システムソフトウェアと
オペレーティング・システム 88−14
（２００１．７．２７）

複数タスクの特徴に動的適合可能なマルチポリシメモリ管理システム

唐野 雅樹，小林 良岳，結城 理憲，中山 健，前川 守

電気通信大学大学院 情報システム学研究科 情報システム設計学専攻

要 旨

近年のコンピュータの性能向上により，マルチメディアやデータベース等の様々なアプリケーションが１つのシステム上で実行されるようになってきた．これらのアプリケーションでは，その用途の違いからメモリのアクセス特性に差が生じる．従って，１つのメモリ管理方針しか提供できないシステムでは，全てのタスクに対し，最適なメモリ管理を提供することは困難である．そこで我々は，各タスクに対して最適なメモリ管理方針を提供し，それらが共存可能なメモリ管理システム「覚 (*Kaku*)」を設計し，*Aya* オペレーティングシステム上に実装した．本稿では，*Kaku* の実装について詳細を述べ，その評価を行う．

キーワード: メモリ管理，オペレーティングシステム，マルチメディア

A Multi-Policy Memory Management System Dynamically Adaptable to Task Characteristics

Masaki Touno, Yoshitake Kobayashi, Yoshinori Yuuki, Ken Nakayama, Mamoru Maekawa

Department of Information Systems Science, Graduate School of Information Systems, University of Electro-Communications

Abstract

In contemporary computers, various applications (ex. multimedia, database) are processed in one system. These applications have different characteristics in memory access because they are used for different goals. Hence, it is very hard for systems which have only one memory management policy to provide good performance. Therefore, we propose multiple memory management policy to solve the problem above. In particular, we allow a policy for multimedia applications to coexist with other policies by controlling the time for calling swap-out function. This is implemented in the *Kaku* memory management system within the *Aya* operating system.

keywords: memory management, operating system, multimedia

1 はじめに

近年，PC 性能の向上により，一台の PC 上で様々な種類のアプリケーションが同時に動作するようになった．また，PC を使用するユーザの要求も多様化しており，ユーザが使用するアプリケーションの中には，ソフトリアルタイム性を必要とする連続メディアアプリケーションなども含まれるようになった．

リアルタイム性を要求するアプリケーションを動作させるには，OS がリアルタイム性をサポートする必要がある．このため，リアルタイム OS に関してスケジューラ [1，2] や IPC[3] 等，様々な分野で研究がなされてきた．しかし，メモリ管理のリアルタイム性に焦点を当てた研究や，リアルタイムアプリケーションとその他のアプリケーションの共存を意識したものは少ない．

これらシステムリソースに対する要求の異なるアプリケーションを，それぞれ効率的に動作させるためには，各アプリケーション毎に，その特性に応じたメモリ管理を行う必要がある．

そこで本稿では，メモリの利用においてリアルタイム性を要求するアプリケーションとその他のアプリケーションの効率的な共存を実現するメモリ管理システム「覚 (*Kaku*)」を提案し，その実装の詳細について述べる．*Kaku* ではメモリ管理のポリシを，各タスク毎に独立して設定することが可能であり，メモリスケジューラによって，各タスク間の全体の管理を集中して行なう．これにより，各アプリケーション毎に適切なメモリ管理を行う事ができ，メモリ管理システムの効率的な動作が可能となる．

2 関連研究

2.1 リアルタイム処理

リアルタイム処理におけるメモリ管理に焦点をおいたものとしては，[4] がある．これは，メモリ管理の機能をユーザ空間に置き，それぞれに独立して動作させる事で後に述べる QoS クロストーク問題を避けている．しかし，ページフォルト等メモリ管理に必要な割り込み処理や，ディスク I/O 処理は，一旦カーネルを経由しなければならないため，システムコール呼び出しのオーバーヘッドが余計にかかる．また，[4] では，物理メモリが枯渇した時に他のアプリケーションに対して時間保証付きの解放を求めるが，これを守れなかった場合，そのアプリケーションを強制終了させてしまう．

しかし，リアルタイムアプリケーションと，その他のアプリケーションとの共存を考えたとき，その時間保証のために他のアプリケーションが不安定になることは好ましくない．よって，マルチメディアアプリケーションなどのソフトリアルタイム性を要求するアプリケーションとその他アプリケーションを，厳しい時間保証の元で共存させて動作させることは難しいと考えられる．

これに対し，スワップ処理のタイミングを調節する事でメモリ管理の時間保証を行おうとしたものもある [5]．この研究は，対象がソフトリアルタイム性を求めたものであるため，[4] よりも実用的であると考えられる．しかし，どのページをページアウトするか等のポリシは従来の方法を使っており，この部分に関する考察は述べられていない．

2.2 スワップ処理の効率化

スワップ処理の効率化という問題にも多くの研究が発表されている．この問題において，メモリアクセス特性に応じてスワップポリシを動的に変化させる事は，スワップ処理をアプリケーションに特化できるという面で非常に有用である．

1. SPIN[6] では，専用の言語でモジュールを書き，イベントハンドラに登録する事でカーネルに動的に組み込み，これを利用してスワップ処理のポリシを動的に変更可能である．
2. VINO[7] では，Address Map Resorce オブジェクトに対する操作を，アプリケーションが上書き可能である．これにより，スワップポリシを変更する事が可能である．
3. [8] では，自動的にアプリケーションのメモリアクセス特性を計測していき，そのデータに基づいてスワップ処理のポリシを自動的に変更していくシステムを提案している．

しかし，上に述べたものは全てリアルタイム性を考慮しておらず，時間保証の概念を導入する事ができない．また，[6] では，システムに 1 つしか存在しないスワップポリシを動的に変更するため，様々なメモリアクセス特性を持つアプリケー

ションが同時に複数動作する環境では，あるアプリケーションに特化すると，他のアプリケーションの性能が落ちてしまう可能性がある．また，[7] や [8] では，スワップポリシの操作がアドレス空間全体を対象としているため，細かな最適化が不可能である．

3 リアルタイムアプリケーションとの共存

3.1 リアルタイムアプリケーション

本稿で扱うリアルタイムアプリケーションは，デッドラインミスがシステムに重大な影響を与えるハードリアルタイムアプリケーションではなく，ビデオ再生ソフトのような，デッドラインミスを起こしても復元作業を行う事で処理を継続可能なソフトリアルタイムアプリケーションを示す．以降，特に断りがない限り，リアルタイムと記した時はソフトリアルタイムを表わすものとする．

ビデオ再生ソフト等のリアルタイムアプリケーションでは，画像を扱うために多くのメモリリソースを必要とする．また，アプリケーションによって要求されるリソース量が違うため，リアルタイムアプリケーションでは仮想メモリ機構を利用する可能性がある．

従って，リアルタイムアプリケーションに対応するためには，システムは以下の機能について時間保証を考慮しなければならない．

- 物理メモリの割り当て・解放
- 仮想メモリ空間の割り当て・解放
- ページイン・ページアウトの動作

また，これらのリアルタイムアプリケーションは，その他のタスクと共存するため，時間的効率・空間的効率が良くなくてはならない．

3.2 複数のスワップポリシの適用

リアルタイムタスクと，非リアルタイムタスクとの共存を考える時，この 2 種類のタスクにおいて共通に考えなければならない事は，ページアウト・ページインにおけるメモリ空間の利用効率である．この利用効率を上げるためには，適切なページをページアウト・ページインしなければならない．

効率的な処理を行うために，UNIX 等では次のような方法をとる．ページアウトでは，最も最近使われていないページを選択する LRU アルゴリズムを利用し，ページアウト対象ページを決める．また，ページインにおいてはページインするページの近辺のページを同時に読み込む事でページフォルト回数を減らして効率を上げている．これらの方法は一般的なプログラムにおける参照の局所性といったアクセス特性に基づいている．しかし，データベース等のアプリケーションでは，このような局所性を持たないアクセス特性を持つものもある [10].

そこで本研究では，このようなアプリケーションに対しても効率的なページングを提供する為に，ページイン・ページアウトを行うページを決定するスワップポリシを複数存在させ，これをシステム上で動作するタスクに対して適切に割り当てる方法を採る．

3.3 QoS クロストーク問題の解決

共有サービスに対して複数のタスクがサービスを要求する時，QoS クロストーク問題 [9] が生じる．また，共有サービスがカーネルに実装されている場合，タスクがカーネルのサービスを受けている時間はカーネルの時間としてカウントされてしまい，タスクそのものの時間予測が複雑になる．このため，可能な限り共有サービスを排除し，各タスク毎に独立して割り当てることで，予測が複雑になる時間を少なくする必要がある．

また，共有サーバにおいて他のタスクとの競合解決にかかる時間は，競合するタスクの動作によって変化する．例えば，あるタスクがメモリ割り当て要求を出し，カーネルがそれを処理している最中に他のタスクが割り当て要求を出した場合を考える．この時，後から要求を出したタスクは先に要求を出したタスクの処理が終了するまで待たなければならない．この競合処理の解決をカーネル内で行うと，待ち時間はシステムの状況によってランダムになってしまう．

これに対処するために，本研究ではカーネル内でリアルタイムタスクに対し，タスク毎にメモリ資源の管理を独立させ，可能な限り競合に対する処理を無くす方法を取る．ここで，時間の制約が厳しいタスクに対してはカーネルは，メモリを即座に割り当てるか，そうでなければ割り当て失敗

と返事をすることで，処理にかかる時間を一定にする．また，時間に対する制限が緩い場合は，最初にタスク毎のポリシで確保しているメモリ資源で解決を試みることで QoS クロストーク問題を抑え，必要に応じて他のタスクからメモリを取得する方法を取る．

3.4 ページアウトの効率化

UNIX 等のシステムでは，スワップアウト処理はスワップデーモンによって定期的に行われることが多い．この時，全てのタスクから公平に不要な物理メモリを解放することにより，スラッシングの状態に陥ることを避けている．しかし，リアルタイムタスクと，その他のタスクが共存して動いている場合，リアルタイムタスクからは可能な限りメモリを解放させない方が好ましい．

そこで，本研究で実装しているスワップデーモンは，QoS クロストーク問題を避けるためにリアルタイムタスクの動作中にはスワップデーモンの起動を停止させている．また，スワップデーモンの処理中でも，リアルタイムタスクのデッドラインが近づけば処理を中止し，そのタスクに制御を移すことでデッドラインミス率を低減させる．

4 *Kaku* メモリマネージャ

4.1 システムの概要

ここまでの議論をもとに，複数のメモリ管理ポリシを管理し，それぞれのタスクに個別に適用可能な「覚 (*Kaku*)」メモリマネージャを実装した．*Kaku* は，*Aya* オペレーティングシステム [11] 上で動作しており，次の 4 つの部分から構成される (図 1).

- ページアロケータ (Page Allocator)
- ポリシハンドラ (Policy handler)
- スイッチ (Switch)
- メモリスケジューラ (Memory Scheduler)

4.1.1 ページアロケータ

ページアロケータは，空き物理ページを保持している．そして，ポリシハンドラからメモリ割り当ての要求があったときに，ポリシハンドラに対して物理ページの割り当てを行う．割り当てた物理ページがポリシハンドラ内でどのような使われ方をされるかについては，ページアロケータは関与しない．

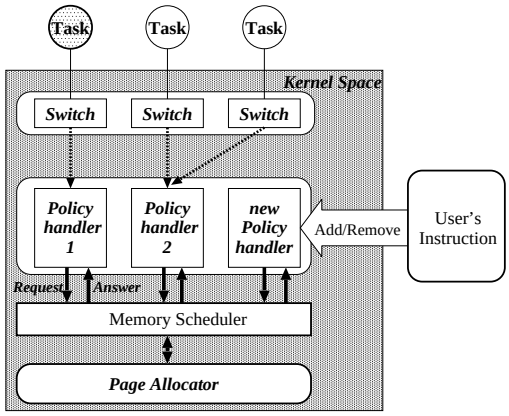

図 1: *Kaku* メモリマネージャの構成

また，ポリシハンドラからのメモリ割り当て要求に対し十分なメモリが確保できない場合は，ページアロケータはメモリスケジューラに対して物理ページの解放を依頼する．

4.1.2 ポリシハンドラ

ポリシハンドラは，ページイン・ページアウトにおける実際の作業を担当する．具体的には以下の機能を持つ．

- ページアウト対象ページの選択
- ページインにおける，プリフェッチ対象ページの選択
- ページのバッキングストアへのセーブ／ロード

ポリシハンドラはスイッチ (Switch) によりタスクと関連づけられ，ユーザによる指示によって登録・削除が可能である．ポリシハンドラ登録の際，メモリスケジューラ内にあるリストに登録される．現在の実装でメモリスケジューラは，リアルタイムポリシ用と非リアルタイムポリシ用の 2 種類のリストを保持している．もし，リアルタイムタスク用に作成されていないポリシハンドラを，リアルタイムポリシ用リストに登録した場合，それはリアルタイムポリシとして使用されることとなる．また，リアルタイムポリシとしてメモリスケジューラに登録されたポリシは，タスクと 1 対 1 で対応付けられる (図 2).

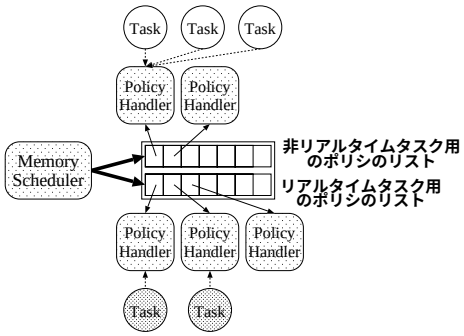

図 2: タスクとポリシハンドラの関連付け

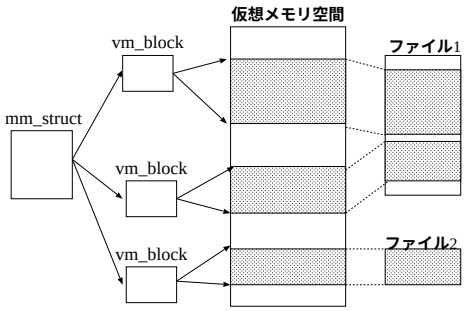

図 3: メモリ管理記述子の関係

4.1.3 メモリスケジューラ

メモリスケジューラは，ページアロケータやスワップデーモンから呼び出され，ポリシハンドラ間の競合の解決を担当する．ページアロケータがメモリの割り当てに失敗した時は，メモリスケジューラに対して物理ページ解放要求が出される．この要求に応えるために，メモリスケジューラはポリシハンドラのリストを保持しており，ポリシハンドラ内の手続きと関連づけられている．現在の実装では，非リアルタイムポリシハンドラとリアルタイムポリシハンドラの 2 つが関連づけられている．

またメモリスケジューラは，それぞれのポリシハンドラに対してメモリ割り当てを行われた際に，実際にどれだけのメモリが割り当てられたか監視して，これを各ポリシハンドラが保持している物理ページ数として保持している．ただし，リアルタイムポリシハンドラが持つ物理ページ数は，実際に保持するページ数から保証された物理ページ数を引いたものとなる．

メモリスケジューラに対してページアウト要求があると，最初にどのタスクから要求が行われたかを確認する．この時，ポリシハンドラの属性チェックが行なわれるが，リアルタイムポリシハンドラに関連づけられたタスクの場合，そのタスクの属するポリシハンドラに対してスワップアウトを要求する．これが失敗した時，他からの要求と同様に全てのポリシハンドラに対してスワップアウト要求を行う．

その他からの要求の場合，それぞれのポリシハンドラが保持する物理ページ数を比較し，保持数の多い方を対象とする．この時，各ポリシハンドラが保持しているスワップカウンタを取得する．そしてスワップカウンタが最も大きなものをスワップアウト対象とし，そのポリシハンドラのページアウトオペレーションを呼び出す．探索が失敗した場合，スワップカウンタの初期化を行い，もう一度探索を繰り返す．それでも失敗した場合はエラーとなる．

4.1.4 ページデーモン

ページデーモンは，*Kaku* とは独立して動作するカーネルスレッドとして実装されている．これは空き物理メモリ量を監視するスレッドである．もし空き物理メモリ量が一定以下になると，スラッシングが起きないようにするために，メモリスケジューラに対しページアウト要求を出す．但し，リアルタイムタスクのデッドラインが近い場合，この処理を先伸ばしにし，そのタスクのデッドラインの処理が終了するまで待つ．

4.2 *Kaku* で用いるデータ構造

全てのポリシハンドラは，メモリ空間の管理において全て同一のメモリ管理記述子を使用している．このメモリ管理記述子は，次の 2 つの記述子によって構成され，仮想メモリ空間を管理している (図 3)．

- `mm_struct`
  `mm_struct` は仮想メモリ空間 1 つに対して 1 つ割り当てられる．これは，`vm_block` 記述子を管理するための情報や，仮想メモリ全体に対する情報を含む．
- `vm_block`
  `vm_block` 仮想メモリ空間上に作成された領域 1 つに対して 1 つ割り当てられる．これには，領域の保護属性，ファイルのメモリマッピングに関する情報，その領域に対する各

種操作へのポインタを保持する．vm_block 記述子は map_file() 関数によって生成される．vm_block で定義される操作は次の６つである．

- unmap
  マップされているファイルを解除して vm_block 記述子を解放する．
- swi_mapped
  ファイルがマップされた領域に対してページインを実行する．
- swi_anon
  匿名マッピングされた領域に対してページインを実行する．
- swap_out
  ページアウトを実行する．
- cowr
  コピー・オン・ライトを実行する．

これらは一般的な UNIX オペレーティングシステムが採用しているものとほぼ同じであり，*Kaku* でもメモリマッピングや匿名マッピング，コピー・オン・ライト [12] 等の機能を備えている．また，実装において，*Kaku* における複数ポリシをサポートできるように，柔軟性を持たせている．

4.3 ファイルマッピング

ポリシハンドラの管理する，全ての仮想メモリ領域はファイルと対応する．従って，vm_block 記述子の生成はファイルマッピングを行う関数 file_map() によって行われる．file_map() 関数はマッピングの種類として，以下の４つを指定することが出来る．

- FMAP_PRIVATE
- FMAP_SHARED
- FMAP_FIXED
- FMAP_STICKY

FMAP_PRIVATE，FMAP_SHARED はそれぞれプライベートマッピング，共有マッピングを行う．FMAP_FIXED は指定したアドレスにマッピングを行う，もしこれが指定されていなければ，file_map() 関数は仮想メモリ空間中の空きを検索しそこに自動的にマッピングを行う．

FMAP_STICKY は，リアルタイムタスクでページフォルトによる遅延を発生させたく無い場合に使用する．この属性を指定すると，file_map() 関数は領域を確保するとそこにファイルの内容をロードし，ページアウト不可の属性を vm_block 記述子に付け加える．また，指定されたページ分だけ，ポリシハンドラの保証ページを増加させる．この属性を指定しなければ，その領域はデマンドページングによりロードされる．

4.4 ポリシハンドラの実装

4.4.1 ポリシハンドラの扱うデータ構造

メモリスケジューラから見たポリシハンドラの実体は，ページアウト・ページインを行うためのデータ構造体の集合となる．このデータ構造体には，スワップカウンタ，物理ページ保持数，保証された物理ページ数，管理している mm_struct のリストの先頭へのポインタと，ページアウト・ページインをするページ選択ための関数へのポインタが含まれている．

保証された物理ページ数は，現在はリアルタイムタスクに対するポリシハンドラでのみ使われており，初期化時に保証された物理ページの数を表している．また，スワップカウンタは，メモリスケジューラにより，ポリシハンドラの物理ページ保持数で初期化される．

さらに，実際にディスクへのページアウト・ページインを行なう作業を分離して管理していることにより，ポリシハンドラ内で行う作業を，対象ページの選択だけに限定する可能である．

4.4.2 非リアルタイムタスクのページアウト処理

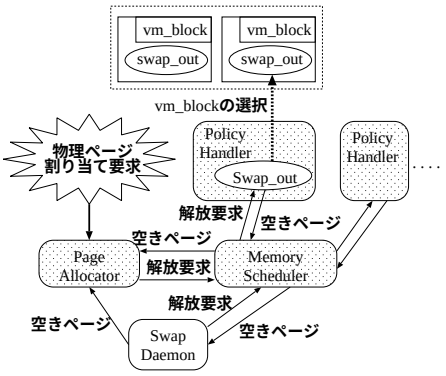

図 4: 非リアルタイムポリシのページアウト処理

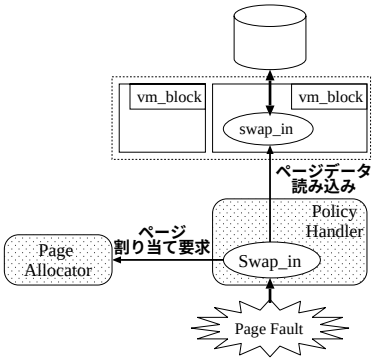

図 5: 非リアルタイムポリシのページイン処理

ページアウト処理での要求の流れは非リアルタイムタスクの場合，図 4のようになる．メモリスケジューラは，ページアロケータに対する物理ページの割り当て要求によってだけでなく，空き物理メモリ量を監視するページデーモンからも呼び出される．

そして，ポリシハンドラではポリシに従って適切な `mm_struct` 構造体， `vm_block` 構造体を選択し，最終的にスワップアウト対象ページを決定する．ページが決定すると `vm_block` 構造体に定義されるスワップアウトオペレーションを開始し，実際にバッキングストアにページをセーブしてページの解放を行う．そして，解放されたページはページアロケータに返される．

4.4.3 非リアルタイムタスクのページイン処理

ページインは基本的にページフォルトをトリガとして実行される．ページフォルトが起こるとページフォルトハンドラが呼び出される．ページフォルトハンドラはフォルトの起こしたタスクの関連づけられたポリシハンドラを探し，ページイン操作を呼び出す (図 5)．

そして，ページインを操作を行なう際には，フォルトを起こしたページだけでなく，今後参照されるであろうページも同時にページインしておく事で，ページフォルトの回数を減らすことが可能である．

さらに，ページイン・オペレーション処理中には，ページアロケータに物理ページ割り当ての要求が出され，それにより得た物理ページをページテーブルに登録する．次にそのページが属する `vm_block` 構造体のスワップインオペレーションを呼び出し，バッキングストアからページのデータを読み出す．

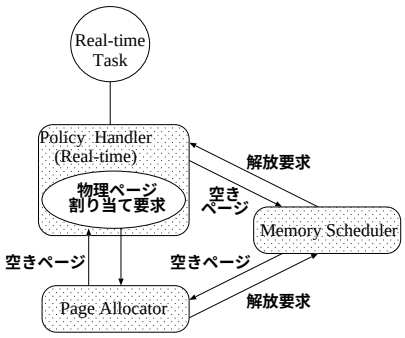

図 6: リアルタイムポリシにおけるページング

4.4.4 リアルタイムタスクのページング

リアルタイムタスクでは，ページアロケータにおけるメモリ割り当てに失敗に対して，非リアルタイムタスク用のポリシとは異なる動作を行う．リアルタイムタスクにおけるページングを図 6に示す．

最初にポリシハンドラはページアロケータからのメモリ割り当てを試みる．ここで，リアルタイムタスクからのメモリ割り当て要求に失敗した場合，メモリスケジューラは他のポリシハンドラではなく，呼び出し元タスクに対応するポリシハンドラに対してページアウトできるものがないか問い合わせを行う．この時，ページアウト対象を見つけられなければ，非リアルタイムタスクの場合と同様にメモリスケジューラにより他のポリシハンドラにスワップアウト要求を出す．

もし，リアルタイムタスクに関連づけられているポリシハンドラによりページアウト対象ページを見つけることができれば，物理ページの流れをポリシ内で閉じることが可能となる．これにより，他のタスクによって影響を受けることがなく，メモリ割り当てに要する時間を一定とすることができる．限定的ではあるが，この場合に QoS クロストーク問題を解決することが可能となる．

5 評価

実際に複数のタスクに対して個々にポリシを適用し，最適化が可能かどうかを調べるために実験を行った．この測定は全て Pentium 90MHz を搭載した IBM PC/AT 互換機を使用し，メモリ 64MByte の環境を使用した．

表 1: ページフォルト起動回数

	NRU	線形
Task1	0 回	10 回
Task2	20 回	0 回

まず，タスクに対して適用するポリシとして，次の 2 つのポリシハンドラを用意した．どちらのポリシも，スワップインはデマンドページングで行っている．

1. NRU(Not Recently Used) によってスワップアウト候補を選ぶポリシハンドラ
2. 単純に下位アドレスから順に線形にスワップアウトしていくポリシハンドラ

そして，次の 2 つのプログラムを作成し，それぞれのポリシに関連づけ，ページフォルト回数を測定した．

1. 局所性のあるデータ参照を行うプログラム (Task1)[1]．
2. データを上から順に参照していき，一度参照したデータは二度と参照しないプログラム (Task2).

測定結果を表 1に示す．

表 1の結果より， 1のプログラムに対して NRU ポリシを適用し， 2のプログラムに対して線形ポリシを適用する方が，より性能をだせるということがいえる．

6 まとめ

本稿では，複数のメモリ管理ポリシをリアルタイム性を考慮して共存させるメモリ管理システム覚 (*Kaku*) について述べた．

そして実験により，メモリ管理ポリシを複数共存させる事でタスクの特徴に合わせ，適切なページのページアウト，ページインが行えることを示した．また，リアルタイムタスクについても，独立してメモリ管理を行うことができる機構を提供することにより， QoS クロストークによる予測不可能な時間を最小限に抑えることができ，時間保証を効率的に実行することが可能となることを示した．

[1] 80Kbyte のメモリ空間を参照範囲として同一範囲を 10 回参照し，その範囲を上位アドレスに向かって 9Kbyte づつずらした．

今後は，未実装のスワップファイルシステムの実装，ページデーモンのタイミングの調整， *Aya* の動的再構成の機構を利用したポリシハンドラの動的追加／削除に対するユーザインタフェイスの設計／実装を行い評価を行う予定である．

参考文献

[1] Hideyuki Tokuda, Tatuo Nakajima, and Prithvi Rao. Real-Time Mach:Towards a Predictable Real-Time System. In *Proceedings of USENIX*, pp. 73–82, October 1990.

[2] Jason Nieh and Monica S.Lam. The Design,Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. In *Proceedings of ACM SIGOPS Symposium on Operating Systems Principles*, pp. 184–197, October 1997.

[3] Takuro Kitayama, Tatsuo Nakajima, and Hideyuki Tokuda. RT-IPC:An IPC Extension for Real-Time Mach. In *Proceedings of Workshop on Micro-Kernel and Other Kernel Architectures*, pp. 91–104, September 1993.

[4] Steven M.Hand. Self-Paging in the Nemesis Operating System. In *USENIX Operating Systems Design and Implementation*, pp. 73–86, February 1999.

[5] 谷出新, 芝公仁, 大久保英嗣. リアルタイムオペレーティングシステム Easel におけるメモリ管理機構. 情報処理学会研究報告, No. 2001-OS-86, pp. 91–98, 2001.

[6] *Extensibility, Safety and Performance in the SPIN Operating System*. Department of Computer Science and Engineering, University of Washington, 1996.

[7] M. Seltzer, Y. Endo, C. Small, and K. A. Smith. A Introduction to the Architecture of the VINO Kernel. Technical Report TR-34-94, Harvard University, 1994.

[8] M. Seltzer and C. Small. Self-Monitoring and Self-Adapting Operating Systems. Technical report, Harvard University, 1996.

[9] Timothy Roscoe. *The Structure of a Multi-Service Operating System*. PhD thesis, University of Cambridge Computer Laboratory, April 1995.

[10] Michael Stonebraker. Operating System Support for Database Management. *CACM*, Vol. 24, No. 7, pp. 412–418, 1981.

[11] 小林良岳, 佐藤友隆, 唐野雅樹, 結城理憲, 前川守. 彩：コンパイル時に自動生成される Portal をもとに動的再構成可能なオペレーティングシステム. 電子情報通信学会, Vol. J84-D-I No.6, pp. 605–616, June 2001.

[12] ユーレッシュ・ヴァハリア. 最前線 UNIX のカーネル. ピアソン・エデュケーション, 2000.