリングの記述から開放される。一方、低いレベルの機能を利用するほど、きめ細かい制御を行うことができる。

（レベル５）予め用意されているスケジューラから選択して利用する。（現在、代表的なスケジューリング方式として、ＦＩＦＯ、および、固定優先順位方式を用意している。）

（レベル４）マイクロプロセス・ライブラリ中のレディ・キュー・モジュール（図３－３）と同じ外部仕様をもつモジュールを記述し、残りのマイクロプロセス・ライブラリと結合して利用する。３．９．２項において、このレベルを用いて記述した優先順位式スケジューラについて述べる。

（レベル３）第２層の sleep/wakeup 同期プリミティブと同じレベルに新たなプリミティブを追加する。たとえば、sleep_and_wakeup のように、複合的なプリミティブを追加する。

（レベル２）マイクロプロセス・ライブラリ中のスケジューラ・モジュール（図３－３）と同じ外部仕様をもつモジュールを記述し、残りのマイクロプロセス・ライブラリと結合して利用する。

（レベル１）第１層を利用して、スケジューラを記述する。

（レベル０）第０層（カーネル・コール）だけを利用して、スケジューラを記述する。

３．９．２　応用固有の軽量プロセス・スケジューラの開発例

ここでは、３．８．３項において述べたＳＭＡＳＨシステムにおけるスケジューラの記述を例に、応用固有のスケジューラの開発について述べる。

次の様な３種類のスケジューラを記述してみた。

（１）優先順位なし（有限バッファ、eager評価方式）
このスケジューラでは、データ、デマンドの到着に関わらず、マイクロプロセスを実行可能にし、レディ・キューに登録する。到着していないデータを待つ時、および、出力用のバッファが満たされている時に、待ち状態に移す。

（２）優先順位付き（有限バッファ、eager評価方式）
これは、（１）に、優先順位を導入したものである。確実に必要な計算を行うマイクロプロセスに高い優先順位を与える。それ以外のマイクロプロセスも、実行可能になった段階で低い優先順位を与え、レディ・キューに登録する。

（３）優先順位付き（有限バッファ、要求の先出し付き要求駆動）
このスケジューラでは、根に近い関数インスタンス（マイクロプロセス）から要求（demand）を受け付けて初めて実行可能状態にする。並列性を抽出するために、要求の先だしを行う。これにより、確実に必要な計算を行うマイクロプロセスだけが実行可能となり、レディ・キューに登録される。

ここで、（３）により実現される方針が、ＳＭＡＳＨシステムの本来のスケジューリング方針に近い。（２）では、ＳＭＡＳＨにおける要求の発行時に優先順位を上げ、要求されたデータの生成の完了時に、優先順位を下げる。（１）、および、（２）では、結果的に無駄となる計算が行われることがある。（２）では、優先順位を利用することにより、無駄になる可能性が小さい計算を先に実行する。

ここで（１）を、あらかじめ用意してあるＦＩＦＯスケジューラを用いて実現した。これは、３．９．１項で述べたレベル５に対応する。また、（２）、（３）を、第２層（図３－３）のレディ・キュー・モジュールを入替えることで実現した。これは、３．９．１項で述べたレベル４に対応する。

（２）では、マイクロプロセスの優先順位は、動的に決定される。これは、実時間システムで用いられる**優先順位継承方式**（Priority Inheritance Protocols）の単純な場合になっている［７４］。（ｃ）の実現を、（ｂ）において、優先順位が上がるまでマイクロプロセスの実行を遅延することにより行った。

ここで実現したスケジューラの性能を調べる実験を行った。その結果については、３．１２．７項において延べる。

３．１０　仮想プロセッサを実現するカーネルの構成法

この節では、仮想プロセッサを実現するカーネルの構成法について述べる。この構成法の特徴は、利用者プロセスとカーネルの構造が相似である点にある。ここでは、まず仮想プロセッサの固有領域の実現について述べる。次に、これと相似の関係にある実プロセッサの固有領域について、その外部仕様、利用、および、内部実現について述べる。最後に、カーネル内部の軽量プロセス・スケジューラを用いたプロセスと仮想プロセッサのスケジューラの実現について述べる。

この節では、３．６節において述べた仮想プロセッサの外部仕様を例として、カーネルの構成法を述べる。しかしながら、ここで述べる方法は、汎用性があり、他の外部仕様を持つ仮想プロセッサやカーネル制御方式の軽量プロセスを実現する場合にも利用可能である。

３．１０．１　仮想プロセッサの固有領域の内部実現

この節では、３．６節で述べた仮想プロセッサの固有領域の外部仕様を実現する方法について述べる。論理的には、図３－１０に示したようなアドレス変換（図３－１０の⑧と⑨）を実現するようにＭＭＵ（Memory Management Unit）の変換表（translation table）を作成すればよい。しかしながら、単純に仮想プロセッサごと変換表を作成したならば、１つのプロセスについて仮想プロセッサの数だけの変換表が存在することになり、次の様な問題点が生じる。

（１）変換表の操作の実行速度が低下する。たとえば、プロセスにメモリを割り当てる場合、そのプロセスに属するすべての仮想プロセッサについて変換表を書き換えなければならない。そのためメモリ資源を割り当てる操作の実行時間は、仮想プロセッサの数に比例することになる。

（２）変換表のためのメモリ資源が無駄になる。

この項では、変換表の、利用者空間における共有領域に対応する部分を共有することにより上記の問題を解決する方法を示す。

３．１０．１．１　２段階の変換表を利用するＭＭＵ

ここでは、図３－１８に示すような２段階の変換表を利用するＭＭＵにおける変換表の設定について述べる。最近のマイクロプロセッサの多くは、多段の変換表を持っており、ここで述べる方式を利用することができる［９９］。たとえば、ＳＰＡＲＣでは、３段階、ＭＣ６８０３０では、４段階（０段階から４段階まで可変）、インテル８０３８６では２段階の変換表を利用する。実際に利用したＭＭＵは、２段階の変換表を利用する、モトローラ社のＭＣ８８２００である［５２］。

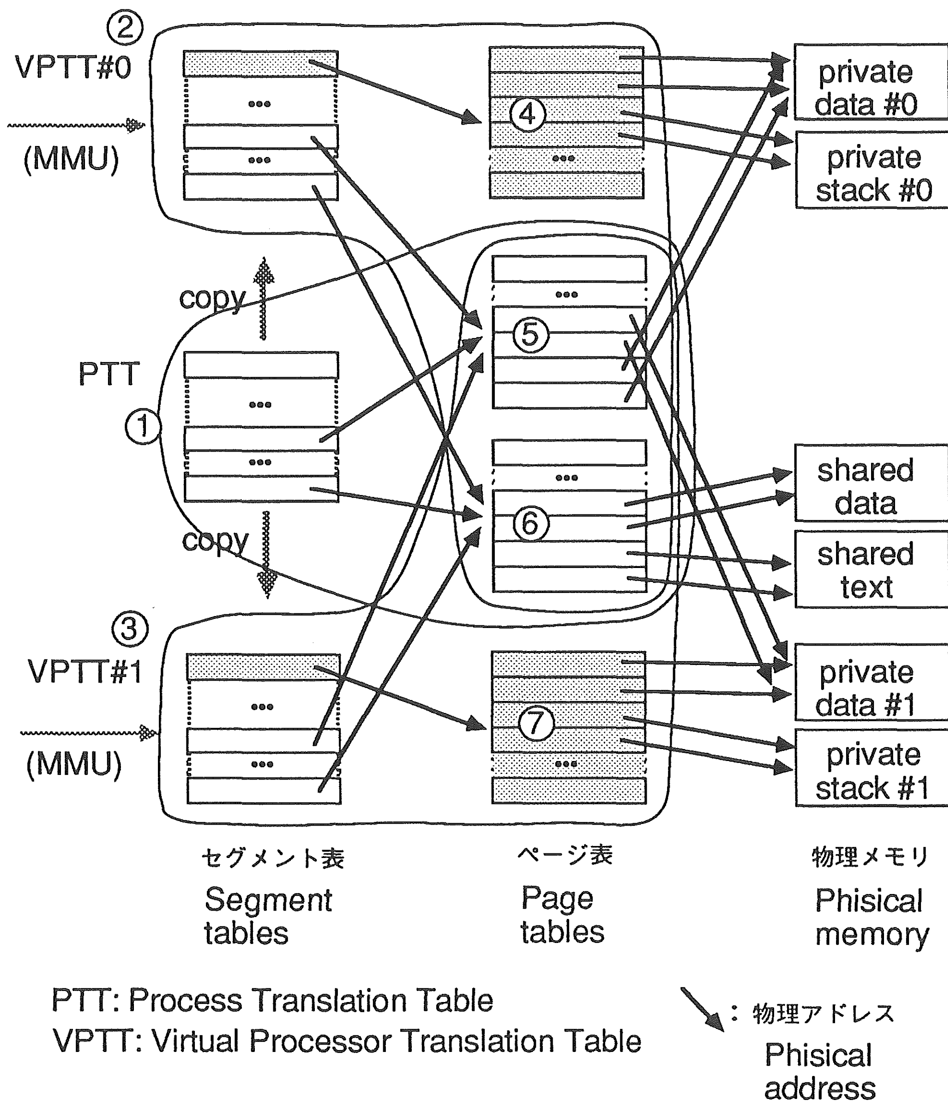


図３－１８　プロセス変換表と仮想プロセッサ変換表

Figure 3-18: Process and virtual processor translation tables

ここでは、２段階の変換表を用いることを仮定する。２段階以上の変換表の場合、ある段階で区切ることにより、２段階と同等に扱うことが可能である。ここでは、変換表の第１段をセグメント表（segment table）、第２段をページ表（page table）とよぶことにする。ＣＰＵが生成する論理アドレスは、上位ビットから

- セグメント番号
- ページ番号
- ページ内オフセット

に分けられる。アドレス変換は、次のようにして行われる。

（１）論理アドレス中のセグメント番号を添え字としてセグメント表のエントリを選択する。

（２）そのエントリには、ページ表の物理アドレスが含まれている。それを用いて、ページ表を選択する。

（３）そのページ表において、論理アドレスのページ番号を添え字としてエントリを選択する。

（４）そのエントリには、メモリの物理アドレスが含まれている。このアドレスとページ内オフセットを加えて、物理アドレスを生成する。

３．１０．１．２　プロセス変換表と仮想プロセッサ変換表

この項では、プロセス用と仮想プロセッサ用の２種類の変換表を用いて仮想プロセッサのアドレス空間を設定する方法を示す。前者をプロセス変換表（PTT, Process Translation Table, 図３－１８の①），後者を仮想プロセッサ変換表（VPTT, Virtual Processor Translation Table, 図３－１８の②と③）とよぶことにする。プロセス変換表は、共有領域の管理を行うためのものであり、ＭＭＵによって参照されることはない。仮想プロセッサ変換表は、その仮想プロセッサを実プロセッサが実行する時にＭＭＵにより参照するための変換表であり、共有領域と固有領域の両方のアドレス変換に関する記述を含む。

図３－１０に示した論理アドレスから物理メモリへの変換を実現する変換表を図３－１８に示す。図３－１８は、図３－１０と同様に、アドレスが小さい（０に近い）方が下になるように描かれている。図３－１０の⑧、⑨に示した変換を実現しているのは、図３－１８の②、③に示す２つの仮想プロセッサ変換表である。それらの変換表（②、③）は、２つのページ表（⑤、⑥）を共有している。そして、物理ページも共有している。この部分は、３．６．３項において述べた共有領域に対応している。

図３－１８において、影を付けた部分が固有領域に関係している部分である。たとえば、図３－１０の⑤に示した固有データ・セグメントの論理アドレスは、②と④を用いて物理アドレスに変換される。

３．１０．１．３　共有領域へのメモリの割り付け

仮想プロセッサのアドレス空間の大部分は、共有領域である。したがって、共有領域への物理メモリの割り付けの速度が重要となる。ここでは、ページ表を共有している利点を活用し、高速化を図る手順を示す。

（１）プロセス変換表（図３－１８の①）を根として、ページ表を共有しない方式と全く同じ手順を用いて、変換表を構築する。

（２）（１）において、プロセス変換表のセグメント表のエントリの中で変更したものを、そのプロセスに属するすべての仮想プロセッサの変換表のセグメント表（図３－１８では②、③）にコピーする。（１）においてセグメント表を変更しなかった場合は、なにもしない。

（３）実プロセッサが割り当てられている仮想プロセッサについて、ＭＭＵ中の変換表のキャッシュ（ＴＬＢ、Translation Look-aside Buffer、ＭＣ８８２００では、ＰＡＴＣ（Page Table Translation Cache）と呼ばれている）をフラッシュする。

（４）実プロセッサが割り当てられていない仮想プロセッサについて、変換表が変更されたことを示すマークを付ける。

これらの操作の中で、（２）の操作においてページ表を共有している利点が活かされている。もしページ表を共有していなかったならば、（１）の操作を仮想プロセッサの数だけ繰り返す必要が生じる。

ページ表を共有するためには、ある制限が必要である。それは、セグメント表の1エントリによって記述されるアドレス空間内に固有領域と共有領域の混在を許さないことである。この制限により、上の（２）において単純にセグメント表のエントリのコピー（浅いコピー、shallow copy）を行うだけでよいことになる。

３．１０．２　実プロセッサの固有領域

３．６．３項と３．６．４項では、利用者プロセスにおける仮想プロセッサごとの固有領域の外部仕様と利用について述べた。前項（３．１０．１項）では、仮想プロセッサごとの固有領域の内部実現について述べた。一方、カーネルの中においては、実プロセッサごとに必要となるデータが存在する。ゆえに、カーネルにおいて実プロセッサごとの固有領域を設定することは、有効である。この項では、その外部仕様、利用、および、内部実現について述べる。

３．１０．２．１　実プロセッサの固有領域の外部仕様

実プロセッサの固有領域とは、実プロセッサのアドレス空間（カーネルの論理空間）の一部に存在する、実プロセッサごとに異なる物理メモリが参照される領域のことである。これに対して、実プロセッサの固有領域以外の領域を実プロセッサの共有領域とよぶことにする。

実プロセッサの固有領域は、利用者レベルのアドレス空間の設定から独立している。すなわち、他の外部仕様を持つ仮想プロセッサを提供するカーネルや、カーネル制御方式の軽量プロセスを提供するカーネルの実現においても、実プロセッサの固有領域を利用することができる。たとえば、Ｍａｃｈシステムのカーネルを実現する場合においても、実プロセッサの固有領域を利用することが可能である。

３．１０．２．２　実プロセッサの固有領域の利用

３．６．４項で行った議論は、仮想プロセッサを実プロセッサに置き換えてもそのまま成り立つ。

３．１０．２．３　実プロセッサの固有領域の内部実現

実プロセッサの固有領域は、利用者空間とカーネル空間の変換表の形式が同じ場合、３．１０．１項で述べた、仮想プロセッサの固有領域の実現とほとんど同じ技術を用いて実現することが可能である。これにより、利用者プログラムに機能を提供するためのモジュールとカーネル自身の動作のために必要なモジュールの共通化を図ること

ができるという利点が生じる。

利用者空間とカーネル空間の変換表の形式が異なる場合には、共通化を図ることができない。しかしながら、カーネル空間においてＭＭＵが利用可能であるならば、実プロセッサの固有領域を実現することは、可能である。

固有領域の実現において、仮想プロセッサと実プロセッサで異なる操作が必要な点が存在する。これを以下にまとめる。

（１）カーネルには、固有スタック領域（スケジューラ用のスタック）が不用である。これは、自力で立上がる（bootstrap）時に利用するスタックをそのままスケジューラ用のスタックとして利用できるからである。

（２）カーネル用の変換表が完成するまで、セグメント表やページ表に用いるページの属性の設定を遅らせる必要がある。セグメント表やページ表に用いる物理メモリは、キャッシングの機能を停止しなければならない。そのためには、カーネル用のセグメント表やページ表があるページの属性を変更する必要がある。ところが、カーネル用の変換表を作成している途中の段階では、この属性変更を行うことは不可能である。したがって、１度変換表を作成した後に改めてセグメント表とページ表の物理アドレスを調べ、該当するページの属性を変更する必要がある。

３．１０．３　仮想プロセッサとプロセスのスケジューラ

３．１０．２項では、利用者レベルの抽象である仮想プロセッサの固有領域と同一の技術を用いて実プロセッサの固有領域を実現する方法について述べた。この項では、利用者レベルの軽量プロセスのスケジューラ（マイクロプロセス・スケジューラ）の技術を用いて、仮想プロセッサとプロセスのスケジューラを実現する方法について述べる。

３．１０．３．１　プロセスと仮想プロセッサのスケジューラの外部仕様

プロセス・スケジューラ（大域スケジューラ（global scheduler））とは、資源割当ての単位であるプロセスの間で公平なＣＰＵ時間の分配を実現するためのスケジューラである。公平な分配を実現するために、実プロセッサの横取り（preemption）が行われる。

仮想プロセッサ・スケジューラとは、同一プロセス内において、別の仮想プロセッサへの制御の移動を行うためのスケジューラである。（本システムの仮想プロセッサ・スケジューラについては、３．４．２項において述べた）。制御の移動の原因としては、同期式入出力、同期式プロセス間通信、仮想記憶システムではページ・フォールト等に関する仮想プロセッサの停止／再開、および、制御の移動を要求するカーネル・コール（本システムでは、３．６．１項述べた vp_switch()）の発行があげられる。

ここでは、それぞれのスケジューラにおける方針（policy）とは独立した構成法を示す。ここで述べる構成法は、プロセスと仮想プロセッサという概念が存在するなら

ば、他の方針においても利用可能である。

３．１０．３．２　プロセス・スケジューラと仮想プロセッサ・スケジューラの内部構造

カーネル内の軽量プロセスとしての仮想プロセッサ

オペレーティング・システムのカーネルは、利用者からのトラップを扱う視点から、次の２つの方式に分類される［１７］。

（１）**プロセス・モデル：**利用者プロセスと１対１に対応し、固有のスタックを持つプロセスをカーネル内に設ける。後述する割込みモデルと比較して、実現が容易である。Ｍａｃｈ２．５システム［２］［６７］、ＵＮＩＸシステム［４］［４６］において採用されている。

（２）**割込みモデル：**カーネルは、利用者からのトラップを割込みとして扱う。カーネル内には、プロセッサごとにスタックを設ける。前述のプロセス・モデルと比較して、効率的な実現が可能である。Ｍａｃｈ３．０システム［１７］、Ｖシステム［１２］等で採用されている。

本システムは、（１）を採用する。その理由は、仮想プロセッサをカーネル内の軽量プロセスとして実現することで、仮想プロセッサとプロセスのスケジューラをカーネル内の軽量プロセスのスケジューラとして実現可能であるからである。さらに、オペレーティング・システムのカーネルを１つの並列プログラムと考えた場合、（利用者レベルの抽象である）仮想プロセッサは、並列処理の単位となり得るため、軽量プロセスにより実現することが自然である。（カーネル中には、仮想プロセッサ以外にも軽量プロセスが存在する。）

カーネル内の軽量プロセスのグループとしての利用者プロセス

実プロセッサの制御という視点では、プロセスは、仮想プロセッサを実現している軽量プロセスのグループとしてとらえることができる。たとえば、プロセスを削除することは、そのグループに属するすべての（仮想プロセッサを実現している）軽量プロセスを削除することに相当する。プロセスを停止させることは、そのグループに属する実行中の（仮想プロセッサを実現している）軽量プロセスから、実プロセッサを奪いとることに相当する。

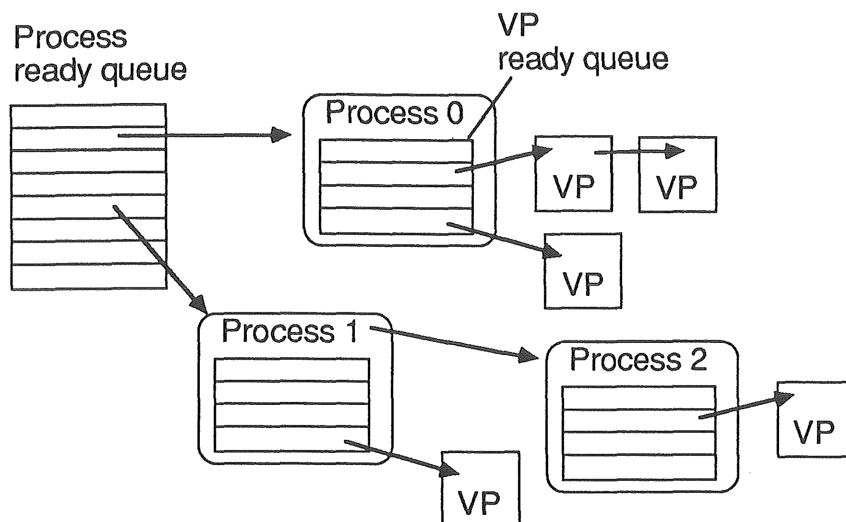


図３－１９　２段階のレディ・キュー
Figure 3-19: Two-level ready queues.

２段階のレディ・キューを持つスケジューラ

上で述べたように、仮想プロセッサは、カーネル内の軽量プロセスであり、プロセスは、カーネル内の軽量プロセスのグループである。この構造に従い、図３－１９に示すような２段階のレディ・キューを用いてスケジューラを構築することができる。第１段は、（実行可能な仮想プロセッサを保持している）プロセスを保持するレディ・キューである。これを操作する部分が、３．４．２節で述べたプロセス・スケジューラに相当する。第２段は、各プロセス内部の実行可能な仮想プロセッサを保持するレディ・キューである。これを操作する部分が、３．４．２節で述べた仮想プロセッサ・スケジューラに相当する。

図３－１９は、プロセスのレベルにおいて８つ、仮想プロセッサのレベルにおいて４つの優先順位に分割されたレディ・キューを示している。実プロセッサは、プロセス・レディ・キューからプロセスを選ぶ。そしてそのプロセス中の仮想プロセッサ・レディ・キューから実行可能な仮想プロセッサを得て、それを実行する。

プロセスの操作

このような構造を持つスケジューラでは、プロセスの操作は、以下のように実現される。

（１）停止：停止するプロセスをプロセス・レディ・キューから削除する。そのプロセスの実行中状態の仮想プロセッサについて、それを実行している実プロセッサに割込みを行う。割込まれた実プロセッサは、実行していた仮想プロセッサを、そのプロセスの仮想プロセッサ・レディ・キューに入れる。実プロセッサは、他のプロセスの仮想プロセッサに制御を移す。

（２）実行再開：実行を再開するプロセスを単にプロセス・レディ・キューに接続

するだけでよい。

（３）生成：アドレス空間や最初の仮想プロセッサを生成した後、（２）を行う。

（４）終了、および、削除：（１）の操作を行った後、プロセスが保持している資源を開放し、終了コードを保存する。

このように、プロセスの操作は、プロセス・レディ・キューへの操作により容易に実現される。

３．１１　実現

３．３節から３．１０節において述べたマイクロプロセス／仮想プロセッサ方式に基づき、軽量プロセス機能を実現した。この節では、はじめに、裸の共有メモリ型マルチプロセッサにおける仮想プロセッサを提供するカーネルの実現について述べる。次に、カーネル外のマイクロプロセス・ライブラリの実現について述べる。最後に、既存のシステムにおける仮想プロセッサ機能のエミュレーションについて述べる。

３．１１．１　仮想プロセッサを実現するカーネルの実現

３．１１．１．１　実現環境

３．１０節で述べた構成法に基づき、共有メモリ型マルチプロセッサＬｕｎａ８８ｋ［３２］［１０５］において仮想プロセッサを提供するカーネルの実現を行った。実現に用いたＬｕｎａ８８ｋは、４プロセッサ構成で、主記憶容量は、３２Ｍバイトである。各プロセッサは、命令とデータのそれぞれに１６ｋバイトのキャッシュを供えている。Ｌｕｎａ８８ｋに付属のオペレーティング・システムであるＵｎｉＯＳ－Ｍａｃｈ１．１０（Ｍａｃｈ２．５相当）を開発システムとして利用した。

３．１１．１．２　モジュール構成

カーネルの開発には、Ｃ言語を用いた。全体の大きさは、約２万１０００行である。モジュール構成を、以下に示す。

メモリ管理

bitmap	メモリ管理用ビットマップ操作モジュール	５８５行
mo	メモリ・オブジェクト管理	４１１行

ハードウェア操作

cmmu	CMMU M88200操作モジュール	２５１５行
dev	デバイス管理	１７８０行

fp	浮動小数点用カーネル内ルーチン	２５９０行
m88k	MPU M88100関連手続き	１４７７行
env	コンテキスト切り替え	２１９行

割込み受付け

locore	割込み受付けモジュール	２７１５行
main	カーネル・コール受付け	２６４０行

プロセスと仮想プロセッサの実現

mp	カーネル内マイクロプロセス	７８６行
vp	仮想プロセッサ操作	１１１０行
ps	プロセス・オブジェクト操作	１２６６行
rp	実プロセッサ操作（スケジューラ）	８７８行

カーネル・サーバ呼び出し受付け部

con	コンソール・ファイル・サーバ	１５８行
time	時間サーバ	８４行
mem	メモリ・サーバ	５１行
stdps	プロセス・サーバ	５４０行

その他

misc	その他ユーティリティ手続き	１５８８行

合計　２０９０７行

これらのモジュールの中で、浮動小数点用カーネル内ルーチン（fp）と割込み受付けモジュール（locore）については、オムロン社より提供されたプログラムを変更して利用した。デバイス管理については、オムロン社より提供されたプログラムを参考にして開発を行った。

３．１１．１．３　カーネル利用者用ライブラリ

カーネル利用者用ライブラリとは、上で述べたＬｕｎａ８８ｋ上のＲｅＳＣカーネルを利用してプログラムを開発するためにＣ言語から利用可能な関数を集めたものである。これには、上記のカーネルを呼び出すためにトラップ命令を含むものと、一般のＣ言語用のライブラリが含まれる。

services	インタフェース記述	２１６行

clients	利用者用ライブラリ	６９６行
合計		９１２行

上記の他に、カーネルと共通のルーチンがある。さらに、開発システム上のＣ言語のライブラリ関数から、ＵＮＩＸのシステム・コールを含まないものを抜出して利用している。このような関数の例としては、文字列操作ライブラリ関数群（string(3)）があげられる。

３．１１．１．４　考察

３．１０．１項、および、３．１０．２項において述べた仮想プロセッサと実プロセッサの固有領域の実現においては、ハードウェア（ＭＭＵ）を操作する部分を除き、その大部分を開発システム上で利用者プログラムとしてデバッグと動作の確認を行うことができた。この時、ハードウェアを操作する手続きについては、ハードウェアのレジスタに相当するダミーの変数を設け、その内容を確認することにより、動作の確認を行った。また、一部の手続きについては、ハードウェアのエミュレートを行うことにより、デバッグを行った。エミュレートを行った手続きの例きとしては、ＣＭＭＵのアドレス変換操作や文字入出力があげられる。

３．１０．３項において述べたスケジューラについても、割込み制御の部分を除き、その大部分を開発システム上でデバッグと動作の確認を行うことができた。これらの実現を通じて、カーネルを利用者プログラムと同様に開発することが可能であることを確認することができた。

今回の実現には、ディスク入出力を行うファイル・サーバの機能が含まれていない。このファイル・サーバの機能は、２．１０．４項で述べたように、ＮＦＳサーバ［５９］中のファイル・サーバの機能からキャッシングの機能を取り外したようなものになる。（ＮＦＳサーバは、他に名前サーバとしての機能を持つ。）したがって、ＮＦＳサーバの一部を抜き出すことにより、ファイル・サーバを開発することが可能であると思われる。

３．１１．２　マイクロプロセス・ライブラリの実現

３．１１．２．１　実現環境

３．５章で述べたマイクロプロセス・ライブラリを、５種類のソフトウェア環境（前項で述べた本ＲｅＳＣシステムのカーネルを含む）と４種類のハードウェア・アーキテクチャにおいて実現を行った。すなわち、５種類の第０層と４種類の第１層を開発した。第２層以上は、これらのソフトウェア環境とハードウェア・アーキテクチャから独立している。

第０層の実現を行ったソフトウェア環境を以下に示す。

（１）ＲｅＳＣカーネル（本システム）
（２）Ｄｙｎｉｘ（Ｓｅｑｕｅｎｔ　Ｂａｌａｎｃｅ　８０００）
（３）ＳｕｎＯＳ（Ｓｕｎ３，Ｓｕｎ４）
（４）Ｍａｃｈ２．５（Ｌｕｎａ８８ｋ，ＮｅＸＴ）
（５）ＮｅｗｓＯＳ（Ｎｅｗｓ）

（１）については、３．１１．１で述べたように、トラップ命令を含む手続きをアセンブリ言語により記述した。その他については、ＵＮＩＸの機能を利用してＲｅＳＣカーネルの機能のエミュレーションを行った。エミュレーションの詳細については、３．１１．３項において述べる。

第１層の実現を行ったハードウェア・アーキテクチャを以下に示す。

（１）ＭＣ８８１００（Ｌｕｎａ８８ｋ）
（２）ＮＳ３２０３２（Ｓｅｑｕｅｎｔ　Ｂａｌａｎｃｅ　８０００）
（３）ＭＣ６８０００（Ｓｕｎ３，ＮｅＸＴ，Ｎｅｗｓ）
（４）ＳＰＡＲＣ（Ｓｕｎ４）

３．１１．２．２　モジュール構成

マイクロプロセス・ライブラリは、３．５章において述べたように、４つの層から構成されている。それぞれの層におけるモジュールの名前、機能、および、大きさは、以下のようになっている。

第０層

resc	カーネル利用者ライブラリ	（３．１１．１項）
	ＲｅＳＣカーネルのエミュレーション	１１００行
mem	メモリ管理	３７６４行

第１層

env	コンテキスト切り替えルーチン	８６８行
lock	スピンロック・ルーチン	４１８行

第２層

main	生成消滅と同期、初期化	１６６０行
trace	実行の軌跡の記録と解析	１０２５行
misc	その他ユーティリティ手続き	６９３行

第３層

mon	モニタ（１）	２２４行
mutex	モニタ（２）	２１８行
sem	セマフォ	１２２行
msg	ランデブ	３０６行
pipe	ストリーム	８７７行

合計

合計	１１２７５行

■第０層

カーネル利用者ライブラリについては、３．１１．１．３において述べた。ＲｅＳＣカーネルのエミュレーションに関しては、１１．３節において述べる。メモリ管理（mem）は、仮想プロセッサの共有領域の実現と汎用メモリ管理ルーチン（ＵＮＩＸにおけるＣ言語のライブラリ関数 malloc(3)、free(3) に対応）を含んでいる。なお、後者については、ＧＮＵ　Ｅｍａｃｓに含まれていたものにスピンロックによる相互排除機能を付加することにより、実現を行った。

■第１層

上で示したモジュールの行数は、３．１１．２．１において述べた４種類のアーキテクチャの合計である。コンテキスト切り替えについては、アセンブリ言語により記述を行った。主に、レジスタの退避、スタックの切替え、レジスタの回復から構成されている。スピンロックについては、原子的な命令を用いて実現を行っている。ただし、図３－９に示したものとは異なり、ロックの解除を待つ部分を、通常のロード命令を用いて実現している。このため、キャッシュの効果により、共有バスへのトラヒックが抑えられる。

■第２層

mainが、第２層の主要部である。trace は、３．５．３項で述べたマイクロプロセスの実行の軌跡の記録と解析を行うモジュールである。最後のmiscには、第２層を動作させるためのユーティリティが含まれている。たとえば、キューの操作、メッセージの表示、シンボル・テーブルの解析、環境変数の解析を行う関数が含まれている。

■第３層

ここで、モニタ（１）は、高レベルの、モニタ（２）は、低レベルの機能を使ったモニタ同期プリミティブの実現である。前者は、第２層において提供するデッドロックを検出する機能が有効であるのに対して、後者は無効である。また、前者は、

ＳｕｎＬＷＰにおけるモニタと、後者は、Ｃ－Ｔｈｒｅａｄｓにおけるｍｕｔｅｘと似た内部構造を持つと思われる。

３．１１．２．３　考察

この実現においては、層化プログラミング（layered programming）の技術により、移植性と可読性が高いライブラリを構築することができた。カーネルに依存する部分は、第０層に、そしてプロセッサのアーキテクチャに依存する部分は、第１層に隠蔽されている。第３層の各同期通信プリミティブの構築が非常に簡単になっていることから、第２層において提供した機能の妥当性を確認することができた。

３．１１．３　ＵＮＩＸにおける仮想プロセッサのエミュレーション

本ＲｅＳＣシステム以外においてもマイクロプロセス・ライブラリを利用するために、ＵＮＩＸのプロセスを用いて仮想プロセッサのエミュレーションを行った。実現を行ったのは、Ｄｙｎｉｘシステム［６］、Ｍａｃｈ２．５システム［１０５］［５８］、ＳｕｎＯＳ［９３］である。

基本的には、ＵＮＩＸのプロセスを仮想プロセッサとして利用し、そしてマイクロプロセス用のメモリを、ＵＮＩＸのプロセス間共有メモリにより実現する方法を用いた。これは、文献［１０６］で用いられている方法と同じ方法である。

３．１１．３．１　カーネル・コール

vp_allocate() カーネル・コールを、ＵＮＩＸの fork() システム・コールにより実現した。vp_sleep() と vp_switch() カーネル・コールを、ＵＮＩＸの usleep() ライブラリ関数により実現した。（usleep()は、ソフトウェア割込みにより実現されている。）

３．１１．２．２　仮想プロセッサの固有領域と共有領域

Ｄｙｎｉｘでは、Ｃ言語により提供されている private/shared キーワード、および、shm_alloc() ライブラリ関数を用いて仮想プロセッサの固有領域と共有領域を実現した。内部的には、それらは、ＵＮＩＸの mmap() システム・コールをインタフェースとして実現されている。

Ｍａｃｈシステム、および、ＳｕｎＯＳでは、固有領域を通常のＵＮＩＸのデータ・セグメントにより実現した。共有領域については、それぞれ vm_alocate() カーネル、および、mmap() システム・コールを用いて、プロセス間共有メモリを実現し、その上に共有領域を割り当てた。共有領域上の静的変数については、データ・セグメントからある定数のオフセットを加えた領域にプロセス間共有メモリを割り当て、その部分を参照することで実現した。

３．１１．２．３　資源割当て

プロセスを用いて仮想プロセッサを実現した場合、もっとも大きな問題は、資源の割当てである。たとえば、ある１つの仮想プロセッサにおいて、ファイルを開いたとしても、他の仮想プロセッサには伝わらない。ファイルを閉じる操作についても同様である。この問題は、エミュレーションに起因するものである。本ＲｅＳＣシステムでは、カーネルにおいて資源割当ての単位としてのプロセスと実プロセッサのエントリとしての仮想プロセッサを実現しているため、このような問題は生じない。

この問題を解決するために、ＵＮＩＸのシステム・コール sendmsg()、 recvmsg() を用いて、ファイル記述子（資源へのポインタ）を他のプロセスへ転送する方法を用いた。以下に、ファイルを開く操作（openシステム・コール）を例に、手順を説明する。

準備：

仮想プロセッサごとに、双方向の通信が可能なソケットを生成しておく。

ファイルを開く仮想プロセッサ：

（１）ＵＮＩＸの open() システム・コールを発行し、ファイルを開く。エラーが生じた時には、これを返す。

（２）自分以外の仮想プロセッサ（を実現しているプロセス）のソケットへ sendmsg() システム・コールを利用して、ファイル記述子を送る。

（３）自分以外の仮想プロセッサ（を実現しているプロセス）へ、kill() システム・コールを用いて、ソフトウェア割込みを行う。

（４）recvmsg() システム・コールを用いて、自分以外の仮想プロセッサからの応答を待つ。

それ以外の仮想プロセッサ：

（１）それ以外の仮想プロセッサ（を実現しているプロセス）は、ソフトウェア割込みを受け付けると、自分自身のソケットから recvmsg() システム・コールを用いて、メッセージを受け取る。これにより、ファイル記述子が渡されることになる。

（２）ソフトウェア割込みを行った仮想プロセッサに対して、sendmsg() システム・コールにより、応答を送る。

以上は、ファイルを開く操作の手順であった。その他に、システム・コール creat(), pipe(), dup(), dup2(), socket() が、ほぼ同様の手順により実現される。システム・コール close() の場合、ファイル記述子ではなく、閉じるべき番号を整数として送り、ソフトウェア割込みを受付けた仮想プロセッサにおいて、close() システム・コールを発行する。

３．１１．２．４　限界

ＵＮＩＸのシステム・コールを利用したＲｅＳＣカーネルのエミュレーションには、限界がある。第１に、最も深刻な問題は、プロセス全体の操作である。たとえば、プロセスを終了させる場合、仮想プロセッサを構成している全てのＵＮＩＸのプロセスを終了させる必要がある。この時、終了させることに失敗したり、あるいは、個々のＵＮＩＸのプロセスが勝手に終了する危険性がある。エミュレーションを行なっている限り、この問題を解決することは、不可能である。

第２に、メモリ資源割当ての問題がある。１度並列処理を開始した後（fork() システム・コールを発行して、複数の仮想プロセッサを生成した後）では、利用するメモリ資源の量を拡張することができない。この問題は、仮想記憶を利用することで、ある程度緩和される。

第３に、Ｍａｃｈシステム、および、ＳｕｎＯＳでは、仮想プロセッサ間の共有領域に対応するコンパイラ、アセンブラ、および、リンケージ・エディタが存在しない。したがって、こららのシステムでは、メモリの無駄が生じている。

最後に、マイクロプロセス・スケジューラが働かなくなるという問題がある。マイクロプロセス・スケジューラは、特定のマイクロプロセスに多くの資源を与えることで、その並列応用プログラム全体の実行時間の短縮を試みる。一方、カーネルは、仮想プロセッサを実現しているプロセスを公平に実行しようとする。このことは、１つの並列応用プログラムの実行において、利用者レベルとカーネル・レベルの２つのスケジューラが動作していることを意味する。その結果、利用者レベルのマイクロプロセス・スケジューラが働かなくなる。

３．１２　性能

この節では、実現した軽量プロセス機能の性能について述べる。

３．１２．１　実験環境

実現したマイクロプロセス、および、仮想プロセッサの性能を調べる実験を行った。実験に用いたシステムのプロセッサのアーキテクチャ、既存のオペレーティング・システム、主記憶容量、キャッシュ容量（制御方式）を以下に示す。

共有メモリ型マルチプロセッサ

（１）Ｓｅｑｕｅｎｔ　Ｂａｌａｎｃｅ８０００、
Ｄｙｎｉｘ２．１、
４×ＮＳ３２０３２　１０ＭＨｚ、
８Ｍ　Ｂｙｔｅｓ、
８ｋ　Ｂｙｔｅｓ（ｗｒｉｔｅ－ｔｈｒｏｕｇｈ）

（２）Ｏｍｒｏｎ　Ｌｕｎａ８８ｋ、
Ｍａｃｈ２．５　（ＵｎｉＯＳ－Ｍａｃｈ　１．１０）、
４×Ｍ８８１００　２５ＭＨｚ、
３２Ｍ　Ｂｙｔｅｓ、
３２ｋ　Ｂｙｔｅｓ　（ｃｏｐｙ－ｂａｃｋ）

単一プロセッサ

（３）ＳＰＡＲＣｓｔａｔｉｏｎ　２、
ＳｕｎＯＳ　４．１．１、
ＳＰＡＲＣ　４０ＭＨｚ、
３２Ｍ　Ｂｙｔｅｓ、
６４ｋ　Ｂｙｔｅｓ

（４）Ｓｕｎ３／６０（以下Ｓｕｎ３と略記する）、
ＳｕｎＯＳ　４．０．３、
Ｍ６８０２０　１６ＭＨｚ、
１２Ｍ　Ｂｙｔｅｓ

（５）ＮｅＸＴ（ｃｕｂｅ）、
　　Ｍａｃｈ　２．５、
　　Ｍ６８０３０　２５ＭＨｚ、
　　８Ｍ　Ｂｙｔｅｓ、
　　ｏｎ－ｃｈｉｐ

このように、様々な環境において実験を行うことにより、軽量プロセスの実現方式の間の、プロセッサのアーキテクチャや具体的な実現から独立した比較が可能になる。（１）と（２）は、共に4プロセッサ構成の共有メモリ型マルチプロセッサであるが、プロセッサのアーキテクチャが大きく異なる。（１）は、ＣＩＳＣ（Complex Instruction Set Computer）系のプロセッサを、（２）は、ＲＩＳＣ（Reduced Instruction Set Computer）系のプロセッサを備えている。ＲＩＳＣの場合、ＣＩＳＣと比較してキャッシュが性能に大きく影響すると予想される。

（３）と（４）では、同一のオペレーティング・システムが動作しているが、プロセッサのアーキテクチャが大きく異なる。（３）は、ＲＩＳＣ系のプロセッサを、（４）は、ＣＩＳＣ系のプロセッサを備えている。特に（３）の ＳＰＡＲＣプロセッサは、巨大なレジスタ・ファイルを持っていることから、軽量プロセスのコンテキスト切替えの処理時間が長くなることが予想される。ＳＰＡＲＣプロセッサでは、手続き１個あたり１６個のレジスタを局所変数として利用することができる。コンテキスト切換えにおいては、本来、利用しているレジスタのみを保存／回復すれば十分である。しかしながら、レジスタの利用状況を得る手段がないため、全てのレジスタを保存／回復しなければならない。さらに、レジスタ・ファイルのフラッシュを特権モードで行う必要があるため、コンテキスト切換えにおいて必ずカーネル・コールが必要となる。

（５）と（２）では、同一のオペレーティング・システムが動作しているが、プロセッサのアーキテクチャと数が異なる。

以下の**３．１２．３項**、**３．１２．４項**、**３．１２．５項**、および、**３．１２．７項**の実験では、既存のシステム上に実現したマイクロプロセス・ライブラリを用いて実行時間を測定した。**３．１２．６項**の実験では、裸の計算機上に実現したカーネルの性能を測定した。

以下の実験においては、実行時間を、ＵＮＩＸのシステム・コール gettimeofdate()、あるいは、ＲｅＳＣカーネルの同等の機能を用いて測定した。実行時間の精度が異なるのは、システムごとに測定可能な最小時間が異なること、および、繰り返し回数が異なることによる。既存のシステムでは、マルチユーザ・モード（いくつかのサーバ・プロセス（ＵＮＩＸでは、デーモン（daemon））が生成されている状態）において、システムの負荷が低い状態で実験を行った。Ｌｕｎａ８８ｋ上のＲｅＳＣカーネルでは、測定対象となるプロセスのみを生成して実験を行った。

既存のシステム上での実験では、あらかじめメモリを参照することで、測定時間に

仮想記憶の処理時間が含まれないようにした。（Ｌｕｎａ８８ｋ上のＲｅＳＣカーネルは、実記憶システムであり、仮想記憶の影響はない。）さらに、プロセス生成・消滅の実験では、仮想記憶の影響により実行時間が長くならない範囲内において、生成する軽量プロセスの数を大きくした。以下で示す実験結果では、１回あたりの実行時間に加えて、反復の回数を示す。これらの結果は、並列処理に用いることが可能な軽量プロセス数や同期処理の頻度の目安を与える。本方式では、軽量プロセスの数の上限は、利用者プロセスのアドレス空間の大きさによってのみ制約される。カーネル制御方式では、しばしばカーネル内に静的に確保された表の大きさにより、制約を受ける。

３．１２．２　比較対象

以下の３．１２．３項、３．１２．４項、および、３．１２．５項では、マイクロプロセスの基本的な性能を論じる。比較対象としては、カーネル制御方式のＭａｃｈのＣ－Ｔｈｒｅａｄｓ［１０５］［５８］、コルーチン方式のＳｕｎＯＳのＳｕｎＬＷＰ［９３］、および、重量プロセスのＵＮＩＸのプロセス［４６］を取り上げる。

Ｃ－Ｔｈｒｅａｄｓは、Ｌｕｎａ８８ｋ、および、ＮｅＸＴシステムで利用可能である。ＳｕｎＬＷＰは、ＳＰＡＲＣｓｔａｔｉｏｎ　２、および、Ｓｕｎ３で利用可能である。Ｓｕｎ３の実行形式の一部は、ＮｅＸＴの atom コマンドを利用することにより、ＮｅＸＴシステム用の実行形式へ変換することが可能である。この機能を利用して、ＳｕｎＬＷＰをＮｅＸＴシステム上でも動作させ、実行時間を測定した。なお、Ｂａｌａｎｃｅ　８０００では、他の軽量プロセスを利用することができなかったので、ＵＮＩＸの重量プロセスとの比較だけを行った。

■Ｃ－Ｔｈｒｅａｄｓ

Ｃ－Ｔｈｒｅａｄｓでは、軽量プロセスの生成・消滅、および、コンテキスト切替えには、カーネル・コールが必要である。コンテキスト切替えを伴わない同期は、カーネル・コールを用いることなく実現される。

Ｃ－Ｔｈｒｅａｄｓでは、軽量プロセスのキャッシングを行っている。すなわち、終了した軽量プロセスを破壊せずにそのまま保存しておき、次に生成要求を受け付けた時にそれを再利用している。

この実験において用いたＣ－Ｔｈｒｅａｄｓは、Ｍａｃｈシステムの第２．５版に付属のものであり、Ｃ言語レベルのＣスレッドとカーネル・レベルのスレッドの間には、１対１の対応関係が存在する。第３版に付属のものでは、両者の間に１対１の対応関係が存在しない。

■ＳｕｎＬＷＰ

ＳｕｎＬＷＰは、ＳｕｎＯＳ４．ｘに付属している軽量プロセス・ライブラリであ

る。これは、コルーチン方式により実現されている。軽量プロセスのスタックをレッド・ゾーン方式により保護する機能がある。この機能は、ＳｕｎＯＳ固有のシステム・コールｍｐｒｏｔｅｃｔ（）により実現されている。ＳｕｎＬＷＰでは、そのスタックのキャッシングを行う機能がある。

ＳＰＡＲＣｓｔａｔｉｏｎ　２では、コンテキスト切換えにおいて、レジスタ・ウインドウをフラッシュするために、カーネル・コールを発行する。本方式においても、同一のカーネル・コールを利用した。

■ＵＮＩＸのプロセス

従来の重量プロセスとの性能を比較するために、ＵＮＩＸのプロセスを利用した。プロセスの生成のプリミティブとしては、ＵＮＩＸの fork() システム・コールを利用した。プロセス間の同期のプリミティブとしては、セマフォ関係のシステム・コールを利用した。なお、Ｌｕｎａ８８ｋとＮｅＸＴでは、セマフォが利用できなかったので、比較を行わなかった。

３．１２．３　プロセスの生成・消滅

■性能測定用プログラム

２重ループ構造をもつプログラムを用いて、プロセスの生成・消滅操作の性能を測定した。内側のループでは、プロセスの入れ子を作り、共有メモリ型マルチプロセッサにおける並列処理の効果、および、軽量プロセスやスタックのキャッシングの効果を抑えている。逆に、外側のループの実行回数を変化させることで、キャッシングの効果を調べることができる。このプログラムを用いて、プロセスの生成処理、コンテキスト切替え（親から子へ、子から親へ）、同期処理（子の終了待ち）、消滅処理の全体の処理時間を測定した。付録Ｄ－１に、使用した性能測定用のプログラムの骨格を示す。

表３－１（ａ）と表３－１（ｂ）に、実験の結果を示す。これらの表で、上段が実行時間、下段が生成したプロセスの数を表す。下段の数字が

$$m \times n$$

とは、外側のループをｍ回、内側のループをｎ回実行したことを意味する。

表３－１（ｂ）における実行時間を、次の式により計算した。

$$\frac{(m+1)\times n\text{回の実行時間} - 1\times n\text{回の実行時間}}{m\times n}$$

表３－１　プロセスの生成・消滅の実行時間の比較
上段：プロセス１個当りの実行時間（単位：ミリ秒）
下段：繰返し回数（単位：回）

Table 3-1: Comparisons of execution times of process creation and termination.
Upper: Execution times of process creation and termination. (In milliseconds)
Lower: The numbers of iteratins.

表３－１（ａ）　１回目
Table 3-1(a): In the first creation of processes.

	SPARCstation 2	Sun3	NeXT	Luna88k	Balance 8000
Microprocess (Layer 2)	0.149 (1x1k)	0.400 (1x500)	0.444 (1x500)	0.19 (1x1k)	1.5 (1x100)
SunLWP (mprot)	0.97 (1x100)	3.2 (1x100)	— —	— —	— —
SunLWP (none)	0.43 (1x100)	1.0 (1x100)	1.03 (1x100)	— —	— —
C-Thread	— —	— —	10.0 (1x10)	10.5 (1x20)	— —
UNIX (heavy-weight)	6.60 (1x50)	16. (1x20)	27.7 (1x20)	16. (1x20)	38. (1x10)

■他の軽量プロセスとの比較

表３－１（ａ）からわかるように、スタックの保護を行ったSunLWP(mprot)と比較すると、本実現（Microprocess）が１０倍から２０倍高速になっている。主な原因は、SunLWPでは、レッド・ゾーン方式によりスタックの保護を行っていることにある。SunLWP において、スタックの保護機能を外したとしても（表３－１（ｂ）の SunLWP(none)）、本実現は、約３倍高速になっている。この差は、軽量プロセスの実現方式の違いによるものではなく、両実現の機能の違いによる。たとえば、SunLWP では、ＵＮＩＸ固有のソフトウェア割込みや例外を扱う機能があり、本マイクロプロセスよりも複雑になっている。軽量プロセスの実現方式の違いだけならば、３．４．１項で述べたように、コルーチン方式の SunLWP の方の効率がよくなる。

表３－１　プロセスの生成・消滅の実行時間の比較
上段：プロセス１個当りの実行時間（単位：ミリ秒）
下段：繰返し回数（単位：回）

Table 3-1: Comparisons of execution times of process creation and termination.
Upper: Execution times of process creation and termination. (In milliseconds)
Lower: The numbers of iteratins.

表３－１（ｂ）　２回目以降
Table 3-1(b): In the second creation of processes.

	SPARCstation 2	Sun3	NeXT	Luna88k	Balance 8000
Microprocess (Layer 2)	0.149 6 ([11-1]x1k)	0.384 ([11-1]x500)	0.362 ([11-1]x500)	0.130 ([11-1]x1k)	1.49 ([11-1]x100)
SunLWP (mprot)	0.443 ([11-1]x100)	1.46 ([11-1]x100)	── ──	── ──	── ──
SunLWP (none)	0.339 ([11-1]x100)	0.98 ([11-1]x100)	1.01 ([11-1]x100)	── ──	── ──
C-Thread	── ──	── ──	2.37 ([11-1]x10)	1.9 ([11-1]x20)	── ──
UNIX (heavy-weight)	6.56 ([11-1]x50)	16.2 ([11-1]x20)	28.11 ([11-1]x20)	16.25 ([11-1]x20)	38. ([11-1]x10)

C-Threads と比較すると、キャッシングが有効でない場合（表３－１（ａ））、本実現が約２０から５０倍高速になっている。この差は、軽量プロセスの実現方式の違いによる。本方式では、カーネル・コールのオーバヘッドがなく、必要となるデータ構造を全て利用者空間に置くことができるので、高速になっている。

また、表３－１（ａ）と表３－１（ｂ）を比較すると、C-Threads、および、SunLWP（mprot）において、２回目以降の実行時間が１回目と比較して著しく短縮していることがわかる。これは、C-Threads では、軽量プロセスのキャッシング、SunLWP（mprot） では、スタックのキャッシングが有効に働いているからである。本実現では、軽量プロセスやスタックのキャッシングを行っていないが、メモリのキャッシュの効果により、処理時間が短くなっている。

■ＵＮＩＸのプロセスとの比較

表３－１において、マイクロプロセスとＵＮＩＸの重量プロセス（UNIX heavy-weight）を比較すると、マイクロプロセスが２５倍から８０倍高速であることがわかる。表３－１（ａ）と（ｂ）を比較すると、ＵＮＩＸでは、プロセスのキャッシングが行われていないことがわかる。

３．１２．４　コンテキスト切替えを伴わないプロセス間の同期

■性能測定用プログラム

コンテキスト切り替えを伴わないプロセス間の同期操作の性能を、次の様な同期プリミティブを繰り返し実行することにより測定した。

（１）（スピン）ロックのロック操作／アンロック操作
　　本マイクロプロセス・ライブラリの第１層（図３－３）で利用可能である。

（２）セマフォのＰ命令／Ｖ命令
　　本マイクロプロセス・ライブラリの第３層（図３－３）、
　　および、ＵＮＩＸで利用可能である。

（３）モニタの入口操作／出口操作
　　本マイクロプロセス・ライブラリの第３層（図３－３）、ＳｕｎＬＷＰ、
　　および、Ｃ－Ｔｈｒｅａｄｓで利用可能である。

実験では、１つのプロセスにおいて、これらの操作を繰り返し行い、実行時間を測定した。実験の結果を、表３－２に示す。これらの表の上段は、同期操作１回当たりの実行時間、下段は、その実行時間を測定した時に行った操作の回数である。ここでは、ロックとアンロック、Ｐ命令とＶ命令、入口操作、出口操作を合せて１回の操作と数える。

表３－２において、mon(1)とmon(2)は、共にモニタの実現であるが、機能と実現が異なる。前者では、第２層で提供されるデッドロックを検出する機能が有効であるのに対して、後者では無効である。それは、後者では、スピンロックにより、モニタ入口手続きを実現しており、スピンロックが保持できなかった場合、待ち状態にするのではなく、mp_yield() により自分自身を実行可能状態に置いたまま、他のマイクロプロセスへ制御を移しているからである。このような実現は、単一プロセッサでは、有効である。また、前者は、内部的な実現が SunLWP のものに近いと思われる。後者は、C-Threadsのものに近くなっている。

なお、第２層には、コンテキスト切替えを伴わないプロセス間の同期を実現するプリミティブが存在しないので、実行時間を測定していない。この層では、必要ならば、第１層のスピンロックがそのまま利用される。

測定に用いたプログラムの骨格を、付録Ｄ－２に示す。

表３－２　コンテキスト切替えを伴わないプロセス間の同期の実行時間の比較
上段：同期操作の実行時間（単位：μ秒）．
下段：繰返し回数（単位：回）．

Table 3-2: Comparisons of execution times of inter-process synchronization without context switches.
Upper: Execution times of synchronization operations.(In microseconds)
Lower: The numbers of iteratins.

	SPARCstation 2	Sun3	NeXT	Luna88k	Balance 8000
Microprocess Layer 3/mon(1)	27.1 (100k)	31.2 (100k)	29.9 (10k)	8.3 (100k)	232. (10k)
Microprocess Layer 3/mon(2)	0.672 (1000k)	6.2 (100k)	5.48 (100k)	2.22 (1000k)	57. (10k)
Microprocess Layer 3/sem	3.13 (100k)	34. (10k)	27.1 (10k)	9.7 (1000k)	150. (10k)
Microprocess Layer 1/spin	0.672 (1000k)	4.8 (100k)	3.92 (100k)	2.05 (1000k)	54. (10k)
SunLWP (monitor)	5.6 (100k)	40.8 (100k)	39.9 (10k)	― ―	― ―
C-Thread (mutex)	― ―	― ―	4.3 (100k)	2.1 (100k)	― ―
UNIX (semaphore)	221. (10k)	1120 (1k)	― ―	― ―	1930. (1k)
UNIX (getpid()x2)	31.84 (100k)	196. (10k)	158. (1k)	68.4 (100k)	530. (10k)

mon: monitor
sem: semaphore
spin: spin lock

■マイクロプロセスの基本的な性能

本マイクロプロセス・ライブラリの各層の機能を比較すると、第１層のスピンロック Layer 1/spin と第３層のモニタ mon(2) の性能がよいことがわかる。セマフォとモニタ mon(1) は、SPARCstation 2 を除いて同程度のオーバヘッドになっている。SPARCstation 2 において、このように大きな差が生じるのは、レジスタ・ファイル

のフラッシュに関係している。すなわち、本ライブラリが層構造を持っているため、層に対応した手続き呼出しのネストが深くなっているからである。

■他の軽量プロセスとの比較

表３－２において、本実現 Microprocess Layer 3/mon(1) と SunLWP(monitor) と比較すると、SPARCstation 2 では、本実現が５倍程度遅くなっているのに対して、逆に、Sun3 と NeXTでは、本方式が２５％程度高速になっている。この差は、軽量プロセスの実現方式の違いによるものではなく、両者の機能と内部実現の違いによる。実現方式としては、相互排除のオーバヘッドが不用なコルーチン方式の方が効率がよくなる。

表３－２において、本実現 Microprocess Layer 3/mon(2) と C-Threads(mutex)を比較すると、ほぼ同程度の性能であることがわかる。これは、３．５．２項で述べたように、カーネル制御方式の C-Threads においても、コンテキスト切替えを伴わないプロセス間の同期においては、カーネル・コールを発行することなく実現することが可能になっているからである。

■ＵＮＩＸのプロセスとの比較

表３－２において、本実現 Microprocess Layer 3/sem と、UNIX semaphore を比較すると、本実現が１０倍から７０倍程度高速になっている。ＵＮＩＸにおける最も軽いシステム・コール getpid() と比較しても、さらに高速になっている。このことは、カーネル・コールを減らす事が高速処理につながることを示している。

３．１２．５　コンテキスト切替えを伴うプロセス間の同期

■性能測定用プログラム

コンテキスト切り替えを伴うプロセス間の同期操作の性能を、次の様な同期プリミティブを繰り返し実行することにより測定する。

（１）２つの私有セマフォを用いる。
　　本マイクロプロセス・ライブラリの第３層、ＵＮＩＸにおいて利用可能である。

（２）１つのモニタと２つの条件変数を用いる。
　　本マイクロプロセス・ライブラリの第３層、Ｃ－Ｔｈｒｅａｄｓ、および、ＳｕｎＬＷＰにおいて利用可能である。

（３）コンテキスト切り替えを行う機能を呼出す。
　　本マイクロプロセス・ライブラリの第１層、第２層、本仮想プロセッサ、Ｃ－Ｔｈｒｅａｄｓ、および、ＳｕｎＬＷＰにおいて利用可能である。

測定に用いたプログラムの骨格を、付録Ｄ－３に示す。

表３－３　コンテキスト切替えを伴うプロセス間の同期の実行時間の比較
上段：同期操作の実行時間（単位：μ秒）．
下段：繰返し回数（単位：回）．

Table 3-3: Comparisons of execution times of inter-process synchronization with context switches.
Upper: Execution times of synchronization operations. (In microseconds)
Lower: The numbers of iteratins.

	SPARCstation 2	Sun3	NeXT	Luna88k	Balance 8000
Microprocess Layer 3/mon(1)	265.8 (10k)	322. (1k)	305. (1k)	66. (10k)	1660. (1k)
Microprocess Layer 3/mon(2)	126.4 (10k)	276. (10k)	246.5 (10k)	48. (10k)	1220. (1k)
Microprocess Layer 3/sem	120.9 (10k)	240. (1k)	184. (1k)	63. (10k)	1700. (100)
Microprocess Layer 2	99.2 (10k)	164. (10k)	154. (10k)	31.4 (100k)	740. (1k)
Microprocess Layer 1	42.1 (10k)	27.1 (10k)	25.45 (100k)	7.3 (100k)	182. (10k)
SunLWP (monitor)	80.2 (10k)	236. (1k)	231. (1k)	— —	— —
C-Thread (mutex)	— —	— —	1146. (1k)	170. (1k)	— —
UNIX (semaphore)	646 (1k)	3200. (1k)	— —	— —	7400. (100)

mon: monitor
sem: semaphore

実験では、２つのプロセスを用いて、これらの操作を繰り返し行い、実行時間を測定した。実験の結果を、表３－３に示す。これらの表の上段は、同期操作１回当たりの実行時間、下段は、その実行時間を測定した時に行った操作の回数である。ここでは、コンテキスト切り替え２回（往復）合せて１回の操作と数える。すなわち、（１）では、セマフォ・プリミティブが２つのプロセスで合計４回、（２）では、条件変数

の操作が２つのプロセスで合計４回、（３）では、コンテキスト切り替えプリミティブが２つのプロセスで合計２回実行される。

共有メモリ型マルチプロセッサでは、これらの同期プリミティブを発行したとしても、実際には、コンテキスト切替えが起こらない可能性がある。すなわち、２つのプロセスに２つの実プロセッサが割り当てられ、プロセスとプロセッサの対応関係が変化しない可能性がある。しかしながら、利用した性能測定用プログラムにより、これらの同期プリミティブの本質的なオーバヘッドの測定と実現方式の比較を行うことが可能である。

■マイクロプロセスの基本的な性能

本実現 Microprocess の各層の機能を比較すると、第１層、第２層の性能がよいことがわかる。コンテキスト切替えにおいては、セマフォ sem とモニタ mon(1), mon(2) は、いずれのアーキテクチャにおいても、同程度のオーバヘッドになっている。プロセッサの性能と比較して、SPARCstation 2 におけるコンテキスト切替え操作のオーバヘッドが大きくなっている。この理由は、この操作において手続き呼出しの深度が深く、その結果、レジスタ・ファイルのフラッシュの処理時間が長くなっているからである。

■他の軽量プロセスとの比較

表３－３において、本実現 Microprocess Layer 3/mon(2) と C-Threads を比較すると、本実現が約５倍高速に高速になっていることがわかる。この差は、軽量プロセスの実現方式の違いによる。カーネル制御方式の C-Threads では、コンテキスト切替えが必要となる場合、必ずカーネル・コールを発行しなければならないのに対して、本方式のマイクロプロセスでは、その必要がない。

本実現 Microprocess Layer 3/mon(1) と SunLWP を比較すると、本実現の方が効率が悪くなっている。この差は、軽量プロセス実現方式の違いによる。本方式では、共有メモリ型マルチプロセッサ上の並列処理を行うので、細かい単位で共有変数の排他制御を行っている。

■ＵＮＩＸのプロセスとの比較

表３－３において、本実現 Microprocess Layer 3/sem と UNIX semaphore を比較すると、本実現の方が５倍から１３倍高速になっている。

表３－４　仮想プロセッサの生成・消滅の性能（単位：ｍ秒）

Table 3-4: Performance of virtual processor creation. (In milliseconds)

	Luna88k
ReSC Process	15.
Virtual Processor	7.7
Microprocess (Layer 2)	0.19
UNIX Process	16.
Mach C-Thread	10.5

表３－５　仮想プロセッサのコンテキスト切替えの性能（単位：μ秒）

Table 3-5: Performance of virtual processor creation. (In microseconds)

	Luna88k
ReSC Process	121.
Virtual Processor	68.
Microprocess (Layer 2)	31.
UNIX Process	
Mach C-Thread	170.

３．１２．６　仮想プロセッサの性能

実現したシステムの性能を、表３－４と表３－５に示す。実現システムのプロセスと仮想プロセッサ（Virtual Processor）は、開発システムとして用いた Mach2.5 システムのプロセス（UNIX prorocess）、および、Mach C-Threads と同程度の性能が得られることが確認された。

利用者レベルの軽量プロセスであるマイクロプロセスと比較すると、仮想プロセッサは、生成において４０倍、コンテキスト切替えにおいて２倍程、実行時間が大きくなっている。これは、カーネル・コールに伴うトラップ処理とパラメタ・チェックのオーバヘッドによる。

３．１２．７　応用固有のスケジューラの性能

３．８．３項と３．９．２項において述べたデータベースの並列処理システムＳＭＡＳＨの応用固有のスケジューラを記述した。さらに、そのスケジューラの効果

を調べる実験を行った。

実験に用いた問い合せの構造を、図３－２０に示す。このように、８個の軽量プロセスより構成されている。

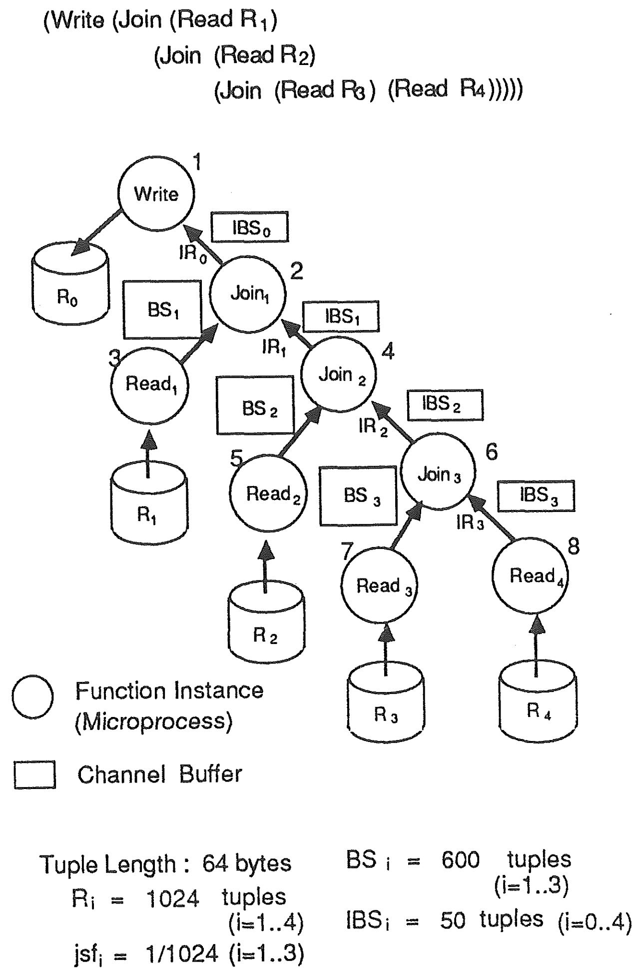


図３－２０　実験に用いた関係データベースへの問い合せの構造

Figure 3-20: The structure of a query for relational databases.

表３－６に実験結果を示す。表の結果より、優先順位を用いないスケジューラ（1）と比較して、優先順位を用いるスケジューラである（２）、または、（３）の性能がよいことがわかる。特に、利用可能なプロセッサの数が少ない場合において、優先順位を用いるスケジューリングの効果が大きいことがわかる。（この実験では、他にプロセスが存在しないので、仮想プロセッサの数と利用可能な実プロセッサの数は、一致している。）これは、動的に優先順位を決定することにより、無駄な計算を避けることができたからである。

以上のことから、本方式において、今回利用した並列応用プログラムについて、応用固有のスケジューラが容易に記述できること、および、そのスケジューラにより効率が改善されることが示された。

表３－６　データベースの並列処理の実行結果（単位：秒）

Table 3-6: Execution times of a parallel application program (database processing). (in seconds)

	Balance 8000			Luna88k		
# of active VP	Scheduler 1	Scheduler 2	Scheduler 3	Scheduler 1	Scheduler 2	Scheduler 3
1	40	38	38	4.5	3.7	3.7
2	22	21	26	3.2	2.9	2.7
3	18	16	19	3.0	3.0	2.9

３．１３　関連した研究

３．１３．１　他の軽量プロセス実現方式との比較

他の実現方式との比較について、３．４．１項では、その概略を述べた。
３．５．２項では、セマフォ同期プリミティブの実現を通じて、比較を行った。
３．１２．４項、３．１２．５項、３．１２．６項では、性能面の比較を行った。この項では、簡単にまとめを行う。また、これらの項で触れなかった面についても、比較を行う。

■カーネル制御方式との比較

カーネル制御方式と比較して、本方式の利点は、効率的な実現が可能である点にある。この理由は、本方式では、軽量プロセスをサブルーチンと同等に扱い、保護の機能を設けていないことによる。保護とは、軽量プロセス操作を実現している手続き、または、カーネル・コールの入口において、パラメタのチェックを行うことで、不正な操作から利用者プロセスが破壊されないようにすることである。カーネル制御方式では、パラメタのチェックを省略することができないため、結果として同一プロセス内において保護が行われていることに相当する。また、本方式の方が、軽量プロセスの数に関する制約が緩いという利点がある。本方式では、メモリ容量の許す限り多くの軽量プロセスを生成することができる。

カーネル制御方式の利点は、横取りを行うスケジューリング、および、実時間性を保証するスケジューリングの実現が容易である点にある。また、複数のプロセスから構成される応用プログラムを記述する場合も、カーネル制御方式が有利である。それは、カーネルが別のプロセスの中に直接、軽量プロセスを生成する機能を提供することができるからである。本方式では、プロセス間通信を行い、間接的に軽量プロセスを生成することになる。

複数のプロセスを用いて処理を行う場合、カーネル制御方式の方が適している。これは、カーネルにおいて軽量プロセスの優先順位を一元的に管理することが可能であるからである。本方式の場合、異なるプロセスに属する軽量プロセスの間に論理的な関係を記述することができない。この点は、１つのプロセス（資源割当てと保護の単位）を用いて並列処理を行う場合には、なんの問題にもならない。しかしながら、クライアント・サーバ間の遠隔手続き呼出しの実行のように、複数のプロセス間の論理的な処理単位が相互作用を行う時には、カーネル制御方式の方が有利である。

本方式における仮想プロセッサは、カーネル制御方式の軽量プロセスと類似している。相違点は、本方式では、軽量プロセスが利用者レベルにおいて実現されていることを前提にして、機能が設計されていることである。たとえば、カーネル制御方式においては、各軽量プロセスに固有領域を設ける必要はない。本方式の仮想プロセッサ

では、利用者レベルにおいて軽量プロセスを実現するために、仮想プロセッサに固有領域を設ける必要がある。また、カーネル制御方式の軽量プロセスでは、スケジューリングにおいて優先順位が重要な役割を果たすのに対して、本方式の仮想プロセッサでは、優先順位の概念が存在しない。

プロセッサのアーキテクチャによっては、カーネルにおいて制御を行った方が効率がよい場合がある。たとえばＳＰＡＲＣプロセッサ［９３］［７１］では、レジスタ・ファイルのフラッシュを特権モードで行う必要があるため、コンテキスト切替えにおいて必ずカーネル・コールが必要となる。この場合、カーネルにおいてコンテキスト切替えを制御する方が効率が良いことが予想される。

■コルーチン方式との比較

コルーチン方式と比較して、本方式の利点は、共有メモリ型マルチプロセッサにおいてＣＰＵ処理の並列実行が可能である点にある。コルーチン方式の利点は、実行時のオーバヘッドが本方式よりもさらに小さい点にある。この理由は、コルーチン方式では、動作が完全に逐次的であるため、軽量プロセス間の共有変数の相互排除を行う必要がないからである。

本方式において、相互排除を行っている部分を削除することにより、完全にコルーチンに還元することができる。すなわち、並列処理を行う必要がない場合には、本方式においても、コルーチン方式と同等の性能を得ることが可能である。

コルーチン方式の利点の１つとして、プログラム実行の再現性がある点があげられる。本方式では、３．８．２項で述べたように、この性質を利用して、効率的にマイクロプロセス間の同期・通信プリミティブの開発を行うことができる。

３．１３．２　他の仮想プロセッサ方式を用いる方式との比較

カーネルにおいて仮想プロセッサを提供し、利用者レベルの軽量プロセスを実行する方式は、文献［８８］［８２］［８１］において提案した本方式の他にも提案された［３］［２３］［３０］［５３］。これらの方式と比較して、本方式の特徴を、以下にまとめる。

（１）利用者レベルの軽量プロセスが、層構造を持つライブラリにより提供されている。これにより、各並列プログラムのプログラマは、それぞれの要求に応じてシステムの機能を利用することができる。

（２）本方式では、コルーチンの性質を利用して、軽量プロセス間の同期・通信プリミティブを構築することができる。

（３）本方式では、並列応用プログラムとほとんど同じ技術により、仮想プロセッサを実現するカーネルを構成している。これにより、カーネル上で動作する並列応用プログラムとしてカーネルの一部を実行し、動作の確認とデバッグを行うことができる。

（４）仮想プロセッサと実プロセッサに固有領域がある。これにより、それぞれ並

列応用プログラムとカーネルのプログラムが書きやすくなっている。この固有領域固有領域とカーネル内軽量プロセス・スケジューラによる仮想プロセッサとプロセスのスケジューラを用いる方式は、異なる外部仕様を持つ仮想プロセッサ、あるいは、カーネル・レベルの軽量プロセスを提供するカーネルの実現においても利用することが可能である。たとえば、スケジューラ活動体（scheduler activations）［３］やＭａｃｈのＴｈｒｅａｄｓ［２］と同じ外部仕様をもつものを３．１０節で述べた固有領域や２段階のレディ・キューを持つスケジューラを用いて実現することも可能である。ただしこの場合、カーネルと利用者プロセスの構造の対称性は、失われる。

（５）本方式では、カーネルと利用者プログラムが分散的に動作する。すなわち、カーネルと利用者の間に共有領域がなく、かつ、カーネルと利用者の間の相互作用は、小数のカーネル・コールに限らている。これにより、カーネルと利用者プログラムの双方が書きやすくなっている。しかしながら、共有領域を用いることにより、効率的にカーネルと利用者の間の情報交換を行うことが可能になる局面が存在する。たとえば、カーネルから利用者に対して提供する情報としては、文献［３０］において提案されている横取りの情報や、３．５．３項において述べた時刻の提供などがあげられる。利用者からカーネルへの情報としては、文献［２３］のように、現在（実プロセッサが割り当てられている）仮想プロセッサが実行している軽量プロセスの優先順位があげられる。

３．１３．３　スピンロックとプロセッサの横取りの問題

多重プログラミング・システムにおいて軽量プロセスを使って並列処理を行う場合、スピンロックとプロセッサの横取りの問題を解決する必要がある。この問題とは、スピンロックにより実現された際どい部分（critical section）を実行中に実プロセッサが横取りされることにより生じる問題である。実プロセッサが割り当てられている仮想プロセッサが際どい部分に入り、スピンロックを保持したとする。この時、プロセス・スケジューラ等により、その仮想プロセッサから実プロセッサが横取りされることがある。この時、他の仮想プロセッサがそのスピンロックを保持しようと試みると、大量のＣＰＵ時間がスピンのために浪費されることになる。

この方式への対応方法は、大きく次の３つに分類される。

（１）防止（prevention）

１量子時間には、１つの並列応用プログラムにそのプロセッサ（サイト）に存在する全ての実プロセッサを割り当てる。１つのプロセスでは、実プロセッサが同時に横取りされるため、上記の問題が防止される。

（２）回避（avoidance）

利用者プログラムに対して、カーネルが、カーネル・レベルで横取りが行われたこ

とをソフトウェア割込みや上向き呼出し（upcall）により告知する。利用者プログラムは、カーネルからの情報を元に、上記の問題を回避するようにスケジューリングを行う。たとえば、スケジューラ活動体［３］では、カーネル・レベルにおいて、スケジューラ活動体（一種の仮想プロセッサ）から実プロセッサが横取りされた場合、（最後の１個の時を除いて）新たにスケジューラ活動体を生成し、他の実行中のスケジューラ活動体から実プロセッサを横取りし、利用者に報告する。この報告は、カーネルから利用者への１種の上向き呼出しにより行われる。この時、引数として２つのスケジューラ活動体の情報が利用者に渡される。２つのスケジューラ活動体とは、１つは、最初に実プロセッサが横取りされたもの、もう１つは、利用者への報告のために、実プロセッサが横取りされたものである。新たに生成されたスケジューラ活動体では、直前に動いていた２つのスケジューラ活動体の情報を解析し、動作していた利用者レベルの軽量プロセスや、際どい部分（critical section）の情報を得る。この時、スピンロックによる際どい部分を実行していたならば、その部分へ制御を移す。このようにして、スピンロックとプロセッサの横取りの問題を回避する。

（３）検出（detection）と解消
スピンロックのルーチンの中で、上記の問題が生じたことを検出し、解消を行う。

本方式では、文献［４０］と同様に、（３）より対応する。３．６．２項において述べたように、スピンロックを実現するルーチンにおいて、この問題が生じたことを検出する。そして、カーネル・コール vp_switch() を発行することにより、他の仮想プロセッサへ制御を移し、この問題の解消する。

本方式において（３）を採用した理由を、以下に示す。

- 本方式では、３．４．２項において述べたように、１つのプロセスの内部は、横取りを行わないスケジューリングを行う。従って、この問題が生じる可能性が小さい。
- ３．１２．６項において示したように、この問題が生じた時に復旧するための代償が小さい。この問題が生じた時に起きる仮想プロセッサ間のコンテキスト切替えは、ソフトウェア割込みやスケジューラ活動体の生成と比較して、オーバヘッドが小さい。

３．１３．４　共有ｆｏｒｋとの比較

３．６．３項で述べたアドレス空間の構造は、Ｄｙｎｉｘシステム［６］における fork システム・コール、および、Ｓｐｒｉｔｅシステム［６０］における共有fork（shared fork）の結果と類似している。相違点は、ＤｙｎｉｘやＳｐｒｉｔｅの場合、新たにプロセスが作られる点にある。すなわち、元のプロセスと新たに生成されたプロセスは、異なるプロセス識別子を持ち、それらが協調して動作していることは、外部から観測している限り分らない。本方式では、プロセスと仮想プロセッサは、異

なるレベルにあり、任意のプロセスを、内部の仮想プロセッサの数にかかわりなく一様に操作することが可能である。たとえば、プロセスが終了すると、内部のすべての仮想プロセッサも自動的に消去される。これに対して、ＤｙｎｉｘやＳｐｒｉｔｅでは、個々のプロセスが別々に終了する。

３．１３．５　軽量プロセスの集合によるカーネルの構築

分散型オペレーティング・システムを軽量プロセスの集合として実現する方法は、文献［９７］において既に提案されている。その方法では、相互排除アクセスされる資源の管理単位に軽量プロセスを配置する。軽量プロセス間のデータのやりとりは、軽量プロセス間の共有メモリではなく、プロセス間通信で行う。このため、オペレーティング・システムのカーネル自身をネットワーク上の別のプロセッサに配置することが可能となっている。

この構成法と３．１０節で述べた本構成法の最大の違いは、本構成法では、並列処理の単位に軽量プロセスを配置する点にある。また、本構成法では、軽量プロセスが同一アドレス空間に存在することを利用して、軽量プロセス間のデータのやり取りを軽量プロセス間の共有メモリを用いて行う。

３．１４　まとめ

この章では、マイクロプロセスと仮想プロセッサの概念に基づく軽量プロセスの実現方式について述べた。本方式では、利用者レベルにおいて軽量プロセスを実現することにより、高い性能を得ることができる。さらに、利用者プロセス内の軽量プロセスのスケジューラにより、応用固有のスケジューリングが実現される。本システムでは、層構造をもつライブラリにより軽量プロセスの実現を支援する。データベースの並列処理システムの構築を例として、そのライブラリを用いた応用固有の軽量プロセス間の同期・通信プリミティブの開発や、応用固有のスケジューラの開発の方法について述べた。

次に、仮想プロセッサを提供するカーネルの構成法について述べた。本構成法の特徴は、利用者プロセスとカーネルが同一の構造を持っている点にある。両者とも軽量プロセスから構成される。そして、利用者レベルの軽量プロセスは、仮想プロセッサにより、カーネル内の軽量プロセスは、実プロセッサにより実行される。各仮想プロセッサと各実プロセッサは、固有のメモリ領域を持ち、そこにそれぞれの識別子や固有のデータを保持する。仮想プロセッサは、カーネルの内部において並列処理の単位であり、カーネル内の軽量プロセスとして実現される。プロセスは、仮想プロセッサを実現している軽量プロセスの集合として実現される。利用者レベルの抽象であるプロセスと仮想プロセッサのスケジューリングは、カーネル内の軽量プロセスのスケジューラにより行われる。そのスケジューラは、プロセス・レベルと仮想プロセッサ・

レベルの2段階のレディ・キューから構成される。

提案した軽量プロセス実現方式、および、カーネル構成法に従って、軽量プロセスを実現するライブラリと仮想プロセッサを提供するカーネルの実現を行った。そして、その基本的な性能を測定した。本方式の軽量プロセスと、カーネル制御方式、および、コルーチン方式の軽量プロセスの基本的な性能を比較する実験を行った。さらに、1つの並列応用プログラムについて応用固有のスケジューラの記述とその性能を調べる実験を行った。これらの実現と実験を通じて、本方式の有効性を示した。

第4章　マッピング・コントローラ

この論文において、マッピングとは、ネットワークに結合された複数のプロセッサに対して、プロセスやデータを割り当てることを意味するものとする。マッピング・コントローラとは、１つの応用プログラム中にある、マッピングの方針を決定するモジュールである。この章では、マッピング・コントローラの基本概念について整理を行い、ＲｅＳＣシステムにおけるマッピング・コントローラに対する方針と支援について述べる。次に、並列シェルの機能とそのプロトタイプを用いた実験について述べる。並列シェルは、内部にマッピング・コントローラを持たない応用プログラムに代わり、マッピングの最適化を行うものである。

４．１　マッピング・コントローラの概念

ネットワーク上に分散した複数のサイト（プロセッサ）を用いて並列処理や分散処理を行う応用プログラムにとって、そのプログラムのプロセスやデータのサイトへの割当ては、非常に重要である。このようなプログラムの中にあり、割当て方針を決定するモジュールをマッピング・コントローラ（mapping controller）、あるいは、分散オプティマイザ（distribution optimizer）とよぶことにする［７９］［８２］［８６］。

マッピング・コントローラは、システムの資源の利用状況を観察し、プロセスやデータの最適な割当てを行う。この時、各プロセスの重さやプロセス間の通信量の予測を元に、プロセスやデータ・ファイルの割当ての最適化を行う。たとえば、ＣＰＵ処理が中心のプログラムのマッピング・コントローラは、ＣＰＵの能力が高く負荷が軽いサイトを探し、プロセスを生成する。大きなファイルを検索する応用プログラムのマッピング・コントローラは、ネットワーク通信のオーバヘッドを削減するために、ファイルが存在するサイトにプロセスを生成する。

マッピング・コントローラは、マッピングの方針（policy）を決定するものである。具体的なデータ・ファイルやプロセスの生成という仕組み（mechanism）は、ファイル・サーバやプロセス・サーバにより提供される。

４．２　マッピング・コントローラに対するＲｅＳＣシステムの方針

マッピング・コントローラに対するＲｅＳＣシステムの方針を、以下にまとめる。

（１）ＲｅＳＣシステムのカーネルは、マッピング・コントローラを含まない。

（２）各並列／分散応用プログラムは、固有のマッピング・コントローラを備える。

（３）システムは、各応用プログラムのマッピング・コントローラに対してシステム構成や資源の利用状況などの情報を積極的に提供する。

（４）マッピング・コントローラを含まない応用プログラムに対して、システムは、並列シェルを提供し、ネットワーク上に分散した資源の有効利用を促進する。

２．４節、および、３．４節において述べたように、カーネルは、１つのサイト内における応用プログラム間の公平なＣＰＵ資源のスケジューリングを行う。しかしながら、サイトへのマッピングに関して、カーネルは、仕組み（mechanism）を提供するファイル・サーバやプロセス・サーバを含んでいるが、方針（policy）を決定するマッピング・コントローラを含んでいない。すなわち、マッピングに関して方針と仕組みの分離（policy/mechanism separation）が実現されている［１０８］。

上記の方針により、以下のような利点が生じる。

（１）各応用プログラムが、その応用プログラム固有の性質を利用するようなマッピング・コントローラを構築することができる。

（２）マッピングを制御する時に、システム定義の汎用の記述ではなく、応用プログラム固有の記述を用いることができる。

（３）マッピング・コントローラを持っていない応用プログラムに対しても、並列シェルによりネットワーク上の資源の有効利用が実現される。

上記（１）と（２）は、マッピングに関して、システムと並列／分散プログラム間の調和が実現されることを意味する。すなわち、システムは、並列／分散応用プログラムと競合することなく、そのような応用プログラムに対して、高信頼性や高速化のためにマッピングを制御する仕組みを提供している。また、上記（３）は、マッピングに関して、逐次応用プログラムと並列／分散応用プログラムの間の調和が実現されていることを意味する。すなわち、マッピング・コントローラを持たない逐次応用プログラムは、並列シェルを利用して、システムを集中型システムと同等に扱うことができる。同時に、並列／分散応用プログラムは、カーネルを直接利用することで、並列シェルに妨害されることなく、システムを分散型システム利用することができる。

４．３　応用固有のマッピング・コントローラの支援

単一プログラミング・システムでは、各並列／分散応用プログラムは、全ての資源の利用状況を常に把握することができる。マッピング・コントローラは、この情報を利用して、最も効率的な実行が可能なように、プロセスの配置やデータの配置の最適化を行う。一方、多重プログラミング・システムにおいては、オペレーティング・システムがそのような情報を提供しなければならない。それは、システムの利用状況が刻々と変化するからである。

ＲｅＳＣシステムは、以下の情報を応用固有のマッピング・コントローラへ提供す

る。

（１）各サイトのプロセッサの型、数、および、速度

（２）各サイトの実メモリ容量

（３）ネットワークの形態（topology）

（４）各サイトの稼働状況（動作中かどうか）とプロセッサの負荷

（５）各サイトのメモリの利用状況

（６）ネットワークのトラヒック

（７）資源の位置

項目（１）から（３）までは、静的な情報である。これらは、公のファイルに格納しておく。項目（４）から（６）は、システムの動的な情報である。これらは、カーネル・サーバにより提供される。項目（７）は、２．６節で述べたオブジェクト識別子のサイト識別子を通じて提供される。

４．４　並列シェル

並列シェル（parallel shell）は、協調して動作する複数の逐次プログラム（コマンド）のためのコマンド・インタプリタである［７９］［８２］［８６］［３６］。並列シェルは、ＲｅＳＣシステムの構成要素であると同時に、カーネル上で動作する１つの並列応用プログラムでもある。並列シェルは、内部にマッピング・コントローラを持たない（逐次）応用プログラムに代わり、マッピング・コントローラとしての役割を果たす。たとえば、その応用プログラムをどのサイトで実行するかを決定したり、出力ファイルを作成するサイトを決定する。

４．３節で述べた情報に加えて、並列シェルは、次の様な情報を利用してマッピングの最適化を行う。

（１）各コマンドの負荷の予測

（２）それらの間の通信量の予測

（３）結合関係

これらは、並列シェルにとって、応用固有の情報に相当する。すなわち、他の並列／分散応用プログラムには存在しない、並列シェルに特有の概念である。

４．４．１　並列シェルの構文と機能

並列シェルの構文と意味（機能）は、ＵＮＩＸのシェルのものを拡張したものである。ＵＮＩＸのシェルから次のような機能と構文を引き継いでいる。

（１）パイプ（|）

（２）入出力の切替え（>, <）

（３）コマンドの並列実行（&）

これらの加えて、次のような拡張を行った。

（１） on 文

（２） siteof 組込関数

（３） processor_allocate 組込関数

on 文は、サイトのリストを受け取り、そのサイトでコマンドを実行する。siteof組込関数は、そのオブジェクトの位置を返す。processor_allocate 組込関数は、高速で負荷が小さいサイトの識別子を返す。

以下に、on() と siteof() を使った例を示す。このようなシェル・スクリプトが与えられると、並列シェルは、ストリームにより結合された３つのコマンドを起動する（図４－１（ａ））。

```
on(siteof(infile)) com-1 < infile | ¥
com-2 | ¥
on(siteof(outfile)) com-3 > outfile
```

ここで、"infile" と "outfile" は、そのストリームの入力となるファイルと出力となるファイルの名前である。この例では、入力ファイルが存在するサイト siteof(input) において com-1 を実行し、 出力ファイルが存在するサイト siteof(output) において com-3 を実行する。com-2 を実行するサイトは、記述されていない。このような場合、並列シェルにより決定される。

一方、図４－１（ｂ）は、簡単な負荷分散を行うマッピング・コントローラによるコマンドのマッピングの例である。このマッピング・コントローラは、２つのサイトの負荷均衡を試みている。しかしながら、コマンド間の結合関係の情報を利用していないため、結果として、ネットワーク通信量が増加している。

以下に、processor_allocate() 組込関数の例を示す。

```
on(processor_allocate(1,1,2)) com-1 & com-2 & com-3
```

この例において、processor_allocate() 関数は、負荷の予測を受け取り、軽いサイトのリストを返す。そして、on() 文により、それらのサイト上で３つのコマンドを並列に実行している。

ここで述べた on文、siteof組込関数等は、並列シェルに固有のマッピングの記述方法である。

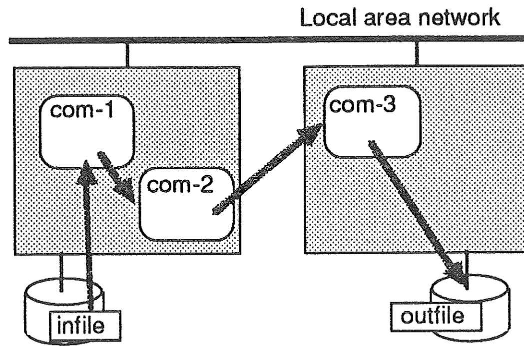


図４－１（ａ）　並列シェルによるもの
Figure 4-1(a): By the parallel shell.

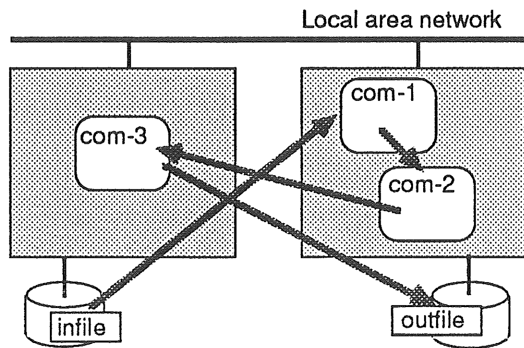


図４－１（ｂ）　簡単な負荷分散を行うマッピング・コントローラによるもの
Figure 4-1(a): By a simple mapping controller.

図４－１　ストリームで結合されたコマンドのマッピング
Figure 4-1: Mapping commands connected with streams.

４．４．２　プロトタイプの実現

並列シェルのプロトタイプを、イーサネットにより結合されたワークステーション上に実現した。実現においては、ＳｕｎＲＰＣ［６９］を利用して、２．７節において述べたＲｅＳＣシステムの遠隔手続き呼出しのエミュレーションを行った。
ネットワーク上の各サイト（ワークステーション）には、ＲｅＳＣのプロセス・サーバのエミュレータを配置する。並列シェルは、遠隔手続き呼出しを用いて、各サイト上のプロセス・サーバに対してプロセス生成要求を送る。要求を受け付けたプロセス・サーバは、ＵＮＩＸのforkシステム・コール、および、execveシステム・コールを用いて、プロセスを生成する。パイプの実現には、ＴＣＰ／ＩＰ［１０１］のストリームを利用した。

4．4．3　実験

実現した並列シェルを用いて、コマンド間の結合関係の情報を利用することによる処理効率の違いを調べる実験を行った。実験に用いたシェル・スクリプト（shell script, 並列シェル・インタプリタのプログラム）の性質を、表4－1に示す。ここでは、両スクリプトともストリームにより結合された8個のコマンドを含んでおり、それらのコマンドは、全体でパイプラインを構成する。パイプラインの各コマンドは、1ブロック（1ｋバイト）のデータを入力し、そのデータに関して処理を行い、同じ大きさのデータを出力する。パイプラインの先頭のコマンドの入力は、読み出すと0で埋められたデータを返す特殊なファイル（/dev/zero）である。パイプラインの末尾のコマンドの出力は、書き込んだデータを吸い込む特殊なファイル（/dev/null）である。

表4－1　実験に用いたスクリプトの性質

Table 4-1: The behavior of experimental scripts.

	Script 1	Script 2
Load (m sec/KB)	2	200
IPC (KB)	200	2

スクリプト1では、各コマンドの入出力データに対する計算深度が浅い。すなわち、各コマンドは、入力したデータに対して、単純な処理を行う。プロセス間の通信量も大きい。そして、パイプラインの定常状態が長い。スクリプト2は、スクリプト1とは逆に、計算深度が深く、プロセス間通信量が小さい。パイプラインの定常状態は、存在しない。

実験に用いたマッピングを以下に示す。

```
s0,s1=processor_allocate(1,1)

Sequential:
        on($s0) source | p | p | p | p | p | p | sink

Parallel(a):
        on($s0) (source | p | p | p) | ¥
        on($s1) (p | p | p | sink)

Parallel(b):
        on($s0) source | on($s1) p | on($s0) p | on($s1) p | ¥
```

```
on($s0) p | on($s1) p | on($s0) p | on($s1) sink
```

このスクリプトにおいて、括弧は、ＵＮＩＸのシェルとは異なり、結合性を変えることを意味し、新しいシェルを作ることを意味しない。

表４－２に、実験結果を示す。このように、両スクリプトのいずれのマッピングにおいても、並列処理により実行時間が短縮されていることがわかる。スクリプト１では、マッピング Parallel(a) の方が実行時間が短い。これは、ネットワーク通信のオーバヘッドが削減されたことによる。対照的に、スクリプト２では、マッピング Parallel(b) の方が実行時間が短い。この理由は、このスクリプトではパイプラインの定常状態がなく、実行の途中においても、Parallel(b)では、２つのサイトの負荷が均衡しているからである。Parallel(a) では、実行の前半では、サイト $s0 に、後半では、サイト $s1 に負荷が集中している。

このように、スクリプトの性質により、最適なマッピングが異なる。このことは、スクリプトの中のコマンド間の結合関係の情報を利用することにより、資源割当ての最適化が可能であることを示している。

表４－２　スクリプトの実行時間

Table 4-1: Execution times of the scripts.

mapping	Script 1	Script 2
Sequential	6.2	5.5
Parallel (a)	3.2	3.9
Parallel (b)	4.0	3.6

４．５　関連した研究

４．５．１　従来の分散型オペレーティング・システムとの比較

従来の分散型オペレーティング・システム［９８］［２２］［４３］では、システムがマッピング・コントローラを含んでいる。すなわち、システムがマッピング・コントローラとして、マッピングの方針（policy）を決定することを意味する。このため、並列／分散応用プログラムを走らせると、システムのマッピング・コントローラと各応用プログラムのマッピング・コントローラが競合することになる。これに対して、本ＲｅＳＣシステムでは、カーネルのレベルではマッピング・コントローラを含んでいない。このため、従来の分散型オペレーティング・システムにみられたような競合が解消される。

Ａｍｏｅｂａシステムは、従来の定義と目標そった典型的な分散型オペレーティン

グ・システムである［５４］。（本システムにおける分散型オペレーティング・システムの定義と目標については、１．５節において述べた。）Ａｍｏｅｂａシステムは、２．２．１項で述べたサーバ・モデルにより構築されているため、ネットワークの位置を利用する状況が少ない。すなわち、ファイル・サーバやウインドウ・サーバのプロセスを除き、応用プログラムは、全てプロセッサ・プールとよばれるネットワークで結合された計算サーバ群の１つで実行される。システムは、プロセッサ・プールの利用状況を常に監視しており、応用プログラムからのプロセッサ割当て要求に対して、負荷が小さいプロセッサを割り当てる。すなわち、システムがマッピング・コントローラを含んでいる。

Ａｍｏｅｂａでは、プロセッサ・プールとそれを管理するオペレーティング・システムの機能により、従来の逐次的なプログラムを複数個、並列に実行し、システム全体の処理能力を改善することを試みる。しかしながら、応用プログラムのレベルでマッピング・コントローラを支援する機能が存在しない。これに対して、本システムでは、並列／分散応用プログラムのマッピング・コントローラに対して、資源の利用状況や位置の情報を積極的に提供する。

Ａｍｏｅｂａシステムにおけるマッピングの機能は、本システムでは、並列シェルのレベルにおいて提供される。ゆえに、本システムの並列シェルとカーネルは、従来の定義における分散型オペレーティング・システムを構成するための方法と見ることも可能である。

Ａｍｏｅｂａシステムでは、プロセス生成のレベルにおいて、プロセスを単位としてマッピングの最適化を行う。これに対して、本システムでは、並列シェルのレベルにおいて、複数のコマンドのプロセスの結合関係を利用して、マッピングの最適化を行う。マッピングの最適化を行う場合、シェルのレベルで行う方が、プロセス生成のレベルで行うよりも、より簡単で効果的な最適化が可能になる。それは、シェルのレベルでは、コマンドの入出力ファイルの位置や、コマンド間の結合関係を得ることが容易であるからである。プロセス生成のレベルでは、そのプロセスがどのようなファイルを読み書きするのか、あるいは、他のプロセスとどのようなプロセス間通信を行うかを調べることは、困難である。

４．５．２　負荷均衡の方針の研究

４．４節で述べた並列シェルは、主に負荷均衡の仕組みを提供するものであり、それを用いてさまざまな方針のマッピングを記述することが可能になっている。マッピングの方針としては、様々な研究が行われている［２２］［４３］。並列シェルには、これらの研究成果を取り入れることが可能である。

文献［３６］では、独立に実行可能なコマンド群とパイプラインを形成しているコマンド群について、簡単なマッピングの方針を提案した。前者の特徴は、受け付けたコマンドを全て一度に実行するのではなく、最大のスループットが得られるように各サイトの多重度を調整しながらプロセスを割り当てる点にある。そして、プロセスが

終了したサイトに、順次プロセスを割り当てることにより、各プロセスの実行時間の違いや動的な負荷の変動への適応が実現される。後者については、４．４．３項の実験で用いたような２種類のマッピングの方針（サイト間通信を減らすものとパイプライン処理を考慮した負荷均衡を図るもの）を提案した。

４．５．３　負荷均衡の仕組みの研究

４．４節において述べた並列シェルとプロセス・サーバは、本ＲｅＳＣシステムの一部として設計されたものである。そのプロトタイプは、ＳｕｎＲＰＣとＴＣＰ／ＩＰによる通信機能があるワークステーションにおいて利用可能である。したがって、本システムの並列シェルとプロセス・サーバを、そのような環境における負荷均衡の仕組みを提供するものとしてとらえることが可能である。

他の負荷均衡の仕組みの研究［２２］［１９］と比較した場合、本並列シェルの特徴は、ストリームにより結合されたコマンド群の並列実行が可能である点にある。他の多くの研究では、主に独立して動作する（プロセス間通信を行わない）複数のプロセスが対象となっている。

４．６　まとめ

この章では、マッピング・コントローラの概念を整理し、ＲｅＳＣシステムにおけるマッピング・コントローラに対する方針と支援について述べた。ＲｅＳＣシステムでは、カーネルがマッピング・コントローラを含んでいない。カーネル上で動作する、ネットワーク上の複数のプロセッサ（サイト）を利用する各並列／分散応用プログラムは、内部に固有のマッピング・コントローラを持つことになる。これにより、システムと並列／分散応用プログラムの間の、マッピングの方針に関する競合が解消され、調和が図られる。

並列シェルは、マッピング・コントローラを持たない応用プログラムに代わって、マッピングの最適化を行う。シェル・スクリプトのレベルでマッピングの最適化を行うので、プロセス生成のレベルでマッピングの最適化を行う方法と比較して、より高度な最適化が容易に達成される。

第５章　オブジェクトの堆積

この章では、オブジェクトの堆積というモデル化の手法について述べる。ＲｅＳＣシステムにおいて、オブジェクトの堆積は、外部サーバが提供する高度な機能を統合するために用いられる。そして、統合された機能は、主に逐次応用プログラムにより利用される。

この章では、はじめに、オブジェクトの堆積の目標と背景となる技術について述べる。次にオブジェクトの堆積の基本概念を、例と比較を用いて説明する。そして、分散型オペレーティング・システムにおけるオブジェクトの堆積の利用について述べる。

５．１　目標と背景

オブジェクトの堆積は、object-based システムにおいて、複数のサーバの持つ機能を統合して利用するためのモデル化の手法である［７８］［８０］［８３］［８５］。オブジェクトの堆積は、単純なモデルであるが、非常に強力である。本ＲｅＳＣシステム以外のシステムにおいても、広く利用することが可能である。

２章で述べたように、ＲｅＳＣでは、カーネルが、分散核としてプロセス間通信機能を提供する。同時に、カーネルは、ファイル・サービスやプロセス・サービスなど基本的な機能を提供する。これらの機能は、カーネルで実現されるため、オーバヘッドが小さく、効率を重視する並列／分散応用プログラムによる利用に適している。しかしながら、逐次応用プログラムの高度な要求を全て満足することはできない。たとえば、ファイルのキャッシングやファイルの複製（replication）、オブジェクトの移動（migration）といった機能を提供していない。ＲｅＳＣでは、このような高度な機能は、カーネル外のサーバにおいて提供される。オブジェクトの堆積は、この高度な機能を実現するモデル化の手法として開発されたものである。

オブジェクトの堆積により、逐次応用プログラムと並列／分散応用プログラムの間の調和が実現される。すなわち、逐次プログラムは、オブジェクトの堆積により統合された、高度な外部サーバの機能を利用することができる。同時に、並列／分散応用プログラムは、効率的なカーネル・サーバを直接利用することができる。また、システムの一部である外部サーバは、並列／分散応用プログラムとしてカーネルの機能を利用する。これにより、システムと他の並列／分散応用プログラムの間の調和が図られる。

オブジェクトの堆積の背景となった技術を、以下にまとめる。

（１）object-based 分散型オペレーティング・システム

最近の分散型オペレーティング・システムの多くは、object-basedシステムとして構成されている［９８］［２２］［４３］。オブジェクトの堆積は、このようなobject-basedシステムにおいて、利用可能である。object-based システムは、サーバ、クライアント、カーネルから構成される。サーバは、オブジェクトを管理し、クライアントへサービスを提供する。主要なサービスとして、ファイル・サービス、プロセス・サービス、ディレクトリ・サービスがあげられる。カーネルは、プロセス間通信と物理的な入出力機能を提供する。オブジェクトの堆積では、システムによって提供される基本的なオブジェクト、オブジェクト識別子、および、クライアント・サーバ間のプロセス間通信機能が必要となる。

一方、分散ファイル・システムに関する研究が盛んに行われている［１３］［２４］［４７］［９４］。このような研究の多くは、複製（replica, コピー）間の一貫性を保証する効率的なアルゴリズムを与えるものである。オブジェクトの堆積は、そのようなアルゴリズムを与えるものではなく、分散ファイル・システムをカーネル外で実現することを可能するためのモデル化の手法を提供する。さらに、オブジェクトの堆積を使うことにより、汎用の複製ファイル・サーバを構築することができる。従来のファイルの複製の研究では、１種類のファイルに対してのみ複製を作ることができた。これに対して、オブジェクトの堆積では、様々な種類のファイルの複製を作ることが可能になる。

（２）機能を組み合わせる仕組み

オペレーティング・システムにおいて、機能を組み合わせる仕組みとして、もっとも成功した例の１つが、ＵＮＩＸのパイプである。パイプを使うことにより、単純なコマンドを組み合わせて、複雑なコマンドを作ることが可能となる。これに対して、オブジェクトの堆積では、単純なオブジェクトを組み合わせて、複雑なオブジェクトを作ることが可能となる。

（３）層化プログラミング

層化プログラミングでは、巨大なシステムを複数の層に分割して開発することで、再利用可能性が高いモジュールが構築される。たとえば、ＩＳＯのＯＳＩ参照モデルでは、通信プロトコルを７つの層に分割することで、異機種間の通信を可能にしている［１６］［９２］。これに対して、オブジェクトの堆積では、再利用可能性が高いモジュールをサーバとして開発することが可能となる。たとえば、キャッシング機能と複製機能があるファイル・サーバを構築する場合、キャッシング機能しかないサーバと複製機能しかないサーバを組み合わせることで、両者の機能を併せ持つサーバを構築することができる。

5．2　基本概念

オブジェクトの堆積は、次の４つの基本概念から構成される。

（基本概念１）オブジェクト

オブジェクトの堆積において、オブジェクトとは、手続きとデータがカプセル化されたものである。オブジェクトのデータの参照と更新は、外部に公開された手続きによってのみ行われる。object-based 分散型オペレーティング・システムでは、オブジェクトは、サーバ・プロセスによって管理と保護が行われる。図５－１において、Server A は、ファイル・サーバであり、ファイル・オブジェクトを管理している。そして、クライアントからの読み込み手続き（read()）と書き込み手続き（write()）を受け付け、実行する。そして、それ以外の不正な操作からオブジェクトを保護している。

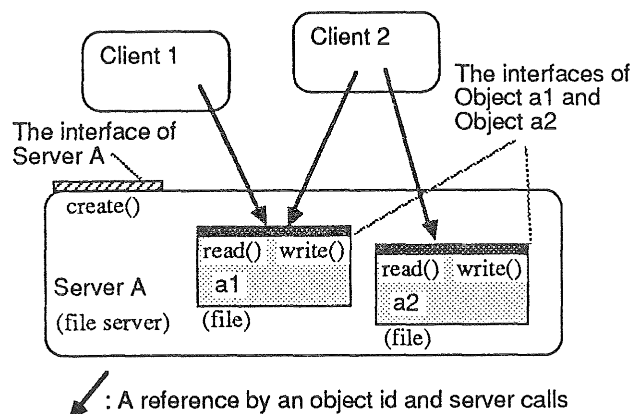


図５－１　オブジェクト、サーバ、クライアントの関係
Figure 5-1: Objects, their server and clients.

オブジェクトの堆積では、オブジェクトは、オブジェクト識別子（object identifier, object id）により常に間接的に参照される。図５－１において、Client 1 は、Object a1 のオブジェクト識別子を保持しており、それを使ってそのファイルに読込み／書込み要求のメッセージを送る。そのメッセージは、サーバに送られ、サーバにより解釈される。

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）、サーバに対する手続きをサーバ手続き（server procedure）とよぶことにする。これらの手続きは、それぞれオブジェクト指向プログラミング言語におけるインスタンス・メソッド、および、クラス・メソッドに対応する。

あるオブジェクトが受け付けることができる手続きの集合を、オブジェクトのイン

タフェースとよぶ。たとえば、ファイル・オブジェクトは、インタフェースとして読込み手続きと書込み手続きを持つ。同様に、サーバが受け付けることができる手続きの集合を、サーバのインタフェースとよぶ。たとえば、ファイル・サーバのインタフェースには、ファイルの生成を行う手続きが含まれている。

（基本概念２）積み重ねる、積む、堆積を作る。

オブジェクトa1の上にオブジェクトb1を**積み重ねる**（to stack b1 on a1）とは、次の２つの条件を満たすことである。

（１）オブジェクトb1がオブジェクトa1のオブジェクト識別子を保持している。

（２）オブジェクトb1が自分自信の機能を実現するために、必ずオブジェクトa1を呼び出す。

この時、オブジェクトb1のサーバは、クライアントとしてオブジェクトa1を利用する。

オブジェクトa1の上にオブジェクトb1が**堆積可能**である（be stackable on）とは、オブジェクトb1 の機能を実現するために、十分なインタフェースと機能をオブジェクトa1が持っていることと定義する。（b1が使わないインタフェースと機能をa1が持っていてもよい。）

オブジェクトがオブジェクト識別子を含んでいたとしても、必ずしも積み重ねることを意味しない。たとえば、ディレクトリ・オブジェクトは、その中にオブジェクト識別子と文字列の組を保持している。しかしながら、ディレクトリ・オブジェクトは、そのオブジェクト識別子で示されたオブジェクトの上に積まれたオブジェクトではない。なぜならば、そのオブジェクト識別子は、単なるビット列であり、そのディレクトリ・オブジェクト自身の機能を実現するために、そのオブジェクト識別子で示されたオブジェクトへメッセージを送ることがないからである。

（基本概念３）堆積をオブジェクトと同じと考える

あるオブジェクトの上に、別のオブジェクトを積む場合、下位層オブジェクトが、システムにより実現された単純なオブジェクトではないかもしれない。すなわち、既にいくつかのオブジェクトが積み重ねられたものの上に、新たにオブジェクトを積むことができる。

別の言葉では、オブジェクトが堆積したものもまたオブジェクトであり、その上に新たにオブジェクトを積み重ねることが可能である。その結果できたものもまたオブジェクトである。今後、堆積によってできたオブジェクトを特に区別して扱う場合、**堆積オブジェクト**（stack object）とよぶ。単にオブジェクトという場合、単純なオブジェクトと堆積オブジェクトの両方を含むものとする。

あるオブジェクトの下のオブジェクトを**下位層のオブジェクト**（lower object）とよぶ。あるオブジェクトの上に積まれているオブジェクトを**上位層のオブジェクト**（upper object）とよぶ。最上位層のオブジェクトを**頂上オブジェクト**（top object）、最下位層のオブジェクトを**基底オブジェクト**（bottom object）とよぶ。

基底オブジェクトは、他のオブジェクトに依存していない。そして、他のオブジェクトのオブジェクト識別子を含んでいない。基底オブジェクトは、堆積により作ることができないので、システムにより提供されなければならない。

図５－２において、Object a1 は、Server A によって管理されている基底オブジェクトであり、その上に Server B によって管理されている Object b1 が積まれ、１つの堆積オブジェクトが作られている。これを堆積オブジェクトb1/a1と表す。Object b1 は、a1 の識別子を保持しており、b1自身の機能を実現するために、a1の機能を呼び出す。堆積オブジェクトb1/a1 の上に、さらに Object c1 が積まれ、堆積オブジェクトc1/b1/a1 が形成されている。

堆積オブジェクトの識別には、その頂上オブジェクトのオブジェクト識別子を利用する。たとえば、図５－２の堆積オブジェクトc1/b1/a1 の識別子として、c1 の識別子が使われる。これにより、クライアントは、そのオブジェクトが堆積オブジェクトであるか、単純なオブジェクトであるかにかかわらず、一様に扱うことができる。

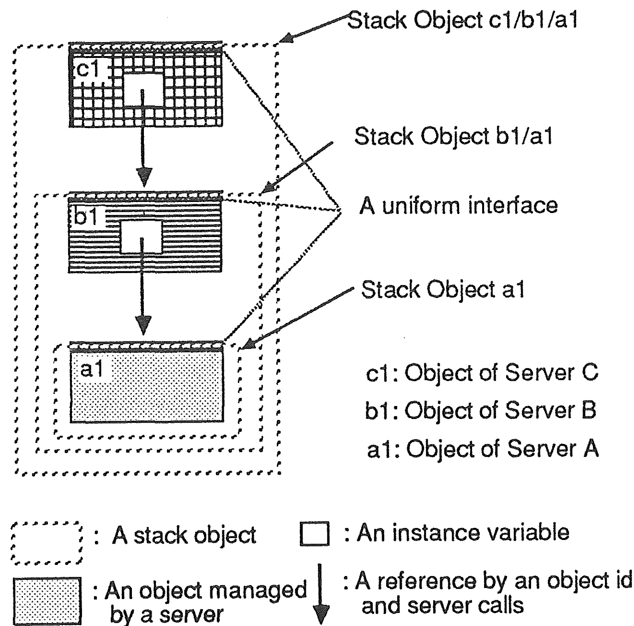


図５－２　サーバに管理されているオブジェクトと堆積オブジェクト

Figure 5-2: Objects managed by servers and stack objects.

（基本概念４）一様なインタフェース

オブジェクトを自由に組み合わせて利用するためには、それらのオブジェクトのインタフェースが一様（uniform）である必要がある。一様なインタフェースのオブジェクトを利用することにより、積み重ねるオブジェクトの構成と順序を自由に選択することが可能となる。もし、インタフェースが異なれば、あるオブジェクトの上に積むことができるオブジェクトの種類が制約される。また、実行される手続きに着目すると、一様なインタフェースにより、手続きの実行の順序を積み重ねるオブジェクト

の順序として、個々の堆積オブジェクトごとに設定することが可能となる。

一様なインタフェースを持つということは、自分自身のインタフェースと下位層のオブジェクトのインタフェースが同一であることを意味する。たとえば、自分自身がファイルというインタフェースを持つ場合、下位層のオブジェクトのインタフェースもまたファイルである。図５－２において、Object a1, b1, c1 は、一様なインタフェースを持っている。

■堆積可能オブジェクト

自分自身のインタフェースと下位層オブジェクトのインタフェースが一様であるようなオブジェクトを、**堆積可能オブジェクト**（stackable object）とよぶ。同様に、堆積可能ファイル・オブジェクトとは、自分自身のインタフェースがファイルで、下位層のオブジェクトのインタフェースもファイルであるようなオブジェクトである。図５－２において、Object b1、および、Object c1 が堆積可能である。

堆積可能オブジェクトを管理しているサーバを**堆積可能サーバ**（stackable server）とよぶ。同様に、堆積可能なファイル・オブジェクトを管理しているサーバを、堆積可能ファイル・サーバとよぶ。図５－２では、Server B、および、Server C が堆積可能サーバである。

５．２．１　ＳｔｄＦＳ、ＺＦＳ、ＣＦＺ

ここでは、以下の３つのファイル・サーバを使って、object-basedシステムにおけるオブジェクトの堆積の例を示す（図５－３）。

（１）標準ファイル・サーバ(StdFS:Standard File Server)

これは、システムが提供する標準的なファイル・サーバである。ＲｅＳＣシステムでは、カーネルにおいて提供される。標準ファイル・サーバでは、各オブジェクトは、下位層のオブジェクトを必要としない。ファイルのデータは、ディスク・ブロックに格納される。

（２）圧縮ファイル・サーバ(ZFS:Compression File Server)

これは、ファイルの内容を圧縮して保存するようなファイル・サーバである。圧縮ファイル・サーバでは、各オブジェクトは、圧縮したデータを格納するために、下位層のファイル・オブジェクトを１個必要とする。クライアントから書込み要求を受け付けると、サーバは、書込み要求の中のデータを圧縮し（compress）、対応する下位層のオブジェクトに書込み要求を送る。クライアントから読込み要求を受け付けると、下位層のオブジェクトに読込み要求を送り、圧縮されたデータを得る。そして、圧縮されたデータを解凍し（uncompress）、その結果をクライアントに返す。

（３）暗号ファイル・サーバ(CFZ:Encryption File Servere)

これは、ファイルの内容を暗号化して保存するようなファイル・サーバである。サーバの構造と動作は、ＺＦＳにおける圧縮と解凍を暗号化と復号化に置き換えたものである。

ＺＦＳでは、普通、各オブジェクトは、下位層のオブジェクトとしてＳｔｄＦＳのオブジェクトを保持する。すなわち、ＺＦＳの各ファイルについて、圧縮した内容を保存するために１つのＳｔｄＦＳのファイルが利用される。これにより、圧縮ファイルが作られる（図５－３：①）。ここで、ＳｔｄＦＳの代りに、ＣＦＳのオブジェクトを下位層オブジェクトとして保持するならば、圧縮された暗号化ファイルが作られる（図５－３：②）。

オブジェクトの堆積では、堆積可能オブジェクトの順序は、固定されていない。この例では、図５－３に示すように、ＺＦＳ／ＣＦＳ／ＳｔｄＦＳの順（図５－３：②）でも、ＣＦＳ／ＺＦＳ／ＳｔｄＦＳの順（図５－３：③）でもかまわない。（ただし、圧縮と暗号の場合、順序が効率に影響を与える。先に圧縮を行う方が先に暗号化を行うよりも圧縮効率がよい。）

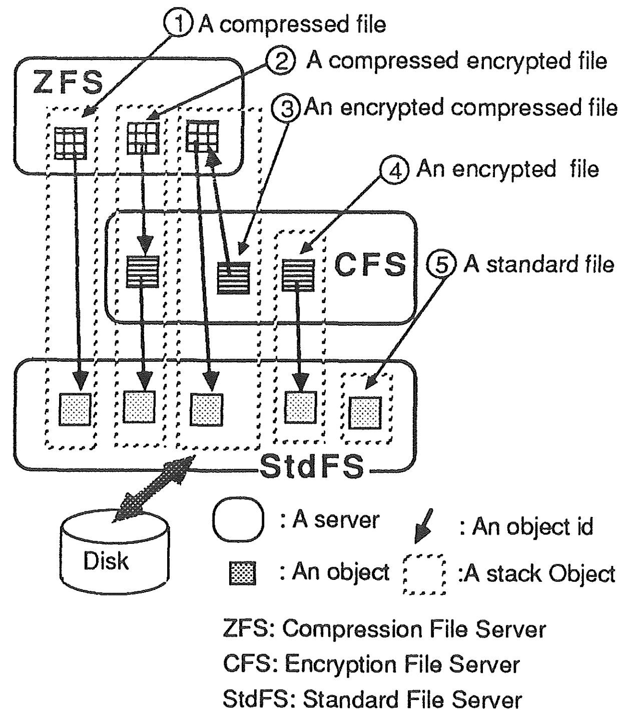


図５－３　圧縮、暗号、標準ファイル・サーバ

Figure 5-3: Compression, Encryption and Standard File Servers.

５．２．２　サーバの堆積とオブジェクトの堆積

オブジェクトの堆積では、堆積がオブジェクトを単位として作られる。一方、個々のオブジェクトではなく、サーバ全体を単位として堆積を作る方法がある。これを、**サーバの堆積（server-stacking）**とよぶことにする（図５－４（ａ））。サーバの堆積とオブジェクトの堆積の概念は、ほぼ同じ時期に、独立に考案された［２４］［７０］［８５］。

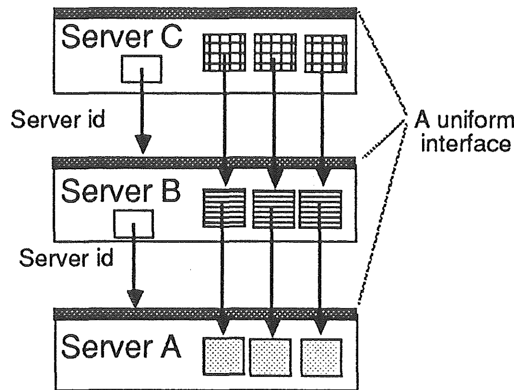


図５－４（ａ）　オブジェクトの概念があるサーバの堆積

Figure 5-4(a): Server-stacking with objects.

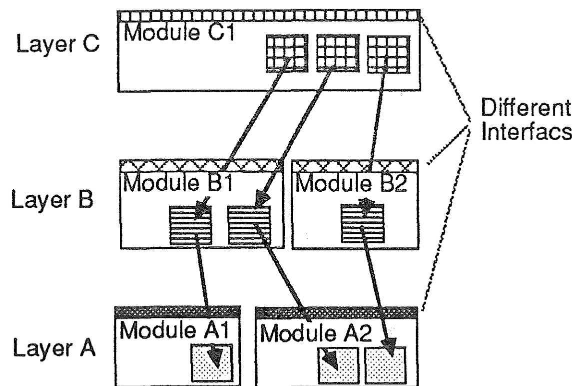


図５－４（ｂ）　オブジェクトの概念がある層化プログラミング

Figure 5-4(b): Layered programming with objects.

図５－４　サーバの堆積と層化プログラミング

Figure 5-4: Server-stacking and layered programming.

オブジェクトの堆積では、オブジェクトを生成する時に、そのオブジェクトの下位層のオブジェクトの識別子をサーバに渡す。これに対して、サーバの堆積では、サーバを起動する時に、下位層のサーバの識別子を引数として渡す。この結果、オブジェクトの堆積と同様に、堆積可能なサーバを用いることによって、サーバの構成と順序を自由に選択することが可能となる。ここでサーバが堆積可能であるとは、自分自身

のインタフェースと、下位層のサーバのインタフェースが同一であることである。

サーバの堆積では、必ずしもサーバがオブジェクトを扱う必要はなく、単なる手続き呼出しにより機能が提供されることも考えられる。しかしながら、オブジェクトを扱うサーバの場合、サーバの堆積では、サーバ単位の自由しかなく、すべてのオブジェクトが同じ形になってしまう。図５－４（ａ）に、オブジェクトを扱うサーバを示す。たとえば、５．２．１項で述べたＺＦＳとＣＦＳをサーバの堆積により重ねた場合、すべてのファイルが暗号化された圧縮ファイルになる。暗号化だけされたファイル、あるいは、圧縮だけされたファイルを作ることができない。

オブジェクトの堆積においても、結果として、同一の形の堆積しか作られないこともある。ゆえに、サーバの堆積は、オブジェクトの堆積の保護された特殊形である。ここで保護とは、柔軟性を犠牲にして、不正な堆積が生成されないようにすることである。また、サーバ全体をオブジェクトと考えると、１つのプログラムから１つのインスタンスしか生成されないような、オブジェクトの堆積そのものであると見ることもできる。

５．２．３　層化プログラミングとオブジェクトの堆積

サーバの堆積は、サーバを１つの層と考えると、層化プログラミング（layered programming）と似ている（図５－４（ｂ））。相違点は、層化プログラミングの場合、層の構成と順序が固定されているのに対して、サーバの堆積では、サーバの構成と順序が自由に選択可能になっていることである。

層化プログラミングでは、必ずしもオブジェクトの概念が必要ではない。層化プログラミングにおいて、オブジェクトの概念を導入し、１つの層について複数のサーバを実現するならば、オブジェクトの堆積と同様に、サーバを自由に組み合わせて利用することが可能になる。また、１つの層の内部の実現において、オブジェクトの堆積、または、サーバの堆積の技術を取り入れることも可能である。

５．２．４　ＵＮＩＸのパイプラインとオブジェクトの堆積

オブジェクトの堆積の枠組みでは、ＵＮＩＸにおけるパイプラインを構成する（パイプで結合された）プロセス群は、ストリームという通信プリミティブにより結合されたプロセス・オブジェクトの堆積と見ることが可能である。ＵＮＩＸのパイプでは、プロセス・オブジェクトという揮発性（volatile）のオブジェクトが堆積を形成する。これに対して、ファイル・オブジェクトを積み重ねた場合、ファイルという安定（stable）なオブジェクトが堆積を形成する。たとえば、ＵＮＩＸにおけるプロセス・オブジェクトの堆積では、コマンドを実行する度に新しいプロセス・オブジェクトが生成される。一方、ファイル・オブジェクトの堆積の場合、何度アクセスされても同じオブジェクトが参照される。この性質を利用してキャッシングを行うことで効率を改善することができる。

５．２．５　抽象データ型、部分型とオブジェクトの堆積

オブジェクトの堆積は、object-basedシステムだけではなく、プログラミング言語においても利用することができる。その条件は、**抽象データ型**（ADT, Abstract Data Types）、および、**部分型**（subtypes）を支援していることである［４２］。抽象データ型とは、実現方法によって型を記述するのではなく、その型に属する値を使って利用できる操作のリスト、および、その操作の公理を示すことよって型を記述するものである。抽象データ型のデータでは、内部の状態が公開された手続きを通じてのみ操作可能である。このことは、object-basedシステムにおける遠隔手続き呼出しと対応する。

object-basedシステムにおけるオブジェクトとサーバは、プログラミング言語におけるインスタンス（オブジェクト）とクラス（抽象データ型の具体的な実現）に対応する。たとえば、ファイル抽象データ型には、read(),write() のような手続きが定義されており、すべてのファイル型のインスタンスは、この手続きを受け付けなければならない。５．２．１項で述べた例においては、標準ファイル・クラス、圧縮ファイル・クラス、暗号ファイル・クラスは、ファイル抽象データ型の３つ実現である。

抽象データ型には、この節の冒頭で述べた基本概念（２）と（４）が存在しない。（（３）は、部分型と解釈される。）オブジェクトの堆積では、堆積可能オブジェクトは、インスタンス変数として、別のオブジェクトの識別子を保持し、自分自身の機能を実現するために、そのオブジェクトを呼び出さなければならない。さらに、そのオブジェクトが自分と同じ抽象データ型（のある部分型）のインスタンス（オブジェクト）でなければならない。たとえば、堆積可能なファイル・オブジェクトは、ファイル抽象データ型の部分型のインスタンス変数を持っており、その変数が指しているオブジェクトを呼び出して、自分自身の機能を実現する。

５．２．６　Ｃ＋＋によるオブジェクトの堆積の例

図５－５に、Ｃ＋＋によるオブジェクトの堆積の実現の例を示す。

第６行から第１０行において、Service1 は、Ｃ＋＋の抽象クラス（abstract class）として宣言されている。Ｃ＋＋の抽象クラスとは、抽象データ型に対応するもので、他のクラスの基底クラス（base class）となるだけで、インスタンスが生成されることがないようなクラスである。図５－５第９行の仮想関数（virtual function）に０が代入されている。これにより、このクラス Service1 は、抽象クラスとなる。

図５－５第１５行から第３６行において、サービス Service1 を具体的に実現する３つのクラス ServerA, ServerB, ServerC が宣言されている。ServerA は、基底オブジェクト（bottom object）のサーバ、ServerB と ServerC は、堆積可能オブジェクトのサーバになっている。各コンストラクタは、ServerA を除き、下位層のオブジェクトの識別子を１個引数として取る。オブジェクト識別子は、Ｃ＋＋では、オブジェクトへのポインタにより実現されている。

```
 1: #include "std.h"
 2:
 3: //
 4: //      Service
 5: //
 6: class Service1                          // abstract class in C++
 7: {
 8: public:
 9:         virtual void print( char *s ) = 0 ;
10: };
11:
```

図5－5（a） 抽象データ型の定義

Figure 5-5(a): The definition of the abstract data type "Service1".

```
12: //
13: //      The interfaces of three servers.
14: //
15: class ServerA : public Service1         // Bottom server
16: {
17: public:
18:         ServerA() {}
19:         virtual void print( char *s );
20: };
21:
22: class ServerB : public Service1
23: {
24:         class Service1 *lower ;
25: public:
26:         ServerB( Service1 *x ) { lower = x; }
27:         virtual void print( char *s );
28: };
29:
30: class ServerC : public Service1
31: {
32:         class Service1 *lower ;
33: public:
34:         ServerC( Service1 *x ) { lower = x; }
35:         virtual void print( char *s );
36: };
37:
```

図5－5（b） サーバのインタフェースの定義

Figure 5-5(a): The definitions of the interfaces of the servers.

図5－5 C＋＋言語によるオブジェクトの堆積の実現

Figure 5-5: Object-stacking in C++.

```
38: //
39: //      The implementations of the servers.
40: //
41: void
42: ServerA::print( char *s )
43: {
44:         printf("%s/A¥n", s );
45: }
46:
47: void
48: ServerB::print( char *s )
49: {
50:     char buff[strlen(s)+1+2] ;
51:         sprintf( buff, "%s/B", s );
52:         lower->print( buff );
53: }
54:
55: void
56: ServerC::print( char *s )
57: {
58:     char buff[strlen(s)+1+2] ;
59:         sprintf( buff, "%s/C", s );
60:         lower->print( buff );
61: }
62:
```

図５－５（ｃ）　サーバの実現

Figure 5-5(c): The implementations of the servers.

```
63: //
64: //      The client.
65: //
66: main()
67: {
68:         Service1 *a1 = new ServerA();
69:         Service1 *b1 = new ServerB( a1 );
70:         Service1 *c1 = new ServerC( b1 );
71:         c1->print("hello[1]");
72:
73:         Service1 *a2 = new ServerA();
74:         Service1 *c2 = new ServerC( a2 );
75:         Service1 *b2 = new ServerB( c2 );
76:         b2->print("hello[2]");
77:
78:         Service1 *b3 = new ServerB( b2 );
79:         Service1 *c3 = new ServerC( b3 );
80:         c3->print("hello[3]");
81: }
```

図５－５（ｄ）　クライアント

Figure 5-5(d): The client.

```
hello[1]/C/B/A
hello[2]/B/C/A
hello[3]/C/B/B/C/A
```

図５－６　ｍａｉｎプログラムの実行結果

Figure 5-6: Results of the main program.

ServerB, ServerC では、各オブジェクトは、抽象データ型 Service1 のインスタンスへのポインタを持っている（第２４行、第３２行）。これらは、下位層のオブジェクトを保持するための変数である。ServerA のオブジェクトの場合、下位層のオブジェクトを保持する変数を持っていない。

各クラスのコンストラクタは、ＲｅＳＣシステムにおける堆積を生成するサーバ手続き create_with_lower に相当する。Ｃ＋＋では、コンストラクタ関数の名前がクラスの名前と同じという制約があるため、それぞれのクラスで別々の名前になっている。create_with_lower については、５．４．２項において詳しく述べる。

第４１行から第６１行において、各サーバの実現が記述されている。第４１行の基底オブジェクトの関数 print では、他のオブジェクトを呼び出すことなく、機能が実現されている。ここでは、Ｃ言語の printf 関数を呼び出している。第４７行、および、第５５行の関数 print では、lower->print という記述により、下位層のオブジェクトの機能が呼び出されている（第５２行、第６０行）。Ｃ＋＋における演算子 "->" によるメンバ関数の呼出しが、object-basedシステムにおける遠隔手続き呼出しに対応する。このように、堆積可能オブジェクトのメンバ関数では、下位層のオブジェクトを呼び出すことにより自分自身の機能を実現している点に特徴がある。

第６６行以下、関数 main の部分が、上で宣言されたサーバを利用するクライアントに相当する。第６８行において、サーバ ServerA のコンストラクタを呼び出し、基底オブジェクトを生成している。そして、生成されたオブジェクトへのポインタを Service1 型のポインタを保持する変数 a1 に格納に格納している。第６８行では、第６７行において生成されたオブジェクトへのポインタを引数として、ServiceB のコンストラクタが呼び出され、堆積オブジェクト b1/a1 が生成されている。同様に、第７０行では、堆積オブジェクト c1/b1/a1 が生成されている。

第７１行において、この３層のオブジェクトから構成される堆積オブジェクトの頂上オブジェクト c1 に print メッセージを送っている。その結果を、図５－６の第１行に示す。このように、c1/b1/a1 という堆積が生成されていることが分る。

図５－５の第７３行から第７５行では、b2/c2/a1 という堆積オブジェクトが生成されている。このことは、図５－６の第２行の実行結果にも現れている。堆積オブジェクト c1/b1/a1 と 堆積オブジェクト b2/c2/a1 では、ServerB::print() と ServerC::print() の実行順序が逆になっている。このように、オブジェクトの堆積では、オブジェクトを積み重ねる順序に制限がなく、実行される手続きの順序をオブ

ジェクトごとに設定することが可能である。

図５－５第７８行、第７９行では、既存の堆積オブジェクト b2/c2/a2 の上にさらに、オブジェクト b3, c3 を積んでいる。その結果として、堆積オブジェクト c3/b3/b2/c2/a2 が生成されている。第８０行の c3->print の実行の結果を、図５－６の第３行に示す。このように、ServerB::print と ServerC::print が再帰的に呼び出されている。

この例では、Ｃ＋＋言語の抽象データ型と部分型の機能が使われている。継承の機能が利用されているように見えるが、それは、部分型を定義するためにのみ利用されている。すなわち、部分クラスの機能は、インタフェースを一様にするためにのみ利用されていおり、各関数の実現を継承するためには利用されていない。

５．２．７　継承とオブジェクトの堆積

オブジェクトの堆積は、オブジェクト指向システムとオブジェクト指向プログラミング言語における継承（inheritance）とは関係がない。たとえば、オブジェクト指向プログラミング言語において、５．２．１項で述べた圧縮ファイル・サーバように、圧縮ファイル・クラスを作ることを考える。この場合、標準ファイル・クラスの部分クラスとして圧縮ファイル・クラスを開発することは、可能である。しかしながら、標準ファイル・クラスのメソッド（手続き）の実現を継承することはできない。すなわち、read()やwrite()といったメソッドを全て書き換える必要がある。この時、圧縮ファイル・クラスのインタフェースは、標準ファイル・クラスのインタフェースと同一のものになる。しかしながら、このことは、標準ファイル・クラスも、圧縮ファイル・クラスも、ファイル抽象データ型の１つの実現であることを示しているだけである。継承とは、インタフェースだけではなく、メソッドの実現が共有される必要がある［１４］。

オブジェクト指向プログラミング言語において、継承の機能を用いることなく、５．２．１項で述べたものと同じ様なプログラムを記述することが可能である。すなわち、たまたま自分と同じ型（の部分型）のオブジェクトの識別子をインスタンス変数として保持しており、そのオブジェクトへメッセージを送ることで、自分自身のオブジェクトの機能を実現するようなプログラムを記述すればよい。これは、オブジェクト指向プログラミング言語における抽象データ型を支援する機能を利用していることに相当する。

オブジェクト指向では、継承という機能を利用して、プログラム・コードの共有を行う。これにより、ソフトウェアの再利用可能性を高めることができる。一方、オブジェクトの堆積では、継承の機能、すなわち、メソッドのプログラムの共有は、行われない。しかしながら、インタフェースを統一することにより、汎用で独立性の高いサーバを構築することができる。これにより、プログラム・コードの共有を行うことなく、ソフトウェアの再利用可能性を高めることができる。

５．２．８　委譲とオブジェクトの堆積

オブジェクトの堆積は、オブジェクト指向プログラミング言語における委譲（delegation）と類似点がある［４８］。委譲とは、継承とは異なる仕組みにより、機能の再利用をはかるものである。委譲に基づくモデルでは、クラスの概念が存在しない。新しいオブジェクトは、既存のオブジェクトをプロトタイプとして、そのプロトタイプとの違いを記述することにより作られる。オブジェクトに対してメッセージを送ると、まずそのオブジェクト自身が検索される。そして、対応するメソッドが見つかった場合、それが実行される。見つからなかった場合、そのオブジェクトのプロトタイプが呼び出される。

オブジェクトの堆積における積み重ねるという動作は、委譲におけるプトロタイプからオブジェクトを生成する動作と類似している。そして、オブジェクトの堆積における下位層のオブジェクトが、委譲におけるプロトタイプ・オブジェクトに相当する。オブジェクトの堆積と委譲との相違点を以下にまとめる。

（１）オブジェクトの堆積には、クラス（サーバ）の概念がある。オブジェクトは、クラス（サーバ）により管理される。委譲には、クラスの概念が存在しない。委譲では、クラスに相当するようなオブジェクトを生成するオブジェクトを定義することも可能ではある。しかしながら、この場合においても、クラスとオブジェクトは、区別されない。

（２）委譲には、基本概念（４）がない。すなわち、一様なインタフェースを用いるという制約が存在しない。委譲では、一般に、オブジェクトとそのプロトタイプ・オブジェクトは、異なるインタフェースを持つ。このようなインタフェースの拡張は、継承における特殊化に相当する。

（３）オブジェクトの堆積では、未定義の手続きに対する呼出しは、下位層のオブジェクトに転送されるのではなく、サーバにより拒絶される。委譲では、自動的にそのプロトタイプに転送される。

（４）委譲には、ｓｅｌｆという変数が存在する。その変数は、一連の手続き呼出しにおいて、最初に手続き呼出しを受け付けたオブジェクトを保持している。これに対して、オブジェクトの堆積では、そのような変数は存在しない。すなわち、上位層のオブジェクトのサーバは、一般のクライアントとして、下位層のオブジェクトのサーバを呼び出す。

５．２．９　メタ・オブジェクトとオブジェクトの堆積

オブジェクトの堆積における下位層のオブジェクトは、メタ・オブジェクトとは関係がない。メタ・オブジェクトを利用するシステムとしては、Ａｐｅｒｔｏｓシステム（Ｍｕｓｅシステム）があげられる［１０９］。Ａｐｅｒｔｏｓは、平行オブジェクト指向計算（concurrent object-oriented computing）とリフレクティブ計算に（reflective computing）に基づいている分散型オペレーティング・システムである。

Ａｐｅｒｔｏｓが与える単一の抽象は、オブジェクトである。各オブジェクトは、１つ、あるいは、複数のメタ・オブジェクトにより支えられている。支えているオブジェクトにより実現される環境は、メタ空間とよばれる。メタ空間は、そのオブジェクトの実行環境を提供するオブジェクトであり、インタプリタであり、そして、そのオブジェクトの実行に最適なオペレーティング・システムである。メタ・オブジェクト自身も、他のオブジェクト（メタ・メタ・オブジェクト）により、支援されている。この結果、オブジェクトは、メタ階層構造を構成する。

オブジェクトの堆積におけるサーバは、オブジェクトのメタ・オブジェクトに相当する。オブジェクトの堆積におけるサーバとＡｐｅｒｔｏｓのオブジェクトを比較すると、オブジェクトの堆積におけるサーバは、互いに依存し合っているのに対して、Ａｐｅｒｔｏｓのオブジェクトは、因果律（the law of causality）により、依存関係が決定されている。オブジェクトの堆積では、例えば５．２．１項で述べたＺＦＳとＣＦＳは、互いに呼び出し合うことがある。これに対して、Ａｐｅｒｔｏｓでは、２つのオブジェクトが互いに相手を支援し合うことはない。

オブジェクトの堆積における下位層のオブジェクトは、Ａｐｅｒｔｏｓのメタ・オブジェクトに似ている側面がある。大きな違いは、オブジェクトの堆積では、各オブジェクトの振舞がそのサーバにより決定されるのに対して、Ａｐｅｒｔｏｓでは、メタ・オブジェクトにより決定される点にある。

Ａｐｅｒｔｏｓでは、オブジェクトの性質を変化させる時に、別のメタ空間に移動（migrate）させることがある。たとえば、オブジェクトに永続性を与える場合、そのような機能を提供するメタ空間にオブジェクトを移動する。オブジェクトａ１を、空間Ａから別の空間Ｂに移動することができる時、ＡからＢへ移動可能であると呼ぶことにする。この移動可能性とオブジェクトの堆積における堆積可能性は、類似点がある。すなわち、両者とも柔軟に移動／堆積するためには、一様なインタフェースが必要となる。

５．３　システムに対する要求

オブジェクトの堆積を実現するために必要とされるシステムの機能は、以下のようにまとめられる。ＲｅＳＣシステムのカーネルは、これらの条件を全て満足する。

（１）オブジェクトの概念の存在

オブジェクトとは、データと手続きがカプセル化されたものである。オブジェクトのデータの参照と更新は、外部に公開された手続きによってのみ行われる。

分散型オペレーティング・システムでは、オブジェクトは、サーバにより管理されている。サーバは、アドレス空間の壁により隔離されている。クライアントは、公開された（export された）遠隔手続き呼出しによってのみオブジェクトの内部のデータを参照したり更新したりすることができる。オブジェクト指向プログラミング言語、あるいは、抽象データ型を支援したプログラミング言語では、コンパイラや実行時システムによりオブジェクトへのアクセスが公開された手続きに制限される。

ＲｅＳＣは、object-basedシステムであり、システムの資源は、オブジェクトとして提供される。オブジェクトの概念が実現ないプログラミング言語やシステムでは、オブジェクトの堆積を利用することができない。ＵＮＩＸシステムは、ファイルやプロセスをオブジェクトと考えることが可能であるので、この条件を満足する。

（２）オブジェクト識別子

オブジェクトは、一様なオブジェクト識別子により参照されなければならない。これは、任意のサーバのオブジェクトを下位層オブジェクトとして利用するためである。プログラミング言語では、オブジェクトへのポインタをオブジェクト識別子として利用する。この場合、型がないか、あるいは、部分型を支援している必要がある。

ＲｅＳＣでは、２．６節で述べたように、システム・レベルのオブジェクトは、一様なオブジェクト識別子により識別される。ＵＮＩＸシステムにおいては、ファイル名をオブジェクト識別子として利用することが可能である。この場合、ファイル・オブジェクトとディレクトリ・オブジェクトを一様に扱うことができる。しかしながら、プロセス・オブジェクトを一様に扱うことは不可能である。また、効率の面では、文字列の名前よりも数値の名前の方が有利である。

（３）メッセージによる間接的な操作

オブジェクトの操作は、サーバに対して要求メッセージ送ることで間接的に行われる。オブジェクトをクライアントのアドレス空間にマップし、直接操作することを許すシステムが存在する。機能の観点から見ると、オブジェクトに対する直接操作は、なんら問題がない。しかしながら保護の観点からは、オブジェクトに対する不正な操

作が行われる危険性がある。プログラミング言語においては、コンパイラや実行時システムによりオブジェクトに対する不正な操作が防止される必要がある。

ＲｅＳＣカーネルは、通信プリミティブとして遠隔手続き呼出しを提供する。クライアントは、操作対象のオブジェクトの識別子を引数として、サーバに遠隔手続き呼出しを行うことで、オブジェクトを間接的に操作する。オブジェクトに対する不正な操作は、サーバにより拒絶される。

プロセス間通信機能は、非対称的でよい。すなわち、クライアントがサーバを呼び出す機能があればよく、逆に、サーバからクライアントを呼び出す機能は、不用である。

（４）基底オブジェクト

オブジェクトの堆積を使っても、基底オブジェクトのサーバを作ることができないので、システムが基底オブジェクトを提供しなければならない。２．１０節で述べたように、ＲｅＳＣでは、カーネル中のサーバ群が、基底オブジェクトを提供する。プログラミング言語では、５．２．６項で述べたように、なんらかの方法で基底オブジェクトを実現する必要がある。５．２．６項の例では、Ｃ言語のライブラリ関数を用いて、基底オブジェクトを実現している。

基底オブジェクトは、単に基本的な機能を提供するだけではなく、そのインタフェースも重要である。たとえば、Ｖシステムのカーネルが提供しているような単純なブロック・デバイスというインタフェースでは、不十分である。Ｖシステムの場合、カーネル中のサーバのインタフェースが、カーネル外で構築されるファイル・サーバによる利用を前提として設計されているため、一般利用者による利用には向いていない。この場合、カーネル外の一般利用者向けのファイル・サーバのオブジェクトが、基底オブジェクトとして適している。

（５）１つのサービスについて複数のサーバの存在を許す機能

ＲｅＳＣでは、オブジェクト識別子にサービス識別子ではなくサーバ識別子が含まれているので、自由にサーバを追加することができる。たとえば、ファイル・サービスやネーム・サービスのように、システム中に既に存在しているサーバと同じサービスを提供するサーバを追加することが可能である。その追加されたサーバは、既存のシステムのサーバと全く対等に扱われる。

インターネットのサーバは、１つのサービスについて、各ホストで１つだけ走るように設計されている［１０１］。このサービスとサーバの１対１対応は、オブジェクトの堆積に適していない。たとえば、メール・サービスの場合、ＴＣＰ／ＩＰのポート番号が、２５番と固定されている。（ＵＮＩＸでは、この対応関係が、/etc/services というファイルに格納されている。）オブジェクトの堆積により、既存のメール・サーバの上に堆積する新たなメール・サーバを構築した場合、新たなサーバには、異なるサービス名とポート番号を割り当てなければならない。これにより、一般のク

ライアントは、両者を対等に扱うことができなくなる。

プログラミング言語において、この条件は、1つのプログラム内部で1つの抽象データ型に対して複数の実現が同時に利用可能であることに相当する。例えばC++では、5．2．6項で示したような抽象クラスを用いることにより、この条件を満足することができる。しかしながら、5．2．6項の例では、仮想関数 print の定義が各クラスのインタフェースに繰り返し現れている。このことは、一様なインタフェースを実現する妨げとなる。Eiffel言語における暫定クラス（deferred class）では、この点が改善されている［44］。

NFSが利用可能でないUNIXシステムは、この条件を満たさない［59］。すなわち、UNIXでは、カーネルが提供するファイル・オブジェクトやディレクトリ・オブジェクトの他に、同じインタフェースのファイル・オブジェクトやディレクトリ・オブジェクトを定義することはできない。NFSが利用可能なUNIXでは、利用者プロセスとしてNFSサーバを実現することが可能になっている。よって、この条件を満足する。SunOSでは、利用者プロセスによるNFSサーバを利用して、自動マウント（automount）や半透明ファイル・サービス（translucent file service）を実現している［93］。

（6）下位層のオブジェクトを自由に選択する機能

ReSCのサーバは、新しいオブジェクトを作る手続きにおいて、下層のオブジェクトの識別子を引数として受け取る。基底オブジェクトの場合も、インタフェースとしては0個のオブジェクト識別子を受け取るようになっている。これにより、下位層のオブジェクトを自由に選択することができる。新しいオブジェクトを作る手続きについては、5．4．2項で詳しく述べる。

考察

分散核を持つ分散型オペレーティング・システムの多くは、上記（1）から（4）の条件を容易に満足する。しかしながら、条件（5）は、本質的であり、インターネットのように、この条件を満たさないために、オブジェクトの堆積が実現できないことがある。条件（6）を達成するには、それぞれの分散型オペレーティング・システムのサーバ呼出しのインタフェースを変更して、オブジェクト生成時に下位層のオブジェクトを引数として取るようにしなければならない。

5．4　オブジェクトの堆積による分散型オペレーティング・システムの構築

5．3節では、オブジェクトの堆積を実現する条件について述べた。この節では、分散型オペレーティング・システムを構築する際に望まれる、オブジェクトの堆積を実現する環境の性質について論じる。

５．４．１　望まれる性質

この項では、必要条件ではないが、分散型オペレーティング・システムを構築する際に望まれる、オブジェクトの堆積を実現する環境の性質について述べる。ＲｅＳＣカーネルは、ここであげる条件を全て満足する。

（１）安定なオブジェクトには、安定な識別子

ファイルのような安定（stable）なオブジェクトを扱う場合、オブジェクト識別子も安定していることが望ましい。安定とは、サーバがクラッシュして再起動した後も、変らずに存在し続けることである。たとえば、ファイルやディレクトリは、安定なオブジェクトである。これに対して、プロセスは、揮発性（volatile）のオブジェクトである。安定しているオブジェクトの識別子は、安定していることが望ましい。その理由は、サーバがクラッシュして再起動した場合、クラッシュの前とオブジェクト識別子が変らない方が好ましいからである。これに対して、プロセスのように揮発性のオブジェクトの場合、揮発性のオブジェクト識別子で十分である。

２．６節で述べたＲｅＳＣのオブジェクト識別子の中で、システム・レベルで定義されているサイト識別子とサーバ識別子は、安定である。オブジェクト識別子の安定性は、各サーバにおけるオブジェクト番号の管理方法に依存している。

（２）コネクションレスの通信プリミティブ

通信プリミティブとしては、コネクションレスのものが望ましい。その理由は、上記（１）と同様に、サーバのクラッシュと再起動に耐えられるからである。コネクション付きの場合、サーバの再起動により、コネクションを維持することができない。ＲｅＳＣでも、コネクションレスの通信プリミティブである遠隔手続き呼出しを採用している。

（３）位置依存のオブジェクトと位置独立のオブジェクト

オブジェクトの堆積を使って、分散処理を行うサーバを構築する場合、位置独立のオブジェクトと位置依存のオブジェクトの両方を扱う必要がある。たとえば、ファイルの複製では、１つのファイルの複製を複数のサイトに配置する。そして、あるサイトに障害が発生しても、別のサイト上にある複製（replica）により、そのファイルへのアクセスが可能になる。このことは、位置に依存したオブジェクトを生成する必要があることを示している。逆に、自分自身のオブジェクトは、位置独立でなければならない。それは、せっかく複製を作っても、自分自身が位置依存では、そのサイトが落ちた場合、複製が働かないからである。

２．６節で述べたように、ＲｅＳＣでは、位置依存のオブジェクトと位置独立のオブジェクトの両方が識別可能な識別子を利用している。分散処理を行うサーバについては、５．６節において述べる。

（４）位置独立のプロセス間通信機能

オブジェクトの堆積に基づき、分散型オペレーティング・システムを構築する場合、オブジェクトの位置に独立したプロセス間通信機能が不可欠である（単にオブジェクトの堆積を実現するためには、この機能は不用である。）。２．７節で述べたように、ＲｅＳＣでは、位置依存なオブジェクト識別子により指し示されているオブジェクトに対しても、位置に依存せず遠隔手続き呼出しを行うことができる。

（５）軽量プロセス

軽量プロセスは、サーバ・プロセスのように、複数のクライアントを同時に扱う必要があるプログラムを記述する際に、有効な方法であることが知られている［９８］。それは、独立したクライアントからの要求を並列に処理することができるからである。単一プロセッサにおいても、軽量プロセスを利用することにより、複数のクライアントとの通信処理が並列に実行される。

３章で述べたように、ＲｅＳＣは、マイクロプロセスと仮想プロセッサという概念に基づく軽量プロセス機能を提供する。軽量プロセスを用いたサーバの構成については、５．９．１項で述べる。

（６）要求メッセージの転送（forward）

Ｖシステムのように、遠隔手続き呼出しを転送（forward）する機能があると便利である。その理由は、オブジェクトの堆積では、何もしないで下位層オブジェクトへ要求メッセージを伝えることが多いからである。この場合、転送機能を利用することで、応答メッセージが直接最初のクライアントに戻されるので、メッセージの受渡しのオーバヘッドが軽減される。さらに、サーバを軽量プロセスで実現する場合、同時に活動する軽量プロセスの数を少なくすることができる。その理由は、下位層オブジェクトからの応答を待つ必要がなくなるからである。２．７節で述べたように、ＲｅＳＣカーネルは、要求メッセージの転送機能を持っている。

（７）分離されたネーム・サーバ

ネーム・サーバ（ディレクトリ・サーバ）は、オブジェクトの管理と独立している方が望ましい。その理由は、オブジェクトの堆積では、サーバの種類が非常に多くなるからである。仮にサーバごとに名前空間を構成したとすると、オブジェクトのサーバの種類ごとに名前空間が構成され、利用者インタフェースの質が低下する。ＲｅＳＣのネーム・サーバも、オブジェクトのサーバと分離している。

５．４．２　堆積を生成する手続き

オブジェクトの堆積では、クライアントが堆積の構造を意識して利用することも、意識しないで利用することも可能にすることが望まれる。ＲｅＳＣでは、各サーバに

次の３つのサーバ手続きの定義を義務付けることにより、これを実現している。

（１）oid_t create_with_lower(oid_t lowers[], opt_t options[])
これは、オブジェクトの堆積を直接的に実現する手続きである。この手続きは、与えられたオブジェクト識別子で指定されたオブジェクトを下位層オブジェクトとするような新たなオブジェクトを生成し、生成したオブジェクトの識別子を返す。

（２）oid_t copy(oid_t old)
これは、与えられた引数 old で示されたオブジェクトに対して、いわゆる「深いコピー（deep copy）」を行う事で、新しいオブジェクトを生成する手続きである。下位層オブジェクトがある場合、それらは、再帰的にコピーされる。

（３）oid_t create_like(oid_t old)
これは、手続き（２）と同じく、引数 oid で示されたオブジェクトと同じ形のオブジェクトを生成する手続きである。（２）との違いは、オブジェクトの内容がコピーされない点にある。結果として、空のオブジェクトが生成される。

手続き（１）を使うことにより、クライアントは、堆積の構造を意識してオブジェクトを生成することができる。手続き（２）、または、手続き（３）を使うことにより、クライアントは、そのオブジェクトの堆積の構造を気にすることなく、同じ構造を持つ新しいオブジェクトを生成することができる。手続き（２）は、コピー・コマンドで使われる。手続き（３）は、エディタなどで、編集しているファイルと同じ堆積の構造を持つファイルを作るときに使われる。

５．４．３　ひな形オブジェクト・ベース

ＲｅＳＣでは、頻繁に利用されるオブジェクトを予め生成し、あるディレクトリに登録しておく。このようなディレクトリを、ひな形オブジェクト・ベース（template object bases）とよぶ。たとえば、次の様な名前のファイルが登録されている。

（１）std-file

（２）compressed-std-file

（３）encrypted-compressed-std-file

上記（１）は、５．２．１節で述べた標準ファイル・サーバＳｔｄＦＳの空のオブジェクトである。すなわち、ＳｔｄＦＳの空のファイルが "std-file" という文字列の名前でそのディレクトリに登録されている。（２）は、ＺＦＳのオブジェクトで、その下位層オブジェクトがＳｔｄＦＳの空のオブジェクトである。（３）は、ＺＦＳの

オブジェクトで、下位層オブジェクトとしてＣＦＳのオブジェクトを持っている。ここで、そのＣＦＳのオブジェクトは、下位層オブジェクトとしてＳｔｄＦＳのオブジェクトを持っている。

ここに登録されているオブジェクトは、５．４．２節で述べたサーバ手続き create_with_lower() を利用し、システム管理者がオブジェクトを１個１個積み上げることで作られたものである。一方、一般の利用者がこれらのオブジェクトを使う場合、必要なオブジェクトをコピー・コマンドでコピーして利用する。そのコピー・コマンドでは、サーバ手続き copy() が利用されている。

５．４．４　書込み時コピー

Ｍａｃｈでは、書込み時コピー（copy-on-write）という技術を使って、ＵＮＩＸのプロセス生成システム・コールｆｏｒｋを効率的に実現している［６６］。ｆｏｒｋシステム・コールでは、親プロセスのメモリ・イメージを子プロセスへコピーする。書込み時コピーでは、実際にコピーを行うことを、書込みが起こるまで遅らせることで、書込みが起こらないページが親子間で共有される。そのため、親プロセスと子プロセスの両方の読み書き可能領域に、書込み保護を掛けておく。どちらかのプロセスがその領域に対して書込みを行うと、ページ・フォールトが発生する。カーネルは、ページ・フォールトの処理の中で、新しいページを割り当て、そのページの内容をコピーし、以後、親プロセスと子プロセスが別々のページを参照するようにアドレス変換表を再設定して、実行を再開する。書込みが行われない限り、ページは、親プロセスと子プロセスの間で共有される。この結果、ｆｏｒｋシステム・コールの効率が改善される。

ＲｅＳＣでは、５．４．２節で述べた copy() 手続きの実現において、この手法を利用することができる。この場合、プロセス・オブジェクトだけでなく、ファイルやディレクトリにおいても利用することができる。たとえば、あるファイルがコピーされ、元のファイルに対しても新たに生成されたファイルに対しても書込みが行われなかった場合、それらのファイルの実体が同一のディスク・ブロックを共有するような実現を行うことが可能となる。

５．５　種々の堆積可能オブジェクトとサーバ

この節では、オブジェクトの堆積に基づき高度な機能を提供するオブジェクトとそのサーバについて述べる。ここで述べるサーバのオブジェクトは、堆積可能である。すなわち、それらのサーバのオブジェクトを積み重ねることで、それらのサーバの機能を統合して利用することができる。ＲｅＳＣシステムでは、このような高度な機能は、主に逐次応用プログラムにより利用される。

ここで述べるサーバのいくつかは、ＲｅＳＣ以外の分散型オペレーティング・システムでも利用可能であろう。ただし、５．４節で述べたような機能がない場合、一部の機能を利用することができない。

分類の観点

堆積可能なオブジェクトとサーバは、次のような観点から分類される。

（１）機能

クライアントからメッセージを受け取り、対応する下位層のオブジェクトにメッセージを送る時に、メッセージの内容になんらかの操作をほどこすものをフィルタ・オブジェクト（filter object）とよぶ。これに対して、ほとんど変換を行わないようなオブジェクトを間接オブジェクト（indirection object）とよぶ。そして、それぞれのオブジェクトのサーバをフィルタ・サーバ（filter server）、および、間接サーバ（indirection server）とよぶ。

（２）対象となるオブジェクトの型

下位層のオブジェクトとして、例えば、ファイルという特定の型に対して動作するオブジェクトをファイル・オブジェクトとよぶ。これに対して、下位層に任意の型のオブジェクトを取ることができるオブジェクトを総称オブジェクト（generic object）とよぶ。そして、それぞれのサーバをファイル・サーバ、総称サーバ（generic server）とよぶ。

（３）各オブジェクトごとの下位層のオブジェクトの数

１つのオブジェクトが下位層のオブジェクトとして複数のオブジェクトを利用するものをグループ・オブジェクト（group object）とよぶ。そして、それを管理するサーバをグループ・サーバ（group server）とよぶ。

（４）分散処理を行うかどうか

下位層のオブジェクトが異なるサイト（ネットワーク上に分散したプロセッサ）にあることを利用して分散処理を行うオブジェクトを分散型オブジェクト（distributed object）とよぶ。そして、それを管理するサーバを分散型サーバ（distributed server）とよぶ。

この節では、上記（１）から（３）について、それぞれ代表的なサーバの機能と内部動作を説明する。（４）については、５．６節で述べる。

５．５．１　フィルタ・サーバ

フィルタ・サーバとは、受け付けた要求を下位層オブジェクトへ中継する際に、さまざまな変換処理を行うサーバである。

５．５．１．１　フィルタ・ファイル・サーバ

ＵＮＩＸでは、多くのコマンドがパイプにより結合することで組み合わせて利用することが可能になっている。このようなコマンドは、フィルタ型コマンド（filter）と呼ばれる。フィルタ型コマンドの例としては、tr, sort, sed があげられる。

ＵＮＩＸにおける各フィルタ型コマンドから、堆積可能サーバを構築することができる。たとえば、次のＵＮＩＸのコマンド行：

```
% tr A-Z a-z <from >to
```

に対応して、ファイル中の大文字を小文字に変換するファイル・サーバを構築することができる（図５－７（ａ））。このように、ファイルを対象とし、フィルタ型コマンドと同等の機能を提供するサーバを、フィルタ・ファイル・サーバ（filter file server）と呼ぶことにする。５．２．１項で述べたＺＦＳとＣＦＳも、フィルタ・ファイル・サーバに分類される。

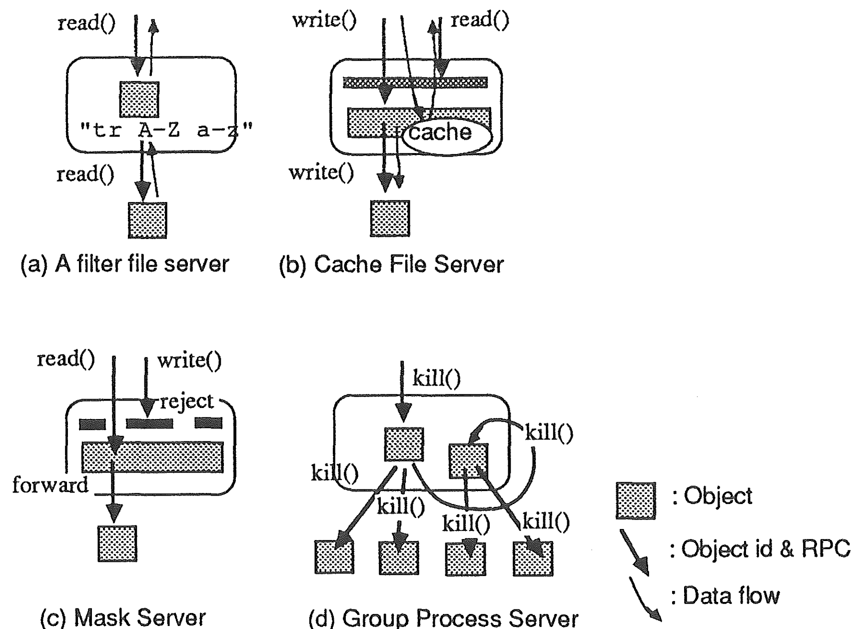


図５－７　さまざまな堆積可能サーバ

Figure 5-7: Various stackable servers.

一般のフィルタ型コマンドには、逆関数が存在しないので、それから構築されるファイル・サーバは、読込み専用（または、書込み専用）になる。５．２．１項で述べた圧縮ファイル・サーバと暗号ファイル・サーバでは、compress, encrypt に対して、uncompress, decrypt という逆関数が存在するので、読み書き可能なファイル・サーバを構築することが可能である。

５．５．１．２　フィルタ・ディレクトリ・サーバ

ファイルと同様に、ディレクトリに対してもフィルタを考えることができる。たとえば、あるディレクトリについて、"*.[ch]" というパタンにマッチするファイルだけを含むようなディレクトリを作ることが考えられる。

５．５．１．３　フィルタ・ウインドウ・サーバ

ウインドウ・システムでは、次の様なウインドウ・サーバを構築することが考えられる。

（１）スケールを変える。
（２）回転する。
（３）色を変える。

このようなサーバを組み合わせることにより、このような機能を持たないウインドウに対して、上記の機能を付加することができる。

５．５．２　間接サーバ

間接サーバとは、受け付けたメッセージの内容を変更することなくそのまま下位層オブジェクトへ転送するようなサーバである。この時、単純に転送するだけでなく、２次的な操作を行う事で、様々な機能を提供することができる。

５．５．２．１　キャッシュ・ファイル・サーバ

キャッシュ・ファイル・サーバとは、オブジェクトに対する要求を受け付けると、対応する下位層オブジェクトに問い合せを出さずに自分自身で答えるようなサーバである。これを、図５－７（ｂ）に示す。クライアントから書込み要求を受け付けると、それをキャッシュに書き込み、下位層オブジェクトに転送する。読込み要求を受け付けると、まずキャッシュを調べ、ヒットした場合はそれを返す。ミスした場合、下位層オブジェクトに読込み要求を送り、その内容を自分自身のキャッシュに書き込みを行った後、クライアントに返す。

このキャッシュ・ファイル・サーバの特徴は、どのようなファイル・オブジェクトについてもキャッシングを行うことができる点にある。堆積可能ではないファイル・サーバでは、そのサーバが管理するファイルについてしかキャッシングを行うことができない。

５．５．２．２　キャッシュ・ディレクトリ・サーバ

キャッシュ・ファイル・サーバと同様に、ディレクトリ・オブジェクトについても、キャッシュ・ディレクトリ・サーバを構築することができる。

５．５．２．３　マスク・サーバ

マスク・サーバとは、オブジェクトに送られてきた要求の中で特定の要求だけを下位層オブジェクトへ転送し、そのほかの要求をさえぎるようなサーバである。たとえば、図５－７（ｃ）では、読込み要求をそのまま転送し、書込み要求をさえぎることで、書込み禁止ファイルが作られている。

マスク・サーバは、総称サーバ（generic server）である。すなわち、このサーバは、任意の型のオブジェクトを扱うことができる。たとえば、ディレクトリ・オブジェクトに対しても、ファイル・オブジェクトに対しても、マスク・サーバを利用することができる。

５．５．３　グループ・サーバ

上記のサーバでは、各オブジェクトについて下位層オブジェクトを１個取るものであった。しかしながら、オブジェクトの堆積において、下位層オブジェクトの数が１個に制限されているわけではない。すなわち、複数の下位層オブジェクトを取ることにより、オブジェクトのグループを作ることができる［７５］。

オブジェクトのグループ化の基本的な考え方は、次の通りである。

（１）１つのオブジェクトに対して、複数の下位層オブジェクトを保持する。

（２）要求を受付けると、それを下位層オブジェクトへ多重送信（multiplex）する。

（３）下位層オブジェクトからの応答を多重受信（demultiplex）し、クライアントへ応答する。

５．５．３．１　ディレクトリの融合

ディレクトリの融合とは、複数のディレクトリの内容を全て含むような１つのディレクトリを作ることである。堆積可能なグループ・サーバの１つとして、ディレクトリの融合を行うサーバを構築することができる。この時、融合されるディレクトリは、単純なディレクトリでも既に融合されたディレクトリでもよい。すなわち、階層的にディレクトリの融合を行うことができる。

５．９．２項において、キャッシングの技術を用いたディレクトリの融合を行うサーバの実現について述べる。

５．５．３．２　ファイルの結合

ファイルの結合とは、複数のファイルを順に結合したような１つのファイルを作ることである。ファイルの結合を行うサーバは、５．５．１．１で述べたフィルタ・フ

ァイル・サーバにおいて、ＵＮＩＸの cat コマンド（concatenate）に対応したものとして構築される。

５．５．３．３　プロセス・グループ

プロセス・グループとは、そのグループに属する複数のプロセスを同時に操作するための仕組みである。オブジェクトの堆積では、堆積可能なプロセス・グループを実現することが可能である。図５－７（ｄ）では、そのプロセス・グループに属するプロセスに対して、プロセスを殺す操作（kill）が行われている。

Ｖシステム［１２］は、カーネルにおいてプロセス・グループの機能を提供している。Ｖシステムでは、プロセス・グループを、マルチキャスト通信を実現するために利用している。Ｖシステムのプロセス・グループは、堆積可能ではない。すなわち、他のプロセスの実現やプロセス・グループ自身をメンバとするグループを構築することができない。

５．５．３．４　ウインドウの飾り付け

ウインドウ・システムでは、ウインドウ・マネジャにおいてタイトル・バー、スクロール・バーなどを付加する、ウインドウの飾り付け（decoration）を実現している。これは、オブジェクトの堆積の枠組みでは、ウインドウの本体、タイトル・バー、スクロール・バーを下位層オブジェクトとしてもつウインドウ・オブジェクトを生成することに相当する。こうして生成されたウインドウ・オブジェクトは、堆積可能であり、他の堆積可能なウインドウ・オブジェクトと組み合せて利用することが可能である。

５．６　分散型堆積可能オブジェクトとサーバ

この節では、分散型堆積可能オブジェクトとそれを管理するサーバについて述べる。分散型堆積可能オブジェクト（distributed stackable object）とは、５．５節の冒頭で述べたように、下位層のオブジェクトが異なるサイトにあることを利用して分散処理を行うようなオブジェクトである。そして、それを管理するサーバを分散型堆積可能サーバ（distributed stackable server）とよぶことにする。

５．６．１　複製ファイル・サーバ

複製ファイル・サーバ（Replication File Server）とは、ファイルの複製を行い、フォールト・トレラントなファイル・サービスを提供するサーバである。複製ファイル・サーバは、１つのオブジェクトについて、複数のサイト上の複数の下位層オブジェクトを利用する。これを、図５－８に示す。このサーバは、書込み要求を受け付けると、全ての下位層オブジェクトへ送る。読込み要求を受け付けると、いくつかの下

位層のオブジェクトに要求を送り、そのうち少なくとも1個のオブジェクトから応答が得られれば十分である。このことにより、ある下位層オブジェクトが存在するサイトが落ちている場合も、他の下位層のオブジェクトの働きにより、そのファイルをアクセスすることが可能となる。

この複製ファイル・サーバの動きから、複製とは、次の様な意味であると考えることができる。

（１）複数のサイトに、同一の内容を持つオブジェクトを生成する。

（２）クライアントに対して、それらがあたかも１つのオブジェクトであるかのごとく見せる。

（３）具体的には、受け付けたメッセージを下位層オブジェクトへ多重送信（multiplex）し、下位層オブジェクトからの応答を多重受信（demultiplex）することで実現することができる。

オブジェクトの複製では、下位層オブジェクトの位置が重要である。たとえば、あるファイル・オブジェクトの全ての下位層オブジェクトが同一サイトにある場合、複製が無効になる。したがって、下位層のオブジェクトを指し示すために、２．６節で述べた位置依存のオブジェクト識別子を用いる。これとは対照的に、複製サーバ自身のオブジェクトを指し示すためには、位置独立のオブジェクト識別子を用いる。

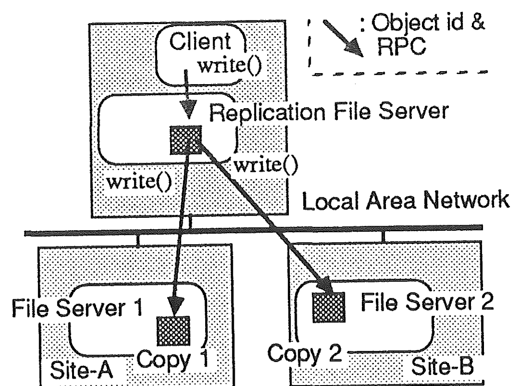


図５－８　複製ファイル・サーバ

Figure 5-8: Replication File Server

５．６．２　間接オブジェクトによるオブジェクトの移動

オブジェクトの移動（object migration）は、凍結（freeze）、解凍（unfreeze）、下位層の変更（change_lower）という３つの手続きを持つ間接オブジェクトを利用して実現することができる（図５－９）。

このオブジェクトは、通常、受け付けた要求を自ら解釈することなく、下位層のオブジェクトへ転送（forward）する。たとえば、読込み要求や書込み要求を下位層へ

転送する。しかしながら、凍結（freeze）、解凍（unfreeze）、下位層の変更（change_lower）という３つの手続きだけは、自ら解釈する。オブジェクトの移動は、次のような手順で実現される。

（１）オブジェクトの移動の管理者（initiator）は、移動するオブジェクト（間接オブジェクト）に凍結要求（freeze）を送る。それ以降、そのオブジェクトへの通常のメッセージは、下位層のオブジェクトへ転送される代わりに、間接サーバの中の待ち行列に保持される。

（２）管理者は、下位層のオブジェクトの内容を新しいオブジェクトにコピーする。

（３）管理者は、新しい下位層のオブジェクトの識別子を引数として、下位層の変更要求（change_lower）を間接オブジェクトに送る。間接オブジェクトのサーバは、そのオブジェクトの下位層のオブジェクトを新しいものに置き換える（図５－９）。

（４）管理者は、その間接オブジェクトに解凍要求（unfreeze）を送る。この時、そのオブジェクトの待ち行列に要求が保持されていた場合、間接オブジェクトのサーバは、それらの要求を新しい下位層のオブジェクトへ転送する。

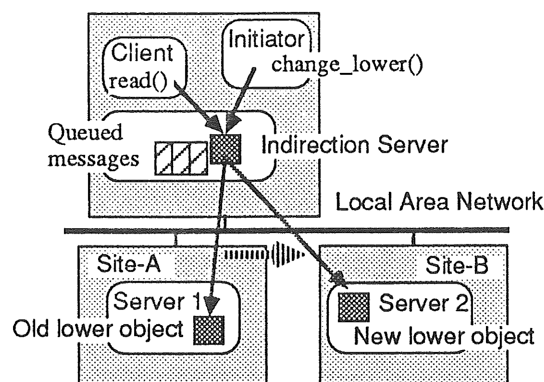


図５－９　間接サーバによるオブジェクトの移動

Figure 5-9: Object migration by an indirection server.

この間接サーバの動きより、オブジェクトの移動とは、次のような意味であると考えることができる。

（１）元のオブジェクトが存在するサイトとは別のサイト上に新しいオブジェクトを作る。

（２）クライアントに対して、２つのオブジェクトがあたかも同一のものであるように思わせる。

（３）具体的には、オブジェクトの移動は、オブジェクトのアクセスを全て１回間接で行う事で実現される。

ここで説明した間接オブジェクトとそのサーバは、５．５節で述べた総称オブジェクト、および、総称サーバである。この間接オブジェクトは、ファイル・オブジェク

トだけでなく、ディレクトリ・オブジェクトの移動にも利用することができる。

ここで説明した間接オブジェクトとそのサーバは、分散型堆積可能オブジェクト、および、分散型堆積可能サーバである。この間接サーバの場合、同じ時刻には、１つのサイト上のオブジェクトしか利用しない。しかしながら、異なる時刻においては、複数のサイト上のオブジェクトを利用することがある。

５．６．３　分散型堆積可能オブジェクト間の競合

分散型堆積可能オブジェクトは、しばしば互いに競合する。たとえば、５．６．２項で述べた間接オブジェクトを下位層のオブジェクトとして、５．６．１で述べた複製ファイル・オブジェクトを生成することを考える。この場合、間接オブジェクトの働きで下位層のオブジェクトが全て１つのサイトに移動した場合、複製が意味をなさなくなる。

分散型堆積可能オブジェクト間の競合は、従来の分散型オペレーティング・システムにおいてマッピング・コントローラを含む並列／分散応用プログラムを走らせた時に生じる、システムと並列／分散応用プログラムの間の競合と同じものである。今後の研究の発展の１つとして、この競合を解消し調和をはかる方法の研究があげられる。

５．７　多重視点

５．５．３項で述べたオブジェクトのグループ化では、複数の下位層オブジェクトを管理するオブジェクトを利用した。逆に、１つの下位層オブジェクトが複数の上位オブジェクトから参照されることも考えられる。これは、オブジェクトの多重視点（multiple views）を実現していることに相当する。たとえば、図５－１０では、１つの下位層のファイル・オブジェクトについて、単純な間接オブジェクト、書込みをマスクするオブジェクト、大文字を小文字に変換するフィルタ・ファイル・オブジェクトを利用することにより、次の３つの視点が得られる。

（視点１）読み書き可能なファイル

（視点２）読込み専用ファイル

（視点３）小文字だけからなるファイル

このように、オブジェクトの堆積を使うことにより、多重視点を直接的に表現することが可能となる。

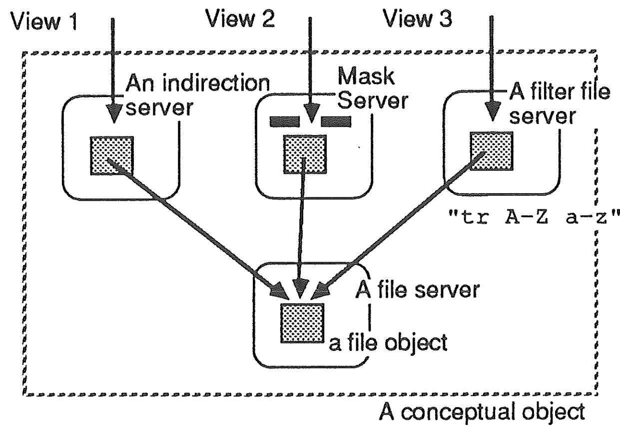


図５－１０　３つの上位層のオブジェクトによる３つの視点の実現

Figure 5-10: An object with three views by three upper objects.

５．７．１　堆積の上下を参照することによる多重視点

全ての堆積オブジェクトは、途中の層をアクセスすることにより多重視点を実現していると考えることができる。図５－１１に、３つのオブジェクトから構成されている堆積オブジェクトの３つの視点を示す。

むやみに堆積の途中のアクセスを許した場合、問題が生じることがある。たとえば、マスク・サーバを用いて、あるファイルを読込み専用にしたとしても、その下位層オブジェクトへのアクセスを許せば、読込み専用にした意味が失われる。このような問題を未然に防ぐためには、下位層オブジェクトのオブジェクト識別子を秘密にする必要がある。このことは、Ａｍｏｅｂａシステム［５４］［５５］におけるケーパビリティと同様に、暗号の技術を用いることで実現可能である。

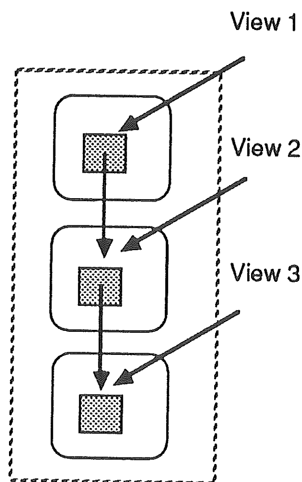


図５－１１　堆積オブジェクトによる３つの視点の実現

Figure 5-11: A stack object with three views.

５．７．２　多重視点とキャッシュ

多重視点を実現する場合、問題となることの１つは、キャッシュの一貫性である［１０７］。たとえば、図５－１０の例において、個々の視点ごとに５．５．２．１で述べたキャッシュ・オブジェクトを積み重ねることを考える。この時、読み書き可能ファイルの視点に対して書込みが行われた場合、他の視点のキャッシュの一貫性が保たれなくなる可能性がある。

この問題の解決方法の１つは、同期型キャッシュ・ファイル・サーバ（coherent cache file server）を構築することである。このサーバでは、図５－１２に示すように、すべての視点のキャッシュが関連付けられて管理される。そして、ある視点に書込みが行われた場合、他の視点のキャッシュが無効化される。

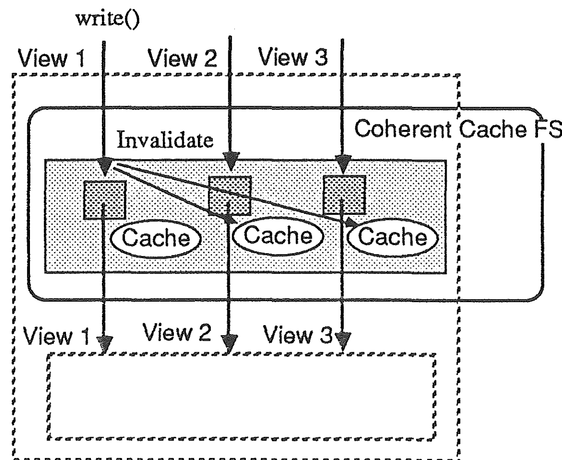


図５－１２　同期型キャッシュ・サーバ

Figure 5-12: Coherent Cache File Server.

５．８　遠隔手続き呼出しスタブ生成器とインタフェース記述言語

分散型オペレーティング・システムにおいて、オブジェクトの堆積を利用する場合、普通、システムにより提供される遠隔手続き呼出しが利用される。多くの分散型オペレーティング・システムでは、遠隔手続き呼出しのインタフェースを記述するための言語と、その言語から遠隔手続き呼出しのスタブを自動的に生成するプログラムを備えている。このプログラムは、スタブ生成器（stub generator）、あるいは、スタブ・コンパイラと呼ばれている。このようなインタフェース記述言語やスタブ生成器の例としては、ＡｍｏｅｂａのＡＩＬ（Amoeba Interface Language）［５４］、ＭａｃｈシステムのＭｉＧ（Mach Interface Generator）［１０５］［５８］、ＳｕｎＲＰＣの rpcgen （RPC protocol compiler）［６９］の記述言語があげられる。

従来のスタブ生成器のインタフェース記述言語は、サーバを中心にインタフェースを記述するものであった。このことは、オブジェクトの堆積を利用する場合、一様なオブジェクトのインタフェースの記述を困難にする。そこで、オブジェクトの堆積に適したインタフェース記述言語の設計を行った。この節では、設計したインタフェース記述言語の構文と意味について述べる。そして、従来のインタフェース記述言語として
ＳｕｎＲＰＣのインタフェース記述言語をとりあげ、比較を行う。

５．８．１　インタフェース記述言語の構文と意味

本インタフェース記述言語において扱えるデータ型は、Ｃ言語、および、ＳｕｎＲＰＣのインタフェース記述言語［６９］を拡張したものになっている。拡張した型構成子は、proc, sproc, set of, server である。付録Ｂ－１に、本インタフェース記述言語の詳細を示す。

本記述言語において、インタフェースは、次のような文の並びにより記述される。

　　＜名前＞ ： ＜型＞ ＝ ＜値＞ ；

この構文は、ｆｕｎ［９］に準じている。ｆｕｎは、１階型付きλ計算に基づく、多相（polymorphism）とオブジェクト指向言語をモデル化するために設計された言語である。ただし、ｆｕｎにおいて関数を意味する "fun" と記述するところを、本インタフェース記述言語では、手続きを意味する "proc", あるいは、"sproc" と記述する。また、ｆｕｎでは、型と値を明確に区別しているのに対して、本インタフェース記述言語では、型を型型の値として扱う。

図５－１３に示した簡単なファイル・サーバの記述を例に、この言語の特徴について述べる。

```
 1: buff_t: type = array of char ;
 2:
 3: kill:    proc( void ) void                      = 14 ;
 4: create: sproc( lower:array of oid_t ) new:oid_t =  1 ;
 5: copy:   sproc( orig:oid_t )  new:oid_t          =  2 ;
 6:
 7: any_server: set of sproc = { copy, create };
 8: any_object: set of  proc = { kill };
 9:
10: read:  proc( where,count:int ) buff_t = 52 ;
11: write: proc( where,count:int;buff:buff_t ) void = 53 ;
12:
13: file: set of proc = any_object + { read,write };
14: read_only_file: set of proc = file - { write };
15: file_server: type = server( any_server + file );
16:
17: StdFS: file_server = 3 ;
18: ZFS  : file_server = 5 ;
19: CFS  : file_server = 6 ;
20: ReadOnlyFS : server( any_server+read_only_file ) = 7 ;
```

図５－１３　簡単なファイル・サーバのインタフェース記述

Figure 5-13: The interface description of simple file servers.

型構成子 proc は、オブジェクト手続きを、型構成子 sproc は、サーバ手続きを定義する。オブジェクト手続き／サーバ手続きとは、５．２節の冒頭で説明したように、それぞれオブジェクト／サーバに対する遠隔手続き呼出しのエントリである。それぞれは、オブジェクト指向言語におけるインスタンス・メソッド／クラス・メソッドに相当するものである。たとえば、第１０行では、手続きreadが２つの整数型の値を入力パラメタとし、バッファ型（buff_t）の値を出力パラメタとするオブジェクト手続きとして定義されている。

オブジェクト手続きは、隠れたパラメタとしてオブジェクト識別子（oid_t）を含むようなサーバ手続きである。たとえば、第１０行に定義されているオブジェクト手続きreadは、次の様なサーバ手続きと等価である。

```
read: sproc( this:oid_t; where,count:int ) buff_t = 52 ;
```

オブジェクト手続き型とサーバ手続き型の変数の値（たとえば、readでは52）は、遠隔手続き呼出しの手続き番号である。この値は、システム全体で一意でなければならない。スタブ生成器は、この一意性を確認する。

この言語の特徴は、型構成子 set of を用いて手続きの集合としてインタフェースを記述する点にある。たとえば、第７行では、any_server という変数にサーバ手続き create とサーバ手続き copy を要素とする集合が代入されている。これは、任意のサーバがサーバ手続き create とサーバ手続き copy をインタフェースに含んでいることを意味している。

この言語では、手続きの集合を扱う演算として、集合の和と集合の差がある。たと

えば、第１３行では、任意のオブジェクトに対して、読み込み、および、書き込みを行う手続きを追加して、普通のファイル・オブジェクトのインタフェースが定義されている。次の行では、読込み専用のファイル・オブジェクトのインタフェースが、普通のファイル・オブジェクトから書込みを行う手続きを削除することによって定義されている。

型構成子serverは、手続きの集合を引数としてサーバ（サーバ識別子型）を定義する。第１５行では、ファイル・サーバが、任意のサーバとしての手続きとファイルの読み書きを行う手続きを受け付けることができるサーバとして定義されている。（オブジェクト手続きは、オブジェクト識別子を隠れた引数として持つような特殊なサーバ手続きであることから、サーバ手続きの集合とオブジェクト手続きの集合の和を求めることができる。）ＲｅＳＣでは、この型の変数の値は、サーバ識別子として利用されるので、全体で唯一でなければならない。スタブ生成器は、その唯一性を確認する。

このスタブ生成器は、以下の点においてオブジェクトの堆積に適している。

（１）サーバではなく、オブジェクトを中心にインタフェースを定義することが可能となっている。このことは、オブジェクトを中心に考えるオブジェクトの堆積に合致している。

（２）オブジェクトの堆積において重要となる一様なインタフェースを持つオブジェクトを容易に定義することができる。

（３）オブジェクト手続きが、特殊なサーバ手続きであるというobject-basedシステムの性質を直接的に反映した記述が可能となっている。

図５－１３の第４行において定義されているサーバ手続きcreateが、オブジェクトを積み重ねる操作を行うものである。これは、５．４．２項で述べたcreate_with_lower を簡略化したものである。この図の create は、引数としてオブジェクト識別子の配列を受け取り、それらを下位層のオブジェクトとして新しいオブジェクトを生成し、その識別子を返すサーバ手続きとして定義されている。

５．８．２　ＳｕｎＲＰＣのインタフェース記述言語との比較

図５－１３に示したファイル・サーバの、ＳｕｎＲＰＣにおけるインタフェースの記述を、図５－１４に示す。図５－１３に示した本記述言語では、第１７行と第１８行において、StdFS と ZFS が同一のインタフェースを持っていることが明示的に示されている。これに対して、図５－１４に示したＳｕｎＲＰＣの記述言語では、StdFS と ZFS の記述が第１８行と第２８行に散在している。さらに、それぞれのサーバにおいて定義されている手続きのインタフェースが同一であるとことを保証する手段が存在しない。

```
 1: struct oid_t {
 2:         site_t          site_id;
 3:         server_t        server_id;
 4:         object_number_t object_number;
 5: };
 6:
 7: typedef opaque buff_t<>;
 8: struct create_arg_t
 9: {
10:         oid_t   lower<>;
11: };
12: struct read_arg_t {
13:         oid_t   this ;
14:         int     where;
15:         int     count ;
16: };
17:
18: program StdFS_PROGRAM {
19:     version StdFS_VERSION {
20:         void_t  KILL(oid_t)             = 14 ;
21:         oid_t   CREATE(create_arg_t)    = 1 ;
22:         oid_t   COPY(oid_t)             = 2 ;
23:         buff_t  READ(read_arg_t)        = 55 ;
24:         void_t  WRITE(write_arg_t)      = 56 ;
25:     } = 1;
26: } = 3 ;
27:
28: program ZFS_PROGRAM {
29:     version ZFS_VERSION {
30:         void_t  KILL(oid_t)             = 14 ;
31:         oid_t   CREATE(create_arg_t)    = 1 ;
32:         oid_t   COPY(oid_t)             = 2 ;
33:         buff_t  READ(read_arg_t)        = 55 ;
34:         void_t  WRITE(write_arg_t)      = 56 ;
35:     } = 1;
36: } = 5 ;
```

図５－１４　ＳｕｎＲＰＣにおける簡単なファイル・サーバのインタフェース記述
Figure 5-14: The interface description of the simple file servers in SunRPC

本記述言語のその他の特徴としては、引数と結果の数が可変であることがあげられる。rpcgenでは、入力パラメタと出力パラメタの数がそれぞれ１個に制限されている。たとえば、図５－１３の第１０行における手続きreadの入力パラメタに対応するものは、図５－１４の第１２行で定義されている構造体read_arg_tにまとめられている。図５－１４の第１３行のthisは、５．８．１項で述べたオブジェクト手続きの隠されたパラメタである。

5．9　堆積可能サーバの実現

この節では、典型的な堆積可能サーバの構造と実現方法を示し、その実現を支援するために構築したライブラリの機能について述べる。ここで示す堆積可能サーバの実現において特徴的なことは、キャッシングの技術を利用している点、および、3章で述べた軽量プロセスを利用している点にある。

堆積可能サーバとライブラリは、現在、ＲｅＳＣカーネルではなく、既存のシステム上に構築したＲｅＳＣカーネルの機能をエミュレートする環境において利用可能となっている。この節では、この環境についても述べる。

5．9．1　典型的な堆積可能サーバの構成要素

典型的な堆積可能サーバは、次のようなモジュールから構成されている（図5－15）。

- 遠隔手続き呼出し受付け軽量プロセス
- 遠隔手続き呼出し入口手続き
- メモリ中のオブジェクトに対する手続き
- 局所オブジェクト管理部

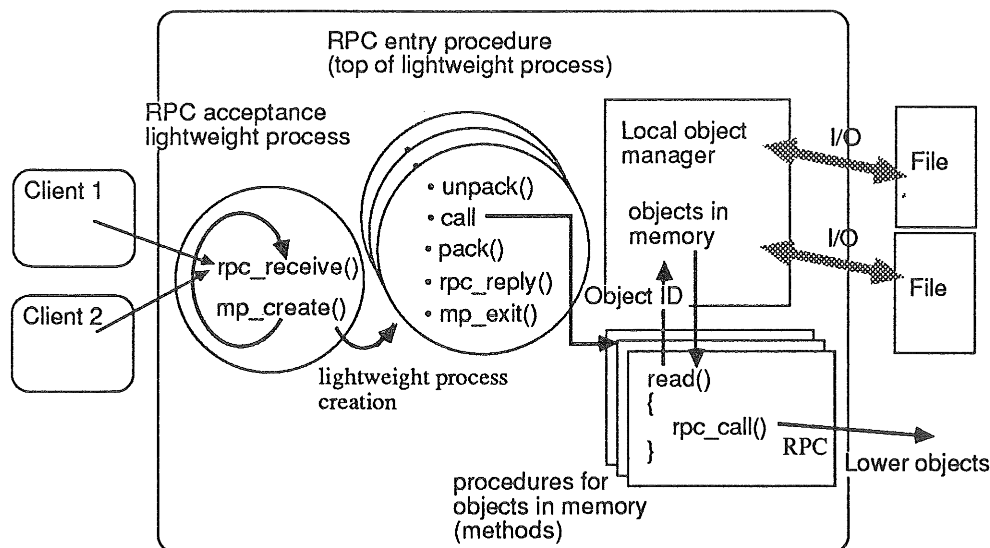


図5－15　典型的な堆積可能サーバの構造

Figure 5-15: The structure of a typical stackable server.

■遠隔手続き呼出し受付け軽量プロセス

遠隔手続き呼出し受付け軽量プロセス（RPC acceptance lightweight process）とは、クライアントからの遠隔手続き呼出しの受付けを行う軽量プロセスである。この

軽量プロセスは、通常、クライアントからの遠隔手続き呼出しを受け付けるカーネル・コールを発行している状態にある。遠隔手続き呼出しを受け付けると、手続き番号を確認し、以下で述べる遠隔手続き呼出し入口手続きをトップとする軽量プロセスを生成する。その手続きが存在しなければ、クライアントへエラーを返す。

■遠隔手続き呼出し入口手続き

遠隔手続き呼出し入口手続き（RPC entry procedure）とは、遠隔手続き呼出しのセッションに対応して最初に呼出される手続きである。これは、軽量プロセスのトップとして、遠隔手続き呼出し受付け軽量プロセスから呼び出される。

遠隔手続き呼出し入口手続きでは、次のような処理を行う。

（１）クライアントから１個にまとめられて送られてきた遠隔手続き呼出しの引数を解く（unpack, unmarshall）。

（２）対応するメモリ中のオブジェクトに対する手続きを呼び出す。

（３）（２）手続きの結果を１個にまとめ（pack, marshall）、遠隔手続き呼出しの結果としてクライアントに返す（rpc_reply()）。

（４）軽量プロセスを終了する。

■メモリ中のオブジェクトに対する手続き

メモリ中のオブジェクトに対する手続き（procedures for objects in memory）とは、オブジェクト指向プログラミング言語、あるいは、抽象データ型を支援しているプログラミング言語における、手続き（メソッド）に対応するものである。通常のプログラミング言語における手続きと比較して、特徴的な処理としては、次のようなものがある。

（１）アクセス権のチェック

サーバとして、遠隔手続き呼出しを行ったクライアントがそのオブジェクトに対してその手続きを実行する権利があるかどうかを調べる。

（２）相互排除

複数のクライアントが１つのオブジェクトに対して同時に遠隔手続き呼出しを発行することがある。並列に実行することができない遠隔手続き呼出しの場合、ロック等を用いて相互排除を行う。相互排除は、軽量プロセス間の同期プリミティブにより実現される。

■局所オブジェクト管理部

局所オブジェクト管理部（local object manager）とは、メモリ中のオブジェクトを管理するモジュールである。このモジュールは、次の様な機能を提供する。

（１）新たにオブジェクトを生成する場合、オブジェクト識別子の割当てを行う。

（２）与えられたオブジェクト識別子に対応するオブジェクトのデータをファイルから読込み、メモリ中のオブジェクトに変換する。ファイルに格納されるオ

ブジェクトのデータを、ファイル構造体（file structure）とよぶ。

（３）メモリ中のオブジェクトをファイルへ構造体として保存する。

（４）メモリ中のオブジェクトのキャッシュを管理する。

このモジュールは、ＵＮＩＸのアイノード（inode）とディスク・アイノードの管理とよく似ている。このモジュールにおけるオブジェクト識別子、メモリ中のオブジェクト、ファイル構造体が、ＵＮＩＸにおけるアイノード番号、アイノード、ディスク・アイノード（disk inode）に対応する。ＵＮＩＸとの違いを、以下にまとめる。

・ＵＮＩＸでは、アイノードもアイノードから参照されるデータ（ファイルのデータ）も、ディスク・ブロックに格納される。これに対して、本モジュールでは、ディスク・ブロックではなく、ファイルに格納される。

・ＵＮＩＸでは、アイノード番号は、カーネル内にあり、利用者から直接利用されることはない。これに対して本モジュールが扱うオブジェクト識別子は、利用者により操作される。これにより、利用者によるオブジェクト識別子の改変（forging）や偽造（tampering）に対処する必要がある。

図５－１６に、典型的な堆積可能サーバにおけるオブジェクト識別子の構造と、２つのファイルを用いたオブジェクトの格納の様子を示す。メモリ中のオブジェクトのデータは、固定長部分（fixed length part）と可変長部分（variable length part）に分けられ、別々のファイルに保存される。オブジェクト識別子のオブジェクト番号の構造は、個々のサーバにより決定される。一般には、次の３つ組から構成される。

＜シリアル番号，ノード番号，チェック＞

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。ノード番号は、ＵＮＩＸのアイノード番号と同様に、固定長部分を格納するファイル中の構造体の位置を示すインデックスとして利用される。最後のチェックは、オブジェクト識別子の改変と偽造を防ぐためのものである。この部分を利用して、Ａｍｏｅｂａシステム［５４］［５５］におけるケーパビリティで用いられている技術と同様の技術により、利用者によるオブジェクト識別子の改変と偽造に対処する。

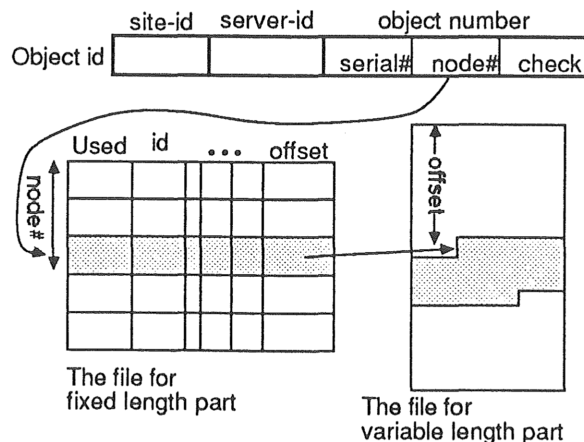


図５－１６　オブジェクト識別子とファイル構造体

Figure 5-16: An object id and its file structure.

以下に、与えられたオブジェクト識別子をメモリ中のオブジェクトに変換する手順を示す。

（１）オブジェクト番号の中のチェックを利用して、オブジェクト識別子が破損していないかどうかを調べる。破損していた場合は、エラーを返す。

（２）メモリ中のキャッシュを検索する。ヒットした場合は、それを返す。

（３）オブジェクト番号の中のノード番号から、オブジェクトのデータの固定長部分を読み込む。固定長部分には、オブジェクト識別子が含まれている。これと現在変換中のオブジェクト識別子を比較して、等しいことを確認する。等しくない場合は、エラーを返す。

（４）必要ならば、オブジェクトのデータの可変長部分を読み込む。

（５）ファイルに格納されないオブジェクトのデータを初期化する。

（６）オブジェクトをキャッシュに登録する。

５．９．２　キャッシングを利用した融合ディレクトリ・サーバの実現

この項では、５．５．３．１において述べたディレクトリの融合を行うサーバを例に、キャッシングを利用した堆積可能サーバの構築について述べる。ここでは、特徴的な手続きであるcreate_with_lower, dir_readdir, dir_lookup を取上げる。

create_with_lower: この手続きは、５．４．２項で説明したように、下位層のオブジェクトとなるオブジェクトの識別子を引数として、新しいオブジェクトを生成するものである。この手続きが呼び出されると、サーバは、与えられたオブジェクトがディレクトリかどうかを確認する。局所オブジェクト管理部を呼び出して、オブジェクト識別子を割り当て、メモリ中にオブジェクトを生成する。最後に、割り当てられたオブジェクトの識別子をクライアントに返す。

dir_readdir: この手続きは、ディレクトリの一覧を返すものである。まず、そのディレクトリに対するキャッシュがメモリ中に存在するかどうか調べる。存在しない場合、下位層のオブジェクトに dir_readdir メッセージを送り、それらの結果を融合し、結果をキャッシュとしてメモリ中に保存する。キャッシュが存在する場合、下位層のオブジェクトの最終更新時刻を調べ、キャッシュの整合性を確認する。キャッシュが古くなっていた場合、それを再構成する。最後に、キャッシュの内容をクライアントに返す。

dir_lookup: この手続きは、文字列の名前を受け取り、対応するオブジェクト識別子を返すものである。まず、dir_readdir と同様に、整合性の取れたキャッシュを作る。次に、与えられた文字列をキーとしてキャッシュを検索する。最後に、見つかったエントリのオブジェクトの識別子をクライアントに返す。

５．９．３　外部のオブジェクトへのキャッシュの保存

前項（５．９．２項）で述べたディレクトリの融合の場合、扱う対象がディレクトリであるため、全てのキャッシュをサーバのメモリ中に置くことが可能である。しかしながら、大きなデータを必要とする場合、キャッシュをサーバのメモリ中に格納することは困難になる。この場合、キャッシュをサーバの外部のオブジェクトに格納する。

図５－１７は、５．２．１項で述べた圧縮ファイル・サーバ（ＺＦＳ）の、キャッシュを用いた実現を示している。この実現では、解凍したデータをキャッシュとしてサーバの外部のファイル・オブジェクトに格納している。ＺＦＳのオブジェクトは、クライアントから読込み、あるいは、書込み要求を受け付けると、下位層のオブジェクトに読込み要求を送り、圧縮されたデータを読み込む。次に、そのデータを解凍し、外部のファイル・オブジェクト（一時ファイル（temporary file））に書き込む。そして、受け付けた読込み要求や書込み要求を一時ファイルへ転送（forward）する。クライアントからのアクセスが途絶えた場合、ＺＦＳサーバは、一時ファイルの内容を読み込み、圧縮し、対応する下位層のオブジェクトへ書き戻す。

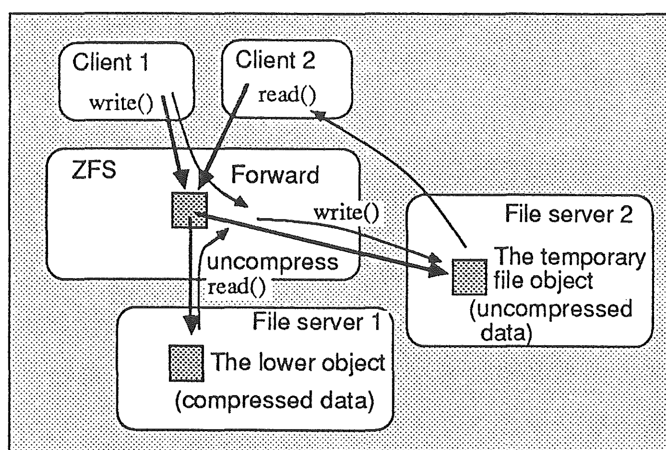


図５－１７　外部のオブジェクトへキャッシュを保存するＺＦＳの実現
Figure 5-17: The implemenation of ZFS by using external objects as caches.

５．９．４　オブジェクトの堆積を実現する環境

オブジェクトの堆積を実現する環境を、ＳｕｎＲＰＣとＮＦＳを利用して実現した。２章で述べたＲｅＳＣの遠隔手続き呼出しを、ＳｕｎＲＰＣを用いてエミュレートした。基底オブジェクトを提供するファイル・サーバとして、ＮＦＳサーバを直接呼び出すものを実現した。

この環境では、ＲｅＳＣのオブジェクト識別子は、次の様に実現されている。

・サイト識別子：
インターネット・ホスト・アドレス（４バイト）を利用した。

・サーバ識別子：
各サーバについて、４バイトの小さな整数を割り当てた。

・オブジェクト番号：
３２バイト長の不定型（opaque）（バイト列）を利用した。これは、ＮＦＳにおけるオブジェクト識別子である file handle と同じ長さである［５９］。これにより、オブジェクト番号にＮＦＳの file handle を格納することで、ＮＦＳサーバを直接呼び出すような基底ファイル・サーバの実現が容易になっている。

ＳｕｎＲＰＣのライブラリ関数では、静的変数を利用している。したがって、３．６．４項で述べたように、そのままでは、マイクロプロセス・ライブラリと共存できないものがある。そこで、ＳｕｎＲＰＣのソース・コード確認して、共存不可能なものを削除した。

ＳｕｎＲＰＣでは、スタブ生成器（rpcgen）により、サーバの主プログラムが自動生成される。しかしながら、この主プログラムは、軽量プロセスを利用するようにはなっていない。これに代わるものとして、５．９．１項で述べた、遠隔手続き呼出し受付け軽量プロセスと遠隔手続き呼出し入口手続きをライブラリ関数として用意した。サーバの開発者は、従来のＳｕｎＲＰＣのサーバ側の手続きに相当する、メモリ中のオブジェクトに対する手続きを開発する。そして、その手続きの表を、そのライブラリ関数に渡すことで、サーバを記述する。

局所オブジェクト管理部は、各サーバにより個別に記述されるものである。しかしながら、オブジェクトのデータをファイルに格納する部分には、共通な操作が多い。そこで、それを行うものを、ライブラリ関数として用意することにした。構造体のファイルへの格納には、ＳｕｎＲＰＣにおける標準データ表現形式であるＸＤＲ（eXternal Data Representation）を利用した。

５．１０　関連した研究

５．２節では、オブジェクトの堆積のモデルとして特徴を、サーバの堆積、層化プログラミング、ＵＮＩＸのパイプ、抽象データ型、継承、委譲、メタ・オブジェクトとの比較しながら述べた。この節では、その他の関連した研究との比較を行う。

５．１０．１　Ｐｌａｎ９

Ｐｌａｎ９は、ＡＴ＆Ｔベル研究所で開発されているオペレーティング・システムである。Ｐｌａｎ９では、ディレクトリの融合を行う機能をカーネルにおいて提供している。本システムでは、カーネルの外の堆積可能サーバにより、ディレクトリの融合を実現している。Ｐｌａｎ９のディレクトリの融合の機能は、堆積可能ではない。すなわち、カーネルが扱う１種類のディレクトリについてのみ融合を支援している。堆積可能サーバでは、様々な種類のディレクトリを融合することが可能になっている。

Ｐｌａｎ９では、ウインドウ・システムのインタフェースとして、ファイル・サーバと同一のインタフェースを利用している。これにより、ファイルを操作するプリミティブと同一のプリミティブを用いて、ウインドウを操作することが可能になっている。

Ｐｌａｎ９では、ウインドウ・システムの開発において、サーバの堆積と似た手法を利用している。新たにウインドウ・システムを開発する場合、まず、既存のウインドウ・サーバ上で動作するウインドウ・サーバを開発する。この瞬間には、２つのバージョンのウインドウ・サーバが堆積している。このような環境において、ウインドウ・システムのクライアントを開発する。その後、新しいウインドウ・サーバを単独で起動する。ただし、サーバの堆積を利用しているのは、ウインドウ・システムの開発の局面だけであり、５．５．１．３で述べたフィルタ・ウインドウ・サーバのようなサーバは、利用されていない。

５．１０．２　interposition

文献［３３］［３４］では、interposition という概念が提案されている。interposition とは、仮想計算機、あるいは、５．２．９項で説明したメタ空間に近い概念である。interposision では、プロセスが発行したシステム・コールやオブジェクトに対するアクセスを、捕捉（trap）し、プロセスのサーバへ伝える。サーバでは、捕捉したシステム・コールやオブジェクトの呼出しに変換を加えたり、あるいは、チェックを行い、再び、別のシステムやサーバを呼び出す。この時、このシステムやサーバの呼び出しが、さらに別のサーバにより捕捉され、変換が加えられる可能性もある。

５．２．９項において述べたように、メタ空間の階層とオブジェクトの層は、互いに直交した概念である。同様に、interposition とオブジェクトの堆積も直交した概念である。たとえば、オブジェクトの堆積では、複数のサーバを同時に利用することができるが、interposition では、複数の仮想計算機を同時に利用することはできない。一方、interposition では、全てのシステム・コールやオブジェクトへの参照を捕捉することが可能であるが、オブジェクトの堆積では、不可能である。

５．１０．３　多重継承

オブジェクト指向プログラミング言語では、複数のクラスが提供する機能を同時に

利用するために、多重継承（multiple inheritance）が用いられる。多重継承では、メソッドを組み合わせる方法（メソッド結合）が重要な研究課題となっている。Ｆｌａｖｅｒでは、before、after、primary 等の指定を行い、多重継承におけるメソッドの実行順序を決定している［１０４］。この方法は、非常に複雑でわかりにくいものであった。新Ｆｌａｖｅｒでは、利用者によるメソッド結合の定義が可能になっている。

オブジェクトの堆積における、下位層のオブジェクトの同名の手続きを呼び出すことが、（多重）継承におけるスーパークラスの同名のメソッドを呼び出すことに対応する。オブジェクトの堆積では、個々の手続きの実現において、手続き呼出しの順序だけでなく、呼出しそのものを行うかどうかを自由に制御することができる。また、同名の手続きを呼び出さず、他の名前の手続きを呼び出すことで、その手続きを実現することもある。例えば、５．９．２項で述べた融合ディレクトリ・サーバの dir_lookup 手続きの実現では、下位層の dir_lookup 手続きを呼び出すのではなく、dir_readdir を呼び出している。このように、オブジェクトの堆積では、一般的なメソッド結合のパタンを定義することはできないが、個々の手続きにおいて、自由に手続きを組み合わせることが可能になっている。

実時間Ｍａｃｈのスケジューラの実現では、オブジェクトの堆積と類似の技術が利用されている［５６］［１０２］。実時間システムにおけるスケジューラを構築する場合、様々なスケジューリングの方針（policy）を組み合わせる必要がある。実時間Ｍａｃｈでは、方針を実現するものが、一様な外部仕様を持つオブジェクトとして定義されている。利用者は、それらのオブジェクトをいくつかのリンクを用いて結合することで、それらのオブジェクトが提供する機能を組み合わせて、実時間スケジューラを構築する。

この方法は、文献［５６］では、多重継承の柔軟な実現方法として提案されている。しかしながら、その論文で提案されている方法は、５．２．７項で行った継承の定義に従えば、継承ではなく、オブジェクトの堆積に分類される。すなわち、継承では、インタフェースだけでなく、メソッド（手続き）の実体（プログラム）が共有されることが必要である。文献［５６］で提案されている方法では、インタフェースが一様になるが、メソッドは、共有されない。

５．１０．４　手続きの堆積

オブジェクトの堆積では、オブジェクト識別子、あるいは、オブジェクトへのポインタを用いて、複数のオブジェクトを結合し、それらのオブジェクトが持つ機能を組み合わせて利用する。オブジェクトの堆積では、オブジェクト、および、オブジェクト識別子という概念が必須である。これに対して、オブジェクトやオブジェクト識別子を一切用いることなく、手続きを積み重ねて、複数の手続きの機能を統合して利用する方法がある。これを手続きの堆積（procedure-stacking）とよぶことにする。

手続きの堆積を支援しているプログラミング言語として、Forth がある［２０］［３１］。Forthでは、手続き（Forthでは、ワードと呼ばれる）の中で同名の手続きを呼び出した場合、再帰呼出しではなく、直前に定義された同名の手続きの呼出しとなる。次の例では、（２）において W1 を呼び出しているが、それは、自分自身の呼出し（再帰呼出し）ではなく、（１）において定義されている W1 の呼出しになる。

```
: W1 ＜処理１＞ ;                              ＼（１）
: W1 ＜処理２＞ W1 ＜処理３＞ ;                ＼（２）
```

ここで、コロン ":" に続く文字列が手続きの名前である。それ以降、セミコロン ";" までが手続きの処理を表す。バックスラッシュ "＼" 以降は、コメントである。上記の定義は、次の定義と等価である。

```
: W1 ＜処理２＞ ＜処理１＞ ＜処理３＞ ;         ＼（３）
```

手続きの定義中で同名の手続きを呼び出した場合に再帰呼出しではなく既に定義されている別の手続きを呼び出すという意味付けは、オブジェクト指向プログラミング言語における差分プログラミングと同等の機能を提供する。たとえば、次の例では、Ｃ言語の構文を用いて、マルチプロセッサ用にロック機能を付加した１文字出力手続きを定義している。

```
putchar( c )
{
        lock( &putchar_lock );
            putchar( c );        /* call the non-lock version of putchar */
        unlock( &putchar_lock );
}
```

この例では、putchar 手続き（Ｃ言語では、関数）にロックの機能を付加している。

オブジェクトの堆積と手続きの堆積を比較すると、オブジェクトの堆積では、オブジェクト識別子を使って手続きを呼び出す順序を制御している点に特徴がある。これにより、同じサーバが管理するオブジェクトでも、オブジェクトごとに組み合わされる手続きの種類と順序が異なる。

５．１１　まとめ

オブジェクトの堆積とは、object-based システムにおいて、複数のオブジェクト

が持つ機能を統合して利用するためのモデル化の手法である。オブジェクトの堆積は、非常に単純なモデルであるが、非常に強力であり、応用範囲も広い。この章では、オブジェクトの背景となる技術と基本概念について述べた。重要な基本概念は、積み重ねること、および、一様なインタフェースであった。次に、オブジェクトの堆積による分散型オペレーティング・システムの構築について述べた。オブジェクトの堆積により、キャッシングやオブジェクトの移動といった高度な機能が提供される。これらのサーバの実現の特徴は、キャッシングの技術が利用されている点にある。関連した研究として、サーバの堆積、層化プログラミング、継承、委譲、メタ・オブジェクト等を取りあげ、比較を行った。このようにして、オブジェクトの堆積の特徴と有効性を明らかにした。

第６章　結論

本研究では、並列／分散応用プログラム、および、逐次応用プログラムを対象としたオペレーティング・システムの設計と実現を行った。第１章では、研究の背景、目標、および、関連した研究について述べた。

並列／分散応用プログラムの特徴は、内部に複数の処理単位と資源割当てモジュールを含んでいる点にある。これに対して、逐次応用プログラムは、処理単位を１つだけ含み、資源割り当てモジュールを含んでいない。従来のオペレーティング・システムは、主に集中型システムで開発された逐次応用プログラムを想定してシステムの機能が設計されていた。このため、並列／分散プログラムという、内部に資源割当てモジュールを含むプログラムへの対応が不十分であった。具体的には、オペレーティング・システム内部の資源割当てモジュールの方針と並列／分散応用プログラムの資源割当てモジュールの方針が競合することがあった。これをシステムと並列／分散応用プログラムの間の競合とよんだ。また、並列／分散応用プログラムにとっては、オペレーティング・システムの資源割当てモジュールが不用であるが、逐次プログラムにとっては、必要となる。これを逐次応用プログラムと並列／分散応用プログラムの間の競合とよんだ。

本研究では、逐次応用プログラムに加えて、並列／分散応用プログラムという、内部に資源割当てモジュールを含むプログラムを対象とした分散型オペレーティング・システムの設計と実現を行った。このシステムをＲｅＳＣと名付けた。ＲｅＳＣが対象とするハードウェア環境は、共有メモリ型マルチプロセッサと単一プロセッサが高速ＬＡＮにより結合されたものである。
ＲｅＳＣが対象とする応用プログラムは、並列／分散応用プログラム、および、逐次応用プログラムである。

本研究の目的を、上記の２種類の競合を解消し、次の２種類の調和を実現することに設定した。

（１）オペレーティング・システムと並列／分散応用プログラムの間の調和：
　オペレーティング・システムは、並列／分散応用プログラムと競合することなく、そのような応用プログラムに対して、高速化や高信頼性を実現するためにマッピングやスケジューリングを制御する機能を提供する。

（２）逐次応用プログラムと並列／分散応用プログラムの間の調和：
　これらの応用プログラムが、必要に応じてシステムの機能を利用することができるようにする。システムを集中型システムとも分散型システムとも見ることを可能にする。

この目標を設定したことは、従来の分散型オペレーティング・システムの定義と研究

目標を設定し直すことを意味する。従来の分散型オペレーティング・システムは、分散透明性が実現され、仮想的な集中型システムというモデルを提供するオペレーティング・システムとして定義されていた。そして、分散透明性を実現することが研究の目標であった。本研究では、並列／分散応用プログラムを考慮して、従来の定義と目標であった透明性の実現を、調和の実現へ改めた。

上記の調和を実現するために、本研究では、従来の分散型オペレーティング・システムを、カーネル上で走る並列／分散応用プログラムとして構成する方法を用いた。これは、次の3つの意味を持つ。

（1）選択可能化：

各応用プログラムがシステムの機能を選択的に利用することを可能にする。

（2）隔離：

競合する主体の間の情報交換を制限し、独立に動作させる。

（3）オペレーティング・システムの応用プログラム化：

システムの構成要素と応用プログラムが対等な立場で計算機資源を利用する。オペレーティング・システムを並列／分散応用プログラムと同一の技術を用いて構築する。

本論文の第2章から第5章では、様々な局面で現れる競合に対する調和を図る方法を示した。

第2章では、本システムの全体構成とカーネルの機能について述べた。本システムは、カーネル、外部サーバ、並列シェル、ライブラリから構成される。カーネル化されたカーネルと比較して、本システムの特徴は、効率を重視する並列／分散応用プログラムのために、カーネル中に完全なサーバを置いている点にあった。また、システム・レベルのオブジェクト識別子は、位置独立のオブジェクトと位置依存のオブジェクトの両者を識別することが可能になっている。このオブジェクト識別子と、位置独立の遠隔手続き呼出しの発行により、分散型ハードウェア環境を、分散型システムとも集中型システムとも見ることが可能になることを示した。

カーネルは、オペレーティング・システムの中で唯一、応用プログラムによる選択ができない部分である。本研究では、選択可能化を進めるにあたり、並列／分散応用プログラムがしばしば自ら制御することを好む機能をカーネル外の構成要素で提供することにした。同時に、公平な資源割当て、応用プログラム間の通信、および、保護を実現する機能をカーネルに残した。これにより、オペレーティング・システムにおけるカーネルの論理的な位置付けを明らかにした。

第3章では、軽量プロセスの実現方式について述べた。軽量プロセスは、多重プログラミング・システムにおける、各応用プログラム内部の並列処理の単位としてのプロセスである。本研究では、マイクロプロセスと仮想プロセッサという概念を用いて軽量プロセスを実現する方式を提案した。これは、従来のカーネル制御方式と比較し

て、より効率的な実現が可能になっている。また、コルーチン方式と比較して、並列処理が可能になっているという利点がある。

本軽量プロセス実現方式の特徴は、応用固有の軽量プロセス間の同期・通信プリミティブの開発、および、スケジューラの開発が容易である点にある。これは、軽量プロセス機能が層構造を持つライブラリにより実現されていることによる。カーネル・レベルの仮想プロセッサにより、利用者レベルの軽量プロセスを実行する方式は、本研究の他にも提案された。これらの方式と比較して、本研究の特徴として、まず、層構造のライブラリにより利用者レベルの軽量プロセスが実現されていることがあげられる。次に、本仮想プロセッサの機能の特徴は、各仮想プロセッサごとに固有のメモリ領域が存在することである。これにより、利用者レベルの軽量プロセスの実現が容易になり、かつ、並列処理の効率を改善することが可能となる。

本研究では、仮想プロセッサ機能を提供するカーネルの構成法を提案した。この構成法の特徴は、カーネル内の軽量プロセスを、固有のメモリ領域を持つ実プロセッサにより実行する点にある。この方式は、ＲｅＳＣシステムの仮想プロセッサだけではなく、他の外部仕様を持つ仮想プロセッサの実現においても利用可能である。この方式では、カーネルの一部を利用者プロセスとして開発することが可能となる。これにより、カーネルの動作の確認とデバッグが容易になる。

実現した軽量プロセスと他の軽量プロセス、および、既存のオペレーティング・システムの重量プロセスとの基本的な性能を比較する実験を行った。また、データベースの並列処理システムを例として、応用固有の同期・通信プリミティブの開発、および、スケジューラの開発を行い、開発したシステムの性能を調べる実験を行った。これらの実現と実験を通じて、提案した軽量プロセス実現方式の有効性を示した。

軽量プロセスの実現についても、選択可能化が適用されている。それは、層構造を持つライブラリに端的に現れている。一方、応用プログラムによる選択が不可能なカーネルには、公平なＣＰＵ資源を実現するプロセス・スケジューラを置いた。システムのスケジューラと複数の応用プログラムのスケジューラは、小数のカーネル・コールによってのみ結合されている。こうしてスケジューラを互いに隔離することにより、それらの間の競合を防止している。また、仮想プロセッサを提供するカーネルの構成法には、オペレーティング・システムの応用プログラム化が適用されている。すなわち、カーネル自身をカーネル上で動作する並列プログラムと同一の技術で開発している。結果として、軽量プロセスに関して、選択可能化、隔離、および、オペレーティング・システムの応用プログラム化が有効であることが、確認された。

第４章では、マッピング・コントローラについて述べた。マッピング・コントローラとは、並列／分散応用プログラムの中に在り、ネットワーク上のサイト（プロセッサ）に対して、プロセスやデータの配置の方針を決定するモジュールである。本研究では、カーネルには全くマッピング・コントローラを含めず、各応用プログラムごとにマッピング・コントローラを開発する方針を提案した。これにより、並列／分散プ

ログラムとシステムの間のマッピングに関する競合が解消される。システムは、応用固有のマッピング・コントローラの開発を支援するために、資源の位置や資源の利用状況の情報を積極的に提供する。

マッピング・コントローラを持たない逐次応用プログラムに対しては、システムは、並列シェルを提供する。並列シェルは、互に協調して動作するコマンド群を並列に実行するためのコマンド・インタプリタである。並列シェルは、マッピング・コントローラを含まない応用プログラムに代わって、マッピングの最適化を行う。

ネットワークで結合されたワークステーション上に並列シェルのプロトタイプを実現した。そして、実現した並列シェルを利用して、並列処理の実験を行った。その結果、プロセス間の結合関係を利用することで、並列処理の効率が改善されることが確認された。

マッピングにおいて、カーネルからマッピング・コントローラの機能を並列シェルに移動した点に、選択可能化が適用されている。一方、それらのマッピング・コントローラは、互いに隔離されており、プロセスやファイルを生成することで資源の利用状況を変化させたり、カーネルに資源の利用状況を問い合せたりする以外に、情報交換を行わない。また、システムの構成要素である並列シェルが、他の並列／分散応用プログラムと対等な立場で動作する点に、オペレーティング・システムの応用プログラム化が適用されている。結果として、マッピングにおいて、選択可能化、隔離、および、オペレーティング・システムの応用プログラム化が有効であることが確認された。

第５章では、オブジェクトの堆積について述べた。オブジェクトの堆積とは、object-basedシステムにおいて、複数のオブジェクトの機能を組み合わせて利用するためのモデル化の手法である。ＲｅＳＣシステムでは、主に逐次プログラムに対して高度な機能を提供するために、外部サーバを統合する場面においてオブジェクトの堆積が利用される。第５章では、オブジェクトの堆積の基本概念、それを利用する条件、分散型オペレーティング・システムにおける様々なサーバについて述べた。また、オブジェクトの堆積に適した遠隔手続き呼出しのインタフェース記述言語や、堆積可能サーバの構造についても述べた。こうして、オブジェクトの堆積の特徴と有効性を明らかにした。

オブジェクトの堆積は、選択可能化を直接的に実現している。応用プログラムは、堆積するオブジェクトを選択することで、必要な機能を選択することができる。また、外部サーバの存在は、オペレーティング・システムの応用プログラム化が行われていることを示している。

本研究では、新しいオペレーティング・システムの全体の機能の提案を行った。そして、それらの機能の部分的な実現を行った。具体的には、軽量プロセスを実現するライブラリ、仮想プロセッサを提供するカーネル、並列シェルのプロトタイプ、いく

つかの堆積可能サーバの実現を行った。

選択可能化、隔離、および、オペレーティング・システムの応用プログラム化は、マッピングとスケジューリングにおける２つの調和を実現する上で有効な手段であることが示された。しかしながら、本研究において、全ての問題が解決されたわけではない。残された問題としては、スケジューリングに関しては、３．１３節において述べた実時間性の保証があげられる。マッピングに関しては、５．６節において述べた複数の分散型堆積可能オブジェクトのマッピング・コントローラの間の協調があげられる。これらの問題を解決するには、隔離の度合いを下げる必要があると思われる。

本研究では、主にプロセッサ資源に関して、競合を解消し調和を実現する方法を提案した。今後の研究の方向としては、プロセッサ以外の計算機資源に関して、競合を解消し調和を実現する方法を研究することがあげられる。たとえば、ディスク資源やメモリ資源があげられる。これらの資源については、並列／分散応用プログラムが現れる以前から、データベース応用プログラムの効率的な処理を実現する局面において個別に対応がなされてきた。今後は、複数の利用者や応用プログラムを扱う局面において、統合的な対応が可能な方法を開発することが望まれる。

ネットワーク資源に関する調和の研究も興味深い。応用固有の情報を利用することにより、効率的なネットワーク通信プリミティブを開発することが可能となる。一方、ネットワーク通信プリミティブは、カーネルにおいて実現する方が効率がよいが、信頼できない応用プログラムをカーネルに置くことは難しい。このような競合状態を解消し、調和を図る方法も、重要な研究課題となるであろう。

軽量プロセスに関しては、ネットワーク機能との統合に関する研究が考えられる。たとえば、マッピング・コントローラにおいて、軽量プロセスの移動の技術を用いて、軽量プロセスを単位として負荷均衡を実現する方法が考えられる。

オブジェクトの堆積では、クライアントとサーバが非対象になっていた。すなわち、クライアントは、オブジェクト識別子を保持し、サーバを知っているが、サーバは、クライアントの情報を一切保持していない。オブジェクトの堆積の変形として、クライアントとサーバの区別がなくなり、オブジェクトが対等に結合される方法が考えられる。これを、オブジェクトの連結（object-cascading）と呼ぶことにする。オブジェクトの連結は、局面に応じてクライアントとサーバの立場が入れ替わるようなウインドウ・システムで有効であると思われる。

今後、オペレーティング・システムは、並列／分散応用プログラムを始めとして、新たに登場する様々な種類の応用プログラムの要求に対応する必要がある。オペレーティング・システムを必要とする並列／分散ハードウェアの進歩も著しい。このような応用プログラムの要求とハードウェアの変化に対応するために、オペレーティング・システムの研究は、ますます重要になってくると思われる。

オペレーティング・システムにおいて、計算機資源の有効利用を図るための資源割当ての最適化は、旧来から最も重要な仕事の１つであった。しかしながら、並列／分

散応用プログラムが登場した現在、オペレーティング・システムは、もはや資源割当ての最適化を行う唯一のプログラムではない。今後、資源割当てに関して、オペレーティング・システムとそれ以外の並列／分散応用プログラムの間で相互利用が可能な技術がますます増えてくることが予想される。さらに、応用プログラムとその実行環境であるオペレーティング・システムの動的な相互作用が重要になってくるであろう。本研究が、このような技術の相互利用と動的な相互作用の研究に貢献することを確信する。

謝辞

本論文は、筆者が筑波大学大学院博士課程工学研究科において行った研究をまとめたものである。筑波大学電子・情報工学系清木康助教授には、研究の開始から現在に至るまで御指導と御助言をして頂いた。同助教授が進めて来られたデータベースの並列処理システムは、単一プログラミング・システムとしての並列／分散応用プログラムの例として、本研究の契機となり、その開発を通じて、本研究への数多くの貴重な示唆を受けることができた。ここに、心から感謝の意を表す。東京大学理学部益田隆司教授には、様々な角度から御指導、御助言を頂いた。そして、オペレーティング・システムの魅力と大域的な思考方法を伝授していただいた。心から感謝の意を表す。筑波大学電子・情報工学系中田育男教授には、本研究に対する有益なご意見を頂いた。同学系佐々政孝教授（現在東京工業大学）には、研究発表に関する御指導をして頂き、また、同教授の研究成果であるコンパイラ生成系Ｒｉｅを使わせて頂いた。同学系井田哲雄教授には、形式化の重要性を教えて頂いた。同学系板野肯三助教授には、本研究に関して有益なコメントを頂き、また、研究発表に関する指導をして頂いた。以上の方々に心から感謝の意を表す。

本論文をまとめるに当たり、筑波大学構造工学系星野力教授、同電子・情報工学系亀田壽夫教授、並びに、電子技術総合研究所情報アーキテクチャ部弓場敏嗣部長には、有益なコメントを頂いた。心から感謝の意を表す。

日本ＩＢＭの田胡和哉博士、並びに、東京大学理学部加藤和彦博士をはじめとする益田研究室の皆様には、様々な機会を通じて熱心に討論していただいた。心から感謝の意を表す。本研究は、筑波大学オペレーティング・システム研究室において、行われたものである。同研究室の卒業生である高野陽介博士、劉澎博士、福田宗弘氏、黒沢貴弘氏、関安宏氏、波内みさ氏をはじめとする多くの人々から協力を得た。また、筑波大学工学研究科佐藤聡氏、同情報学類茹部朋幸氏には、本研究の成果を利用して頂き、また、有益な議論を交わすことができた。以上の方々を始め、同研究室の学生の方々に感謝する。また、ソフトウェア研究室ドメインの計算機利用者の仲間であった中田研究室、井田研究室の学生の方々、並びに、筑波大学工学研究科西山博泰氏、かん暁微氏を始めとする板野研究室の学生の方々に感謝する。

本研究では、ｆｊ／ＪＵＮＥＴにおけるネットワーク・ニュースと電子メールによる議論が不可欠であった。このような環境を整えて頂いた筑波大学電子・情報工学系大田友一教授を始めとするポストマスタ、ならびに、歴代のニュース・システム管理者の方々に深く感謝する。そして、箕原さん＠慶応大学、久野さん＠大塚．筑波大学、平野さん＠電総研、河野さん＠ＣＳＬ．Ｓｏｎｙ、桜井さん＠富士通、住元さん＠富士通研、藤田さん＠オムロン、乾さん＠オムロン、門林さん＠大阪大学、山井さん＠奈良高専をはじめとするｆｊ／ＪＵＮＥＴの皆さんには、ネットワーク・ニュースや電子メールを通じて、熱心に討論に参加して頂いた。深く感謝する。

参考文献

[１] V.Abrossimov, M.Rozier and M.Shapiro: "Generic Virtual Memory Management for Operating System Kernels", SOSP12, ACM Operating System Review, Vol.23, No.5, pp.123-136 (1989).

[２] M.J.Accetta, R.V.Baron, W.Bolosky, D.B.Golub, R.F.Rashid, A.Tevanian, and M.V.Young: "Mach: A New Kernel Foundation for UNIX Development", Proceedings of USENIX 1986 Summer Confernce, pp.93-112 (1986).

[３] T.Anderson, B.Bershad, E.Lazowska and H.Levy: "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism", SOSP13, ACM Operating System Review, Vol.25, No.5, pp.95-109 (1991).

[４] M.J.Bach: "The Design of the UNIX Operating System", Prentice-Hall (1986).

[５] H.E.Bal, J.G.Steiner, and A.S.Tanenbaum: "Programming Languages for Distributed Computing Systems", ACM Computing Surveys, Vol.21, No.3 (1989).

[６] "Balance 8000 Parallel Programming", Sequent Computer Systems, Inc. (1985).

[７] A.P.Black: "Supporting Distributed Applications: Experience with Eden", Proceedings of the Tenth Symposium on Operating Systems Principles, pp.181-193 (1985).

[８] D.Black: "Scheduling Support for Concurrency and Parallelism in the Mach Operating System", IEEE Computer, Vol.23, No.5, pp.35-43 (1990).

[９] L.Cardelli and P.Wegner: "On Understanding Types, Data Abstraction, and Polymorphism", ACM Computing Surveys, Vol.17, No.4, pp.471-522 (1985).

[１０] N.Carriero and D.Gelernter: "How to Write Parallel Programs: A Guide to the Perplexed", ACM Computing Surveys, Vol.21, No.3, pp.323-357 (1989).

[１１] J.S.Chase, F.G.Amador, E.D.Lazowska, H.M.Levy, and R.J.Littlefield: "The Amber System: Parallel Programming on a Networke of Multiprocessors", SOSP12, ACM Operating System Review, Vol.23, No.5, pp.147-158 (1989).

[１２] D.R.Cheriton: "The V Distributed System", Communications of the ACM, Vol.31, No.3, pp.314-333 (1988).

[１３] R.Chin and S.Chanson: "Distributed Object-Based Programming Systems", ACM Computing Surveys, Vol.23, No.1, pp.91-124 (1991).

[１４] S.Danforth and C.Tomlinson: "Type Theories and Object-Oriented Programming", ACM Computing Surveys, Vol.20, No.1, pp.29-72 (1988).

[１５] P.Dasgupta, R.J.LeBlanc,Jr. and W.F.Appelbe: "The Clouds Distributed Operating System: Functional Description, Implementation Details and Related Work", Proc. IEEE 8th Intl. Conf. on Distributed Computing Systems, pp.2-9 (1991).

[１６] J.D.Day: "The OSI Reference Model", Proc. of the IEEE, Vol.71, No.7, pp.1334-1340 (1983).

[１７] R.Draves, B.Bershad, R.Rashid, and R.Dean: "Using Continuations to Implement Thread Management and Communication in Operating Systems", SOSP13, ACM Operating System Review, Vol.25, No.5, pp.122-137 (1991).

[１８] M.A.Ellis and B.Stroustrup: "The Annotated C++ Reference Manual", Addison-Wesley (1990).

[１９] C.J.Fleckenstein and D.Hemmedinger: "Using a Global Name Space for Parallel Execution of UNIX Tools", Communications of the ACM, Vol.32, No.9, pp.1085-1090 (1989).

[２０] "Forth-83 Standard", Forth Standards Team (1983).

[２１] A.Fukuda, K.Murakami and S.Tomita: "Toward Advanced Parallel Processing: Exploiting Parallelisms at Task and Instructuon Levels", IEEE Micro, Vol.11, No.8, pp.16-19,50-61 (1991).

[２２] A.Goscinski: "Distributed Operating Systems -- The logical Design", Addison-Wesley, (1991).

[２３] R.Govindan and D.Anderson: "Scheduling and IPC Mechanisms for Continuous Media", SOSP13, ACM Operating System Review, Vol.25, No.5, pp.68-80 (1991).

[２４] R.G.Guy, J.S.Heidemann, Wai Mak, T.W.Page,Jr., G.J.Popek and D.Rothmeier: "Implementation of the Ficus replicated file system", USENIX 1990 Summer Conf., pp.63-71 (1990).

[２５] S.Habert, L.Mosseri, and V.Abrossimov: "COOL: Kernel Support for Object-Oriented Environments", Proc. OOPSLA/ECOOP'90 Conf., ACM SIGPLAN Notes, Vol.25, No.10 (1990).

[２６] 萩野，山岸："ＴｏＭマイクロカーネル"，電子情報通信学会論文誌Ｄ－１，Vol.J75-D-I, No.8, pp.555-562 (1992).

[２７] 箱根，横山，谷口："軽量プロセスと非完了システムコール機能の比較：並列処理の性能評価"，情報処理学会第４２回全国大会講演論文集（４），4K-11, pp.15-16 (1991).

[２８] B.Hansen: "The Architecture of Concurrent Programs", Prentice-Hall (1997).

[２９] C.A.R.Hoare: "Communicating Sequential Processes", Prentice-Hall (1985).

[３０] S.Inohara, K.Kato, A.Narita and T.Masuda: "A Thread Facility Based on User/Kernel Cooperation in the XERO Operating System", Proc. 15th Intl. Computer Software & Applications Conf., pp.398-405 (1991).

[３１] 井上外志雄："標準ＦＯＲＴＨ"，共立出版 (1985).

[３２] 乾："分散ＯＳ Ｍａｃｈとそのインプリメンテーション"，情報処理学会研究会報告，90-OS-49-1 (1990).

[３３] M.B.Jones: "Inheritance in Unlikely Places: Using Objects to Build Derived Implementations of Flat Interfaces", Proc. 2nd International Workshop on Object Orientation in Operating Systems, pp.341-345 (1992).

[３４] M.B.Jones : "A Toolkit for Interposing User Code at the System Interface", Workshop on Operating Systems and Object Orientation at OOPSLA/ECOOP'90 (1990).

[３５] A.R.Karlin, Kai Li, M.S.Manasse and S.Owicki: "Empirical Studies of Competitive Spining for a Shared-Memory Multiprocesor", SOSP13, ACM Operating System Review, Vol.25, No.5, pp.41-55 (1991).

[３６] 片上，新城，清木："並列処理環境に適したプロセス・サーバと並列シェルの実現"，情報処理学会第４４回全国大会講演論文集，2G-2，pp.17-18 (1992).

[３７] J.Kepecs and M.Solomon: "SODA: A Simplified Operaing System for Distributed Applications", ACM Operating System Review, Vol.19, No.4, pp.45-56 (1985); originally presented at the Third ACM SIGACT/SIGOPS Symp. on Principles of Distributed Computing (1984).

[３８] Y.Kiyoki, T.Kurosawa, K.Kato, and T.Masuda: "The Software Architecture of a Parallel Processing System for Advanced Database Applications", Proc. 7th IEEE Conf. on Data Engineering, pp.220-229 (1991).

[３９] Y.Kiyoki, K.Kato and T.Masuda: "A Relational Database Machine Based on Functional Programming Concepts", Proc. ACM-IEEE Computer Society Fall Joint Computer Conf., pp.969-978 (1986).

[４０] Y.Kiyoki, K.Kato, N.Yamaguchi and T.Masuda: "A Stream-Oriented Approach to Parallel Processing for Deductive Database, Proc. 5th Intl. Workshop on Database Machines, pp.102-115 (1987).

[４１] 清木，加藤："関数型計算モデルのデータベース処理への適用"，情報処理，Vol.29, No.8, pp.897-907(1988).

[４２] 久野："型階層をもつオブジェクト指向言語"，情報処理，Vol.29, No.4, pp.318-324 (1988).

[４３] 前川，所，清水（編）："分散型オペレーティングシステム"，共立出版 (1991).

[４４] B.Meyer: "Object-oriented Software Construction", Interactive Software Engineering (1988).

[４５] B.W.Lampson: "Designing a Global Name Service", Proc. Symp. on Principles of Distributed Computing, pp.1-10 (1986).

[４６] S.J.Leffle, M.K.McKusic, M.J.Karels and J.S.Quarterman: "The Design and Implementation of the 4.3BSD UNIX Operating System", Addison-Wesley (1989).

[４７] E.Levy and A.Silberschatz: "Distributed File Systems: Concepts and Examples", ACM Computing Surveys, Vol.22, No.4, pp.321-374 (1990).

[４８] H.Lieberman: "Using Prototypical Objects to Implement Shared Behavior in Object Oritented Systems", OOPSLA'86, pp.214-223 (1986).

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

[50] P.Liu, Y.Kiyoki and T.Masuda: "Efficient Algorithms for Resouce Allocation in Parallel and Distributed Query Processing Environments", Proc. IEEE Intl. Conf. on Distributed Computing Systems (1989).

[51] J.Magee, J.Kramer and M.Sloman: "Constructing Distributed Systems in Conic", IEEE Transactions on Software Engineering, Vol.15, No.6, pp.663-675 (1989).

[52] "MC88200 Cache/Memory Management Unit User's Manual", Motorola (1990).

[53] B.Marsh and M.Scott: "First-Class User-Level Threads", SOSP13, ACM Operating System Review, Vol.25, No.5, pp.110-121 (1991).

[54] S.Mullender, G.Rossum, A.Tanenbaum, R.Renesse and H.Staveren: "Amoeba: A Distributed Operating System for the 1990s", IEEE Computer, Vol.23, No.5, pp.44-53 (1990).

[55] S.J.Mullender and A.S.Tanenbaum: "The Design of a Capability-Based Distributed Operating System", The Computer Journal, Vol.29, No.4, pp.289-299 (1986).

[56] T.Nakajima and H.Tokuda: "Implementation of Scheduling Policies in Real-Time Mach", Proc. 2nd International Workshop on Object Orientation in Operating Systems, pp.165-169 (1992).

[57] 波内, 清木, 劉: "演繹データベースの並列処理方式の実現と資源割当て方式", 情報処理学会データベース・システム研究会, 89-DBS-72, pp.105-112, July 20-21 (1989).

[58] "The NeXT System Reference Manual", NeXT,Inc.(1988).

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

[60] J.K.Ousterhout, A.R.Cherenson, F.Douglis, M.N.Nelson and B.B.Welch: "The Sprite Network Operating System", IEEE Computer, Vol.21, No.2, pp.23-36 (1988).

[61] T.W.Page Jr., G.J.Popek and R.G.Guy: "Stackable Layers: An Object-Oriented Approach to Distributed File System Architecture", Workshop on Operating Systems and Object Orientation at OOPSLA/ECOOP (1990).

[62] D.A.Patterson, G.Gibson and R.H.Katz: "A Case for Redundant Arrays of Inexpensive Disks (RAID)", Proc. of ACM SIGMOD Conf., pp.109-116 (1988).

[63] R.Pike, D.Presotto, K.Thompson and H.Trickey: "Plan 9 from Bell Labs", proceedings of UKUUG, (1990).

[64] G.J.Popek, B.Walker, J.Chow, D.Edwards, C.Kline,G.Rudisin and G.Thiel: "LOCUS: A Network Transparent, High Reliability Distributed System", Proceedings of the 8th Symposium on Operatiting Systems Principles, pp.160-168 (1981).

[65] G.J.Popek: "The LOCUS Distributed System Architecture", The MIT Press, Cambridge USA (1986).

[６６] R.Rashid, A.Tevanian,Jr., M.Young, D.Golub, R.Baron,D.Black, W.J.Bolosky, and J.Chew: "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocesor Architectures", IEEE Transactions on Computers, Vol.37, No.8, pp.896-908 (1988).

[６７] R.Rashid: "Threads of a New System", UNIX REVIEW, No.8, pp.37-49 (1986).

[６８] "Reference Manual for the Ada Programming Language", United States Depertment of Defense (1983).

[６９] "Remote Procedure Call Programming Guide", Sun Microsystems, Inc. (1988).

[７０] D.S.H.Rosenthal: "Evolving the Vnode Interface", USENIX 1990 Summer Conf. (1990).

[７１] 酒井: "ＳＰＡＲＣプロセッサにおける高速コンテキスト切替え方式", 情報処理学会第４回コンピュータシステムシンポジウム論文集, 情処シンポジウム論文集, Vol.92, No.7, pp.85-92 (1992).

[７２] 佐藤, 清木: "関数型計算に基づく並列型データベースシステムの並列処理実行系の実現方式", 情報処理学会データベース・システム研究会, 92-DBS-89-19, July 24 (1992).

[７３] 関, 関島, 清木: "並列処理システムＳＭＡＳＨにおける機械系ＣＡＤデータベース実現のための一考察", 情報処理学会データベース・システム研究会, 89-DBS-72, pp.177-184, July 20-21 (1989).

[７４] L.Sha, R.Rajkumar and J.Lehoczky: "Priority Inheritance Protocols: An Aproach to Real-Time Synchronizaion", IEEE Transactions on Computers, Vol.39, No.9, pp.1175-1185 (1990).

[７５] K.Shimizu, M.Maekawa and J.Hamano: "Hierarchical Object Groups in Distributed Operating Systems", Proc. IEEE 8th Intl. Conf. on Distributed Computing Systems, pp.18-24 (1991).

[７６] Y.Shinjo and Y.Kiyoki: "Harmonization methods in the ReSC Distributed Operating System", Journal of Concurrency: Practice and Experience, special issue on HPDC-1 (投稿中).

[７７] 新城, 清木: "仮想プロセッサを提供するオペレーティング・システム・カーネルの構成法", 情報処理学会論文誌, Vol.34, No.3 (1993).

[７８] 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).

[７９] Y.Shinjo and Y.Kiyoki: "Harmonizing a Distributed Operating System with Parallel and Distributed Applications", Proc. 1st International Symposium on High Performance Distributed Computing (HPDC-1), pp.114-123 (1992).

[８０] 新城, 清木: "オブジェクトの堆積に適したＲＰＣスタブ生成器", 情報処理学会第４４回全国大会講演論文集（４）, 2G-1, pp.15-16 (1992年)

[８１] 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).

[８２] Y.Shinjo and Y.Kiyoki: "ReSC: A Distributed Operating System for Parallel and Distributed Applications", Proc. First International Conference on Parallel and Distributed Information Systems (PDIS), p.171 (1991).

[８３] 新城, 清木: "分散型オペレーティング・システムにおけるオブジェクトの堆積", 情報処理学会第４２回全国大会講演論文集（４）, 3K-8, pp.15-16 (1991).

[８４] 新城: "分散型ＯＳ　ＲｅＳＣにおけるデータベース処理の支援", 電子情報通信学会データベースワークショップ論文集, pp.118-123 (1991)

[８５] Y.Shinjo and Y.Kiyoki: "Multiple Views of ReSC Distributed Operating System", Workshop on Operating Systems and Object Orientation at OOPSLA/ECOOP'90 (1990).

[８６] 新城, 清木, 益田: "並列・分散処理環境を対象としたＯＳ　ＲｅＳＣにおける分散オプティマイザ", 情報処理学会第４０回全国大会講演論文集, 5G-3 (1990).

[８７] 新城, 清木: "ＲｅＳＣ：並列アプリケーションのためのオペレーティング・システム", 情報処理学会第３９回全国大会講演論文集, 6P-2, pp.1247-1248 (1989).

[８８] 新城, 清木: "データベースの並列処理を支援するオペレーティング・システムの基本機能", 情報処理学会研究会報告, 89-OS-44,89-DBS-73 (1989).

[８９] 新城, 清木, 劉, 益田: "データベースおよび知識ベースを対象としたストリーム指向型並列処理系の共有メモリ・マシン上への実現", 情報処理学会アドバンストデータベースシステム・シンポジウム論文集, Vol.88, No.9, pp.117-126 (1988).

[９０] 新城: "密結合並列処理システム上でストリーム指向型関係演算処理をする際の資源割当て", 第２１回情報処理学会若手の会シンポジウム, 湯河原 (1988).

[９１] 新城, 田胡, 高野, 福田, 関: "分散システムＡｇｏｒａ－Ｉのユーザ環境", 情報処理学会第３６回全国大会講演論文集, 4D-9, pp.325-326 (1988).

[９２] M.Sloman and J.Kramer: "Distributed Systems and Computer Networks", Prentice-Hall International(UK) Ltd. (1987).

[９３] "SunOS Refernce Manual", Sun Microsystems, Inc. (1988).

[９４] V.Srinivasan and J.C.Mogul: "Spritely NFS: Experiments with Cache-Consistency Protocols", SOSP12, ACM Operating System Review, Vol.23, No.5, pp.45-57 (1989).

[９５] M.Stonebraker: "Operating System Support for Database Management", Communications of the ACM, Vol.24, No.7, (1981).

[９６] 田胡, 益田: "分散型オペレーティング・システム", 情報処理, Vol.28, No.4, pp.437-445 (1987).

[９７] 田胡, 益田: "オペレーティング・システムの構造記述に関する一試み", 情報処理学会論文誌, Vol.25, No.4, pp.524-534 (1984).

[９８] A.S.Tanenbaum and R.van Renesse: "Distributed Operating Systems", ACM Computing Surveys, Vol.17, No.4, pp.419-470 (1985).

[９９] A.S.Tanenbaum: "Modern Operating Systems", Prentice-Hall (1992).

[１００] C.P.Thacker, L.C.Stewar, E.H.Satterhwaite,Jr.:"Firefly: A Multiprocessor Workstation", IEEE Transaction on Computers, Vol.37, No.8, pp.909-920(1988).

[１０１] "Transmission Control Protocol", DARPA Internet Program Protocol Specification, RFC 793 (1981).

[１０２] H.Tokuda and T.Nakajima: "Evaluation of Real-Time Synchronization in Real-Time Mach", USENIX 2nd Mach Symposium (1991).

[１０３] A.Tucker and A.Gupta: "Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors", SOSP12, ACM Operating System Review, Vol.23, No.5, pp.159-166 (1989).

[１０４] 梅村, 大里 : "Ｌｉｓｐ上のオブジェクト指向プログラミング", 情報処理, Vol.29, No.4, pp.303-309 (1988).

[１０５] "UniOS-Mach Reference Manual", オムロン (1990).

[１０６] M.Weiser, A.Demers and C.Hauser: "The Portable Common Runtime Approach to Interoperability", SOSP12, ACM Operating System Review, Vol.23, No.5, pp.114-122 (1989).

[１０７] "Workshop on Object-Orientation in Operating Systems", OOPSLA/ECOOP'90 Addendum to the Proceedings, edited by V.Russo and M.Shapiro, pp.81-91 (1990).

[１０８] W.Wulf, E.Cohen, W.Corwin, A.Jones, R.Levin, C.Pierson, and F.Pollack: "HYDRA: The Kernel of a Multiprocessor Operating System", Communications of the ACM, Vol.17, No.6, pp.337-345 (1974).

[１０９] Y.Yokote, F.Teraoka, A.Mitsuzawa, N.Fujinami and M.Tokoro: "The Muse Object Architecture: A New Operating System Structuring Concept", ACM Operating System Review, Vol.25, No.2, pp.22-46 (1991).

[１１０] C.T.Yu and C.C.Chang: "Distributed Query Processing", ACM Computing Surveys, Vol.16, No.4, pp.399-433, (1984).

付録の目次

付録Ａ　遠隔手続き呼出し …… 1
Ａ－１　オブジェクト識別子 …… 1
Ａ－２　マスク …… 1
Ａ－３　利用者識別子 …… 1
Ａ－４　アクセス権のチェックの例 …… 2
Ａ－５　カーネル・コール …… 3
Ａ－６　サーバ・プログラムの例 …… 4
　Ａ－６．１　手続きの表 …… 4
　Ａ－６．２　セッション構造体 …… 5
　Ａ－６．３　遠隔手続き呼出し受付け軽量プロセス …… 5
　Ａ－６．４　遠隔手続き呼出し入口手続き …… 6
　Ａ－６．５　メモリ中のオブジェクトに対する手続き …… 6
Ａ－７　クライアント・プログラムの例 …… 7

付録Ｂ　ＲｅＳＣシステムの標準的なオブジェクトとサーバ …… 9
Ｂ－１　インタフェース記述言語ｐｒｏｃ …… 9
　Ｂ－１．１　構文 …… 9
　Ｂ－１．２　基本データ型 …… 10
　Ｂ－１．３　一般的な型構成子 …… 10
　Ｂ－１．４　拡張した型構成子 …… 11
Ｂ－２　オブジェクト手続きやサーバ手続きで利用されるデータ型 …… 12
Ｂ－３　一般のサーバ …… 13
Ｂ－４　一般のオブジェクト …… 14
Ｂ－５　ファイル・サーバとファイル・オブジェクト …… 15
Ｂ－６　ディレクトリ・サーバとディレクトリ・オブジェクト …… 16
Ｂ－７　メモリ・サーバとメモリ・オブジェクト …… 17
Ｂ－８　プロセス・サーバとプロセス・オブジェクト …… 18
Ｂ－９　カーネル・サーバ群 …… 19

付録Ｃ　マイクロプロセス・ライブラリの機能 …… 21
Ｃ－１　全体の構成 …… 21
Ｃ－２　データ型 …… 21
Ｃ－３　第０層　カーネル・コール …… 22
Ｃ－４　第１層　スピンロックとコンテキスト切替え …… 23
Ｃ－４．１　スピンロック …… 23
Ｃ－４．２　コンテキストの生成と切替え …… 23
Ｃ－５　第２層　マイクロプロセスの生成と基本的な同期 …… 23
Ｃ－５．１　現在実行中のマイクロプロセス …… 24
Ｃ－５．２　マイクロプロセスの生成・消滅 …… 24
Ｃ－５．３　マイクロプロセス間の同期 …… 25
Ｃ－６　第３層　高度な同期通信プリミティブ …… 25
Ｃ－６．１　セマフォ …… 25
Ｃ－６．２　モニタ …… 26
Ｃ－６．３　ランデブ …… 26
Ｃ－６．４　ストリーム …… 27
Ｃ－７　マイクロプロセス・スケジューラ …… 28
Ｃ－８　レディ・キュー・モジュール …… 28

付録Ｄ　プロセス操作の性能を計測するためのベンチマーク …… 31
Ｄ－１　プロセスの生成・消滅 …… 31
Ｄ－２　コンテキスト切替えを伴わないプロセス間の同期 …… 32
Ｄ－３　コンテキスト切替えを伴うプロセス間の同期 …… 32
Ｄ－３．１　セマフォを用いたプログラム …… 32
Ｄ－３．２　モニタと条件変数を用いたプログラム …… 33
Ｄ－３．３　コンテキスト切替えを起こすプリミティブを用いたプログラム・ 34

付録A　遠隔手続き呼出し

ここでは、遠隔手続き呼出し関連のカーネル・コールと、それを利用する典型的なサーバとクライアントの構造を、C言語を用いて示す。

A－1　オブジェクト識別子

システム・レベルのオブジェクトは、全て一様なオブジェクト識別子により識別される。本システムのオブジェクト識別子は、以下の oid_t で示された構造を持つ。

```
typedef int     site_t ;        /* サイト識別子 */

typedef int     server_t ;      /* サーバ識別子 */

typedef struct
{
        char[32];               /* 32 == NFS_FHSIZE */
} inum_t ;                      /* オブジェクト番号 */

typedef struct
{
        site_t          site ;
        server_t        server ;
        inum_t          inum ;
} oid_t ;                       /* オブジェクト識別子 */
```

A－2　マスク

本システムでは、様々な場所でマスク（mask）を用いる。マスクとは、対応する整数の有効範囲を現すものである。マスクのあるビットが１であることは、そのビットが有効であることを意味する。

```
#define All1    (~0)    /* '~' は、ビット反転を行う演算子 */
```

全ビットが１のマスクは、対応する整数の全てのビットが有効であることを意味する。

A－3　利用者識別子

利用者識別子は、オブジェクトへのアクセス権の確認、セキュリティのチェックにおいて、利用者を識別するものである。利用者識別子は、以下の uid_t に示された構造を持つ。

```
typedef struct
{
        int     u_no ;          /* 利用者番号 */
        int     u_mask ;        /* マスク */
} uid_t ;
```

利用者番号 u_no は、各利用者に一意になるように与えられた整数である。マスク u_mask は、利用者番号の有効範囲を示す。

本システムのプロセスは、属性として利用者識別子を持っている。利用者識別子の利用者番号の、マスクにより残された部分がアクセス権のチェックに用いられる。たとえば、システム管理者の場合、マスクとして０が用いられる。これにより、全てのアクセス権のチェックが無効になり、任意のオブジェクトにアクセスすることが可能になる。一般の利用者のマスクは、ALL1 である。

A－４　アクセス権のチェックの例

この節では、A－３節で述べた利用者識別子を用いたアクセス権のチェックの例を示す。たとえば、次の様なオブジェクトを考える。

```
struct an_object
{
        int     owner ;
        int     rmask ;
        int     wmask ;
        int     xmask ;
        ・・・
};
```

属性 rmask, wmask, xmask が、それぞれ読込み、書込み、実行のアクセス可能性を決定する変数である。ある利用者識別子 uid を持つクライアント・プロセスのアクセス権を考える。

```
struct process
{
        uid_t uid ;
        ・・・
};
```

たとえば、読込みの場合、次の式が真ならば、その利用者はオブジェクトへの読込みのアクセス権を持っていると判定する。

```
(uid.u_no  & uid.u_mask & rmask) ==
(owner     & uid.u_mask & rmask)
```

このように、クライアント・プロセスの利用者識別子のマスクとオブジェクトのマスクのビットが１の部分について、クライアント・プロセスの利用者番号 u_no とオブジェクトの所有者の利用者番号 owner を比較して、アクセス権の有無を判定する。したがって、オブジェクトのマスクとして０を指定することにより、全ての利用者からのアクセスを許すことを意味する。 ALL1 を指定することにより、オブジェクトの所有者にみのアクセスを許すことを意味する。０と ALL1 の中間を指定することにより、あるグループによるアクセスを許すことを意味する。

例として、次の様なアクセスが可能なオブジェクトを生成ことを考える。

- 全ての利用者からの読込みを許す。
- そのオブジェクトの所有者だけに書込みを許す。
- 同じグループに属する利用者の実行を許す。

これは、ＵＮＩＸにおいて、ファイルのモードが 754 (rwxr-xr--) として実現される。ここでは、利用者番号とそのマスクを３２ビットの整数、その中で、グループの管理のために、上位１６ビットを用いることにする。上記のようなアクセスが可能

なオブジェクトを実現するためには、次のように、マスクを設定する。

```
rmask = 0 ;
wmask = ALL1 ;
xmask = ALL1 << 16 ;
```

A－5　カーネル・コール

ＲｅＳＣシステムの遠隔手続き呼出し機能は、次の５つのカーネル・コール rpc_send(), rpc_receive(), rpc_reply(), rpc_forward(), rpc_send_uid() により提供される。最初のものが、クライアントにより、残りの４つがサーバにより利用される。特に、最後の２つは、堆積可能なサーバにおいて利用される。

```
int rpc_send( site, server, procno, arg, res, timeout )
site_t site ;
server_t server ;
int procno ;
rpcbuff_t *arg ;
rpcbuff_t *res ;
time_t *timeout ;
```

rpc_send() は、クライアントが遠隔手続き呼出しの呼出し（要求の送信と結果の受信）を行うカーネル・コールである。site は、サーバが存在するサイト、server は、サーバ識別子（遠隔手続き呼出しのポート番号）である。オブジェクト手続きの場合、site と server は、オブジェクト識別子から抜き出されたものを用いる。procno は、遠隔手続き呼出しの手続き番号である。arg は、クライアントからサーバに渡される引数を格納するバッファを、次のような構造体により指定する。

```
typedef struct
{
    char        *m_addr ;
    int          m_len ;
} rpcbuff_t ;
```

res は、サーバからクライアントへ返される結果を格納するバッファを指定する。timeout は、遠隔手続き呼出しの完了の最大待ち時間を指定する。

```
typedef int rpcid_t ;
rpc_id rpc_receive( server, procno, arg, res, timeout, uid )
server_t server ;
int *procno ;
rpcbuff_t *arg ;
rpcbuff_t *res ;
time_t *timeout ;
uid_t *uid ;
```

rpc_receive() は、サーバ側において遠隔手続き呼出しの受付けを行うカーネル・コールである。server, arg, res, timeout の意味は、rpc_send() と同じである。uid は、クライアント・プロセスの利用者識別子である。これはカーネルによって自動的に付加されるものである。サーバは、これを利用してアクセス権のチェックを行うことができる。rpc_receive() の結果は、遠隔手続き呼出しの識別子である。これは、次の rpc_reply(), rpc_forward() において使用される。

```
int rpc_reply( rpcid, errno )
rpcid_t rpcid ;
```

```
int errno ;
```

rpc_reply() は、サーバ側において遠隔手続き呼出しの応答を行うカーネル・コールである。rpcid は、rpc_receive() により返された識別子を指定する。errno は、呼出し側の rpc_send() カーネル・コール自身の結果として、クライアントに渡される。errno が 0 であることは、エラーが発生しなかったことを示す。遠隔手続き呼出しの結果は、rpc_reply() を行う前に、rpc_receive() により受け付け時に res で指定された領域に格納されていなければならない。

```
int rpc_forward( rpcid, site, server, procno, timeout, uid )
rpcid_t rpcid ;
site_t site ;
server_t server ;
int procno ;
time_t *timeout ;
uid_t *uid ;
```

rpc_forward() は、遠隔手続き呼出しの転送を行うカーネル・コールである。rpcid に指定された遠隔手続き呼出しのセッションを、site, server をで指定されたサーバの手続き番号 procno の手続きに転送する。ここで、uid には、転送するときに用いるクライアントの利用者識別子である。これを書き換えることで、受け付けた時とは異なる権限で遠隔手続き呼出しの呼出しを行うことが可能となる。ただし、書き換えが可能な部分は、そのプロセスの利用者識別子のマスクが0の部分だけである。

```
int rpc_send_uid( site, server, procno, arg, res, timeout, uid )
site_t site ;
server_t server ;
int procno ;
rpcbuff_t *arg ;
rpcbuff_t *res ;
time_t *timeout ;
uid_t *uid ;
```

rpc_send_uid() は、rpc_send() と同じく、遠隔手続き呼出しの呼出しを行うカーネル・コールである。rpc_send() との違いは、uid を指定することができる点にある。uid は、rpc_forward() と同じく、遠隔手続き呼出しの呼出しを行う時の利用者識別子である。

A－6　サーバ・プログラムの例

以下にサーバ・プログラムの例を示す。重要なデータ構造としては、手続きの表、セッション構造体がある。重要な手続きとしては、遠隔手続き呼出し受付け軽量プロセス、遠隔手続き呼出し入口手続き、メモリ中のオブジェクトに対する手続きがある。

A－6．1　手続きの表

```
struct rpc_proc
{
        int      procno ;
        int    (*xdr_arg)();
        int      arg_size ;
        int    (*xdr_res)();
        int      res_size ;
        char  *(*local)() ;
};
```

```
typedef struct rpc_proc *rpc_proc_t ;

struct rpc_proc rpc_proctab[] =
{
        {SERVER_NULL, xdr_void, 0,
        xdr_void, 0, server_null },
        {SERVER_CREATE_WITH_LOWER, xdr_oid_array_t, sizeof(oid_array_t),
        xdr_oid_res_t, sizeof(oid_res_t), server_create_with_lower},
        ・・・
};
int rpc_proctab_nentries = sizeof(rpc_proctab)/sizeof(rpc_proct) ;

extern  rpc_proc_t      rpc_proctab_find( /*tab,nent,procno*/ );
```

rpc_proctab は、そのサーバが受付け可能な遠隔手続き呼出しの手続きの表である。procnoは、手続き番号、xdr_arg は、引数を解く（unpack, unmarshall）手続き、arg_size は、引数の大きさ、xdr_res は、結果をまとめる（pack, marshall）手続き、res_size は、結果の大きさ、local は、メモリ中のオブジェクトへの手続きのポインタである。rpc_proctab_find() は、表を検索するライブラリ関数である。

Ａ－６．２　セッション構造体

```
struct rpc_session_t
{
        rpcid_t         rpcid ;
        rpcbuff_t       arg ;
        rpcbuff_t       res ;
        int             procno ;
        uid_t           uid ;
};
```

これは、遠隔手続き呼出しのセッションの情報を保持する構造体である。この構造体は、Ａ－５で述べた rpc_receive() の引数と結果に対応している。

Ａ－６．３　遠隔手続き呼出し受付け軽量プロセス

```
rpc_acceptance_lwp()
{
    rpc_proc_t     p ;
    rpc_session_t *s ;

        while( 1 )
        {
            if( s == 0 )
                s = malloc( sizeof(rpc_session_t) );
            s->rpcid =
                rpc_receive( SERVER1, &s->procno, &s->arg, &s->res,
                                        &TIMEOUT, &s->uid );
            p = rpc_proctab_find( rpc_proctab, rpc_proctab_nentries,
                                  s->procno );
            if( p )
            {
                mp_create( rpc_entry_procedure, STACK_SIZE, M_READY, s, p );
                s = 0 ;
```

```
            }
            else
            {
                rpc_reply( s->rpcid, ERROR_NOPROC );
            }
        }
}
```

これは、遠隔手続き呼出し受付け軽量プロセスの骨格を示している。この軽量プロセスは、このように無限ループを実行しており、通常、rpc_receive() カーネル・コールを発行し、遠隔手続き呼出しの受付けを行っている。クライアントから遠隔手続き呼出しを受け付けると、rpc_proctab_find() により、手続きの表を検索する。対応する手続きが見つかった場合、遠隔手続き呼出しエントリ手続き rpc_entry_procdure() をトップとする軽量プロセスを生成する。見つからなかった場合、エラーを返す。

A－6．4　遠隔手続き呼出し入口手続き

```
void rpc_entry_procdure( s, p )
    rpc_session_t *s ;
    rpc_proc_t     p ;
{
    char        *arg ;
    char        *res ;

        arg = arg_decode( p->xdr_arg, p->arg_size, s->arg );
        res = (*p->local)( arg,s,p );
        res_encode( p->xdr_res, res, s->res );
        rpc_reply( s->rpcid,0 );
        arg_free( p->xdr_arg, arg );
        res_free( p->xdr_res, res );
        free( s );
        return; /* mp_exit() */
}
```

これは、遠隔手続き呼出し入口手続きである。この手続きの役割は、クライアントから送られてきた引数を解き、メモリ中のオブジェクトに対する手続きを呼び出し、結果をクライアントへ返すことである。まず、ライブラリ関数 arg_decode() により、遠隔手続き呼出しの引数を解く。続いて、対応する局所オブジェクトに対する手続き p->local を呼び出す。そして、ライブラリ関数 res_encode() により、遠隔手続き呼出しの結果をまとめ、rpc_reply()カーネル・コールにより、クライアントに結果を返す。その後、不用になった引数、結果、セッション構造体のメモリを開放する。最後に、この関数から復帰することで、現在の軽量プロセスを終了させている。

A－6．5　メモリ中のオブジェクトに対する手続き

```
object_forward( arg, s )
    some_arg_t    *arg ;
    rpc_session_t *s ;
{
    object_t             *obj ;

        obj = obj_fetch( arg );
        if( obj == 0 )
        {
```

```
            rpc_reply( s,ERROR_NOOBJECT );
            free( s );
            arg_free( arg );
            mp_exit();
        }

        obj_lock( obj );
        obj_access( obj,s->uid );
        arg->oid = obj->lower ;
        arg_encode( p->xdr_arg, arg, s->arg );
        rpc_forward( s->rpcid, obj->lower.site, obj->lower.server,
                     s->procno, &TIMEOUT, &s->uid );
        obj_unlock( obj );
        obj_free( obj );
        arg_free( p->xdr_arg, arg );
        free( s );
        mp_exit();
}
```

これは、典型的な転送（forward）を行うようなメモリ中のオブジェクトへの手続きである。まず、関数 obj_fetch() を呼び出し、メモリ中のオブジェクトを得る。失敗すると、エラーを返す。次に、オブジェクトにロックを掛け、アクセス権のチェックを行う。そして、引数の中のオブジェクト識別子の部分（arg->oid）を、下位層のオブジェクト（obj->lower）に書き換え、s->rpcid で指定された遠隔手続き呼出しのセッションを、obj->lower.site, obj->lower.server で指定された、下位層のオブジェクトのサーバに転送（rpc_forward()）している。その後、ロックの開放、オブジェクトの開放、セッション構造体の開放を行い、軽量プロセスを終了している。

A－7　クライアント・プログラムの例

以下に、クライアント・プログラムの例を示す。

```
getattr( oid, attr )
    oid_t *oid ;
    attr_t *attr ;
{
    memmaddr_t arg,res ;
    char arg_buff[MAX_ARG_BUFF_SIZE] ;
    char res_buff[MAX_RES_BUFF_SIZE] ;

        arg.m_addr = arg_buff ;
        arg.m_len  = arg_encode( xdr_oid_t, oid, arg_buff );
        res.m_addr = res_buff ;
        res.m_len  = MAX_RES_BUFF_SIZE ;

        errno = rpc_send( oid->site, oid->server, OBJECT_GETATTR,
                          &arg, &res,TIMEOUT );
        if( errno == 0 )
        {
            res_decode( xdr_attr_t, attr, arg_buff );
        }
        return( errno );
}
```

これは、オブジェクトの属性（attr_t）を得る手続きである。まず、引数（oid）を、ライブラリ関数 arg_encode() によりまとめ、arg_buff に格納している。次に、結果を受け取るためのバッファ res_buff を、res に設定している。そして、カーネル・コール rpc_send() により、遠隔手続き呼出しを行っている。ここで、遠隔手続き呼出しにより呼び出すサーバを、oid->site, oid->server で、オブジェクト識別子の中から抜出している。OBJECT_GETATTR は、手続き番号である。そして、遠隔手続き呼出しが成功すると（errno==0）、結果を解いている。

付録Ｂ　ＲｅＳＣシステムの標準的なオブジェクトとサーバ

ここでは、５．８節で述べたインタフェース記述言語を用いて、ＲｅＳＣシステムで標準的に利用されるオブジェクトとサーバのインタフェースについて述べる。ここでは、簡単のため、結果として返されるエラーを省略する。

Ｂ－１　インタフェース記述言語ｐｒｏｃ

５．８節で述べたインタフェース記述言語をｐｒｏｃと名付ける。これは、プログラミング言語ｆｕｎにおいて関数を意味する fun と記述するところを、本インタフェース記述言語では、手続きを意味する proc と記述することに由来する。

Ｂ－１．１　構文

本言語におけるインタフェースは、次のような文の並びにより記述される。

＜名前＞ ： ＜型＞ ＝ ＜値＞ ；

この構文は、ｆｕｎに準じている。ｆｕｎは、１階型付きλ計算に基づく、多相（polymorphism）とオブジェクト指向言語をモデル化するために設計された言語である。ただし、ｆｕｎにおいて関数を意味する fun と記述するところを、本インタフェース記述言語では、手続きを意味する proc，あるいは、sproc と記述する。また、ｆｕｎでは、型と値が明確に区別されているのに対して、本インタフェース記述言語では、型を型型（型を格納するための変数のための型）の値として扱う。たとえば、次の文：

```
oid_t:type =
struct
{
        site_id       : site_t;
        server_id     : server_t ;
        object_number : object_number_t ;
};
```

では、oid_t という変数が、型を格納するための変数（:type）であり、oid_t の値として、struct以下に記述されている構造体が束縛される。なお、この定義は、Ｃ言語における定義：

```
typedef struct {
        site_t          site_id;
        server_t        server_id;
        object_number_t object_number;
} oid_t ;
```

と同じ意味を持つ。

Ｂ－１．２　基本データ型

本インタフェース記述言語において扱えるデータ型は、Ｃ言語、および、ＳｕｎＲＰＣのインタフェース記述言語（rpcgenコマンドの入力）を拡張したものになっている。基本データ型としては、次のようなものがある。

ｖｏｉｄ型	void
整数型	int, unsigned, long
列挙型	enum { ＜名前＞=値, ... }
実数型	float, double
文字型	char

Ｂ－１．３　一般的な型構成子

Ｃ言語、および、ＳｕｎＲＰＣのインタフェース記述言語と同等の型構成子を以下に示す。

■配列

配列には、次のような種類がある。

array[n] of ＜型＞	＜型＞型のn個の要素からなる固定長の配列
array<n> of ＜型＞	＜型＞型の最大n個の要素からなる可変長の配列
array<> of ＜型＞	＜型＞型の要素からなる可変長の配列 （最大要素数は不明）
array　of ＜型＞	array<> of ＜型＞ と同じ

■文字列型

文字列型は、文字列（０終端された文字の配列）を表す型である。次のような種類がある。

string<n>	最大n文字の可変長の文字列
string<>	最大長が不明の可変長の文字列
string	string<> と同じ

■不定型

不定型（opaque）とは、内容を一切解釈しないバイト列である。次のような種類がある。

opaque[n]	nバイトの固定長のバイト列
opaque<n>	最大nバイトの可変長のバイト列
opaque<>	最大長が不明のバイト列
opaque	opaque<> と同じ

■構造体

これは、Ｃ言語の構造体に相当する。次のような構文により、構造体が定義される。

struct { ＜フィールド並び＞ }

ここで、＜フィールド並び＞とは、

＜名前の並び＞ : ＜型＞ ;

の並びである。

■選択子付き共用体（discriminated union）

これは、ＳｕｎＲＰＣのインタフェース記述言語における選択子付き共用体（discriminated union）に対応するものである。これは、Ｐａｓｃａｌ言語における可変長レコードに相当する。次のような構文により、選択子付き共用体が定義される。

```
union switch(<選択子>)
{
case <選択子１>:
        <フィールド並び１>;
case <選択子２>:
        <フィールド並び２>;
...
default:
        <デフォルトのフィールド並び>;
}
```

Ｂ－１．４　拡張した型構成子

ここでは、Ｃ言語、および、ＳｕｎＲＰＣのインタフェース記述言語には存在しない型構成子 proc, sproc, set of, server について説明する。

■オブジェクト手続き構成子 proc
オブジェクト手続き構成子 proc は、オブジェクトに対する遠隔手続き呼出しを定義する。次のような構文により、オブジェクト手続きの型が定義される。

```
proc( <引数のフィールド並び> )
                        <結果のフィールド並び>
```

これは、以下で述べるサーバ手続きに展開される。

```
sproc( this:oid_t; <引数のフィールド並び> )
                        <結果のフィールド並び>
```

このように、オブジェクト手続きは、第１引数として操作対象となるオブジェクトの識別子をとるようなサーバ手続きである。

■サーバ手続き構成子 sproc
サーバ手続き構成子とは、サーバ手続き型の１つの型を作るものである。次のような構文により、サーバ手続きの型が定義される。

```
sproc( <引数のフィールド並び> ) <結果のフィールド並び>
```

<引数のフィールド並び>では、引数となる構造体を定義する。
<結果のフィールド並び>では、結果となる構造体を定義する。この型の値は、遠隔手続き呼出しの手続き番号を表す整数である。たとえば、次の文：

```
copy:   sproc( orig:oid_t )  new:oid_t  = 2 ;
```

は、copy という名前の遠隔手続き呼出しの手続きを定義している。ここで、変数 copy の値として、２が束縛されている。この２という値は、copy手続きの遠隔手続

き呼出しの手続き番号である。この手続きは、引数としてオブジェクト識別子を取り、結果としてオブジェクト識別子を返す。ここで、次の部分：

```
sproc( orig:oid_t )  new:oid_t
```

が、サーバ手続き型の値となる。したがって、上記の copy 手続きを次のように型変数を用いて宣言することも可能である。

```
proc_oid_oid_t: type = sproc( orig:oid_t ) new:oid_t ;
copy: proc_oid_oid_t = 2 ;
```

なお、sproc により、遠隔手続き呼出しのクライアント側のスタブが生成される。

■集合型 set of proc, set of sproc

集合型は、オブジェクト手続き、または、サーバ手続きの集合を作るための型である。この型の値は、以下で述べるサーバ型構成子の引数となる。次のような構文により、集合型が定義される。

```
set of  {proc | sproc}
```

この型の値は、オブジェクト、または、サーバのインタフェースを表すために用いられる。たとえば、次の文：

```
any_server: set of sproc = { copy, create };
```

では、any_server という名前の変数に、copy, create という要素からなる集合が束縛される。これは、任意のサーバ any_server は、サーバのインタフェースとして、手続き copy と create を持つことを意味する。

■サーバ型構成子 server

サーバ型構成子は、サーバのスタブを生成するための型構成子である。次のような構文により、サーバ型が定義される。

```
server( <サーバ手続きの集合> )
```

このように、<サーバ手続きの集合>を引数として取り、それらの手続きを受け付けるようなサーバのスタブを生成する。この型の変数の値は、サーバ識別子である。たとえば、次の文：

```
StdFS: server( file_server ) = 3 ;
```

では、StdFSは、サーバ型である。その値３は、サーバＳｔｄＦＳのサーバ識別子となる。そして、集合 file_server に保持されているサーバ手続きを受け付けるサーバのスタブが生成される。

Ｂ－２　オブジェクト手続きやサーバ手続きで利用されるデータ型

ここでは、付録Ｂ－１で述べたインタフェース記述言語ｐｒｏｃの基本データ型以外に、オブジェクト手続きやサーバ手続きで利用されるデータ型について述べる。

■オブジェクト識別子

```
site_t: type = int ;
server_t: type = int ;
object_number_t: type = opaque[32] ;

oid_t:type=
struct
{
        site_id       : site_t;
        server_id     : server_t ;
        object_number : object_number_t ;
};
```

これは、２．６節において導入したＲｅＳＣのオブジェクト識別子を表している。site_t は、サイト識別子を、server_t は、サーバ識別子を、object_number_t は、オブジェクト番号を表す型である。そして、oid_t は、オブジェクト識別子を意味する型である。現在、オブジェクト番号は、３２バイトのバイト列である。そして、オブジェクト識別子全体の大きさは、４０バイトである。

■オブジェクト識別子の配列

```
oid_array_t:type = array<> of oid_t ;
```

これは、オブジェクト識別子の可変長の配列である。

■ディレクトリ・エントリ

```
dir_entry_t: type = struct { oid:oid_t; name:string };
```

これは、ディレクトリの１つのエントリを示す型である。エントリとは、ここで示されているように、オブジェクト識別子と文字列の名前の組である。

Ｂ－３　一般のサーバ

この節では、一般的なサーバが全て備えているようなサーバ手続きを示す。

```
server_null: sproc( void ) void = 0 ;
```

これは、何もしないサーバ手続きである。サーバが動作しているかどうかを確認するために利用される。

```
server_create_with_lower:
sproc( lowers: oid_array_t, options: option_t ) new: oid_t = 1 ;
```

これは、オブジェクトの堆積を作るサーバ手続きである。引数として下位層のオブジェクト lowers とオプション options を取り、オブジェクトを生成し、結果として生成したオブジェクトの識別子を返す。

```
server_copy:
sproc( orig:oid_t ) new:oid_t  = 2 ;
```

これは、オブジェクトのコピーを行うサーバ手続きである。与えられた引数 orig で示されたオブジェクトに対して、いわゆる「深いコピー（deep copy）」を行う事で、新しいオブジェクトを生成する。下位層のオブジェクトがある場合、再帰的にコピーされる。結果としてその生成したオブジェクトの識別子を返す。

```
server_create_like:
sproc( orig:oid_t ) new:oid_t  = 3 ;
```

これは、server_copy と同様に、引数 orig で示されたオブジェクトと同じ形のオブジェクトを生成するサーバ手続きである。server_copy とは異なり、オブジェクトの内容は、コピーされない。結果として、空のオブジェクトが生成される。

```
any_server: set of sproc =
{
        server_null,
        server_create_with_lower,
        server_copy,
        server_create_like,
};
```

任意のサーバは、このような４つのサーバ手続きを受け付けることが要求される。

Ｂ－４　一般のオブジェクト

この節では、一般的なオブジェクトが全て備えているようなオブジェクト手続きを示す。これらのオブジェクト手続きは、隠された引数 this:oid_t で指定されたオブジェクトを操作するものである。

```
object_null: proc( void ) void = 10 ;
```

これは、何もしないオブジェクト手続きである。オブジェクトが存在するかどうかを確認するために使われる。

```
object_getattr: proc( void ) attr:attribute_t = 11 ;
```

これは、オブジェクトの属性を得るためのオブジェクト手続きである。結果として、属性を返す。属性としては、所有者、大きさ、許されたアクセス、最終更新時刻等がある。

```
object_setattr: proc( attr:attribute_t ) void = 12 ;
```

これは、属性を変更するオブジェクト手続きである。

```
object_lowers: proc( void ) lowers:oid_attry_t = 13 ;
```

これは、下位層のオブジェクトを返すオブジェクト手続きである。

```
object_kill: proc( void ) void = 14 ;
```

これは、オブジェクトを消去するオブジェクト手続きである。

```
any_object: set of  proc =
{
        object_null,
        object_getattr,
        object_setattr,
        object_lowers,
        object_kill,
};
```

任意のオブジェクトは、このような５つのオブジェクト手続きを受け付けることが要求される。

Ｂ－５　ファイル・サーバとファイル・オブジェクト

ここでは、ファイル・オブジェクトとしてのオブジェクト手続きとファイル・サーバとしてのサーバ手続きを示す。オブジェクト手続きの場合、隠された引数 this:oid_t で指定されたオブジェクトが操作対象となる。

```
file_server_null: sproc( void ) void = 50 ;
```

これは、ファイル・サーバかどうかを確認するためのサーバ手続きである。

```
file_balloc: proc( where,count:int ) void = 53 ;
```

これは、指定された領域を確保するたオブジェクト手続きである。where は、確保する領域の先頭のオフセット、count は、確保する領域の長さである。これ以後、その部分に書込みが行われるまで、読み出すと０で埋められた領域が返される。

```
file_bfree: proc( where,count:int ) void = 54 ;
```

これは、指定された領域を開放するオブジェクト手続きである。where は、開放する領域の先頭のオフセット、count は、開放する領域の長さである。これ以後、この部分を読み出すと、エラーとなる。

```
file_read: proc( where,count: int ) buff:opaque = 55 ;
```

これは、指定された領域を読み込むオブジェクト手続きである。where は、読み込む領域の先頭のオフセット、count は、読み込む領域の長さである。結果として、データが格納された不定型のバッファを返す。

```
file_write: proc( where,count: int; buff:opaque ) void = 56 ;
```

これは、指定された領域に書込みを行うオブジェクト手続きである。where は、確保する領域の先頭のオフセット、count は、書込む領域の長さ、buff は、書込むデータである。

```
file_object: set of proc =
        any_object +
        {
                file_balloc,
                file_bfree,
```

```
        file_read,
        file_write,
    };
```

ファイル・オブジェクトは、任意のオブジェクト any_object に加えて、これら4つのオブジェクト手続きを受け付けることが要求される。

```
file_server_procs: set of sproc =
    any_server +
    { file_server_null } +
    file_object ;
```

ファイル・サーバは、any_serverに指定された任意のサーバに対するサーバ手続きに加えて、サーバ手続きfile_server_null、および、file_objectに指定されたオブジェクト手続きを受け付けることが要求される。

```
file_server: type = server( file_server_procs );
```

server型構成子により、ファイル・サーバのスタブが生成される。

B－6　ディレクトリ・サーバとディレクトリ・オブジェクト

ここでは、ディレクトリ・オブジェクトとしてのオブジェクト手続きとディレクトリ・サーバ（名前サーバ）としてのサーバ手続きを示す。オブジェクト手続きの場合、隠された引数 this:oid_t で指定されたオブジェクトが操作対象となる。

```
dir_server_null: sproc( void ) void = 30 ;
```

これは、ディレクトリ・サーバかどうかを判定するためのサーバ手続きである。

```
dir_lookup: proc( name:string ) oid:oid_t = 34 ;
```

これは、ディレクトリの検索を行うオブジェクト手続きである。引数として与えられた名前 name をキーとして、そのオブジェクトを検索し、結果としてヒットしたオブジェクトを oid に返す。

```
dir_register: proc( oid:oid_t; name:string ) void = 35 ;
```

これは、そのディレクトリに、引数として与えられたオブジェクト識別子 oid と名前 name の組を登録するオブジェクト手続きである。

```
dir_remove: proc( name:string ) oid:oid_t = 36 ;
```

これは、そのディレクトリから、その名前 name のエントリを削除するオブジェクト手続きである。

```
dir_readdir: proc( void ) array of dir_entry_t = 37 ;
```

これは、そのディレクトリの内容を全て返すようなオブジェクト手続きである。

```
dir_object: set of proc =
        any_object +
        {
                dir_lookup,
                dir_register,
                dir_remove,
                dir_readdir,
        };
```

ディレクトリ・オブジェクトは、任意のオブジェクト any_object に加えて、これら4つのオブジェクト手続きを受け付けることが要求される。

```
dir_server_procs: set of sproc =
        any_server +
        { dir_server_null } +
        dir_object ;
```

ディレクトリ・サーバは、any_serverに指定された任意のサーバに対するサーバ手続きに加えて、サーバ手続き dir_server_null、および、dir_objectに指定されたオブジェクト手続きを受け付けることが要求される。

```
dir_server: type = server( dir_server_procs );
```

server型構成子により、ディレクトリ・サーバのスタブが生成される。

B－7　メモリ・サーバとメモリ・オブジェクト

ここでは、メモリ・サーバとしてのサーバ手続きを示す。メモリ・オブジェクトとしての固有のオブジェクト手続きは、存在しない。その代わり、メモリ・オブジェクトは、以下で示すように、ファイル・オブジェクトに対するオブジェクト手続きを受け付ける。

```
mo_server_null: sproc( void ) void = 60 ;
```

これは、メモリ・サーバかどうかを確認するためのサーバ手続きである。

```
mo_server_mo_create: sproc( size:int ) mo:oid_t = 61 ;
```

これは、メモリ・オブジェクトを生成するサーバ手続きである。この手続きは、sizeで指定された長さのメモリ・オブジェクトを生成し、結果として、生成したメモリ・オブジェクトのオブジェクト識別子を返す。

```
mo_server_getpagesize: sproc( void ) pagesize:int = 62 ;
```

これは、ページ・サイズを返すサーバ手続きである。

```
memory_object: set of proc =
        file_object ;
```

メモリ・オブジェクトは、file_object に指定されたファイル・オブジェクトに対する手続きを受け付ける。

```
mo_server_procs: set of sproc =
        any_server +
```

```
	{
		mo_server_null,
		mo_server_mo_create,
		mo_server_getpagesize,
	} +
	memory_object ;
```

メモリ・サーバは、any_serverに指定された任意のサーバに対するサーバ手続きに加えて、３つのサーバ手続き mo_server_null, mo_server_mo_create, mo_server_getpagesize, および、memory_objectに指定されたオブジェクト手続きを受け付けることが要求される。

```
mo_server: type = server( mo_server_procs );
```

server型構成子により、メモリ・サーバのスタブが生成される。

Ｂ－８　プロセス・サーバとプロセス・オブジェクト

ここでは、サーバ手続き、および、遠隔手続き呼出しにより他のプロセスを操作することが可能なオブジェクト手続きを示す。オブジェクト手続きの場合、隠された引数 this:oid_t で指定されたオブジェクトが操作対象となる。この他に、仮想プロセッサ関係の手続きとして、同一のプロセスに対してのみ利用可能なオブジェクト手続きが存在する。それらについては、付録Ｃにおいて説明する。

```
process_server_null: sproc( void ) void = 70 ;
```

これは、プロセス・サーバかどうかを確認するためのサーバ手続きである。

```
process_exit: proc( status:int ) void = 71 ;
```

これは、プロセスを終了するオブジェクト手続きである。status は、終了コードである。

```
process_stop: proc( void ) void = 72 ;
```

これは、プロセスの実行を一時停止するオブジェクト手続きである。

```
process_cont: proc( void ) void = 73 ;
```

これは、一時停止したプロセスの実行を再開するオブジェクト手続きである。

```
process_getpri: proc( void ) pri:int = 74 ;
```

これは、プロセスの優先順位を得るオブジェクト手続きである。

```
process_setpri: proc( pri:int ) void = 75 ;
```

これは、プロセスの優先順位を設定するオブジェクト手続きである。

```
process_get_status: proc( void ) stat:int = 76 ;
```

これは、プロセスの終了コードを得るオブジェクト手続きである。

```
process_mo_map:
proc( mo:oid_t, addr:caddr_t; len:int; prot:prot_t ) void = 77 ;
```

これは、メモリ・オブジェクトをプロセスのアドレス空間にマップするオブジェクト手続きである。moで指定されたメモリ・オブジェクトを addr にマップする。len には、マップする長さを、prot には、保護のフラグを指定する。

```
process_mo_unmap:
proc( addr:caddr_t; len:int ) void = 77 ;
```

これは、メモリ・オブジェクトをプロセスのアドレス空間からアンマップするオブジェクト手続きである。addr から len バイトにマップされているメモリ・オブジェクトをアンマップする。

```
process_object: set of proc =
        any_object +
        {
                file_read,
                file_write,
        } +
        {
                process_exit,
                process_stop,
                process_cont,
                process_getpri,
                process_setpri,
                process_get_status,
                process_mo_map,
                process_mo_unmap,
        };
```

プロセス・オブジェクトは、任意のオブジェクト any_object に加えて、ファイル・オブジェクトに対する file_read, write_write オブジェクト手続き、および、プロセス専用の８つのオブジェクト手続きを受け付ける。

```
process_server_procs: set of sproc =
        any_server +
        { process_server_null } +
        process_object ;
```

プロセス・サーバは、any_serverに指定された任意のサーバに対するサーバ手続きに加えて、サーバ手続き process_server_null、および、process_objectに指定されたオブジェクト手続きを受け付ける。

```
process_server: type = server( process_server_procs );
```

server型構成子により、プロセス・サーバのスタブが生成される。

Ｂ－９　カーネル・サーバ群

ここでは、カーネル中に含まれるサーバを示す。

```
StdPS: process_server = 1 ;
```

カーネル中には、標準プロセス・サーバ StdPS (Standard Process Server) が存在する。これは、プロセス・サーバとしての手続きを受け付ける。

StdDS: dir_server = 2 ;

カーネル中には、標準ディレクトリ・サーバ StdDS (Standard Directory Server)が存在する。これは、dir_server により示されるディレクトリ・サーバとしての手続きを受け付ける。このサーバは、標準名前サーバ (StdNS, Standard Name Server) とも呼ばれる。

StdFS: file_server = 3 ;

カーネル中には、標準ファイル・サーバ StdFS (Standard File Server)が存在する。これは、file_server により示されるプロセス・サーバとしての手続きを受け付ける。

StdMS: mo_server = 4 ;

カーネル中には、標準メモリ・サーバ StdMS (Standard Memory Server)が存在する。これは、mo_server により示されるメモリ・サーバとしての手続きを受け付ける。

KernelTime: server({file_read, file_write}) = 5 ;

カーネル中には、時間サーバが存在する。これは、ファイルに対する読み込み／書込みを行う手続きをインタフェースとして、現在の時刻の取得／設定を行う。

付録C　マイクロプロセス・ライブラリの機能

ここでは、マイクロプロセス・ライブラリに含まれている主要な手続きについて述べる。

C－1　全体の構成

マイクロプロセス・ライブラリは、第０層から第３層までの４層構造になっている。それぞれの層の主な機能は、次の通りである。

第０層では、カーネルによって提供される仮想プロセッサを実現するカーネル・コールが提供される。

第１層では、スピン・ロック（Ｃ－４．１参照）とコンテキスト切替え（Ｃ－４．２参照）を実現する機能が提供される。この層を利用することにより、各プロセッサ・アーキテクチャやバス・アーキテクチャから独立したプログラムを記述することが可能となる。

第２層では、基本的なマイクロプロセスの生成・消滅機能と、sleep/wakeup モデルに基づく簡単な同期機能が提供される。この層は、マイクロプロセス・スケジューラ（Ｃ－７参照）とレディ・キュー・モジュール（Ｃ－８参照）を含んでいる。これらは、応用固有のスケジューリングを行うために利用される。

第３層では、セマフォ（Ｃ－６．１参照）、モニタ（Ｃ－６．２参照）、ランデブ（Ｃ－６．３参照）、バイト・ストリーム（Ｃ－６．４参照）といった同期・通信プリミティブを提供する。これらのプリミティブのソース・コードは、各並列プログラム固有の同期・通信プリミティブを構築するプログラマにプロトタイプを与える。

C－2　データ型

マイクロプロセス・ライブラリでは、次のようなデータ型を利用する。

env_t
　　これは、マイクロプロセスのコンテキスト（環境、environment）を保持する変数ための型である。

lock_t
　　これは、スピン・ロックのための型である。

struct mpcb ;
typedef struct mpcb *mp_t ;
　　これは、マイクロプロセス制御ブロック（mpcb, microprocess control block）を表す型である。マイクロプロセスを操作する時に、この型の値をを指定する。mp_t は、struct mpcb * と同じものである。

enum mpstat
　　これは、マイクロプロセスの状態を表す型である。次の５つの状態がある。

M_FREE	初期
M_READY	実行可能

M_RUNNING　　実行中
M_BLOCKED　　待ち
M_TERMINATE　　終了

usec_t

これは、μ秒単位の時間を表現する型である。（実際に有効な精度は、アーキテクチャによって異なる。）

C－3　第0層　カーネル・コール

仮想プロセッサは、カーネルにより実現される。仮想プロセッサに関連したカーネル・コールには、次の4つがある。

```
int vp_allocate( n )
int n ;
```

これは、n－1個の仮想プロセッサを新たに生成することを要求するカーネル・コールである。返り値は、仮想プロセッサの識別子である。プロセスは、生成されるとまず1個の仮想プロセッサにより実行される。そこで種々の初期設定を行った後で、このカーネル・コールを発行し並列処理を開始する。

```
int vp_sleep( t )
usec_t t ;
```

これは、現在仮想プロセッサと対応している実プロセッサを開放するカーネル・コールである。 t は、開放する時間のヒントである。単位は、μ秒である。このカーネル・コールは、応用プログラム内のマイクロプロセス・スケジューラにおいて、実行可能なマイクロプロセスが存在しなくなり、プロセッサを一時的に開放するときに用いられる。 t は、 単にヒントであり、正確に t μ秒後に実プロセッサが割り当てられ、応用プロセスに実行が戻ってくる保証はない。負荷が小さい時は、指定された時間より先に戻ってくることもある。

```
vp_wakeup( n )
```

これは、vp_switch()、 あるいは、vp_sleep() で実行を停止している仮想プロセッサの実行再開を要求するカーネル・コールである。n は、必要な仮想プロセッサの数の目安である。これは、カーネルにおいてプロセッサ割当てのヒントとして用いられる。この数より少ないプロセッサが割当てられることもあるし、逆に多くのプロセッサが割当てられることもある。ただし、このカーネル・コールを用いなかったとしても、vp_sleep() を発行した仮想プロセッサは、指定された時間が経過した後には、実行を再開する。

```
int vp_switch( vpid )
int vpid ;
```

これは、同一プロセス内の他の仮想プロセッサへ制御を切り替えるためのカーネル・コールである。スピンロックを保持したまま実プロセッサを奪われた仮想プロセッサに制御を移すために用いられる。vpidは、次に実行すべき仮想プロセッサのヒントである。カーネルは、このカーネル・コールが発行されると、以前に実プロセッサを横取りした仮想プロセッサに対して実プロセッサを割り当てる。そのプロセスの他の仮想プロセッサも vp_sleep()、または、vp_switch()を発行してプロセッサを開放している場合は、自分自身で掛けたロックを待っていることを意味する。したがって、この場合はエラーである。それ以外の場合は、カーネル・コールは、同じプロセスの別の仮想プロセッサ、または、他のプロセスの仮想プロセッサにコンテキストを切り替える。

C－4　第1層　スピンロックとコンテキスト切替え

C－4．1　スピンロック

```
void lock_init( lp )
lock_t *lp ;
```

lp で指定されたスピン・ロック変数の初期化を行う。

```
int lock( lp )
lock_t *lp ;
```

lp で指定されたスピン・ロック変数に対して、ロックの保持を試み、成功するまで復帰しない。これは、プロセッサの test-and-set 命令や swap 命令を用いて実現されている。ただし、長い時間ロックが開放されない場合は、カーネル・コール vp_switch() により、他の仮想プロセッサへ制御を移すことがある。

```
int lock_try( lp )
lock_t *lp ;
```

lp で指定されたスピン・ロック変数に対して、ロックの保持を試み、その結果を報告する。成功した場合、返り値として LOCK_SUCCESS を返し、失敗した場合、LOCK_FAILED を返す。

```
void unlock( lp )
lock_t *lp ;
```

lp で指定されたスピン・ロック変数に掛けられているロックを開放する。

C－4．2　コンテキストの生成と切替え

```
void env_make( env,sp,pc,argv )
env_t *env ;
unsigned long int sp,pc ;
int argv[ENV_MAXARGC] ;
```

コンテキスト（環境）を生成する。spは、スタック・ポインタ、pc は、プログラム・カウンタ、argv は、軽量プロセスに与える引数である。最大 ENV_MAXARGC 個（現在は6個）の4バイト整数が渡される。

```
int env_save( env )
env_t *env ;
```

現在のコンテキストを保存する。返り値は、0である。ただし、env_restore() から復帰する時には、その第2引数で指定された値が返される。したがって、env_restore() では、0以外の値を指定すべきである。

```
void env_restore( env,value )
env_t *env ;
int value ;
```

コンテキストを env で指定されたものに切り替える。これにより、制御が env_save() で保存された所にもどる。このとき、value が env_save() の返り値となる。この手続きは、決して復帰することがない。

C－5　第2層　マイクロプロセスの生成と基本的な同期

Ｃ－５．１　現在実行中のマイクロプロセス

```
private extern mp_t current ;
```

この変数は、現在実行中のマイクロプロセスを保持している。private は、仮想プロセッサごとの固有領域に割り付けられる変数であることを示している。

Ｃ－５．２　マイクロプロセスの生成・消滅

```
mp_t mp_create( func, ssize, istat, arg0, arg1, arg2, arg3, arg4, arg5 )
void (*func)();
int ssize ; /* stack size */
enum istat ; /* M_READY or M_BLOCKED */
int arg0, arg1, arg2, arg3, arg4, arg5 ;
```

マイクロプロセスを生成する。func には、最初に実行される関数のポインタを指定する。ssize は、マイクロプロセスのスタックの大きさである。istat は、マイクロプロセスの初期状態である。M_READY が指定され時には、mp_wakeup() が自動的に実行される。arg0 以下は、生成されるマイクロプロセスに対する引数である。最大 ENV_MAXARGC 個（現在は６個）まで、生成された軽量プロセスに渡される。返り値は、mp_t 型である。

```
mp_t mpcb_init( m, func, ssize, argv, vf )
mp_t m ;
void (*func)() ;
int ssize ;
int *argv ;
mpcb_vftab *vf ;
```

標準のマイクロプロセスに新たに機能を付加する時に用る、マイクロプロセスを生成する手続きである。たとえば、

```
struct my_mpcb
{
        struct mpcb      m ;
        <独自のフィールド>
};
```

のように、独自のフィールドをマイクロプロセスに付加することが可能となる。func、ssize は、mp_create() と同じである。func には、最初に実行される関数のポインタを指定する。ssize は、マイクロプロセスのスタックの大きさである。なお、マイクロプロセスの初期状態は、M_BLOCKED である。vf は、Ｃ＋＋言語における virtual function の表と同様の役割を持つものである。この表には、このマイクロプロセスが終了した時に呼出される関数 mvt_final と、デバッグ用の表示関数 mvt_print を登録しておく必要がある。

```
struct mpcb_vftab
{
        void          (*mvt_final)();
        void          (*mvt_print)();
};
typedef struct mpcb_vftab mpcb_vftab ;
```

```
mp_exit()
```

現在実行中のマイクロプロセスを終了する。ただし、mp_create() において func により指定された関数から復帰すると、自動的にこの関数が呼出される。

C－5．3　マイクロプロセス間の同期

```
void mp_sleep( next,wait_for )
mp_t next ;
```

現在実行中のマイクロプロセスの実行を中断する。next は、次に実行するマイクロプロセスのヒントとして使われる。wait_for は、どのマイクロプロセスを待っているかを表もので、実行の軌跡を得るために使われる。内部的には、current で示された現在実行中のマイクロプロセスのコンテキストを保存し、マイクロプロセスの状態を待ち状態にする。プロセッサは、マイクロプロセスのスケジューラの mp_switch() を呼出す。これを呼出す前には、かならず現在実行中のマイクロプロセス current をロックしておく必要がある。

```
void mp_wakeup( w )
mp_t w ;
```

指定されたマイクロプロセス w を実行可能状態 M_READY に変える。すでに実行可能状態 M_READY あるいは、実行中状態 M_RUNNING の時には、なにもしない。

```
void mp_yield( w )
mp_t w ;
```

マイクロプロセス w へコンテキストを切り替える。現在実行中のマイクロプロセス current は、実行可能状態になる。w に 0 が指定されると、現在実行中のマイクロプロセス以外のマイクロプロセスに制御を切替える。マイクロプロセス w は、既に実行可能状態（MP_READY）になっている必要がある。M_RUNNING、M_BLOCKED の時は、mp_yield(0) と同様、別のマイクロプロセスへ切替わる。これを呼出す前には、かならず現在実行中のマイクロプロセス current をロックしておく必要がある。

C－6　第3層　高度な同期通信プリミティブ

C－6．1　セマフォ

このモジュールは、計数セマフォ機能を提供する。

```
sem_t *
sem_init( sem, n )
sem_t *sem ;
int n
```

セマフォの初期化を行う。nは、計数セマフォの初期値である。

```
void    sem_p( sem )
```

セマフォsemに対して、Ｐ命令を実行する。

```
void    sem_v( sem )
```

セマフォsemに対して、Ｖ命令を実行する。

C－6．2　モニタ

このモジュールは、モニタと条件変数（condition variable）の機能を提供する。

```
mon_t *
mon_init( mon )
mon_t *mon ;
```
モニタ構造体 mon の初期化を行う。

```
void
mon_final( mon )
mon_t *mon ;
```
モニタ構造体 mon を破壊する。

```
void
mon_enter( mon )
mon_t *mon ;
```
モニタの入口操作を実現する。

```
void
mon_exit( mon )
mon_t *mon ;
```

モニタの出口操作を実現する。

```
cv_t
cv_init( cv, mon )
cv_t *cv ;
mon_t *mon ;
```
条件変数 cv を初期化する。monには、対応するモニタを指定する。

```
cv_final( cv )
cv_t *cv ;
```
条件変数を破壊する。

```
void cv_wait( cv )
cv_t *cv ;
```
条件変数 cv に、信号が送れるまで現在実行中のマイクロプロセスを待ち状態にする。

```
cv_signal( cv )
cv_t *cv ;
```
条件変数 cv で待っているマイクロプロセスを、実行可能状態にもどす。

C－6．3　ランデブ

このモジュールは、ランデブ機能を提供する。

```
msg_t msg_create( entc )
int entc ;
```
ランデブの受付けポイントを生成する。entc には、受付けを行うエントリの数を指定する。返り値は、ランデブの受付けポイントである。

```
void msg_set_receiver( msg, receiver )
msg_t msg ;
mp_t receiver ;
```
ランデブの受付けポイント msg に対するランデブの受付けを行うマイクロプロセスとして、receiver を登録する。

```
void
msg_send( msg, eno, arg_size, arg_buff, res_size, res_buff )
msg_t msg ;
int eno ;
int arg_size, res_size ;
char *arg_buff, *res_buff ;
```
ランデブの呼出しを行う。eno には、呼出すエントリの番号を指定する。arg_size は、エントリ呼出し引数の大きさ、arg_buff は、引数の先頭番地である。res_size、 res_buff には、それぞれエントリ呼出しの結果の大きさと先頭番地を指定する。

```
int
msg_receive( msg, arg_sizep, arb_buffp, res_sizp, res_buffp )
msg_t msg ;
```

ランデブの受付けを行う。*arg_sizep、 *arb_buffp には、それぞれ呼出し側で用意されているエントリ呼出しの引数の大きさと先頭番地が返される。 *res_sizep、 *res_buffp には、それぞれ呼出し側で用意されているエントリ呼出しの結果の大きさと先頭番地が返される。関数の結果としては、エントリの番号が返される。

```
void
msg_reply( msg, eno )
msg_t msg ;
int eno ;
```
ランデブの応答を行う。eno には、応答を行うエントリを指定する。

```
void
msg_enable( msg, eno )
msg_t msg ;
int eno ;
```
エントリ番号 eno に対するランデブの受付けを可能にする。

```
void
msg_disable( msg, eno )
msg_t msg ;
int eno ;
```
エントリ番号 eno に対するランデブの受付けを不可能にする。

Ｃ－６．４　ストリーム

このモジュールは、ＵＮＩＸのパイプに似たストリーム通信機能を提供する。

```
pipe_t pipe_make( size, npage )
int     size ;
int     npage ;
```

ストリームの通信路であるパイプを生成する。size には、パイプのためのバッファの大きさを指定する。npage には、内部的なバッファの数を指定する。

```
port_t pipe_open( pipe,mode )
pipe_t pipe ;
enum porttype mode ;     /* PORT_INPUT,PORT_OUTPUT */
```

パイプ pipe をモード mode で開く。実際のデータの読み書きは、この返り値である port に対して行われる。mode には、入力モード PORT_INPUT、または、出力モード PORT_OUTPUT を指定する。

```
int port_read( p,buff,count )
port_t p ;
char *buff ;
int count ;
```

ポート p から、buff で指定されたバッファへ count バイト読み込む。

```
int port_write( p,buff,count )
port_t p ;
char *buff ;
int count ;
```

ポート p へ、buff で指定されたバッファから count バイト書き込む。

C－7　マイクロプロセス・スケジューラ

第２層に含まれているマイクロプロセス・スケジューラは、次のような外部仕様を持っている。これらの手続きは、スケジューリングの仕組み（mechanism）を提供するものである。スケジューリングの方針は、次の節で説明するレディ・キュー・モジュールにより決定される。

```
void sched_init()
```

スケジューラの初期化を行う。

```
void sched_main()
```

main() からの呼出しポイントである。復帰することはない。

```
void scheduler()
```

mp_exit() からの呼出しポイントである。復帰することはない。

```
void mp_switch( next )
mp_t next;
```

mp_sleep()、mp_yield() からの呼出しポイントである。現在実行中のマイクロプロセスの実行を中断し、nextで指定されたマイクロプロセスに制御を切り替える。nextに0が指定された場合、レディ・キューより実行可能なマイクロプロセスを選択し、実行する。再びプロセッサが割り当てられた時に、この手続きから復帰する。

C－8　レディ・キュー・モジュール

この節では、マイクロプロセス・スケジューラ・モジュールが使うレディ・キューモジュールの外部仕様を示す。

```
void ready_init()
```

初期化を行う。

```
void ready_enqueue( m )
mp_t m ;
```
実行可能なマイクロプロセスをキューに入れる。

```
mp_t ready_dequeue()
```
実行可能なマイクロプロセスをキューから取りだす。

```
mp_t ready_remove( m )
mp_t m ;
```
キューから指定されたmpを取りだす。

```
void ready_replace( old,new )
mp_t old,new ;
```
キューに含まれているマイクロプロセス old を new で置き換える。これは、

```
ready_remove( old )
ready_enqueue( new )
```
と等価である。

```
void ready_print()
```
デバッグ用に、キューの内容を表示する。

付録D　プロセス操作の性能を計測するためのベンチマーク

ここでは、軽量プロセス、および、重量プロセスの性能を比較する時に用いたベンチマーク・プログラムの骨格を示す。ここであげるプログラムでは、多くの処理系で利用可能な、プロセスの生成と同期のプリミティブを利用している。

D－1　プロセスの生成・消滅

プロセスの生成・消滅の性能を測定するために用いたプログラムの骨格を以下に示す。

```
 1: run( m, n )
 2:     int n,m ;
 3: {
 4:     int j ;
 5:         for( j=0 ; j<n ; j++ )  /* outer loop */
 6:         {
 7:             process_create( child,m );
 8:                         /* create a light- or heavy- weight process */
 9:             wait_for_child_exit();
10:         }
11: }
12:
13: child( rem )    /* inner loop by recursive call */
14:     int rem ;
15: {
16:         if( rem > 0 )
17:         {
18:             process_create( child,rem-1 );
19:                         /* create a light- or heavy- weight process */
20:             wait_for_child_exit();
21:         }
22:         exit();
23: }
```

このプログラムは、外側の明示的なループと内側の暗黙的なループの２重ループ構造をもつ。第5行の for 文が外側のループを形成している。内側のループは、プロセス child の第１８行において、自分自身を再帰的に生成することにより実現されている。この結果、内側のループでは、プロセスの入れ子が形成される。これにより、共有メモリ型マルチプロセッサにおける並列処理の効果、および、軽量プロセスやスタックのキャッシングの効果が抑えられる。逆に、外側のループの実行回数を変化させることで、キャッシングの効果を調べることができる。

このプログラムは、プロセスの生成・消滅の他に、コンテキスト切替え処理（親から子へ、子から親へ）、同期処理（子供の終了待ち）を含んでいる。

D－2　コンテキスト切替えを伴わないプロセス間の同期

コンテキスト切替えを伴わないプロセス間の同期の性能を測定するために用いたプログラムの骨格を以下に示す。

```
 1: run( count )
 2:     int count ;
 3: {
 4:     int i ;
 5:     lock_t l ;
 6:         for( i=0 ; i<count ; i++ )
 7:         {
 8:             lock( &l );
 9:             unlock( &l );
10:         }
11: }
```

このプログラムでは、１つのプロセスにおいてロックに対するロック操作／アンロック操作を行っている。このプログラムでは、ロック操作とアンロック操作を合せて１回の操作と数える。ロック／アンロックの代わりに、セマフォの場合、Ｐ命令／Ｖ命令、モニタの場合、入口操作／出口操作を用いた。

D－3　コンテキスト切替えを伴うプロセス間の同期

コンテキスト切替えを伴うプロセス間の同期の性能を測定するために用いたプログラムの骨格を以下に示す。ここでは、セマフォ、モニタ、および、強制的にコンテキスト切替えを起こすプリミティブを用いる例を示す。ただし、共有メモリ型マルチプロセッサでは、これらのプログラムを用いたとしても、実際にはコンテキスト切替えが起こらない可能性がある。すなわち、２つのプロセスが別々の実プロセッサにより実行され、実プロセッサのレベルでは、コンテキスト切替えが起こらない可能性がある。しかしながら、同期プリミティブのオーバヘッドを測定するためには、このプログラムで十分である。

D－３．１　セマフォを用いたプログラム

セマフォを用いたプログラムを以下に示す。

```
 1: sem_t sem0 ;
 2: sem_t sem1 ;
 3:
 4: p0( count )
 5:     int count ;
 6: {
 7:     int i ;
 8:         for( i=0 ; i<count ; i++ )
 9:         {
10:             P( &sem0 );
11:             V( &sem1 );
12:         }
13: }
14:
```

```
15: p1( count )
16: {
17:         for( i=0 ; i < rcount ; i++ )
18:         {
19:             V( &sem0 );
20:             P( &sem1 );
21:         }
22: }
```

このプログラムでは、２つの私有セマフォを用いている。

Ｄ－３．２　モニタと条件変数を用いたプログラム

モニタと条件変数を用いたプログラムを以下に示す。

```
 1:
 2: monitor_t                mon ;
 3: conditional_variable     cv0 ;
 4: conditional_variable     cv1 ;
 5:
 6: p0( count )
 7:     int count ;
 8: {
 9:     int i ;
10:         mon_enter( &mon );
11:         for( i=0 ; i<count ; i++ )
12:         {
13:             cv_wait( &cv0,&mon );
14:             cv_signal( &cv1,&mon );
15:         }
16:         mon_exit( &mon );
17: }
18:
19: p1( count )
20: {
21:         mon_enter( &mon );
22:         for( i=0 ; i < rcount ; i++ )
23:         {
24:             cv_signal( &cv0,&mon );
25:             cv_wait( &cv1,&mon );
26:         }
27:         mon_exit( &mon );
28: }
```

mon_enter() は、モニタの入口操作、mon_exit() は、出口操作である。cv_wait() は、その条件が起きるまで待つプリミティブ、cv_signal() は、条件を起こすプリミティブである。

Ｄ－３．３　コンテキスト切替えを起こすプリミティブを用いたプログラム

コンテキスト切替えを起こすプリミティブを用いたプログラムを以下に示す。

```
 1: p0( count )
 2:     int count ;
 3: {
 4:     int i ;
 5:         for( i=0 ; i<count ; i++ )
 6:         {
 7:             yield( p1 );
 8:         }
 9: }
10:
11: p1( count )
12: {
13:         for( i=0 ; i < rcount ; i++ )
14:         {
15:             yield( p1 );
16:         }
17: }
```

ここで、yield() がコンテキスト切替えを要求するプリミティブである。これに相当するプリミティブとしては、マイクロプロセス・ライブラリでは、mp_yield()、ＲｅＳＣカーネルでは、vp_switch()、ＳｕｎＬＷＰでは、lwp_yield()、Ｃ－Ｔｈｒｅａｄｓでは、cthread_yield() があげられる。

題目：　　　　並列分散応用プログラムを対象とした
　　　　　　　オペレーティング・システムの研究
著者：　　　　新城　靖
日付：　　　　１９９２年１２月

連絡先

住所：　　　　〒３０５　茨城県つくば市筑波大学
　　　　　　　電子・情報工学系ソフトウェア研究室
電話：　　　　0298-53-5163, 0298-53-6455
Ｆａｘ：　　　0298-53-5206
電子メール：　postmaster@softlab.is.tsukuba.ac.jp

Title: A Study on an Operating System for Parallel and Distributed Applications
Author: Yasushi Shinjo
Date: December 1992

Contact

Address: Software Lab.
Institute of Information Sciences and Electronics
University of Tsukuba
Tsukuba, Ibaraki 305
Japan
Phone: +81 298 53 5163, +81 298 53 6455
Fax: +81 298 53 5206
E-Mail: postmaster@softlab.is.tsukuba.ac.jp