システムソフトウェアと
オペレーティング・システム 70－3
（１９９５．８．２２）

粗粒度投機的処理を支援するオペレーティング・システムにおけるファイル・システム

當眞 聡　新城 靖　溝淵 雅也　根路銘 崇
<{ha,yas,miz,nero}@ocean.ie.u-ryukyu.ac.jp>
喜屋武 盛基　翁長 健治
<kyan@ie.u-ryukyu.ac.jp> <onaga@ie.u-ryukyu.ac.jp>

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

概要　投機的処理とは、その結果が利用されるかどうか確定される前に実行を開始する処理である。われわれは、プロセス単位の投機的処理を支援するオペレーティング・システムを開発している。多重プログラミング環境で投機的処理を扱うために世界という概念を導入した。世界とは、プロセスやファイルを入れるための箱である。本論文では、このオペレーティング・システムのファイル・システムのインタフェースと実現について述べる。インタフェースの特徴は、ファイルが名前と世界を指定して識別されることにある。実現においては、名前サブシステムが中心的な役割を果たしている。

The File System of an Operating System Supporting Speculative Processing

Hajime Toma,　Yasushi Shinjo,　Masaya Mizobuchi,　Takashi Nerome,
<{ha,yas,miz,nero}@ocean.ie.u-ryukyu.ac.jp>

Seiki Kyan　and　Kenji Onaga
<kyan@ie.u-ryukyu.ac.jp> <onaga@ie.u-ryukyu.ac.jp>

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

Abstract　Speculative processing is processing that is started eagerly before it is known to be required. We are developing an operating system that supports process-level speculative processing. We introduce the idea of *worlds* to deal with speculative processing. A world is a container which contains processes and files. This paper describes the interface and implementation of the file system of this operating system. The feature of the interface is that files are identified by their names and worlds. In this implementation, the name subsystem plays the main role.

1　はじめに

今日では、ワークステーションが高速LANにより接続された環境が広く利用されている。また、1000プロセッサ規模の高並列計算機も登場してきている。このような計算機環境では、多くの余剰計算機資源が存在する。

投機的処理（speculative processing）とは、その結果が利用されるかどうか確定される前に実行を開始する処理である。投機的処理ではない処理は、**必須の処理**（mandatory processing）と呼ばれている。余剰計算機資源を投機的処理に割り当てることにより、見かけの処理時間の高速化が望める。

我々は、投機的処理を支援するオペレーティング・システムを研究している。今までソフトウェア・レベルの投機的処理の研究は、並列Prologや並列Lispなど、プログラミング言語レベルにおける細粒度投機的処理を中心に行われてきた。細粒度投機的処理の目的は、主に同期のオーバヘッドを減らすことである。我々が扱う投機的処理は、多重プログラミング環境においてファイルの入出力を行うプロセスを単位とした**粗粒度投機的処理**である。その目的は、並列性を高めることである。

粗粒度投機的処理の実現を支援するために、我々は、**世界**（world）という概念を導入した。世界とは、プロセスやファイルを入れるための箱である。世界を用いることにより、既存のアプリケーションを投機的処理に利用することが可能となる。

我々は、すでに投機的処理を支援するオペレーティング・システムの基本構想を述べている[1]。また、世界やプロセスを扱うためのシステム・コールを提案している[2]。本論文では、粗粒度投機的処理を支援するオペレーティング・システムのファイル・システムの仕様と実現方法について述べる。その特徴は、ファイル名の操作により、ファイル・システムにおいて世界の考え方を実現している点にある。

2　世界の概念を導入したオペレーティング・システム

我々は、オペレーティング・システムのレベルで粗粒度投機的処理の実現を支援するために、**世界**（world）という概念を導入した。世界とは、プロセスやファイルを入れるための箱である。1つのシステムには、同時に複数の世界が存在する。プロセスやファイルは、ある1つの世界に属する。ある世界に属するプロセスは、その世界の中に新たにファイルを生成したり、同じ世界に属するファイルを読み書きすることができる。また、プロセスは、同じ世界に属するプロセスと通信することができる。

本オペレーティング・システムが提供する世界を操作するためのシステム・コールを以下に示す。

world_create()：指定された世界を引き継いだ、新たな世界を生成する。このとき、指定された世界を親、生成された世界を子とする親子関係が作られる。すべての世界の親子関係は、1つのDAG（Directed Acyclic Graph）を構成する。

world_merge()：指定された2つの世界を1つの世界に融合する。

world_delete()　：指定された世界を削除する。

2.1　世界の概念があるファイル・システム

利用者から見ると、1つの世界には、それぞれ独立したファイル・システムが存在するように見える。内部的には、ある世界のファイル・システムは、その親世界のファイル・システムとの差分をファイル単位で保持している。この場合、ファイルの操作は、以下のようになる。プロセスは、自分が属している世界にのみ、新規にファイルを生成することができる（図1）。生成されたファイルは、親世界から見ることはできない。

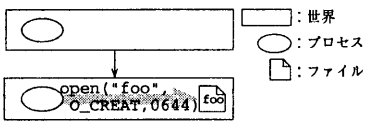

図1．ファイルの生成

プロセスが読み込み専用でファイルを開くとき、開こうとしたファイルがその世界に存在しなければ、そのファイルは、親世界を遡って探される。そして、最初に見つかったファイルが開かれる（図2）。

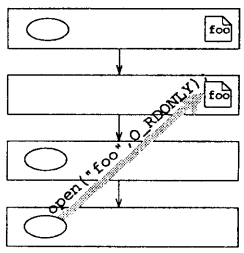

図2．ファイルの読み込み

プロセスが、他の世界に既に存在しているファイルを書き込みモードで開こうとした場合、そのファイルはプロセスと同じ世界に複製される。プロセスは、複製されたファイルに対して、書き込みを行う（図３）。

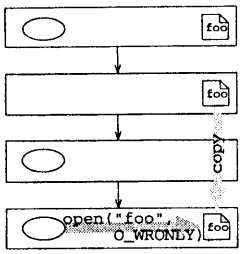

図３．ファイルの書き込み

2.2 システムの構成要素

世界の概念を導入したオペレーティング・システムは、次のようなサーバから構成される。

世界サーバ：世界の関係を保持する。
プロセス・サーバ：プロセスを管理する。
ファイル・サーバ：ファイルを管理する。

ユーザ・プロセスが発行したシステム・コールは、プロセス・サーバにより受け付けられる。プロセス・サーバは、システム・コールを解析し、世界サーバやファイル・サーバ、あるいは自分自身に処理を振り分ける。

世界サーバには、世界の操作のための３つのシステム・コールに対応した３つの手続きがある。その他に、引数に世界をとり、その世界の祖先のリストを返す手続き `get_ancestors()` がある。この手続きは、ファイル・サーバやプロセス・サーバから呼び出される。世界サーバは、世界の状態が変化したこと（生成・削除・融合）を、プロセス・サーバとファイル・サーバに知らせる機能を持つ。

3 世界ファイル・サーバ

２章では、本オペレーティング・システムが提供する世界の概念、利用者プロセスから見たファイル・システムの機能、および、システムの構成についてのべた。この章では、システム内部のファイル・サーバの仕様と実現について述べる。

我々は、NetBSD をもとにシステムの実現を行っている。既存の NetBSD のファイル・システムは、ファイルの実体に対する入出力を行う部分と、ファイル名を扱う部分に分けられる。我々は、世界の概念をファイル・システムにおいて実現するために、主にファイル名を扱う部分を変更することにした。そのために、既存のシステムの構造を整理した。

３.１節では、まず整理した NetBSD のファイル・システムの構造を示す。次に、本ファイル・システムの中心的な部分である名前サブシステムの外部仕様と、その１つの実現を示す。この実現の特徴は、既存のシステムの機能と２つの表を使う点にある。

3.1 UNIXファイル・システム

既存のUNIX[3]（NetBSD）では、カーネル中のファイル・システムとその他の部分の境界が不明確である。２章で述べたように、本システムでは、ファイルを扱う部分をファイル・サーバとして独立させる。この節では、既存のファイル・システムを、ファイル・サーバとして独立させたと考えて、その機能を整理する。ここでは簡単のため、マウント機能、シンボリック・リンク、特殊ファイルについては述べない。

UNIXのカーネルにおいてファイルの実体は、inode または、vnode と呼ばれる構造体で示されている。vnode は、NFS (Network File System) を扱うために inode を拡張したものである。

UNIXにおけるファイルの存在は、ファイル名により決定される。利用者プロセスは、ファイルを操作（読み書き）する前に、まず名前により操作対象のファイルを指定する。利用者プロセスは、inode (vnode) を直接指定してファイルを操作することはできない。UNIXには、ファイル名を消すシステム・コール（`unlink()`）は存在するが、ファイルを消すシステム・コールが存在しない。あるファイルの全ての名前が消されると、ファイルの実体も削除される。

UNIXには、世界の概念がないので、操作対象のファイルを指定する時にファイル名のみを指定する。この様子を、図４に示す。ファイル名としては、パス名が用いられている。パス名には、ルート・ディレクトリからのパス名である絶対パス名と、カレント・ワーキング・ディレクトリからのパス名である相対パス名がある。

図４．UNIXの名前によるファイルの識別

現在のUNIXのファイル・システムの構成を整理

したものを、図5に示す。一番上の層がプロセス・サーバ、残りがファイル・サーバである。ファイル・サーバは、上下2層に分けられる。上位層は、open_file構造体（UNIXでは、file構造体）を扱う層である。下位層は、ファイル入出力を行う部分とファイル名の解決を行う部分に分けられる。それぞれを、**ファイル・オブジェクト・サブシステム**、**名前サブシステム**と呼ぶことにする。前者は、ファイル・オブジェクト(file_t)、後者は、ディレクトリ・オブジェクト(dir_t)を扱う。それらは、それぞれファイルとディレクトリを示すinode (vnode)である。

ファイル・サーバは、プロセス・サーバに対してopen_file構造体を通じてファイル・サービスを提供している。open_file構造体は、開かれたファイルに対応する構造体である。この構造体は、ファイルを読み書きする位置（シーク・ポインタ）、および対応しているinode (vnode)へのポインタを含んでいる。

プロセス・サーバは、ファイルを開くシステム・コール（open()）を受け付けると、引数のファイル名とフラグ、プロセスの属性であるルート・ディレクトリ、カレント・ワーキング・ディレクトリ、**資格**（credential、利用者識別子やグループ識別子の集合）を引数としてファイル・サーバを呼び出す。ファイル・サーバは、内部の名前サブシステムを呼び出し、ファイル名をinode (vnode)へのポインタへ変換する。UNIXでこの操作の中心的な役割を果たしている手続きは、namei()である。そして、ファイル・サーバは、open_file構造体を割り当て、その内部にinode (vnode)へのポインタを保存し、シーク・ポインタなどを初期化したのち、そのopen_file構造体へのポインタをプロセス・サーバへ返す。プロセス・サーバは、返されたopen_file構造体のポインタを、プロセス構造体の中にある配列へ格納する。そして、格納した配列の添字をファイル記述子として利用者プロセスへ返す。

プロセス・サーバは、ファイルを読み書きするシステム・コール（read(),write()）を受け付けると、与えられたファイル記述子を用いてopen_file構造体へのポインタを得る。そして、そのopen_file構造体へのポインタを引数として、プロセス・サーバを呼び出す。プロセス・サーバは、open_file構造体より、inode (vnode)を取り出し、内部のファイル・オブジェクト・サブシステムを呼び出して、入出力を行う。

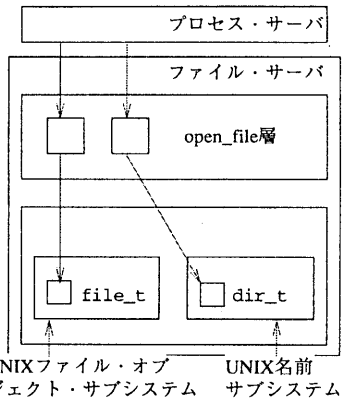

図5．UNIXのファイル・システムの構成

```
interface udir_subsystem {
  void udir_register_by_path(string_t path,
                    dir_t root_dir, dir_t cwd,
                    ucred_t ucred, obj_t obj);
  void udir_unregister_by_path(string_t path,
                    dir_t root_dir, dir_t cwd,
                              ucred_t ucred);
  obj_t udir_lookup_by_path(string_t path,
                    dir_t root_dir, dir_t cwd,
                              ucred_t ucred);
  dir_entry_list_t udir_readdir(dir_t dir);

  void udir_register(dir_t dir, string_t base,
                    ucred_t ucred, obj_t obj);
  void udir_unregister(dir_t dir,
                string_t base, ucred_t ucred);
  obj_t udir_lookup(dir_t dir, string_t base,
                              ucred_t ucred);
};
```

図6．UNIXのファイル・システムの名前サブシステムのインタフェース

ファイル・サーバの内部の名前サブシステムのインタフェースを図6に示す。udir_register_by_path() は、名前の登録、udir_unregister_by_path() は、名前の削除、udir_lookup_by_path() は、名前の検索を行う手続きである。引数のpath は、対象となるパス名を表す文字列、root_dir, cwd は、ルート・ディレクトリとカレント・ワーキング・ディレクトリ、ucredは、資格である。これらの引数は、プロセスの属性であり、プロセス・サーバから渡されたものである。obj_tは、操作対象のオブジェクトへのポインタである。obj_tは、file_tまたはdir_tのいずれかである。udir_readdir() は、ディレクトリ・オブジェクトの内容を読み出す手続きであり、ls コマンドなどから利用される。

udir_{register, unregister, lookup}()は、それぞれ
udir_{register, unregister, lookup}_by_path()におけるパス名を、ディレクトリとそのディレクトリに登録されるエントリに含まれる文字列（UNIXでは、ベース名と呼ばれる）にかえたものである。これらの手続きは、3.4節で述べる世界の概念を実現する名前サブシステムから利用される。

3.2 世界ファイル・システム

UNIXでは、パス名からファイルの実体を指定する。世界を考慮したファイル・システムでは、パス名と世界からファイルの実体を指定する。これを管理するファイル・システムを、世界ファイル・システムと呼ぶことにする。世界ファイル・システムでは、パス名が同じでも、世界が異なれば、ファイルの実体も異なる。図7に、ファイルの指定の様子を示す。ファイルは、世界と名前を軸とした2次元平面上に存在している。図中aは、複数の世界に、同じ名前のファイルが複数存在できることを表している。図中bは、一つの世界に、異なる名前を持つファイルが複数存在していることを示している。世界の融合／削除のシステム・コールの実現では、世界を指定して、ある世界に属するすべてのファイルを操作する必要がある。

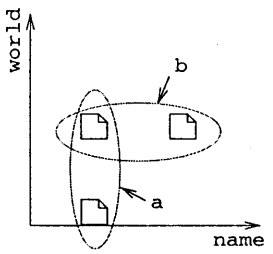

図7．世界ファイル・システムのファイル指定の様子

我々は、世界ファイル・システムを、UNIXのファイル・オブジェクト・サブシステムをそのまま用い、名前サブシステムの仕様を変更することにより実現することにした。3.1節で述べたように、UNIXでは、名前サブシステムにパス名をわたしている。我々の世界ファイル・システムでは、名前サブシステムにパス名と世界をわたす。世界は、プロセス・サーバから、世界オブジェクトへのポインタという形でわたされる。

3.3 名前サブシステムのインタフェース

名前サブシステムのインタフェースを図8に示す。手続きには、名前を登録／削除／検索する手続きwdir_{register,unregister,lookup}()がある。これらの手続きは、図6のUNIXの手続きudir_{register,unregister,lookup}()と対応しているが、引数に世界が必要である所が異なる。また、世界ファイル・システムには、ディレクトリの内容を返す手続きwdir_readdir()がある。

wdir_world_{create,merge,delete}()は、世界サーバから、それぞれ対応する操作が行われたときに呼び出される。

```
interface wdir_subsystem {
  void wdir_register(dir_t dir,string_t base,
                world_t world, ucred_t ucred,
                                obj_t obj);
  void wdir_unregister(dir_t dir,
                string_t base, world_t world,
                                ucred_t ucred);
  obj_t wdir_lookup(dir_t dir, string_t base,
                world_t world, ucred_t ucred);
  obj_t wdir_lookup_by_path(string_t path,
                    dir_t root_dir, dir_t cwd,
                world_t world, ucred_t ucred);

  dir_entry_list_t wdir_readdir(dir_t dir);

  viod wdir_world_create(world_t world);
  viod wdir_world_merge(world_t parent,
                                world_t child);
  viod wdir_world_delete(world_t world);
};
```

図8．名前サブシステムのインタフェース

この名前サブシステムを利用した、ファイルの生成システム・コール（creat()）とファイル名の削除システム・コール（unlink()）の実現の骨格を、それぞれ図9と図10に示す。

図9のfs_create()は、プロセス・サーバより引数として作成するファイルの名前を表す情報（ディレクトリ・オブジェクトとベース名）、モード、世界、資格を受け取る。引数として世界を渡されている点に特徴がある。結果として、open_file構造体へのポインタを返す。fs_create()は、すでに作成するファイルの名前と同じ名前のファイルが存在しなければ、ファイル・オブジェクトを新たに確保（file_obj_allocate()）する。得られたファイル・オブジェクトを、手続きwdir_register()を呼び出して、ディレクトリにファイル名と世界を

キーとして登録する。

ファイル名の削除を行うfs_unlink()の引数は、削除するファイルの名前を表す情報（ディレクトリとベース名）、世界、資格である。fs_unlink()は、与えられたファイル名と世界をキーとしてエントリを検索する。見つかった時には、そのエントリを削除する（wdir_unregister()）。見つからなかった時には、エラー（ENOENT）を返す。

```
fs_create(open_file_t *ofilep/*out*/,
          wdir_t dir, string_t base, int flags,
                    mode_t mode, world_t world,
                                ucred_t ucred) {
  obj_t obj = wdir_lookup(dir, base, world,
                                          ucred);
  if( found(obj) ){
    *filep = 0 ;
    return( EEXIST );
  }else{
    obj = file_obj_allocate(mode, ucred);
    wdir_register(dir,base,world,ucred,obj);
    open_file_t ofile =
       malloc(sizeof(struct open_file));
    open_file_init(ofile, obj, dir, base);
    *ofilep = ofile ;
    return( 0 );
  }
}
```

図9．ファイルの生成

```
fs_unlink(dir_t dir, string_t base,
                world_t world, ucred_t ucred){
  obj_t obj = wdir_lookup(dir, base, world,
                                          ucred);
  if( found(obj) ){
    wdir_unregister(dir, base, world, ucred);
    return( 0 );
  }else
    return( ENOENT );
}
```

図10．ファイルの削除

UNIXと比較して、これらの手続きの実現の特徴は、引数としてプロセス・サーバからプロセスの属している世界を受け取ること、および、名前サブシステムを呼び出す時にその世界を利用することである。他の部分は、UNIXの場合と同じである。

3.4　名前サブシステムの一実現方法

3.3節で示した世界ファイル・システムの名前サブシステムのインタフェースを実現する方法の一つとして、UNIXの名前サブシステムと2つの表を使う方法を提案する（図11）。すべての世界の祖先にあたる世界を根世界（root world）と呼ぶことにする。提案方式では、根世界に属するファイルの名前を管理するためにUNIXの名前サブシステムを用いる。根世界以外の世界に属するファイルの名前を管理するために、2つの表を用いる。

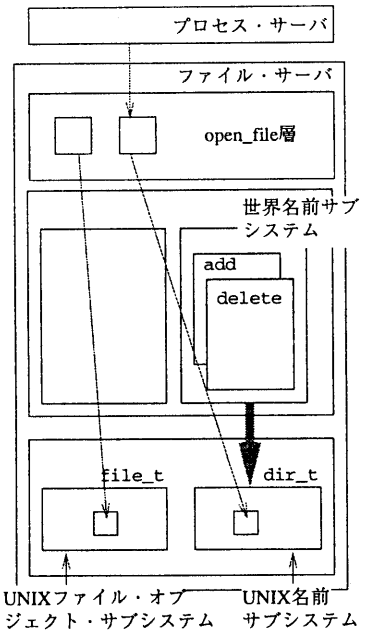

図11．UNIXの名前サブシステムと2つの表を用いた世界ファイル・システムの名前サブシステムの実現

表の構造とそれを操作する手続きを図12に示す。

```
typdef element_t table_t[];
struct element_t {
  world_t   world;
  dir_t     dir;
  string_t  base;
  obj_t     obj;

  element_t table_lookup(table_t,dir_t,
                    string_t,world_t,ucred_t);
  void table_register(table_t,dir_t,string_t,
                      world_t,ucred_t,obj_t);
  void table_unregister(table_t,dir_t,
                    string_t,world_t,ucred_t);
}
table_t  add;
table_t  delete;
```

図12．根世界以外のファイル名を管理する表

表の要素は、世界、ディレクトリ、ベース名、オブジェクトの4つである。表を操作する手続きとしては、世界と名前をキーとする検索、登録、削除がある。これらの手続きは、世界を単にビット・イメージとして用いるだけであり、世界をオブジェクトと

して操作することも、世界のDAGを遡ることもしない。

図１２に示されているように、表には、addとdeleteの２つがある。表addは、ある世界で作られたファイル名とオブジェクトの束縛を保持する。表deleteは、ある世界でそのファイル名が削除されたということを保持する。

この２つの表と、３.１節で述べた既存のUNIXの名前サブシステムを利用したwdir_lookup()の実現を、図１３に示す。get_ancestors()は、世界サーバに定義されている手続きであり、引数で与えられた世界の祖先の世界のリストを返す。wdir_lookup()は、このリストの順にファイル名を検索していく。検索の対象となる世界が根世界であれば、UNIXの名前サブシステムudir_lookup()に処理を委譲する。wdir_lookup()は、検索対象が表deleteに登録されていると、それは見つからなかったことにする。検索対象が表addに登録されているときは、そのオブジェクトを返す。オブジェクトが見つからないときは、次の世界を検索する。どの表にも登録されていない場合は、検索対象は、見つからなかったことにする。

wdir_register()の実現は、単純である。それは、対象が根世界であれば、UNIXの名前サブシステムに委譲し、そうでなければ表addに登録、表deleteから削除するものである。wdir_unregister()は、wdir_register()で、表addと表deleteを逆にしたものである。

```
obj_t wdir_lookup(dir_t dir, string_t base,
                  world_t world, ucred_t ucred)
{
  world_t *ancestors; int count;
  get_ancestors(world, &ancestors, &count);
  for(int i = 0; i < count ; i++){
    if( is_root_world(ancestors[i]) )
      return( udir_lookup(dir, base, ucred) );
    obj_t obj = table_lookup(delete, dir,
                  base, ancestors[i], ucred);
    if ( found(obj) )
      return(0);
    obj = table_lookup(add, dir, base,
                       ancestors[i], ucred);
    if ( found(obj) )
      return(obj);
  }
  return(0);
}
```

図１３. wdir_lookup()の実現

wdir_world_merge()の実現を図１４に示す。

```
wdir_world_merge(world_t parent,world_t child)
{
  if( is_root_world(parent) )
    return wdir_world_merge_root(child);
  for( element_t *child_element in add where
              child_element->world == child )
  {
    element_t parent_element =
       table_lookup(add, child_element->dir,
      child_element->base,parent, root_ucred);
    if( found(parent_element) ){
      if( comare_mtime(parent_element->obj,
                    child_element->obj)> 0 ){
        table_unregister(add,
                         child_element->dir,
                        child_element->base,
                        parent, root_ucred);
        continue;
      }else{
        table_unregister(add,
                         parent_element->dir,
                        parent_element->base,
                         parent, root_ucred);
      }
     child_element->world = parent ;
  }
  for( element_t *child_element in delete
        where child_element->world == child )
  {
    element_t parent_element =
      table_lookup(delete, child_element->dir,
     child_element->base, parent, root_ucred);
    if( found(parent_element) )
      table_unregister(delete,
                         parent_element->dir,
                        parent_element->base,
                         parent, root_ucred);
    child_element->world = parent ;
  }
}
```

図１４. wdir_world_merge()の実現

親世界が根世界であれば、根世界特有の処理を行うwdir_world_merge_root()を呼ぶ（後述）。まず、表addを走査して、子世界のすべてファイル(要素）を取り出す．親世界に同名のファイルが存在する場合、最終更新時刻の新しいものを親世界に残し、古い方を表から削除する。親世界に同名のファイルが存在しない場合は、子世界のファイル（要素）を、親世界のファイルとする。表deleteについても、同様に子世界の要素(ファイル名が削除されたこと)を親世界に移している。

親世界が根世界の時、wdir_world_merge_root()が呼び出される。これは、親世界の名前の操作を、表ではなく、

UNIXの名前サブシステムを呼ぶようにしたものである。

wdir_world_delete()の実現は、簡単である。それは、世界をキーとして表を検索し、ヒットした要素を全て削除すればよい。

wdir_world_create()は、中味が空の手続きである。

3.5 書き込みモードで開くのファイルの扱い

3.4節では、新たなファイルを作る場合につい述べた。既存のファイルを読み込み専用モード開く場合についても、従来の方法に世界の引数を増やすだけで対応することができる。この節では、既存のファイルを書き込みモードで開く場合について述べる。

プロセス・サーバから、根世界以外の世界において既存のファイルを書き込みモードで開くという要求を受け付けると、ファイル・サーバは、新たにファイル・オブジェクトを生成し、祖先の世界にあるファイルの内容を、新たに生成したファイルへコピーする。そして、wdir_register()を用いて、同じ名前でその世界に登録する。世界が融合される時、同一の名前のファイルが存在する場合、ファイルの最終更新時刻が新しい方が残されるので、普通は、結果として世界が融合された場合、子世界で更新されたファイルが残されることになる。

UNIXのopen()システム・コールでは、O_TRUNCというフラグが存在する。このフラグは、ファイルを書き込み（O_RDWR,O_WRONLY）のために開く時に、ファイルの長さを0にするためのものである。この場合、上で述べたコピーを省略することができる。

3.6 子プロセスへ開いたファイルを渡すこと

UNIXでは、fork()システム・コールにおいてプロセスのコピーを作成するときに、親プロセスが開いているファイルは、そのまま子プロセスに引き渡される。この方法は、異なる世界にプロセスを生成する場合に、問題を引き起こす。すなわち、親プロセスと子プロセスの間で、open_file構造体が共有されてしまうので、結果として子世界のプロセスの活動が親世界に影響を及ぼすことになるからである。

そこで本システムでは、異なる世界に対してプロセスを生成しファイル記述子を渡す場合、open_file構造体を共有するのではなく、コピーすることにした。具体的には、wfork()システム・コール[2]では全てのopen_file構造体、ファイル記述子を指定するprocess_create()システム・コール[2]においては、指定されたファイル記述子に対応しているopen_file構造体を、プロセス生成時にコピーする。このため、ファイル・サーバに、次のような手続きを設ける。

```
open_file_t fs_open_file_dup(
        open_file_t old,world_t world)
```

この手続きは、open_file構造体と世界を引数に取る。そして、そのopen_file構造体のコピーを作成し、結果として作成したコピーへのポインタを返す。

子プロセスに開いたファイルを渡すために、open_file構造体に新たにファイル名を表す情報（ディレクトリとベース名）を格納する必要が生じた。その理由は、手続き fs_open_file_dup()の引数として書き込みモードで開かれたファイルが指定された場合、ファイルの実体のコピーを行うと同時に、wdir_register()を呼びだし、ファイルの名前を登録する必要があるからである。このファイル名を表す情報は、ファイルを開く処理の時に、open_file構造体へ記録される。

4 まとめ

この論文では、投機的処理を扱うため世界の概念を導入したオペレーティング・システムについて述べた。そして、このオペレーティング・システムのファイル・システムのインタフェースと実現について述べた。インタフェースの特徴は、ファイルが名前と世界を指定して識別されることにある。実現においては、名前サブシステムが中心的な役割を果たしている。

今後は、本ファイル・システムを実現し、世界を導入したオペレーティング・システムの評価を進めていく。

参考文献

[1] 根路銘、當眞、新城、喜屋武、翁長：“粗粒度投機的並列処理を支援するオペレーティング・システムの構想”, SWoPP'94, 情報処理学会研究報告, 94-ARC-107, pp.89-96 (1994).

[2] 溝淵、新城、當眞、根路銘、喜屋武、翁長：“投機的処理を支援するオペレーティング・システムにおける世界とプロセスの操作”, 情報処理学会研究報告, 95-OS-69, No.18, pp.103-108 (1995).

[3] S.J.Leffler, M.K.McKusick, M.J.Kares, and J.S.Quarterman: “The Design and Implementation of the 4.3BSD UNIX Operating System”, Addison-Wesley (1989).