Vol.2012-OS-122 No.20
2012/8/2

OSの多重化とプロセス移植による フェイルオーバー機構の開発

吉田 健二[1]　加藤 雄大[1]　齋藤 彰一[1]　毛利 公一[2]　松尾 啓志[1]

概要：
一台の計算機に二つのOSを同時に動作させてアクティブ・バックアップ構成を組むことにより，計算機の耐障害性を向上させる手法を提案する．稼働しているOSに障害が発生した際，ホットスタンバイで待機しているOSがフェイルオーバー処理を行い計算機の制御を得る．フェイルオーバー処理では，OSのクラッシュでは通常損失するファイルキャッシュを移植しディスクへの書き出しを行う．また，事前指定された重要なプロセスの移植を行い動作を継続する．これらの処理によってファイルシステムとプロセスの信頼性を向上させ，計算機の耐障害性を向上する．

A Fast and Lightweight Failover Mechanism using Software-based Multiple Operatings System and Process Migration

Kenji Yoshida[1]　Yudai Kato[1]　Shoichi Saito[1]　Koichi Mouri[2]　Hiroshi Matsuo[1]

Abstract: We propose a failover system enhancing reliability. The system is constructed on one machine in which two OSes are simultaneously working. When the primary OS encounters a fatal error, the system automatically starts a failover process and switches the backup OS to active. In failover, the backup OS migrates file caches that will be lost if the machine is rebooted. Migrated file caches are written to disk. Moreover, the backup OS migrates a process and runs continueously. By failover, we enhance reliability of file systems and processes and achive fault tolerant computers.

1. はじめに

OSが安定して動作することは非常に重要である．何故なら，OSは計算機上で動作するすべてのプロセスに対して影響を与え，正常な実行に責任をもつ存在だからである．しかし，OSには必ずバグが存在し，何らかの機能追加や修正を行うたびにバグの数は増え続ける．現在主流となっているOSでバグをすべて消し去るのは不可能といってもよい [1]．よって，バグによる不具合が生じることを前提として耐障害性をもったOSが求められている．

OSレベルで何らかの障害が発生した場合，一般的には計算機のリブートによるリカバリが試行される．しかしリブートによる対処は計算機の実行状態を損失するのみならず，リブート中は計算機が使用不可能になるという問題がある．本稿では，計算機のリカバリをリブートを使用せずに行う手法と，リカバリ後のOSがプロセスとファイルキャッシュの状態を復元する方法について述べる．

既存の計算機の耐障害性向上方式として，高可用システムと呼ばれるものが存在する．Distributed Replicated Block Device（DRBD）[2] は，高可用クラスタ向けの分散ストレージシステムであり，ネットワーク越しにディスクを同期することによってディスクの故障に備える．DRBDはクラスタ管理フレームワークと併用することで障害時のフェイルオーバーが可能となり，クライアントと透過的な通信の継続を行うことができる．この方式の欠点は，ディスク同期の通信コストがかかることとロードバランサなどの高価な機器が必要となることからの導入コスト及び管理

[1] 名古屋工業大学
Nagoya Institute of Technology
[2] 立命館大学
Ritsumeikan University

© 2012 Information Processing Society of Japan

コストの高さである．

ファイルシステムのデータを保護するための既存の方法として，ジャーナリングファイルシステムの使用が挙げられる．Linux の標準ファイルシステムである ext3/ext4 はジャーナリングを実装しており，ext3 では Journal, Ordered, Writeback の三種類のモードを選択できる．Ordered モードはデフォルトのモードでありメタデータの保護しか行わない．Journal モードを選択すればすべてのデータの整合性を保てるが，多大なオーバーヘッドが存在する．このように，ジャーナリングはファイルシステムの整合性を保つことを第一としており，ファイルキャッシュ上のデータを保護することができない．

また，プロセスの実行状態を保護するための手法としてチェックポイント/リスタートが存在する．この手法は，ある時点でのプロセスの状態をスナップショットとしてディスクに保存しておき，障害が発生した際に直近のスナップショットの状態を再現することでプロセスを復元する手法である．例えば，libckpt[3] はユーザレベルでプロセスの実行状態を記録するためのチェックポインティングライブラリである．この手法には，定期的にスナップショットを取るために実行時オーバヘッドが増大することと，リカバリのための時間がかかるという欠点が存在する．

以上より，OS の耐障害性向上には以下の各点が求められる．

- OS のバグに対するプロセスとファイルシステムの保護
- 高速なリカバリ
- 最小限の計算機構成での実現
- 最小限のオーバーヘッド

これらの各項目を達成するために，提案手法では二つの OS を一台の計算機上で実行することによる OS の冗長化を提案する．冗長化におけるリカバリは OS の切り替えによるフェイルオーバーで可能になるが，その際にクラッシュした OS 上で動作していたファイルキャッシュの復元とプロセスの動作の継続を行い，障害後も引き続いての処理を可能にする．

本論文の構成は次のとおりである．まず 2 章で関連研究を述べ，問題点を明確にする．3 章では提案手法の概要と利点について述べる．4 章で提案システムの設計を述べ，5 章で実装方法を説明する．そして 6 章で評価を行い，7 章でまとめと今後の課題を述べる．

2. 関連研究

OS のクラッシュに対しての耐障害性を高めるための手法と，提案手法と同じ技術を使用するシステムについて述べる．

RIO File Cache[4] は sync() システムコールを改良し，OS のクラッシュ後にファイルキャッシュの正常な書き出しを保証する．そのためにカーネルクラッシュ時に sync() が実行されるようにハンドラを設定し，またその際の書き込み処理に故障したカーネル内のデータ構造を使用しない．これらによりファイルシステム自体の冗長性を高めなくても高い信頼性でファイルデータの破壊を防ぐことができる．しかし，この手法ではプロセスの復元を行わない．また，クラッシュしたかどうかの判定を信頼性の薄いクラッシュした OS に任せているという欠点がある．

MINIX[5] はデバイスドライバのマイクロリブート機構を有するマイクロカーネル OS である．一般にデバイスドライバは，コード数が非常に多い上に実装がハードウェア依存であるために，バグを含みやすい．MINIX では，デバイスドライバをユーザ空間のアプリケーションとしてカーネル空間から分離している．そして，何らかの障害が発生した際にそのデバイスドライバのみリブートを行うマイクロリブートを使用することで，ドライバのバグに対する耐障害性を得ている．しかし，この手法ではドライバ以外を原因として OS がクラッシュする場合はやはり計算機のリブートが必要である．ドライバ以外の部分のバグにはこの手法では対処できない．

Otherworld[6] はカーネルのマイクロリブートを行うことによってプロセスの実行状態を保護する手法である．カーネルに障害が発生した際に，kexec[7] により新たなカーネルをウォームブートする．さらに，障害発生時に実行中だったプロセスを障害が発生したカーネルのメモリ空間から取り出し，新たに起動したカーネル上で実行を再開する．これによってカーネルのバグからプロセスを保護する方式である．しかし，この手法ではファイルキャッシュの移植を行わないため，プロセスがファイル操作を行った直後には正常な動作を保証できない．

ReHype[8] では，VMM のクラッシュ時に動作中の VM の実行状態の保護を行う．クラッシュした状態の VMM のメモリ内容からデータを読み出し，マイクロリブートした VMM に移植を行うことで VM の実行継続のためのオーバヘッドを削減して高速なフェイルオーバーを実現している．この手法では VM の実行状態を保護することから，カーネル内部の複雑なデータ構造に左右されないリカバリが可能となっている．しかし，ReHype は VMM のクラッシュから VM を保護する手法であり，VM 内の OS のクラッシュを保護することはできない．

3. 提案システム

計算機が OS のクラッシュによって停止してから復帰するまでの時間は，管理者が停止に気付くまでの時間と計算機をリブートする時間及びプロセスを起動して元の状態に戻すまでの時間を合わせたものである．これらの時間が長ければ長いほど計算機は可用性を失う．また，プロセスを新たに起動することによって，障害が発生した OS で動作していたプロセスの一時データが損失するという

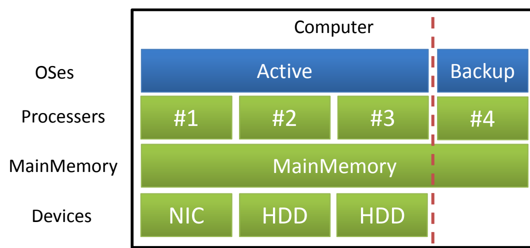

図 1　システム構成例

問題が生じる．これらの問題を解決するために，OS の多重化による耐障害性向上システム **Orthros**（ORganized Transmigratory High Reliable OS）を提案する．本研究室で開発を行っている Orthros[9] は，計算機が停止しているすべての時間の削減を行い，障害発生時のプロセス及びファイルキャッシュの状態を引き継ぐシステムである．以下，Orthros の目的と概要について述べる．

システム構成例を図 1 に示す．フェイルオーバーを行うために，Orthros では一台の計算機上で二つの OS を同時に実行し，それらをアクティブ・バックアップ構成で運用する．ActiveOS は主だってユーザーからの仕事を処理する OS であり，BackupOS は ActiveOS が故障した時に使用する予備の OS である．これらの役割上，通常実行時には ActiveOS は計算機のほぼすべてのリソースの使用権限をもち，BackupOS は動作するために最低限必要な CPU 及びメモリのみを利用する．

管理者が障害に気付くまでの時間を削減するために，BackupOS は ActiveOS を監視して障害の自動検知を行う．障害検知の制御は BackupOS が行うため，RIO File Cache のように自身で検知を行う手法と比べて確実に保護操作を行うことが可能である．

そして障害が検知された際には自動でフェイルオーバー処理を実施し，BackupOS は ActiveOS が保持していたリソースを引き継ぐ．フェイルオーバー処理はプロセスの動作する OS の切り替えによって行われ，これにより計算機上で行われていた処理の継続が可能となる．また，OS 同時実行と切り替えによるフェイルオーバーを行うことで，カーネルのマイクロリブートよりも高速なリカバリを行うことができる．

プロセスの移植を行うために，BackupOS はプロセスの動作していた環境の移植を行う．ここで環境とは，プロセスが使用する実行ファイル，設定ファイル，ライブラリ，IP アドレス，デバイスと定義する．これらは ActiveOS と BackupOS の両方でプロセスが同じ動作をするために必要である．

そして BackupOS 上に移植された環境下にプロセスとファイルキャッシュを移植し，プロセスの実行を継続する．しかし，プロセスの移植のみではプロセスがファイル操作直後であった場合にファイルの内容との間に不整合が生じる．そこで，Orthros ではファイルキャッシュの移植も行うことでフェイルオーバーの成功率を高める．

最小限の計算機構成及び，最小限のオーバーヘッドを達成するために，すべてのシステムを一台の計算機内で完結させる．高可用システムとジャーナリング及びチェックポイント/リスタートは，障害の発生に備えて常にデータの同期または保存を行う実行時オーバーヘッドが存在した．Orthros では，同一の計算機内であることを利用して BackupOS は ActiveOS のメモリ内容に障害発生後のアクセスが可能である．したがって，既存手法における障害発生前のチェックポイント処理を障害発生後の一度だけのデータ復旧に置き換えることができ，実行時オーバーヘッドを削減可能である．このトレードオフとして，BackupOS は通常実行時に CPU の一部のコアとメモリの一部を専有する．しかしこの専有によるトレードオフは，計算機の高性能化によるメニーコア化及び使用可能メモリの増大のため問題にはならないと考える．また，OS の同時実行には特殊な機器を必要とせず，マルチコアの CPU だけあれば動作可能にする．これにより機材導入コストを低く抑えることができる．

Orthros と同様に OS を多重化する方法として，仮想化技術を使用する方法が考えられる．この方法でもまた，OS の障害発生で VM が停止した後にフェイルオーバー処理を行うことが可能である．しかし，VM をまるごと切り替える方法では Orthros で扱っている OS のバグには対応できない．これは，VM の内部が破壊されるため VM 自体を切り替えても OS の実行状態を復旧させることはできないからである．

4.　設計

本章では，まずフェイルオーバー処理の概要について述べる．次に，フェイルオーバーに必要な以下の各機能について述べる．

- 複数 OS 同時実行機構
- 死活監視機構
- デバイス移植機構
- ファイルキャッシュ移植機構
- プロセス移植機構

また，本章の最後に Orthros では対応することができない障害について述べる．

4.1　フェイルオーバー処理の概要

計算機上では複数 OS 同時実行機構によって ActiveOS と BackupOS の二つの OS が動作する．BackupOS は死活監視機構によって常に ActiveOS を監視しており，ActiveOS に異常が発生した場合にはリカバリを開始する．この時 BackupOS が行うフェイルオーバーの動作概要について述べる．

© 2012 Information Processing Society of Japan

BackupOS は ActiveOS と同一のデバイスを利用できないために，そのままでは ActiveOS の処理を肩代わりすることができない．したがって，ActiveOS のクラッシュを検知後に BackupOS はまず ActiveOS の所持するデバイスをデバイス移植機構を用いて引き取り，使用できる状態にする．次に，ActiveOS のクラッシュが原因でディスクに書き出されていない dirty なファイルキャッシュをファイルキャッシュ移植機構によって BackupOS 上に移植する．これによってプロセスが行ったファイル操作を ActiveOS のクラッシュから保護することが可能となる．これらの処理によって ActiveOS と同様のプロセス実行環境を整えた後に，プロセス移植機構により BackupOS はプロセスの移植を行う．停止したプロセスを移植して実行を継続することでフェイルオーバーが完了する．

4.2 複数 OS 同時実行機構

二つの OS を一台の計算機上で実行するために，Orthros では仮想マシンの一種である LPAR[10] を利用する．LPAR はハードウェアの仮想化機構であり，ハードウェアを論理的なパーティションに分割し，各パーティションを仮想的な計算機にする．そしてそれぞれのパーティションで OS を実行することで複数の OS を一台の計算機上で同時動作させる技術である．通常は専用のハードウェアを用いる技術であるが，Orthros は LPAR をソフトウェアで実装した SHIMOS[10] と同様の機構を実現することで，ほぼ機器導入コストを無くして LPAR を実行する．

LPAR 上の各 OS のカーネルは事前に使用可能なデバイスを指定されており，起動時には指定されていないデバイスにはアクセスしない．これにより，中央集権的な仮想化機構をもつことなく，各 OS が一つの計算機を共有することができる．例えば，CPU が 4 コアでメモリが 8GB，HDD を二台もつ計算機上で二つの OS を同時実行する場合，第一パーティションに CPU のコア 0，1 とメモリの 0〜4GB と HDD 一台を割り当て，残りを第二パーティションに割り当てる．この分割によってそれぞれのパーティションで独立して OS が実行可能である．

Orthros で LPAR を用いる理由を述べる．第一の理由は，LPAR の特徴は仮想化オーバーヘッドが非常に小さいことである．これは，LPAR がハードウェアを仮想化せずにパーティションを作成するためである．第二の理由として，LPAR は動的にパーティションを変更することで OS 間でハードウェアの移植を行うことが可能である．この機能を活用して Orthros に必要な他の機構を容易に実現することができる．

4.3 死活監視機構

BackupOS が ActiveOS の状態を監視する機構について述べる．OS が故障によって停止するとき，OS 自らが故障を検知できる場合とできない場合の二つの場合が考えられる．自らが検知できない場合には，何らかの原因でカーネルが急停止した場合やデッドロックに陥った場合が考えられる．このような状況を引き起こすバグは少なくない [1].

ActiveOS が故障を検知できる場合，BackupOS が停止を検知するのは容易である．クラッシュした ActiveOS は，異常の発生を BackupOS に対して通知できるからである．しかし故障を検知できない場合，ActiveOS は能動的に異常通知を行うことができないため，BackupOS は ActiveOS の生存確認を行う必要がある．確認方法として，ActiveOS は定期的な生存確認メッセージを BackupOS に送信する．そして BackupOS は生存確認メッセージを一定時間受け取らなかった場合に，ActiveOS が故障していると判断してフェイルオーバーを開始する．

4.4 デバイス移植機構

デバイス移植機構は，ActiveOS がクラッシュした時に ActiveOS のデバイスを BackupOS に引き継ぐ機構である．

Orthros では ActiveOS と BackupOS が計算機上のデバイスを分割し専有している．これにより通常実行時は使用できるデバイスが異なっている．そこで，フェイルオーバーを行う際に ActiveOS のデバイスを BackupOS に移植する．例えば，NIC を移植することで BackupOS は ActiveOS と同じ IP アドレスと同じ MAC アドレスで通信を行うことができる．これによって外部の計算機からの ActiveOS に対するネットワーク越しの要求に BackupOS が代わりに対応し，リモートプロセスはフェイルオーバーを意識せず引き続きの通信を行うことが可能である．同様に，ディスクを BackupOS に移植することによって，この後の処理で移植するプロセスがそれまで使用していたファイルを継続的に使用することができる．また，ディスクの移植はファイルキャッシュを移植後に書き出すために必ず必要な処理でもある．

IP アドレスを複数の OS 間で共有する他の手法として，仮想 IP アドレスを用いた方法が考えられる．仮想 IP アドレスは一つの NIC に複数の IP アドレスを保有することを可能とする技術であり，Gratuitous ARP[11] を用い必要に応じてマシン間で即座に IP アドレスを移動することができる．仮に Orthros でこの方法を使う場合，少なくとも一つの OS に対して一つの NIC を用意する必要がある．これは，通常時には何も処理を行わない BackupOS に殆ど専用のデバイスを消費することになる．Orthros では，フェイルオーバー時に NIC を移植して最小限のデバイス構成を実現する．

ファイルシステムに関しても，デバイス移植でオーバーヘッドを抑えながら耐障害性を向上させることができる．3 章で言及した DRBD を用いて，ActiveOS と BackupOS が持つディスクを常に同期して同じファイルをもつことは

可能である．しかし，ディスクの同期は計算リソースを消費し，またネットワーク越しのデータ転送が必要なため実行時オーバーヘッドが増大する．一方で Orthros は一台のディスクを ActiveOS から BackupOS に移植することで同じファイルにアクセス可能である．ディスクそのものが故障した場合などを考えると DRDB を用いた方法が耐障害性で優れるが，Orthros は OS の不具合を扱うものであり，ハードウェアの故障に大しては RAID 等の既存技術の組み合わせで信頼性を高めることができると考える．

4.5 ファイルキャッシュ移植機構

ファイルキャッシュは低速な I/O 処理を削減するための仕組みである．ファイルキャッシュに書き込みを行った際のディスクへの変更は，同期的に反映する方法（Write-through）と非同期的に反映する方法（Write-back）がある．Write-through はキャッシュへの変更とディスクへの変更が同時に行われる．一方で Write-back はキャッシュの変更だけがまず行われ，ディスクへの書き出しは遅延して行われる．Write-back は Write-through に比べて効率的であるが，OS がクラッシュした場合に直前の変更が失われるという問題がある．

ファイルキャッシュ移植機構は，Write-back を採用するファイルシステムの信頼性を向上させるものである．つまり，プロセスがファイル操作を行ってから書き出されるまでのデータを保護する．本機構では，ActiveOS 上に存在する書き出されていないファイルキャッシュを BackupOS 上に移植する．これによりリブートでは失われるファイルキャッシュを正常にディスクに書き出すことができる．

ファイルキャッシュの移植は，ActiveOS が専有していたメモリ領域中のファイルキャッシュとそれを取り扱うデータ構造を取り出し，BackupOS が管理できる形に再構成することによって行う．この処理はフェイルオーバー時にのみ行われるため，実行時オーバーヘッドを伴わない．

4.6 プロセス移植機構

プロセス移植機構は，実行途中のプロセスが保持しているデータの保護を行うための機構である．ActiveOS 上に存在するプロセスのメモリ領域とプロセス管理構造体を読み取り，BackupOS が扱える形に再構成を行う．

Orthros では，カーネルの不具合に対する耐障害性の向上を目的としていることから，カーネル空間に存在するプロセス用のデータは使用しない．何故なら，OS に発生する障害はそのほぼすべてがカーネル空間での処理に因るものであり，プロセス移植の際にカーネル空間のデータをそのまま移植すると障害発生の原因となった状況を再現する可能性が高まるからである．よって，プロセスのメモリ領域を移植する際にはユーザ空間のメモリ内容のみを移植してカーネル空間のメモリ内容は破棄する．このため，プロセスがシステムコールを発行していた場合には，システムコールの継続実行は不可能である．したがって，実行途中のシステムコールは失敗したとして再構成する．これは，実用レベルのプログラムならばシステムコールが失敗した際のエラーハンドリングは当然実装されていると考え，プロセス移植後のシステムコールの異常終了に対するリカバリをプロセスに委任するものである．プロセス移植は，ファイルキャッシュを移植する場合と同様に実行時オーバーヘッドは殆ど伴わない．

4.7 Orthros が対象としない障害

本研究の対象はカーネル内のバグによる OS の停止である．一方でファイルキャッシュ及びプロセスに関係するデータ構造を破壊するバグには対処できない．しかし，このような状況は殆ど起こりえないと考える．何故なら，カーネル内のデータを取り扱う関数は，各所で変数の値をチェックすることで，誤った値の伝播を防いでいるからである．これにより，関係のないバグでファイルシステムやプロセスのデータが破壊されることは少ないと期待できる．ただし，それらのデータ構造を扱う部分自体にバグが存在した場合はデータの破壊が発生する．そのため，これらの関数のバグは本研究の対象外である．

また，Orthros ではハードウェアの物理的な故障には原理的に対処できない．しかし，ディスクの故障に対しては RAID 構成をとり，停電に対しては無停電電源装置（UPS）を用いるなど，ハードウェアに対する既存の耐障害性向上手法を組み合わせることが可能である．

5. 実装

4 章で述べた各機構の実装について述べる．実装は Linux（version 2.6.38, processer type x86_64）に行った．

5.1 複数 OS 同時実行機構

OS 同時実行機構は SHIMOS を参考に実装を行った．各 OS が専有する CPU コア，メモリ，デバイスはカーネルパラメータによって起動時に指定される．カーネルは初期化ルーチン内でパラメータを解釈し，自身が専有するハードウェアのみを初期化して起動する．二つの OS の起動はまず ActiveOS を最初に起動し，その ActiveOS が BackupOS を起動することで行う．ActiveOS の起動は通常の OS と同様に行われるが，BackupOS は kexec を改変したブートローダによって起動する．

各 OS が専有する CPU コアはその OS の起動コアの番号と使用可能なコア数を指定することで決定する．Linux カーネルには使用可能なコア数を指定するパラメータ（maxcpus）が存在するため，このオプションの使用とブートローダを改変して実装したコア指定での起動を行うことによって実現した．また，専有するメモリの範囲の設定も

Linux カーネルに備わっているパラメータ（crashkernel, mem, memmap）があり，これを指定してカーネルにその範囲以外を使用させないことで行う．同時に，専有メモリの設定と同じ方法で共有メモリを作成する．共有メモリは，OS 間で通信して情報交換を行うために必要な領域である．専有するデバイスの指定は，ソースコードに ActiveOS と BackupOS が排他的に扱うデバイスのリストをコーディングすることにより行う．パラメータで指定されたリストに記述されたデバイスの初期化は行わないことで，その OS はデバイスを使用不可能として，排他的にデバイスを利用できるようにする．各デバイスの認識に使用する識別子はバス番号とデバイス番号の組を利用している．

5.2 死活監視機構

ActiveOS の生死確認には Inter-Processor Interrupt（IPI）を用いる．Orthros では，ActiveOS が保有するコアと BackupOS が使用するコアの間で通信を行うために専用の割込ハンドラを登録して使用する．通知は二種類あり，ActiveOS がクラッシュしたことを伝える訃報と，ActiveOS が正常に動作していることを伝える心拍である．どちらも ActiveOS から BackupOS に向かって送信される．

訃報は panic() 関数の中で送信する．panic() 関数はカーネル内で深刻なエラーが起きたときに呼び出される関数である．訃報を伝える IPI の宛先は BackupOS に対して割り当てられた CPU コアの中で一番番号が若いものを選ぶ．この IPI を BackupOS が受信することで ActiveOS の異常停止を BackupOS が検知する．

4.3 節で述べたように，ActiveOS がデッドロックなどで停止した場合は panic() 関数が呼ばれないため ActiveOS が BackupOS に対し訃報を送ることができない．この問題に対処するため，ActiveOS は BackupOS に対して定期的に心拍を伝える IPI を送信する．BackupOS はインターバルタイマを用い，一定時間内に IPI が受信されたかどうかを判定している．心拍の間隔は短いほどクラッシュ検知が素早く行われるが，割込処理が増えることは望ましくない．ただし IPI によって呼び出される割込ハンドラはシンプルなものであるため，割込一回当たりのオーバーヘッドは軽微なものである．心拍を送る間隔は変更可能であり，現在の実装では 1 秒間隔で送信している．以上より，BackupOS は訃報を受信するか心拍の受信が一定期間滞った場合に ActiveOS がクラッシュしたとみなし，BackupOS はフェイルオーバーを開始する．

5.3 デバイス移植機構

フェイルオーバー時のデバイス移植では，ActiveOS が専有していたデバイスを BackupOS が扱えるようにする．5.1 節で述べたように，各 OS は他の OS の専有デバイスリストで指定されたデバイスを初期化しないことで排他的なデバイスの使用を実現している．つまり，フェイルオーバー時には専有デバイスリストを参照することで ActiveOS が専有しているデバイスのリストを得ることができる．BackupOS は ActiveOS が専有していたデバイスの再初期化を行うことでデバイスを使用可能にする．再初期化は，BackupOS 起動時に実行されなかった初期化ルーチンを改めて実行することで可能である．

デバイスの再初期化が完了した後，デバイス毎にそれぞれ指定された処理を行う．例えば，NIC を再初期化した後には IP アドレスの割り当てが必要になり，ディスクに対しては適切なパスにマウントする必要がある．本実装では，IP アドレスの割り当ては ifconfig コマンドを用いている．ディスクは ActiveOS と BackupOS の両方で同じパスにマウントする必要がある．そこで，ActiveOS 上で保護が必要なプロセス及びファイルは/export に配置しておき，ディスク移植時に mount コマンドを用いて同じ名前の空ディレクトリにマウントする．これらのコマンドは死活監視機構が異常を検知した際に実行するフェイルオーバースクリプトにて実行される．

5.4 ファイルキャッシュ移植機構

ファイルキャッシュの書き出しを行う動作について述べる．実装に用いたファイルシステムは ext3 である．ファイルキャッシュの移植処理は，指定したパーティション上のディスクに書き出されていない inode を BackupOS が使用できるデータに再構成することで行う．

BackupOS は ActiveOS のメモリマップを知らないため，ActiveOS は通常実行時に一度だけ BackupOS に後述する super block という構造体の位置するアドレスを知らせる必要がある．このアドレス情報の通知は，ActiveOS と BackupOS の共有メモリに書き込むことによって行う．また，ActiveOS の専有メモリ領域を BackupOS が読み取ること自体には特に障害はないが，データ構造体が持っているポインタの変換には注意が必要である．何故なら，各カーネルは独立した仮想メモリ管理を行っており，同じ値のポインタが指す仮想アドレスは OS 毎に別の物理アドレスにマップされているからである．この問題は Linux に存在するストレートマップ領域を用いて解決できる．ストレートマップ領域は，物理メモリ領域と仮想メモリ領域のアドレスが一対一でマップされており，ActiveOS と BackupOS で必ず同一のマッピングとなる．そして OS の扱うデータ構造のアドレスは殆どこの領域にマッピングされた領域を指している．そのためキャッシュを取り出す操作は，キャッシュと関連する構造体のアドレスが分かればその中身を取り出すことができるようになり，仮想メモリのマッピングに対して特別な実装を行う必要はなかった．

Orthros の実装ではディスクの各パーティション毎にファイルキャッシュの移植操作を行う．ext3 においてディ

スクパーティションは super block という構造体で表されている．この super block はファイルキャッシュ書き出し用のスレッドを表す構造体のアドレスを持っており，その構造体は dirty な inode のリストを持っている．そのリストに含まれる inode が移植対象のデータ構造である．この inode をバックアップ OS が使用可能なデータに構成しなおし，BackupOS 上にデバイス移植されたディスクの super block からリンクされた dirty の inode のリストにつなげる．そして，通常の BackupOS の処理によって，この inode が持つ dirty なキャッシュがディスクに書き出されるのを待てばよい．

5.5 プロセス移植機構

プロセスを移植する処理は，ファイルキャッシュと同様に ActiveOS のメモリ上のデータ構造を読み取って BackupOS 上で実行できる形に再編成することで行う．ファイルキャッシュと同様に，カーネル内のデータ構造のアクセスにはストレートマップ領域を利用し，ActiveOS は通常実行時にプロセスを管理する構造体のアドレスを共有メモリに書き込む．

実装は fork() システムコールの仕組みを応用して行った．fork() は呼び出し元の親プロセスと同じ状態の子プロセスを新たに作成するシステムコールである．fork() は新たなプロセスの雛形を作成した後に親プロセスからレジスタの状態及びメモリの内容等の状態をコピーする．本実装ではこの各状態のコピーを行うコードを改変し，親プロセスではなく ActiveOS の専有メモリ領域上のプロセスのコピーを行う．BackupOS は移植用に改変を施した fork() を実行し，ActiveOS の停止したプロセスのコピーを作り出す．これにより，カーネル空間のデータは極力破棄しながらユーザ空間のデータをコピーし，BackupOS 内にプロセスを移植する．

ActiveOS のプロセスをコピーして実行するためには，クラッシュした瞬間のプロセスが持つ各レジスタの状態を得る必要がある．レジスタの保存状態は，四つの場合に分けることができる．まず，保存された状態を利用可能な二つの場合について述べる．プロセスがシステムコールを実行中の場合は，システムコールを終了した時のためにメモリ上にレジスタの状態が保存されているためそれを使用する．また，実行待ち状態の場合も同様に次のスケジューリング時に使用するためにメモリ上に保存されているレジスタの状態を用いることができる．次に，レジスタの状態を容易に取得できない二つの場合について述べる．まず，ユーザ空間で動作中のプロセスのレジスタの状態は，クラッシュした瞬間に消失する．これに対処するため，panic() 関数の中で各 CPU コアに Non-Maskable Interrupt（NMI）を発行してレジスタの状態を保存する．NMI は他のすべての割込に優先するため，確実に保存処理を行うことができる．最後に，ActiveOS がデッドロック等により panic() 関数を呼べない場合には，BackupOS が心拍の遅滞を検知して NMI を送ることによってレジスタの状態を保存する．なお，この場合の実装は今後の課題である．

プロセス構造体は非常に多種多様なデータの集合であり，そのすべてをそのままコピーすることは整合性の観点から難しい．そこで，Orthros では，プロセスが持つレジスタの状態とユーザ空間のメモリ内容，開いているファイルのみを再構成してコピーする．また，移植時のメモリコピーの時間を減らすために後から参照ができるメモリページはコピーを行わず，dirty なページのみをコピーした．さらに，開いているファイルはデータ構造をコピーするのではなく同じファイルを新しく開いた後にシーク位置などの状態を再設定することで ActiveOS のファイル管理構造のデータ破壊に備えた．

Orthros でのシグナルハンドラやソケットの状態のコピーは今後の課題であるが，これらの状態もメモリから読み出して再構成することが可能であると考えている．

表 1 計算機構成

CPU	Intel(R) Core(TM) i5 760 @ 2.80GHz 4 コア
Memory	8GB
Device	SSD(Crucial m4) x2, NIC

表 2 ハードウェアの OS への割り当て

	ActiveOS	BackupOS
CPU	3 コア	1 コア
Memory	7680MB	512MB
Device	SSD, NIC	SSD

6. 評価

評価に使用する計算機の構成を**表 1** と**表 2** に示す．現在は OS 毎に SSD を一台ずつ割り当てるため合計二台搭載しているが，将来的には BackupOS はディスクレスにする予定である．この評価では，システムの移植が正しく行えること及び移植時間を計測する．クラッシュの方法は，ActiveOS 上でプロセスが特定の処理を行っている最中に別のコアで panic() を呼び出すことで実施した．

6.1 移植の正確性

移植が正しく行えることを確かめるために，以下の単純な機能を持つ数種類のプロセスを移植した．

(1) システムコールを実行中のプロセス
(2) ファイルハンドルを扱うプロセス
(3) レジスタの矛盾に敏感なプロセス

(1) プロセスが write()/read() システムコールからの結果を待機している際に OS をクラッシュさせた．移植されたプロセスはシステムコールが失敗したものとして返り値を受け取り，ユーザプログラム内でシステムコールのエラー

処理を実施した．そして，その後は動作の継続に成功した．(2)write() システムコールを繰り返すプログラムにおいて，数回 write() システムコールを行ったところで write() システムコールの処理中に OS をクラッシュさせた．プロセスは移植後に同じファイルの同じシーク位置に対して，失敗した write() システムコールを再度実行した．また，プロセスが処理完了後に使用していたファイルを点検し，平常時と同様の矛盾のない処理結果であることを確認した．(3) 行列積演算を行うプログラムを移植した．最適化された行列積演算は CPU の持つレジスタをフルに活用し，またレジスタの値が違うと異なる結果を出力するためレジスタが正しいことのチェックに利用できる．整数行列と浮動小数点行列を演算したそれぞれの場合で，プロセスがユーザ空間で演算処理途中に OS をクラッシュさせたところ，移植後のプロセスは正しい演算結果を出力することを確認した．また，スケジューリング待機中にも同様の結果を確認した．

6.2 移植所要時間

プロセス及びファイルキャッシュの移植所要時間について，以下の各要素を変化させた場合を評価した．これらの要素の量によって，フェイルオーバー全体の時間は増減するものと考えられる．

(1) dirty なファイルキャッシュの量
(2) プロセスが保持する dirty なメモリページの量
(3) プロセスが開いているファイルの数

それぞれの要素がどの程度移植所要時間に影響するかを測るために，他の要素を固定して対象要素のみを変動させるプログラムを作成し，要素毎に保持量との相関を計測した．その結果を**図 2** と**図 3** に示す．これらの結果より，三つの要素の量とその移植所要時間は比例していることがわかる．図 2 でファイルキャッシュの量とプロセスのメモリ量ではファイルキャッシュの方が比例定数が大きいが，これはファイルキャッシュの管理構造がプロセスのメモリ管理構造よりも複雑であり，単位量当たりのオーバーヘッドが大きいためだと考えられる．

プロセスのメモリとファイルキャッシュについては，コピー量が極端に大きくなることは少ないと考えられる．何故なら，プロセスのメモリは dirty なメモリのみがコピーの対象となるためプログラムが巨大になってもメモリサイズに比例することはないためである．さらに，使用メモリの総量が大きくなると OS がページをディスクにスワップアウトする．これによりコピーが必要なメモリの量は大きくても数 GB に収まり，最悪でも 1 秒程度しか移植時間は延びないと考えられる．また，ファイルキャッシュについても，あくまでディスクへの書き出しを遅延する仕組みであり大きくなりすぎることはないため移植は高速に完了する．

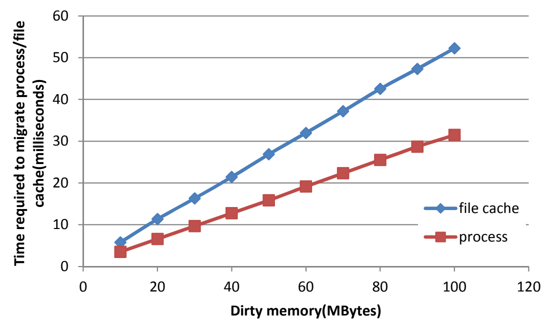

図 2 Dirty なメモリサイズによるファイルキャッシュ移植所要時間プロセス移植所要時間

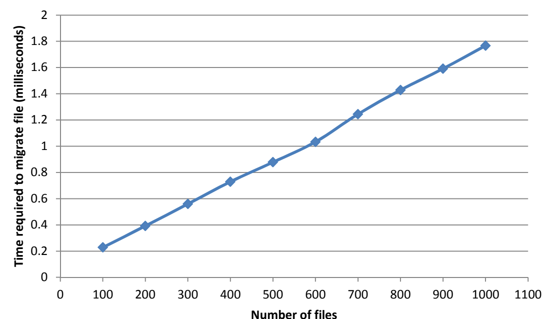

図 3 ファイル数によるプロセス移植所要時間

6.3 DBS への応用

Orthros を適用することでどのように耐障害性が向上するのかについて説明する．Orthros をデータベースシステム（DBS）に適用した場合の効果について検証を行う．

6.3.1 OS クラッシュによる DBS への影響

DBS はトランザクション処理によって OS の障害発生時も DB の整合性を保つことができる．トランザクションでは，DBS に対する一連の操作を不可分な塊として実行し，実行中に不都合が発生した場合にはその実行による影響を消し去り（ロールバック），実行が正常に完了すればそのデータをただちにディスク上に書き出す（コミット）．トランザクション処理を用いることによって，ユーザはその処理の前後での DB のデータの整合性を保つことができる．

逆にいえば，トランザクション処理を行っている最中に OS に障害が発生した場合には処理途中のデータは保護されない．処理中に行った変更はメモリ及びファイルキャッシュに保持され，リブートによって消失するためである．DBS の不具合によって DB に不整合が発生した際にはトランザクションはロールバックによって破棄されるべきだが，OS の障害によって処理が中断した場合の不整合には解決の余地が存在する．Orthros では，OS の障害によって中断されたトランザクションの処理を継続し，従来は破棄されていた状態を復旧することができる．

6.3.2 Orthros による保護

DBS が動作する OS のクラッシュによって，一時データの消失と DBS の一時停止という問題が生じる．この二つの問題を Orthros で解決する．以下，その手順を説明する．

表 3　DBS の構成

DB エンジン	SQLite[12]
フロントエンドプロセス	sqlite3 コマンドを改造したもの
DB ファイル	Chinook Database[13](1066KB)

まず初期状態として，ActiveOS 上で DBS が実行されたとする．また，5.3 節で述べたように，DBS が使用するディレクトリ（/export）は ActiveOS が使用するディスクの一つのパーティションとしてディスク上に存在する．

ActiveOS 上で DBS が動作中に，OS のクラッシュが発生したとする．BackupOS は死活監視機構によって ActiveOS が停止したことを検知し，フェイルオーバーを開始する．まず ActiveOS が使用していたディスクを BackupOS に移植する．ディスクは認識後，ActiveOS がマウントしていたパスと同じパスで BackupOS 上にマウントする．そして ActiveOS が書き出すことができなかった dirty なファイルキャッシュを移植したディスクに対して書き出す．この処理により DBS がファイルに書き込んだデータを保護することができる．

そして次にプロセスの移植を行う．この時点で DBS が実行中の処理及びファイルに書きこむ直前のデータを保護し，DBS が持つ一時データの保護が完了する．プロセス移植終了後，ユーザーは BackupOS 上に存在するコンソールを利用することによってプロセスとの対話を継続することが可能である．

6.3.3　フェイルオーバーの評価

表 3 に示す構成を用いて，フェイルオーバー実験を行った．トランザクション処理により DB に変更を加え，その後のトランザクション途中のクエリを実行中に OS をクラッシュさせてフェイルオーバーした．その後，トランザクションを継続して DB に更なる変更を加えた後にコミットを行った．結果，DB に加えた処理の結果はすべて整合性をもって正しく保持及び永続化されていることが確認できた．

次に，同様の環境を用いてフェイルオーバー所要時間を測定した．移植に要した時間を測る方法として，クエリ実行時間を用いる．1 クエリを実行するのに要した時間について，実行途中に OS をクラッシュさせてフェイルオーバーさせた時間と正常時の時間との差分から移植に要した時間を求めた．この方法で計測した結果，差分は最大でも 1.5 秒であった．これはリブートやウォームブートと比べて非常に高速なリカバリ時間である．この時，クエリに対してプロセスは約 1556KB の物理メモリを使用したが，そのうち dirty で移植されたのは約 168KB である．また，プロセスが開いているファイル数は 7 で，コピーを行った dirty なファイルキャッシュは 64KB であった．このフェイルオーバー時間はクラッシュした瞬間の OS の状態によって増減するが，6.2 節で示したことから最大でも数秒程度しかかからないと考えられる．

7.　まとめ

複数の OS を同時に実行してフェイルオーバーを行うシステムとして Orthros を提案した．Orthros は OS に障害が発生した際の高速なリカバリ及びプロセスとファイルキャッシュの保護を達成する．フェイルオーバーの実行時間は数秒であり，プロセスがフェイルオーバー後も継続して実行可能であることを評価で示した．また，Orthros は常駐オーバーヘッドが少なく，特殊なハードウェアも必要としないため導入コストが低い．

今後は保護できるプロセスの種類を増やすことでさらに様々なシステムに対応できるようにする予定である．

参考文献

[1] Palix, N., Thomas, G., Saha, S., Calvès, C., Lawall, J. and Muller, G.: Faults in linux: ten years later, *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, pp. 305–318 (2011).

[2] DRDB, http://www.drbd.org/.

[3] Plank, J. S., Beck, M., Kingsley, G. and Li, K.: Libckpt: transparent checkpointing under Unix, *Proceedings of the USENIX 1995 Technical Conference Proceedings*, pp. 18–18 (1995).

[4] Chen, P. M., Ng, W. T., Chandra, S., Aycock, C., Rajamani, G. and Lowell, D.: The Rio file cache: surviving operating system crashes, *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pp. 74–83 (1996).

[5] The Minix 3 Operating System, http://www.minix3.org/.

[6] Depoutovitch, A. and Stumm, M.: Otherworld: giving applications a chance to survive OS kernel crashes, *Proceedings of the 5th European conference on Computer systems*, pp. 181–194 (2010).

[7] Reducing system reboot time with kexec, http://www.osdl.org/.

[8] Le, M. and Tamir, Y.: ReHype: enabling VM survival across hypervisor failures, *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pp. 63–74 (2011).

[9] Yudai, K., Shoichi, S., Kouichi, M. and Hiroshi, M.: Faster Recovery from Operating System Failure and File Cache Missing, *International MultiConference of Engineers and Computer Sciencists*, Vol. 1, pp. 218–223 (2012).

[10] Shimosawa, T., Matsuba, H. and Ishikawa, Y.: Logical Partitioning without Architectural Supports, *Proceedings of the 2008 32nd Annual IEEE International Computer Software and Applications*, pp. 355–364 (2008).

[11] Perkins, C.: IP Mobility Support for IPv4, *RFC3344* (2002).

[12] SQLite, http://www.sqlite.org/.

[13] Chinook Database, http://chinookdatabase.codeplex.com/.

© 2012 Information Processing Society of Japan