計算機アーキテクチャ　107－12
（１９９４．　７．　２１）

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

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

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

概要　投機的処理とは，利用されるか利用されないか確定される前に実行を開始する処理である．余剰計算機資源を利用して投機的処理を行なうことによって，ターンアラウンド時間の短縮が可能となる．本研究では，事前実行を伴う投機的処理を扱う．我々は，UNIXのmakeアプリケーションに投機的処理を加えたものを設計している．その実行のモデルを示す．多重プログラミング環境における，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・システムを実現するうえで解決すべき問題について議論する．投機的処理を扱うために，世界という概念を導入する．これにより，ファイル・システムや資源割当てなどの投機的処理における問題点を，単純化して考えることができる．

Ideas of an Operating System Supporting Speculative Processing

Takashi Nerome　<nero@ocean.ie.u-ryukyu.ac.jp>
Hajime Toma　<ha@ocean.ie.u-ryukyu.ac.jp>
Yasushi Shinjo　<yas@ie.u-ryukyu.ac.jp>
Seiki Kyan　<kyan@ie.u-ryukyu.ac.jp>
Kenji Onaga　<onaga@ie.u-ryukyu.ac.jp>

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

Abstract　Speculative processing is processing that is started eagerly before it is known to be required. Speculative processing makes it possible to reduce turnaround time by using surplus computer resources. Speculative processing with pre-execution is discussed. The speculative version of the make application of UNIX is designed, and its model of execution is shown. Issues for supporting speculative processing in multiprogramming environments are discussed. In this paper, the idea of "world" is introduced to deal with speculative processing.

1 はじめに

投機的処理（speculative processing）とは，利用されるか利用されないか確定される前に実行を開始する処理である [2][5]．この論文では，プロセスを単位とした投機的処理を支援するオペレーティング・システムの構想について述べる．従来の投機的処理を支援しないシステムと比較して，資源割り当てやファイル・システムに様々な解決すべき問題がある．我々は，世界という概念を用いてそのような問題点を解決することを試みる．

本研究の背景とそれに関連した目標を以下にまとめる．

（1）余剰の計算機資源とその利用：高性能ワークステーションがネットワークで結合された環境が広く普及してきた．このような環境では，多くのワークステーションの計算機資源が利用されずに遊んでいる．並列計算機においては，並列度が上昇するに従い，余剰の計算機資源が増えてくるものと思われる．このような余剰の計算機資源を投機的処理に利用することで，ターンアラウンド時間を短くすることができる．

（2）細粒度投機的処理から粗粒度投機的処理へ：投機的処理は，主に Lisp や Prolog のようなプログラミング言語レベルにおいて研究が進められてきた．特に，並列 Prolog の OR 並列，AND 並列の処理の技法として有名である．しかしながら，プロセス・レベルの粗粒度の並列処理は，未だに活用されていない．本研究では，オペレーティング・システムのレベルにおいて，プロセスやプロセスの集合を単位とした粗粒度の投機的処理を考える．

（3）値の書き換え（副作用）を含む処理：投機的処理は，関数型言語や論理型言語のような値の書き換え（副作用）の概念がない言語で行われてきた．手続き型言語やオブジェクト指向型言語のように，値の書き換えがある言語では，投機の結果を外に見せないようにすることが難しい．トランザクション処理では，アボートされる可能性がある処理を，陰（shadow）を作る，チェック・ポイントを取る，あるいは，ログを取るといった技法を用いて実現されている．本研究では，トランザクション処理で用いられているような技法を活用し，オペレーティング・システムのインタフェースとして提供することを目標とする．

（4）単一プログラミング・システムから多重プログラミング・システムへ：投機的処理は，単一プログラミング・システム，すなわち，1度に1つの応用プログラムしか走らないようなシステムにおいて利用されてきた．本研究では，多重プログラミングにおける投機的処理を実現することを目標とする．

投機的処理は，ハードウェア・レベルの投機的処理とソフトウェア・レベルの投機的処理に分類される．ハードウェア・レベルの投機的処理では，実行が必要であるかどうか明らかになる以前に機械語命令が実行される．ハードウェア・レベルの投機的処理では，分岐予測によって命令をフェッチし，ハードウェアが発行可能な命令を発見しスケジューリングを行う．この論文では，ソフトウェア・レベルの投機的処理について述べる．

この論文では，投機的処理を行うアプリケーションとして我々が開発している投機的makeを取り上げる．そのようなアプリケーションを走らせる環境としてのオペレーティング・システムの機能に関する問題点を議論する．そして，それらの問題点を解決するために，「世界」という概念を導入した新しいオペレーティング・システムの構想について述べる．

2 投機的処理

投機的処理とは，利用されるか利用されないか確定される前に実行を開始する処理である．これに対して，投機的処理ではない処理は，**必須の処理**(mandatory processing)と呼ばれる．投機的処理は，将来使われずに無駄になる可能性のある処理であるのに対し，必須の処理は，要求されている処理である．

投機的処理の結果が要求された時，投機的処理は，必須の処理に変わる．投機的処理の結果が不要であると確定した時，投機的処理が，**的外れ**（irrelevant）に変わる．投機的処理を実現するためには，次の3つの技術が必要となる．

（1）**隔離**：投機的処理と必須の処理の扱いを分ける技術．投機的処理は，的外れになることがある．その影響によって投機的処理を行わなかったときの結果と違う結果を与えてはいけない．よって，利用されることが確定されるまで，隔離しなければならない．

（2）**出現**：投機的処理を必須の処理にみせる技術．投機的処理をいつまでも隔離していてはいけない．投機的処理の結果を要求された時には，速やかにみせなければならない．

（3）**中止**：要らなくなった投機的処理を消す技術．的外れの処理を実行することは，資源を無駄にするものである．よって，的外れな処理を速やかに消さなければならない．

2.1 投機的処理の分類

これまでにも投機的処理に関して研究がなされてきた．Osborne[5]は，投機的計算を次のように分類している．

（１）複数の選択肢に対する投機的計算
(multiple-approach speculative computing)
これは，いくつかの選択肢を同時に行うものである．典型的な例が論理型並列計算におけるＯＲ並列，ＡＮＤ並列の計算である．

（２）順序に基づいた投機的計算(order-based speculative computing)
これは，（１）の投機的処理の特殊化したものである．処理において優先順位がつけられる．トラベリング・セールスマン問題を分枝探索法により解くことが，その典型的な例である．順序に基づいた投機的処理において重要な事は，投機を行うプログラムが応用固有の優先順位を設定することである．

（３）事前実行(precomputing)
まだ要求の出ていない次に行われるであろう処理について着目している．例としてストリーム処理と，ファイル・システムにおけるディスク・ブロックの先読みが挙げられる．

以上の３つの分類は，重なっている部分がある．この分類には，値の書き換えに関する視点がない．

我々は，値の書き換えを考慮した新しい分類を提案する．図１で示すように我々は，投機的処理をⅠ，Ⅱ，Ⅲ，Ⅳの４つに分類する．この分類は，事前実行と値の書き換えという２つの視点に基づいている．if文で条件が確定される前にthen部やelse部の処理を開始するものが事前実行である．無駄になるかどうかは，他の処理の結果に依存している．これは，Osborneの分類で（３）にあたる．事前実行を伴わない投機的処理として，並列Prologなどがある．並列Prologでは，無駄になるかどうかは，その処理を行ってみなければ判らない．少なくとも１つの処理は、実行されなければならない．

値の書き換えとは，記憶領域の概念があり，その内容を変更することで他の処理に影響を与える操作である．関数型言語や論理型言語には，値の書き換えの概念がない．この場合，２章の冒頭で挙げた隔離と出現の概念がない．よって，投機的処理の実現は，容易である．これに対して，値の書き換えの概念がある場合は，隔離と出現をどのようにして実現するかが，重要な課題となる．

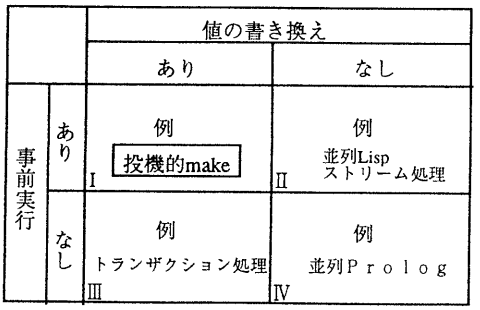


図２　投機的処理の分類

Ⅱの例として並列Lispやストリームなどが挙げられる．Ⅲの例としてトランザクション処理が挙げられる．Ⅳの例としては，並列PrologのＯＲ並列等が挙げられる．

本研究では，図1の分類でⅠで示される事前実行と値の書き換えを行うアプリケーションを対象としたオペレーティング・システムを実現する．その主要なアプリケーションが，投機的makeである．これについて詳細を３章で述べる．

2.2 事前実行を伴う投機的処理の流れ

ここでは，本研究で扱う事前実行を伴う投機的処理の流れを示す．

下のようなプログラムを考える．

```
if( A )
{
      B ;
}
else
{
      C ;
}
```

A，B，Cは，処理を表す．このプログラムの意味は，Aの結果に応じてBまたはCの結果を利用するということである．この場合，Aは，必須の処理である．

図２は，上のif文実行方法を時間軸に沿って表わしたものである．ここではまず，BとCの処理時間がAよりも長い場合を考える．（a）は，if文を逐次的に実行した様子を表わしている．Aの処理の終了後に，Bの処理が開始されている．（b）では，ｉｆ文の実行に投機的処理を用いている．Aの処理が終了する以前にBとCの処理が始められている．図２では，Aの処理の終了後，Bの結果が必要であることが確定している．Bの処理は，投機的な実行から必須の実行に移り変わる．同時にCの処理は，強制終了されせらる．ここで，ｉｆ文の処理全体の実行時間は（a）に比べて（b）が短いことがわかる．　次に，BとCの処理時間がAよりも短い場合を考える．これを（c）に示す．BとCの結果は，Aの処理が終了するまで保存される．Aの処理の実行中は，BとCどちらも必須と判断されていない．よってBとCの結果の保存方法は，普通の方法と違っている．（c）では，Aの処理結果によって，Bの結果を利用すると確定している．Cの処理結果は，破棄される．Bの結果は，必須の処理の結果と変わる．この場合でも，ｉｆ文の処理全体の時間は，逐

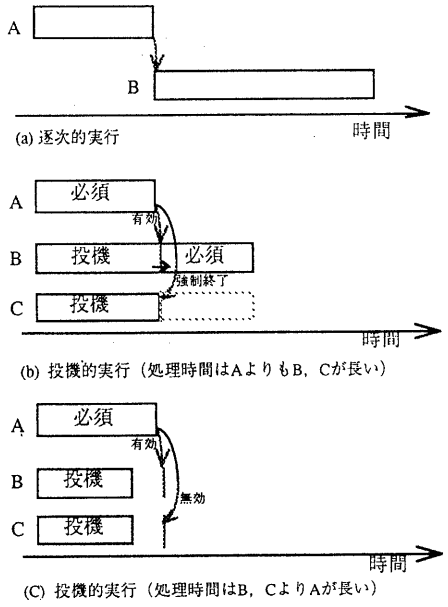


図２ ｉｆ文の実行の方法

次処理と比較して短くなる．

Ａ，Ｂ，Ｃの処理の性質を把握できれば，高度なスケジューリングが可能である．

3 投機的make

我々は，UNIXのmakeアプリケーションに投機的処理を実行する機能を組み込むことにした．これにより，投機的処理を支援するオペレーティング・システムの機能を洗い出す．アプリケーションとしてmakeを選んだ理由は，高速性が要求されているからである．既にmakeの並列化の研究は，盛んに行われている [1][4]．我々は，並列処理に投機的処理を加え，さらに高速化をめざす．

3.1 makeにおいてどこで投機的処理が可能か

makeの主な目的は，実行形式のファイルを作成することである．その主な作業は，コンパイルとリンクである．実行形式のファイルを作成するときには，いくつもの関連ファイルをコンパイルして，その後リンクの処理を行う．ここで例として，１つのヘッダ・ファイルを更新したときを考える．新しく実行形式のファイルを作成するために，複数のファイルに対してコンパイル等の処理を行わなければならない．このような処理を利用者が"make"と打つ前に実行すれば，利用者には，短い時間で処理が行われているかのように見せかけることができる．

このような動作は，次のｉｆ文で説明できる．

```
if( 利用者が "make" と打つ )
{
    ｃｃ ;
}
```

利用者が"make"と打つ前にコンパイル（ｃｃ）を実行するということは，無駄に終わる可能性がある．

3.2 投機的makeの手順

投機的makeは，実行されると，バックグラウンドでファイルの最終更新時刻の監視を始める．投機的makeは，ファイルの時刻等の情報をオペレーティング・システムから入手する．ファイルの生成もしくは変更が起きると，資源の状態の情報をオペレーティング・システムから入手する．投機的makeは，資源の状態の情報によって投機的処理を実行すべきかどうかを判断する．実行すべきでないと判断した場合は何もしない．実行すべきだと判断した場合，更新されたファイルが影響を及ぼすファイルを生成するため，メイクファイルに記載された処理を実行する．

投機的にコンパイル等の処理を行っているときに新たに同じファイルの更新が行われた場合を考える．この場合，投機的makeは，現在行われている処理を強制終了させる（中止）．そしてそれと同時に新しく更新の行われたファイルに対して，メイクファイルに記載された処理を始める．

メイクファイルに記載されたｃｃのような処理を行うとき，処理によって生成されるファイルは，隠される必要がある．利用者が"make"と打つ前にファイルが書きかわっているように見えてはいけない（隔離）．投機的に行っている処理が他の処理に影響を与えてはいけない．要らなくなった処理は，消される．利用者が "make" と打つと、投機的makeは、速やかにコンパイル等の結果を利用者に知らせる．つまり，今まで隠されていたものが，利用者から見えるようになる（出現）．

投機的makeの実行例を図３に示す．この図には，投機的makeとＯＳと利用者の３者の動きを示している．投機的makeは，ＯＳにファイルの更新の情報がないかどうかを問い合わせている（１）．

利用者がファイルの更新を行う（２）と，投機的makeがそのことを検知する（３）．そして投機的makeは，そのファイルに対してｃｃ$_1$の処理を開始する．この処理は，利用者には気付かれないように実行される．図３では，ｃｃ$_1$の実行中に同じファイルに対して更新が行われた（４）．投機的makeがそのことを検知する（５）．投機的makeは，現在実行中であるｃｃ$_1$の処理を強制終了させる．そして，新しく更新されたファイルに対してｃｃ$_2$の処理を始める．ｃｃ$_2$の処理が終了したとき，その結果は，利用者には隠されたままである（６）．その後利用

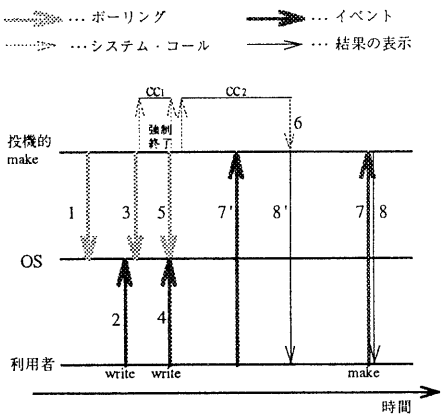


図3 投機的make実行の概観

者が"make"と打った時、（7），投機的makeは，隠していた結果が見えるようにする．具体的には，実行形式のファイルが瞬時に入れ替わり，メッセージが表示される．

実際には，図3においてｃｃ$_2$の処理は，およそ5から6までの時間がかかっている．この時間は，投機的処理を行わないときの時間である．利用者にとっては，7から8までの時間でｃｃ$_2$の処理が行われたかのように見える．もし，ｃｃ$_2$の処理を行っている途中で，利用者が"make"と打った場合（7'）を考える．利用者からみた処理時間は，この時点から，ｃｃ$_2$の処理が終了するまで（8'）である．図3より，この場合でもターンアラウンド時間は，投機的処理を用いなかった場合に比べて短くなっている．

4 粗粒度投機的処理の支援

この章では，粗粒度投機的処理を支援するオペレーティング・システムを実現する上で解決すべき問題について議論する．

投機的処理が必須の処理に変化することを，**投機が成功する**という．投機的処理が無駄であったとわかることを，**投機が失敗する**という．必須の処理をするために生成されたプロセスを**必須プロセス**，投機的処理をするために生成されたプロセスを**投機プロセス**と呼ぶことにする．

2章では，投機的処理で重要なことは，隔離，出現，中止であることを述べた．この章では，ファイル・システム，プロセス間通信，資源割当てにおいて，それらがどのように実現されるかについて述べる．

4.1 ファイル・システム

図3では，利用者が"make"と打つ前に，ファイルのコンパイルを始めている．ｃｃ$_1$，ｃｃ$_2$が投機プロセスである．コンパイルの結果として，実行形式のファイルが作られる．この実行形式のファイルは，利用者が"make"と打つまで隠されていなければならない．投機的makeが利用者によるファイルの書き込みを検知すると（3，5），ファイルのコンパイルのために投機プロセス（ｃｃ$_1$，ｃｃ$_2$）が生成される．投機的処理を支援するファイル・システムは，このプロセスが書き込みを行ったファイルを，利用者に見せないように保存しなければならない（隔離）．

投機が成功した瞬間，すなわち利用者が"make"と打った瞬間（7あるいは7'），投機的処理を支援するファイル・システムは，投機プロセスが書き込みを行ったファイルを，利用者に見せるようにしなければならない（出現）．

最初の投機プロセスｃｃ$_1$が無駄であることが投機的makeから知らされたとき，投機的処理を支援するファイル・システムは，投機プロセスｃｃ$_1$が書き込みを行ったファイルを消去しなければならない（中止）．利用者からは，そのファイルが最初から存在しなかったかのように見える．

投機プロセスは，自分自身が生成された直後は，必須プロセスが使用しているファイルを見る．投機プロセスがファイルに書き込みを行おうとしたとき，ファイル・システムは，このファイルを，投機プロセスだけが見える場所に複製しなければならない．このとき，投機プロセスから元のファイルは，見えなくなる．投機プロセスは，ファイルに対し書き込みを行う．ファイル・システムは，複製したファイルに対して，変更を加える．

必須プロセスからは，投機プロセスが書き込みを行ったファイルは見えない．投機が成功し，投機プロセスが必須プロセスに変化した瞬間に，投機プロセスが書き込みを行ったファイルが見えるようにする．ファイル・システムは，このような変化を扱わなければならない．

投機が失敗し，投機プロセスが無駄であることがわかったら，投機プロセスによって書き換えられたファイルは，すべて消去される．ファイル・システムとしては，このような消去要求に対応しなければならない．

図3で投機的makeは，ファイルの書き込みの検知にポーリングを用いている．ファイル・システムが，ファイルの書き込みが行われたときに，投機的makeなどのアプリケーションにイベントを渡すことができれば，より効率的な動作が期待できる．

4.2 プロセス間通信

必須プロセス間の通信は，特に問題なく行うことができる．必須プロセスと投機プロセスの通信は，必須プロセスにとっては，本来生成されないかもし

れないプロセスとの通信を意味する．従って，投機プロセスが失敗したときは，必須プロセスの状態は，投機プロセスとの通信を最初から行わなかった場合と同じ状態になっていなければならない．

我々は，プロセス間通信を行うこと自体，プロセスの状態の書きかえが行われると考える．必須プロセスと投機プロセスの通信は，必須プロセスの状態を書きかえるので，行わせないことにする．そのようなプロセス間通信は，投機が成功し投機プロセスが必須プロセスに変化するまで遅延される．投機が失敗したときは，プロセス間通信を行っていなかったので，これによって，必須プロセスの状態は，変化しない．

投機プロセス間の通信は，２つに分けて考えなければならない．ひとつは，同一の投機的処理に属する投機プロセス間の通信である．もうひとつは，異なった投機的処理に属する投機プロセス間の通信である．前者は，必須プロセス間の通信と同じように行ってもよい．後者は，同一の投機に属するようになるか，両方とも必須になるまで，遅延される．ここで投機的処理が**同一**とは，１つの処理の結果によって同時に必須に変わるものである．

4.3 資源割当て

投機プロセスを実行することは，それに投入した計算機資源が無駄に使われるかもしれないことを意味する．投機プロセスは，投機が成功すると必須プロセスになる．投機プロセスは，投機が失敗すると強制終了したほうが望ましい．

図３において，本研究の投機的makeでは，cc_1，cc_2を投機プロセスとして実行している．投機的makeは，通常複数のプロセスを生成する．このようなプロセスの集合を，１つのアプリケーションと考える．

１つのアプリケーション内において必須プロセスにより高い優先順位を与えるという方針を実現することを考える．これは，プロセッサ資源が有限なので，計算機資源の無駄遣いをなるべく小さくしようという発想に基づいている．プロセッサ資源が無限であると仮定すると，プロセスは，優先順位を設定せずに全て同時に実行するとよい．オペレーティング・システムは，このような方針を実現する仕組みを提供しなければならない．

アプリケーションが複数存在する場合，プロセスは，それが投機プロセスかどうかということには関係なくアプリケーションごとに設定された優先順位で実行されなければならない．例えば，社長のアプリケーションと平社員のアプリケーションを同一の環境で実行させる状況を考える．このようなときは，社長のアプリケーションに属する投機プロセスと，平社員のアプリケーションに属する必須プロセスでは，社長の投機プロセスのほうが優先的に実行されることもある．すなわち，計算機の外の，アプリケーションの実行者に関連した因子も，プロセスの優先順位を決定する重要な要素である．

5 世界モデルに基づく投機の支援

4章より投機プロセスが用いるファイル・システムやプロセス間通信の扱いは，必須プロセスとは別に考えなければならないことがわかった．複数の必須プロセスや投機プロセスが同時に実行される環境では，ファイル・システムやプロセス間通信の管理は，投機がない場合に比べて非常に複雑なものとなってしまう．本章では，これを単純化し扱いやすくするために，「世界」という概念を導入する．

世界とは，ファイルやプロセスが存在できる箱である．１つの世界は，単一のファイル・システム（名前空間を含む）を持つ．世界は，親子構造を持つ．すなわち世界は，完全な入れ子構造になっている(図4)．

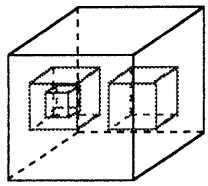


図4　世界の構造

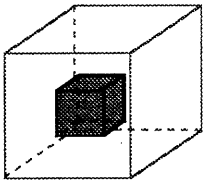
世界の操作には，生成，削除，融合がある（図5）．世界の**生成**とは，箱の中に新たに別の箱を作ることである（図5（a））．世界（箱）が重なりあうことはない．世界の**削除**とは，箱の中から目的の箱を，箱の内容ごと取り去ることである（図5（b））．世界の**融合**とは，箱の内容を残して箱だけ取り去ることである（図5（c））．

この章では，世界の概念を導入することで，投機がうまく扱えることを示す．

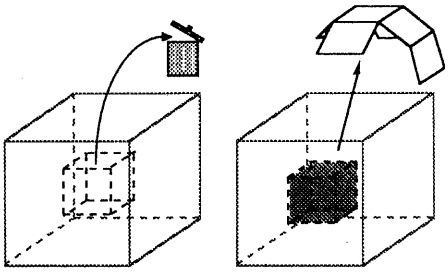
5.1 ファイル・システム

１つの世界には，１つのファイル・システムが存在する．すなわち，同一の世界に属しているプロセスのファイル操作は，投機的処理が支援されていないシステムにおいて実行されている場合と同じように行うことができる．子世界のプロセスは，生成された直後は，親世界のすべてのファイルを見ることができる．プロセスは，同一の箱の中にあるファイルを自由にアクセスすることができる．さらに，プロセスは，生成された直後は，外側の箱の中にあるファイルも見ることができる．

このプロセスがファイルに対して書き込みを行う



(a)世界の生成



(b)世界の削除　　(c)世界の融合

図5　世界の操作

と，親世界の同名のファイルは見えなくなる．新たなファイルを作成すると，子世界でのみ見ることができるファイルが作成される．ファイルを削除すると，親世界では，そのファイルは残っているが，子世界では，そのファイルは，存在しなくなる．ファイル名の変更を行うと，子世界でのみそのファイル名の変更が有効になる．

投機が成功すると，子世界は，親世界に融合される．子世界で行ったファイルへの変更は，親世界で有効になる．

投機が失敗すると，子世界は削除される．結果的に子世界で行ったファイルへの変更は，始めから行わなかったことになる．

図6は，投機的処理を世界で表現したものである．世界は，面で表現される．最下面が，必須プロセスが存在する親世界である．親世界の必須プロセスは，投機を行うために子世界を生成する．図6では，親世界からわかれて立ち上がっている面が，子世界である．投機プロセスは，子世界に生成される．投機プロセスが実行されると，子世界と親世界の差が開いていく．これは，最初は全くの複製であった子世界のファイル・システムが，子世界のプロセスによる変更が進み，次第に親世界と異なっていく様子を示している．

投機が失敗すると，子世界が削除され，投機プロセスは，強制終了させられる．図6では，最初に生成された子世界が削除されている．

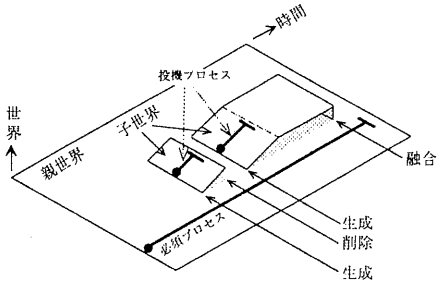


図6　世界を用いた投機の実現

図6の2番目の子世界は，投機プロセスの終了後，親世界と平行になっている．これは，子世界が変化していないことを示している．

投機が成功すると，世界が融合される．図6では，2つの面が1つになっている．これは，世界の差がなくなったことを示している．

5.2 プロセス間通信

4.2節では，プロセス間通信の扱いが，非常に複雑であることを述べた．世界を導入することによって，プロセス間通信の扱いが，単純化される．

同一の世界に属しているプロセス間の通信は，可能である．異なる世界に属しているプロセス間の通信は，世界が融合されるまで遅延される．投機プロセスと必須プロセスが同一の世界に存在することはない．したがって，プロセス間通信の扱いで，投機と必須を区別する必要はない．

5.3 資源割当て

複数のアプリケーションを区別するために，アプリケーション識別子を導入する．各プロセスは，属性としてアプリケーション識別子をもつ．オペレーティング・システムは，同一のアプリケーション識別子を持つプロセスを，同一のアプリケーションとして扱う．

アプリケーションに対しては，優先順位が設定される．投機を行うアプリケーション内には，複数の世界があり、投機プロセスと必須プロセスが存在する．

あるアプリケーションが，4.3節の例で述べたように，投機プロセスよりも必須プロセスに高い優先順位を与えるという方針を実現することを要求したとする．オペレーティング・システムは，同一のアプリケーション内において，一番外側の世界から実行可能なプロセスを探し，実行する．

5.4 関連した研究

ファイル・システム（名前空間）の多重化ついては，多くの研究がなされている．文献[3]では，分

散共有格納庫と呼ばれる永続処理とプロセス間通信を実現すための仕組みに，多重名前空間を導入する方法が提案されている。同文献には，ファイル・システムの多重化ついていくつかの研究が紹介されている．既に広く利用されているファイル・システムの多重化を支援したシステムとしては，Sun Microsystems 社の半透明ファイル・サービス（TFS, translucent file service）があげられる [9].

本研究の特徴は，第１に，ファイル・システム（世界）の融合操作を導入した点にある．それらの研究では，ファイル・システム（名前空間）の融合操作が定義されていない．本研究の特徴は，第２に，プロセスのスケジューリングにおいても，世界の概念を活用していることである．

オペレーティング・システムのレベルでトランザクション処理を支援しているものとしては，LOCUS システムがあげられる [6]．LOCUS では，ファイルの操作にアボート，および，コミットというシステム・コールが追加されている．本研究で導入した世界の概念を利用することにより，LOCUS システムと同等の機能を実現することが可能である．さらに世界を利用することにより，コミットやアボートといったシステム・コールを発行しない既存のアプリケーションを一切変更することなくトランザクション処理に利用することが可能となる．

我々は，既に軽量プロセスのスケジューリングにおいて投機的処理を扱うスケジューラを開発している [7][8]．それらの文献では，投機的処理を行うスケジューラと行わないスケジューラを比較する実験を行っている．それらを２種類の共有メモリ型マルチプロセッサにおいて実行している．１台では投機的処理を行うスケジューラが，もう１台では行わないスケジューラがよい結果を残した．

6 おわりに

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・システムの構想について述べた．投機的makeを例に，投機的処理を行う応用プログラムの要求を調べた．投機的処理を導入することで，余剰の計算機資源の活用が図られ，見かけの処理速度が大きく改善されることが期待される．システムの目標は，プロセス単位の投機的処理を支援すること，および，多重プログラミング環境での投機的処理を扱うことである．

本論文では，投機的処理を扱うために，世界という概念を導入した．世界には，生成，融合，および，削除という操作がある．それぞれ，投機的処理における隔離，出現，中止を実現するために利用される．さらに，世界は，プロセスのスケジューリングにおいて，同一アプリケーション内の優先順位を決定する時に利用することができる．

今後は，本論文で提案した機能を実現していく．世界の概念を持ったファイル・システムを実現することから開始していきたい．同時に，投機的処理を行う応用プログラムとして投機的makeを開発し，投機的処理の効果を調べていく．また，ポーリングすることなくファイルの更新を検知する仕組みについて検討していきたい．さらに，投機的処理を含んだ時の，ＣＰＵのスケジューリングやメモリ割当てなどの資源割当てのよいアルゴリズムを開発していきたい．

参考文献

[1] 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).

[2] R.H.Halstead,Jr.: "New Ideas in Parallel Lisp: Language Design, Implementation, and Programming tools", Proc. US/Japan Workshop on Parallel Lisp ("Parallel Lisp: Languages and Systems",Springer-Verlag LNCS 441, pp.2-57 (1990)).

[3] 加藤，坂田，益田: "分散共有格納庫への多重名前空間の導入について", 情報処理学会第５回コンピュータシステム・シンポジウム, 情処シンポジウム論文集, Vol.93, No.7, pp.115-122 (1993).

[4] 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).

[5] R.B.Osborne: "Speculative Computation in Multilisp", Proc. US/Japan Workshop on Parallel Lisp ("Parallel Lisp: Languages and Systems", Springer-Verlag LNCS 441, pp.103-135, 1990).

[6] 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 Operating Systems Principles, pp.160-168 (1981).

[7] 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).

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

[9] "SunOS Reference Manual", Sun Microsystems, Inc. (1988).