

TruCluster Server

クラスタ高可用性アプリケーション・ガイド

Part Number: AA-RM88D-TE

2002 年 11 月

ソフトウェア・バージョン: TruCluster Server バージョン 5.1B

オペレーティング・システム: Tru64 UNIX バージョン 5.1B

本書では、Tru64 UNIX TruCluster Server バージョン 5.1B クラスタでアプリケーションを高可用性にする方法について説明するとともに、TruCluster Server 製品のアプリケーション・プログラミング・インタフェース (API) ライブラリについて説明します。本書は、高可用性アプリケーションを TruCluster Server 上で稼働させたり、アプリケーションを TruCluster Available Server や TruCluster Production Server から TruCluster Server 環境に移行したりするシステム管理者を対象としています。また、DLM (分散ロック・マネージャ) の同期サービス、クラスタの別名機能、または Memory Channel の高性能な機能を必要とする分散型アプリケーションを作成する開発者を対象としています。

© 2002 Hewlett-Packard Company

本書の著作権は日本ヒューレット・パッカート株式会社が保有しており、本書中の解説および図、表は日本ヒューレット・パッカートの文書による許可なしに、その全体または一部を、いかなる場合にも再版あるいは複製することを禁じます。

日本ヒューレット・パッカートは、弊社または弊社の指定する会社から納入された機器以外の機器で対象ソフトウェアを使用した場合、その性能あるいは信頼性について一切責任を負いかねます。

本書に記載されている事項は、予告なく変更されることがありますので、あらかじめご承知おきください。万一、本書の記述に誤りがあった場合でも、弊社は一切その責任を負いかねます。

本書で解説するソフトウェア(対象ソフトウェア)は、所定のライセンス契約が締結された場合に限り、その使用あるいは複製が許可されます。

COMPAQ, Compaq ロゴ, Digital ロゴは U.S. Patent and Trademark Office に登録されています。Alpha, AlphaServer, NonStop, TruCluster, および Tru64 は米国 Compaq Computer Corporation の商標です。

Microsoft, Windows および Windows NT は米国 Microsoft 社の登録商標です。Intel は米国 Intel 社の登録商標です。Motif, OSF/1, UNIX, The Open Group および X/Open は、The Open Group の米国ならびに他の国における商標です。

このドキュメントに記載されているその他の会社名および製品名は、各社の商標または登録商標です。

目次

まえがき

Part 1 TruCluster Server 上でのアプリケーションの実行

1 クラスタ・アプリケーション

1.1	アプリケーションのタイプ	1-2
1.1.1	シングル・インスタンス・アプリケーション	1-2
1.1.2	マルチ・インスタンス・アプリケーション	1-5
1.1.3	分散型アプリケーション	1-8

2 シングル・インスタンス・アプリケーションの高可用性実現のための CAA の使用

2.1	いつ CAA を使用するか	2-2
2.2	リソース・プロファイル	2-2
2.2.1	リソース・プロファイルの作成	2-3
2.2.2	アプリケーション・リソース・プロファイル	2-4
2.2.2.1	必須リソース	2-7
2.2.2.2	アプリケーション・リソース配置ポリシー	2-8
2.2.2.3	配置決定の際のオプション・リソース	2-9
2.2.3	ネットワーク・リソース・プロファイル	2-9
2.2.4	テープ・リソース・プロファイル	2-11
2.2.5	メディア・チェンジャ・リソース・プロファイル	2-12
2.2.6	プロファイルの検証	2-14
2.3	処理スクリプトの作成	2-14
2.3.1	アプリケーション・リソースの処理スクリプトを作成する 際のガイドライン	2-16

2.3.2	処理スクリプトの例	2-18
2.3.3	環境変数のアクセス	2-18
2.3.4	処理スクリプトの出力先	2-20
2.4	ユーザ定義属性の作成	2-21
2.5	リソースの登録	2-22
2.6	アプリケーション・リソースの起動	2-22
2.7	アプリケーション・リソースの再配置	2-24
2.8	アプリケーション・リソースの分散	2-25
2.9	アプリケーション・リソースの停止	2-28
2.10	アプリケーション・リソースの登録抹消	2-28
2.11	CAA の状態情報の表示	2-28
2.12	グラフィカル・ユーザ・インタフェース	2-31
2.12.1	SysMan Menu による CAA の管理	2-31
2.12.2	SysMan Station による CAA の管理と監視	2-32
2.13	CAA のチュートリアル	2-32
2.13.1	前提条件	2-33
2.13.2	準備作業	2-33
2.13.3	dtcalc の処理スクリプトの例	2-35
2.13.4	ステップ 1: アプリケーション・リソース・プロファイル の作成	2-36
2.13.5	ステップ 2: アプリケーション・リソース・プロファイル の検証	2-36
2.13.6	ステップ 3: アプリケーションの登録	2-36
2.13.7	ステップ 4: アプリケーションの起動	2-37
2.13.8	ステップ 5: アプリケーションの再配置	2-37
2.13.9	ステップ 6: アプリケーションの停止	2-38
2.13.10	ステップ 7: アプリケーションの登録抹消	2-38
2.14	CAA で管理するアプリケーションの例	2-38
2.14.1	OpenLDAP ディレクトリ・サーバ	2-38

2.14.2	CAA を使ってシングル・インスタンスの高可用性 Apache HTTP サーバを作成する	2-41
2.14.3	CAA を使ってシングル・インスタンスの Oracle8i サーバを作成する	2-42
2.14.4	CAA を使ってシングル・インスタンスの Informix サーバを作成する	2-44
3	マルチ・インスタンス・アプリケーションでのクラスタ別名の使用	
3.1	いつクラスタ別名を使うか	3-1
3.2	省略時のクラスタ別名を使用してマルチ・インスタンス Apache HTTP サーバにアクセスする	3-3

Part 2 TruCluster Server へのアプリケーションの移行

4	アプリケーション移行の一般的な問題	
4.1	クラスタ単位のファイルとメンバ固有のファイル	4-1
4.1.1	単一ファイル方式の使用	4-3
4.1.2	複数ファイル方式の使用	4-3
4.1.3	CDSL の使用	4-4
4.2	デバイスの命名規則	4-5
4.3	プロセス間通信	4-7
4.4	共用データへの同期アクセス	4-7
4.5	メンバ固有のリソース	4-8
4.6	拡張 PID	4-8
4.7	削除された DLM パラメータ	4-9
4.8	ライセンス	4-9
4.8.1	TruCluster Server のクラスタ単位でのライセンスはサ ポートされない	4-9
4.8.2	レイヤード・プロダクトのライセンスとネットワーク・ア ダプタのフェイルオーバ	4-9

4.9	ブロッキング・レイヤード・プロダクト	4-10
5	ASE アプリケーションの TruCluster Server への移行	
5.1	ASE と CAA の違い	5-1
5.1.1	ディスク・サービス	5-2
5.1.2	NFS サービス	5-3
5.1.3	ユーザ定義サービス	5-4
5.1.4	DRD サービス	5-4
5.1.5	テープ・サービス	5-5
5.2	ASE サービスから TruCluster Server への移行の準備	5-6
5.2.1	TruCluster Available Server および Production Server のバージョン 1.5 以降での ASE データベースの内容の保存	5-7
5.2.2	TruCluster Available Server および Production Server のバージョン 1.4 以前での ASE データベースの内容の保存	5-8
5.3	ASE スクリプトの検討項目	5-9
5.3.1	ASE コマンドの CAA コマンドへの置き換え	5-10
5.3.2	起動スクリプトと停止スクリプトの結合	5-10
5.3.3	スクリプトの出力のリダイレクト	5-11
5.3.4	nfs_ifconfig スクリプトの置き換え	5-11
5.3.5	適切なエラー処理	5-11
5.3.6	ストレージ管理情報の削除	5-12
5.3.7	デバイス名の変換	5-13
5.3.8	ASE 変数の置き換えまたは削除	5-13
5.3.9	終了コード	5-14
5.3.10	イベントのポスト	5-14
5.4	ネットワークの検討項目	5-15
5.4.1	別名の使用	5-15
5.4.1.1	クラスタ別名	5-16
5.4.1.2	インタフェース別名	5-17

5.4.2	ネットワーク・サービス	5-17
5.5	ファイル・システムのパーティショニング	5-18
6	分散型アプリケーションの TruCluster Server への移行	
6.1	分散型アプリケーションの TruCluster Server への移行の準備	6-1
6.2	TruCluster Server 上での Oracle Parallel Server の実行	6-2
6.3	Oracle Parallel Server の TruCluster Server への移行	6-4
Part 3	クラスタ対応のアプリケーションの作成	
7	プログラミングの検討項目	
7.1	RPC プログラムに必要な変更	7-1
7.2	移植性のあるアプリケーション: スタンドアロンとクラスタ .	7-2
7.3	CLSM のサポート	7-3
7.4	診断ユーティリティのサポート	7-3
7.5	CDFS ファイル・システムの制約	7-3
7.6	/cluster/admin/run ディレクトリから呼び出されるスクリプト	7-5
7.7	ローリング・アップグレード中のクラスタ・メンバの状態のテスト	7-6
7.8	クラスタにおけるファイル・アクセスの障害許容度	7-6
8	クラスタ別名アプリケーション・プログラミング・インタフェース	
8.1	クラスタ別名ポートの用語	8-1
8.2	クラスタ別名関数	8-2
8.3	クラスタ・ポート空間	8-6
8.4	予約ポートへのバインド (512 ~ 1024)	8-8
8.5	setsockopt() のオプション	8-9
8.6	ポート属性: /etc/clua_services , clua_registerservice() , および setsockopt()	8-11
8.7	UDP アプリケーションとソース・アドレス	8-11

9 分散ロック・マネージャ

9.1	DLM の概要	9-2
9.2	リソース	9-5
9.2.1	リソースの細分性	9-6
9.2.2	ネームスペース	9-7
9.2.3	リソースの識別	9-8
9.3	ロックの使用	9-10
9.3.1	ロック・モード	9-11
9.3.2	ロックのレベルと共存性	9-12
9.3.3	ロック管理のキュー	9-13
9.3.4	ロック変換	9-15
9.3.5	デッドロックの検出	9-15
9.4	ロックのキューからの削除	9-18
9.5	変換要求の取り消し	9-19
9.6	高度なロック技術	9-19
9.6.1	ロック要求の非同期完了	9-19
9.6.2	同期完了の通知	9-20
9.6.3	ブロック通知	9-21
9.6.4	ロック変換	9-22
9.6.4.1	ロック変換のキューイング	9-24
9.6.4.2	変換待ちキューへの強制的配置	9-24
9.6.5	親ロック	9-25
9.6.6	ロック値ブロック	9-26
9.7	DLM 関数を使ったローカル・バッファ・キャッシング	9-27
9.7.1	ロック値ブロックを使う方法	9-27
9.7.2	ブロック通知を使う方法	9-28
9.7.2.1	バッファへの書き込みの延期	9-28
9.7.2.2	バッファのキャッシング	9-29

9.7.3	バッファのキャッシング方式の選択	9-29
9.8	分散ロック・マネージャの関数のコーディング例	9-30

10 Memory Channel API ライブラリ

10.1	Memory Channel マルチレール・モデル	10-2
10.1.1	シングルレール・スタイル	10-2
10.1.2	フェイルオーバ・ペア・スタイル	10-3
10.1.3	Memory Channel のマルチレール・モデルの構成	10-4
10.2	Memory Channel API ライブラリの初期化	10-5
10.3	ユーザ・プログラムでの Memory Channel API ライブラリの 初期化	10-6
10.4	Memory Channel 構成のチューニング	10-7
10.4.1	Memory Channel のアドレス空間を拡張する	10-7
10.4.2	固定メモリを増やす	10-8
10.5	トラブルシューティング	10-8
10.5.1	IMC_NOTINIT リターン・コード	10-8
10.5.2	Memory Channel API ライブラリの初期化の失敗	10-9
10.5.3	Memory Channel の致命的なエラー	10-9
10.5.3.1	レールの初期化の失敗	10-10
10.5.4	IMC_MCFULL リターン・コード	10-10
10.5.5	IMC_RXFULL リターン・コード	10-11
10.5.6	IMC_WIRED_LIMIT リターン・コード	10-11
10.5.7	IMC_MAPENTRIES リターン・コード	10-11
10.5.8	IMC_NOMEM リターン・コード	10-12
10.5.9	IMC_NORESOURCES リターン・コード	10-12
10.6	Memory Channel のアドレス空間へのアクセス	10-12
10.6.1	Memory Channel のアドレス空間へのアタッチ	10-14
10.6.1.1	ブロードキャスト・アタッチ	10-14
10.6.1.2	ポイント・ツー・ポイント・アタッチ	10-16

10.6.1.3	ループバック・アタッチ	10-17
10.6.2	初期状態の一貫性	10-18
10.6.3	Memory Channel 領域の読み取りと書き込み	10-19
10.6.4	アドレス空間の例	10-19
10.6.5	遅延と一貫性	10-23
10.6.6	エラー管理	10-27
10.7	クラスタ単位のロック	10-32
10.8	クラスタ・シグナル	10-35
10.9	クラスタ情報	10-36
10.9.1	Memory Channel API の関数を使って Memory Channel API クラスタ情報にアクセスする方法	10-36
10.9.2	コマンド行から Memory Channel の状態情報にアクセスす る方法	10-37
10.10	共用メモリ・モデルとメッセージ渡しモデルの比較	10-38
10.11	よくある質問 (FAQ)	10-39
10.11.1	IMC_NOMAPPER リターン・コード	10-39
10.11.2	効率的なデータ・コピー	10-39
10.11.3	Memory Channel の帯域幅の可用性	10-40
10.11.4	Memory Channel API クラスタ構成の変更	10-40
10.11.5	バス・エラー・メッセージ	10-40

索引

例

9-1	ロック, ロック値ブロック, およびロック変換	9-31
10-1	Memory Channel のアドレス空間の領域へのアクセス	10-20
10-2	System V IPC および Memory Channel のコードの比較	10-24
10-3	imc_rderrcnt_mr 関数を使ったエラー検出	10-28
10-4	imc_ckerrcnt_mr 関数を使ったエラー検出	10-31
10-5	Memory Channel の領域のロック	10-33

10-6	Memory Channel API クラスタ情報を要求し , Memory Channel API クラスタ・イベントを待つ方法	10-37
図		
1-1	シングル・インスタンス・アプリケーションへのアクセス ...	1-3
1-2	CAA を使用したアプリケーションのフェイルオーバー	1-4
1-3	マルチ・インスタンス・アプリケーションへのアクセス	1-6
3-1	マルチ・インスタンス・アプリケーションへのクラスタ別名を介したアクセス	3-2
9-1	データベースのモデル	9-7
9-2	3つのロック・キュー	9-14
9-3	変換デッドロック	9-16
9-4	複数リソース・デッドロック	9-17
10-1	シングルレール Memory Channel の構成	10-3
10-2	フェイルオーバー・ペア Memory Channel の構成	10-4
10-3	ブロードキャスト・アドレス空間のマッピング	10-15
10-4	ポイント・ツー・ポイント・アドレス空間のマッピング	10-16
10-5	ループバック・アドレス空間のマッピング	10-18
表		
1-1	TruCluster Server アプリケーションのタイプ	1-2
1-2	シングル・インスタンス・アプリケーションのアーキテクチャの違い	1-5
1-3	マルチ・インスタンス・アプリケーションのアーキテクチャの違い	1-7
2-1	アプリケーション・プロファイルの属性	2-4
2-2	ネットワーク・プロファイルの属性	2-10
2-3	テープ・プロファイルの属性	2-11
2-4	メディア・チェンジャの属性	2-13
4-1	アプリケーション移行での検討項目	4-1

5-1	ASE サービスとそれに対応する TruCluster Server の機能 ...	5-2
7-1	CDFS ライブラリ関数	7-4
8-1	クラスタ別名ライブラリ関数	8-2
8-2	クラスタ別名ポート属性間の関係	8-11
9-1	分散ロック・マネージャの関数	9-3
9-2	ロック・モード	9-11
9-3	ロック・モードの共存性	9-13
9-4	dln_unlock 関数の呼び出しでの DLM_DEQALL フラグの使用	9-18
9-5	DLM_QUECVT フラグを指定した場合に許される変換	9-24
9-6	ロック変換によるロック値ブロックへの影響	9-26

まえがき

本書では、HP TruCluster Server の機能を使ってアプリケーションの高可用性を実現する方法と、アプリケーションを TruCluster Server 環境に移行する方法を説明します。

また本書では、TruCluster Server アプリケーション・プログラミング・インタフェース (API) を使って、分散ロック・マネージャ (DLM) や、クラスタ別名、Memory Channel などのクラスタ・テクノロジーを利用する方法を説明します。

実際にアプリケーションを TruCluster Server 環境に移行する前には、TruCluster Server クラスタ・サブシステムを理解するために、『クラスタ概要』をお読みください。

対象読者

本書は、高可用性アプリケーションを TruCluster Server 上で稼働させたり、アプリケーションを TruCluster Available Server や TruCluster Production Server から TruCluster Server 環境に移行したりするシステム管理者を対象としています。また、DLM の同期サービス、クラスタの別名機能、または Memory Channel の高性能な機能を必要とする分散型アプリケーションを作成する開発者を対象としています。

新しい機能と変更された機能

本書でバージョン 5.1A のリリースから変更された点は次のとおりです。

- 2.3.3 項では、CAA (Cluster Application Availability) が処理スクリプトに渡す環境変数と、そのスクリプトからプロファイル属性にアクセスする方法と、処理スクリプトが呼び出される理由を調べる方法について説明しています。
- 2.3.4 項では、CAA の処理スクリプトから出力をリダイレクトする方法について説明しています。
- 2.4 節では、カスタム属性を定義してアプリケーション・リソースのプロファイルを拡張する方法について説明しています。

- 2.14 節では、CAA を使用する新しいアプリケーション例を組み込むために更新しました。
- 第 3 章では、クラスタの別名を使用するアプリケーション例を組み込むために更新しました。

本書の構成

本書の構成は以下のとおりです。

Part 1	シングル・インスタンス・アプリケーションまたはマルチ・インスタンス・アプリケーションを TruCluster Server 上で起動して稼働させる方法を説明します。
第 1 章	一般的なクラスタ・アプリケーションのタイプについて説明します。
第 2 章	TruCluster Server 上のシングル・インスタンス・アプリケーションの可用性を高めるために CAA (Cluster Application Availability) 機能を使う方法を説明します。
第 3 章	TruCluster Server 上のマルチ・インスタンス・アプリケーションの可用性を高めるために省略時のクラスタ別名を使う方法を説明します。
Part 2	アプリケーションを TruCluster Server 環境に移行する方法を説明します。
第 4 章	アプリケーションを TruCluster Server に移行する前に考慮すべき、一般的な検討項目について説明します。
第 5 章	ASE (Available Server Environment) スタイルのアプリケーションを TruCluster Server に移行する方法について説明します。
第 6 章	分散型アプリケーションを TruCluster Server に移行する方法について説明します。
Part 3	API を使って TruCluster Server の技術を利用する方法について説明します。
第 7 章	アプリケーションのソース・コードを変更してクラスタ環境で動作するようにする方法を説明します。
第 8 章	クラスタ別名 API の機能を使う方法を説明します。
第 9 章	DLM の機能を使う方法を説明します。
第 10 章	Memory Channel API ライブラリを使う方法を説明します。

関連資料

クラスタの構成，インストール，および管理作業については，次に示す TruCluster Server の各種マニュアルを参考にしてください。

- TruCluster Server 『*QuickSpecs*』 — TruCluster Server バージョン 5.1B について説明するとともに，製品の機能およびサポートするハードウェアに関する情報が記載されています。『*QuickSpecs*』の最新版は，次の URL から入手することができます。
http://tru64unix.compaq.com/docs/pub_page/spds.html
- TruCluster Server 『クラスタ概要』 — TruCluster Server のテクノロジーについて概要を説明しています。
- TruCluster Server 『クラスタ・リリース・ノート』 — 新機能，既知の問題，および回避方法を含めた，TruCluster Server バージョン 5.1B に関する重要な情報を記載しています。
- TruCluster Server 『クラスタ・ハードウェア構成ガイド』 — クラスタのメンバにするプロセッサのセットアップ方法と，クラスタの共用ストレージの構成方法について説明しています。
- TruCluster Server 『クラスタ・インストール・ガイド』 — TruCluster Server ソフトウェア製品のインストール方法について説明しています。
- TruCluster Server 『クラスタ管理ガイド』 — クラスタ特有の管理作業について説明しています。

TruCluster Server の最新ドキュメントは，次の URL から入手することができます。

http://www.tru64unix.compaq.com/docs/pub_page/cluster_list.html

これに加えて，弊社の HP Tru64™ UNIX オペレーティング・システム・ソフトウェアのドキュメント・セットにある次のマニュアルを利用できます。

- Tru64 UNIX 『リリース・ノート』
- Tru64 UNIX 『システム管理ガイド』
- Tru64 UNIX 『ネットワーク管理ガイド：接続編』
- Tru64 UNIX 『ネットワーク管理ガイド：サービス編』
- Tru64 UNIX 『プログラミング・ガイド』

本書の表記法

本書では、次の表記法を使用しています。

#	番号記号は root としてログインした場合のシステム・プロンプトを表します。
% cat	対話式の例における太字(ボールド体)は、ユーザが入力する文字を示します。
<i>file</i>	イタリック体(斜体)は、変数値、プレースホルダ、および関数の引数名を示します。
:	垂直の反復記号は、実際には存在する例の一部が省略されていることを示します。
cat(1)	リファレンス・ページの参照には、該当するセクション番号をカッコ内に示します。たとえば、cat(1) は、cat コマンドについての情報が、リファレンス・ページのセクション 1 に記載されていることを示します。

Part 1

TruCluster Server 上でのアプリケーションの実行



クラスタ・アプリケーション

TruCluster Server 環境を導入すると、最小限の作業で、既存のアプリケーションをクライアントに対して高可用性にすることができます。また、クラスタの性能と高可用性の機能を十分に活用する新しいアプリケーションも稼働させることができます。クラスタ内で実行していることをあまり意識しないで、高可用性アプリケーションを稼働させることができ、アプリケーションは、どのクラスタのメンバからもディスクのデータにアクセスすることができます。

TruCluster Server の CAA (Cluster Application Availability) サブシステムを使用すると、アプリケーションを別のクラスタ・メンバ上で再起動して、メンバとリソースの障害から回復できます。アプリケーションが実行されているクラスタ・メンバに障害が発生するか、または特定の必要なリソースに障害が発生すると、CAA はそのアプリケーションを別のメンバに再配置するか、またはフェイルオーバーします。別のメンバとは、必要なリソースが使用できるか、またはそのメンバ上でリソースの起動が可能なメンバです。

TruCluster Server ではまた、分散型アプリケーションのコンポーネントを並列に実行することができます。これにより、クラスタ特有の同期メカニズムと性能の最適化を利用しながら、高可用性を実現することができます。

TruCluster Server を初めて導入する場合は、TruCluster Server の『クラスタ概要』をまず読んで、TruCluster Server 製品の概要を理解してください。また、アプリケーションの高可用性を実現する TruCluster Server の各種機能については、次の章を参照してください。

- 第2章では、CAA サブシステムを使ってシングル・インスタンス・アプリケーションの高可用性を実現する方法を説明します。
- 第3章では、マルチ・インスタンス・アプリケーションで省略時のクラスタ別名を使って、ネットワーク・クライアントがクラスタを単一システムとして見えるようにする方法を説明します。

TruCluster Available Server または TruCluster Production Server 環境から移行する場合は、Part 2 をお読みください。ASE スタイルのアプリケーションを TruCluster Server に移行する方法について説明しています。

この章では、次のようなクラスタ・アプリケーションについて説明します。各アプリケーションの種類は 1.1 節で詳しく説明します。

- シングル・インスタンス
- マルチ・インスタンス
- 分散型

1.1 アプリケーションのタイプ

クラスタ・アプリケーションは、表 1-1 に示す基本的なタイプに分けることができます。

表 1-1: TruCluster Server アプリケーションのタイプ

タイプ	説明	例
シングル・インスタンス	シングル・インスタンス・アプリケーションは、一度にクラスタの 1 つのメンバ上だけで動作できる。	シングル・インスタンスの DHCP (Dynamic Host Configuration Protocol) サーバ
マルチ・インスタンス	マルチ・インスタンス・アプリケーションは、クラスタの複数のメンバ上で動作できる。また、1 つのクラスタでいくつでもアプリケーションのコピーを実行できる。	マルチ・インスタンスの Apache Web サーバ
分散型	分散型アプリケーションでは、クラスタ・メンバ全体に分散して配置されたモジュールが、独立に連携して動作する。	OPS (Oracle Parallel Server), Oracle 9i RAC (Real Application Cluster)

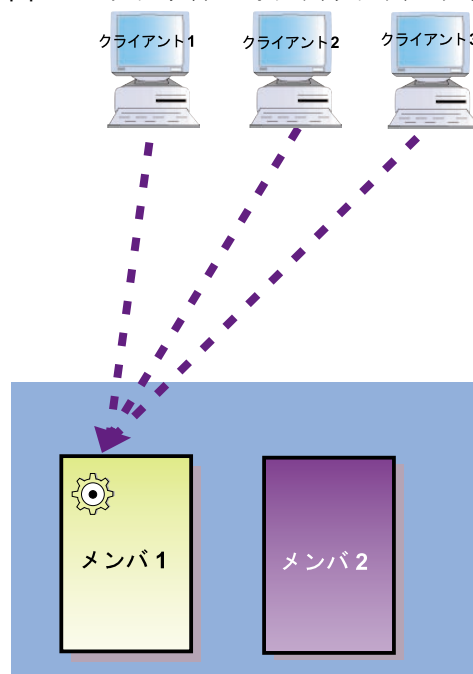
以降の各項では、これらの 3 つのタイプについて詳しく説明します。

1.1.1 シングル・インスタンス・アプリケーション

シングル・インスタンス・アプリケーションは、一度に 1 つのクラスタ・メンバ上でのみ動作します。図 1-1 に示すように、すべてのクライアントが 1 つのメンバ上のシングル・インスタンス・アプリケーションにアクセスします。

1-2 クラスタ・アプリケーション

図 1-1: シングル・インスタンス・アプリケーションへのアクセス

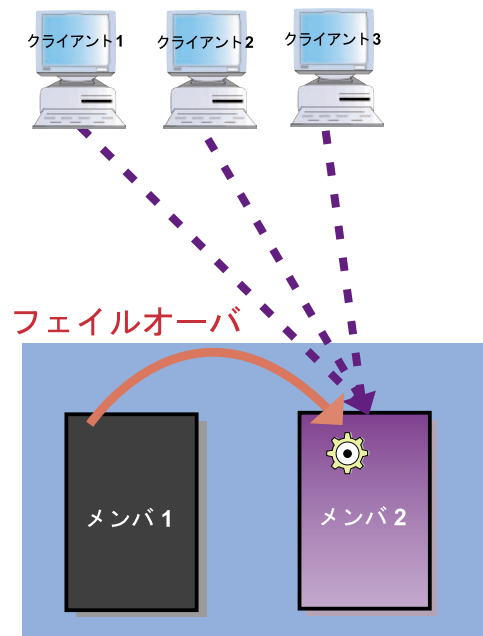


⚙️ シングル・インスタンス・アプリケーション

ZK-1691U-AI

アプリケーションがインストールされているクラスタ・メンバに障害が発生したり，リソースにアクセスできなくなった場合，CAA サブシステムは，動作中の別のメンバへアプリケーションをフェイルオーバーさせることができます。CAA の使用方法についての詳細は，第 2 章を参照してください。図 1-2 に，1 つのクラスタ・メンバに障害が発生したために，アプリケーションが 2 番目のクラスタ・メンバにフェイルオーバーされる方法を示します。

図 1-2: CAA を使用したアプリケーションのフェイルオーバー



⚙️ シングル・インスタンス・アプリケーション

ZK-1692U-AI

表 1-2 では、TruCluster Server と、TruCluster 製品の以前のバージョンとのシングル・インスタンス・アプリケーションのアーキテクチャの違いを説明しています。

1-4 クラスタ・アプリケーション

表 1-2: シングル・インスタンス・アプリケーションのアーキテクチャの違い

TruCluster Server バージョン 5.0 以降	TruCluster Available Server と TruCluster Production Server バージョン 1.6 以前
クラスタ・ファイル・システム (CFS) があるため、クラスタの 1 つのメンバ上にインストールしたアプリケーションとその構成内容を、すべてのメンバ上で見ることができる。ただし、アプリケーションに同期機構が組み込まれていないため、一度に 1 つのメンバでしかアプリケーションを実行できない。	クラスタの 1 つのメンバ上にインストールしたアプリケーションとその構成内容は、そのメンバ上で見たり使用したりすることはできない。
CAA を使って、アプリケーションの起動、停止、およびリソースの依存性の要件を定義する。ストレージは、CFS とデバイス要求ディスパッチャによって透過的に管理される。CAA の処理スクリプト内でクラスタ別名やインタフェース別名を使って、IP 別名の要件を処理することができる。	ASE を使って、アプリケーションの起動、停止、ストレージ、および IP 別名を定義する。

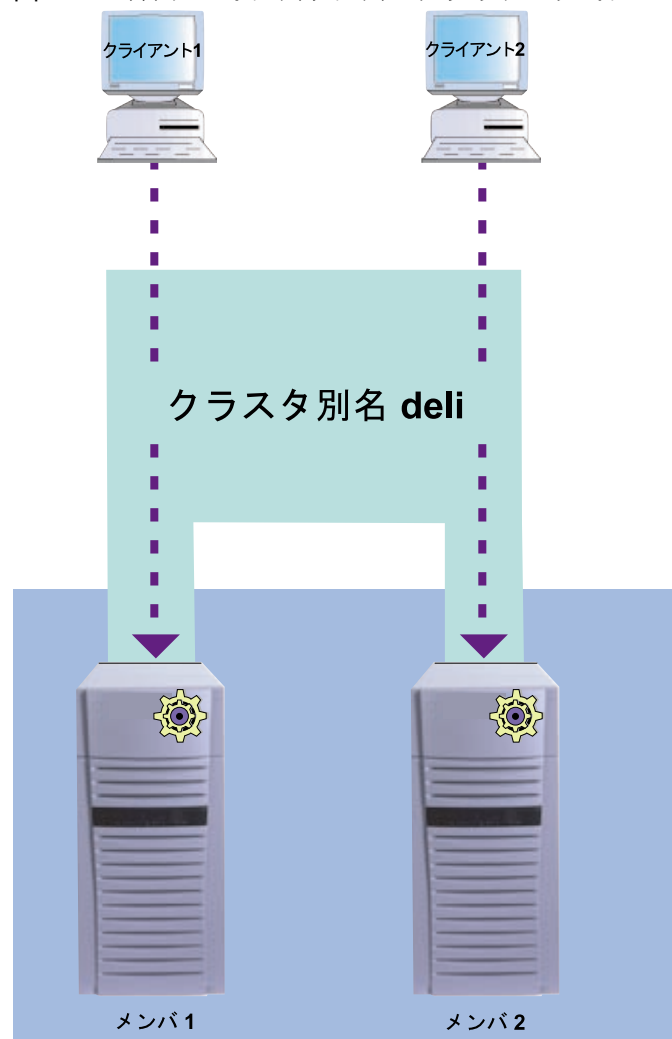
TruCluster Server は、Tru64 UNIX バージョン 5.1B とバイナリ互換性があるため、Tru64 UNIX 上で正常に動作し、新しい形式のデバイス名 (dsk) を認識するアプリケーションは、クラスタ内の少なくとも 1 つのメンバ上で動作します。デバイスの命名規則については、4.2 節を参照してください。


1.1.2 マルチ・インスタンス・アプリケーション

マルチ・インスタンス・アプリケーションは、その複数のインスタンスが 1 つのシステム上で、または、クラスタ内の複数のメンバ上で動作できます。マルチ・インスタンス・アプリケーションを複数のシステム上で実行する場合、アプリケーションはクラスタ対応に修正されています。たとえば、一時ファイル名は、上書きを防ぐために変更されています。

マルチ・インスタンス・アプリケーションは、典型的な高可用性アプリケーションです。1 つのクラスタ・メンバが故障しても、他のメンバ上で動作するそのアプリケーションの別のインスタンスに影響を与えません。図 1-3 に示すように、マルチ・インスタンス・アプリケーションでは、クラスタ別名を利用して、クライアントの要求を他のすべてのクラスタ・メンバに分散させることができます。この図では、クライアントはクラスタ別名 deli を介してマルチ・インスタンス・アプリケーションにアクセスしています。

図 1-3: マルチ・インスタンス・アプリケーションへのアクセス



 マルチ・インスタンス・アプリケーション

ZK-1694U-AI

表 1-3 では、TruCluster Server とそれ以前の TruCluster ソフトウェア製品におけるマルチ・インスタンス・アプリケーションのアーキテクチャの違いを説明しています。

表 1-3: マルチ・インスタンス・アプリケーションのアーキテクチャの違い

TruCluster Server バージョン 5.0 以降	TruCluster Available Server と TruCluster Production Server バージョン 1.6 以前
クラスタ対応のマルチ・インスタンス・アプリケーションでは、ソース・レベルで明示的に分散ロック・マネージャ (DLM) のアプリケーション・プログラミング・インタフェース (API) を使用する。クラスタ別名により、アプリケーションのインスタンス内でのクライアント要求のルーティングが自動化される。	アプリケーションは、固有のインタフェース別名を持つ複数の ASE サービスを作成することにより、マルチ・インスタンスとして実行できる。クラスタ別名はない。負荷分散とクライアント要求の分散は、アプリケーション内に設計しておかなければならない。アプリケーションでは、DRD (Distributed Raw Disk) デバイスをストレージとして使用できる必要があり、UNIX ファイル・ロック・セマンティクス、DLM またはアプリケーション固有のロック・メカニズムを使用する必要がある。
クラスタ対応でないマルチ・インスタンス・アプリケーションでは、標準 UNIX ファイル・ロックまたは DLM を使って、共用データへのアクセスの同期をとる必要がある。	標準 UNIX ファイル・ロックはクラスタ単位で動作するように実装されていないため、クラスタ対応でないアプリケーションでは、DLM を明示的に使用する必要がある。

クラスタ内で複数のアプリケーション・インスタンスを実行できるようにするには、次のことを行わなければなりません。

- プロセス間通信は、リモート・プロシージャ・コール (RPC) またはソケット接続を介して行う。
- 共用ファイルや共用データのアクセスでは、同期をとる必要がある。
flock() システム・コールまたは分散ロック・マネージャ (DLM) を使って、共用データへのアクセスの同期をとります。
- コンテキスト依存シンボリック・リンク (CDSL) または他の方法を使って、アプリケーションのインスタンスごとにログ・ファイルと一時ファイルを作成する。
- アプリケーションの構成に対してメンバごとに固有な要件がある場合は、そのアプリケーションの動作する各メンバにログオンしてアプリケーションを構成する必要がある。

たとえば、アプリケーションのインストール・スクリプトがクラスタ対応でない (つまり、アプリケーションが、クラスタ内にインストールされていることを意識していない) 場合、アプリケーションをインストー

ルした後に、構成を調整する必要があります。詳細は、アプリケーションの構成を説明したドキュメントを参照してください。

- 省略時のクラスタ別名を使ってネットワーク接続要求を他のメンバに分散させるためには、`/etc/clua_services` ファイルに各アプリケーションのポートを定義する必要がある。

`in_multi` 別名属性を付け加え、クラスタ別名サブシステムが、省略時のクラスタ別名宛の接続要求を、その別名のすべてのメンバに分散させるようにします。ポートの定義についての詳細は、`clua_services(4)` を参照し、マルチ・インスタンス・アプリケーションにクライアントを透過的にアクセスさせるために、クラスタ別名を使用する方法については、第 3 章を参照してください。

上記の内容の詳細説明と、アプリケーションの移行に関する一般的な説明については、第 4 章を参照してください。

1.1.3 分散型アプリケーション

分散型アプリケーションは、クラスタ対応のアプリケーションです。つまり、クラスタ上で動作していることを認識し、通常、分散実行により性能の向上を計っています。分散型アプリケーションは、DLM や Memory Channel など、クラスタのアプリケーション・プログラミング・インタフェース (API) を使用します。分散型アプリケーションについての詳細は、第 6 章を参照してください。

2

シングル・インスタンス・アプリケーションの高可用性実現のための CAA の使用

CAA (Cluster Application Availability) サブシステムは、クラスタ内のメンバーリソース (ネットワーク、アプリケーション、テープ装置、およびメディア・チェンジャなど) の状態を維持管理します。CAA は、クラスタ内のアプリケーション・リソースが必要とするリソースを監視し、メンバ上で動作しているアプリケーションが要求を満たせるようにします。

この章では、次のことについて説明します。

- いつ CAA を使用するか (2.1 節)
- リソース・プロファイルの作成 (2.2 節)
- 処理スクリプトの作成 (2.3 節)
- ユーザ定義属性の作成 (2.4 節)
- リソースの登録 (2.5 節)
- アプリケーション・リソースの起動 (2.6 節)
- アプリケーション・リソースの再配置 (2.7 節)
- アプリケーション・リソースの分散 (2.8 節)
- アプリケーション・リソースの停止 (2.9 節)
- アプリケーション・リソースの登録抹消 (2.10 節)
- CAA 状態情報の表示 (2.11 節)
- グラフィカル・ユーザ・インタフェースを使った CAA の管理 (2.12 節)
- CAA の学習 — チュートリアル (2.13 節)
- 高可用アプリケーションの作成 — 例 (2.14 節)

2.1 いつ CAA を使用するか

CAA は、一度に 1 つのクラスタ・メンバだけで動作するアプリケーションで使用するよう設計されています。アプリケーションを実行しているクラスタ・メンバに障害が発生したか、特定の必須リソースにアクセスできない場合、CAA はアプリケーションを別のメンバに再配置、つまりフェイルオーバーします。その別のメンバとは、要求されたリソースを持つメンバか、要求されたリソースを起動できるメンバのどちらかです。

マルチ・インスタンス・アプリケーションでは、アプリケーションを透過的にフェイルオーバーするためには、クラスタ別名を使用する方が便利です。マルチ・インスタンス・アプリケーションは、一般的には、第 3 章で説明するように、クラスタ別名を使用することにより、クライアントに対する高可用性を実現します。ただし、CAA では、アプリケーションの簡略化した集中管理 (起動と停止) と障害の発生したインスタンスの再起動が可能になるため、CAA はマルチ・インスタンス・アプリケーションに有用です。CAA を使用すると、rc3 スクリプトを追加で作成しなくても、ブート時とシャットダウン時に、アプリケーションの自動起動およびシャットダウンを行うことができます。

クラスタ別名サブシステムと CAA の違いについては、TruCluster Server の『クラスタ管理ガイド』を参照してください。また、マルチ・インスタンス・アプリケーションで、省略時のクラスタ別名を使って高可用性を実現する例を第 3 章で示します。

2.2 リソース・プロファイル

リソース・プロファイルには、CAA がリソースを開始して、管理、監視する方法を示す属性があります。プロファイルでは、リソースの依存関係を定義し、依存するリソースにアクセスできなくなったときに、アプリケーションをどうするかを決めます。

リソース・プロファイルには、`application`、`network`、`tape`、および `changer` の 4 種類があります。各リソースには、リソースの属性が定義された独自のリソース・プロファイルがあります。以下の各項の例と表で、それぞれのリソース・プロファイルのタイプとそのプロファイルで利用できる属性について説明します。定義できるプロファイルの詳細な属性リストを含むリソースのタイプごとの定義についての詳細は、2.2.2 項、2.2.3 項、2.2.4 項、および 2.2.5 項を参照してください。

リソース・プロファイルで指定する属性の中には、次のものがあります。

- アプリケーションに必須のリソース (REQUIRED_RESOURCES)。CAA は、必須のリソースが利用できなくなったときにアプリケーションを再配置または停止します。
- アプリケーションを起動または再起動するメンバを選択するルール (PLACEMENT)。
- アプリケーションの起動時またはフェイルオーバー時に選択できるメンバのリスト (HOSTING_MEMBERS)。このリストは、配置ポリシー (PLACEMENT) が *avored* または *restricted* のときに使用され、優先順に並んでいます。

すべてのリソース・プロファイルはクラスタ単位のディレクトリ `/var/cluster/caa/profile` に置かれます。リソース・プロファイルのファイル名は、`resource_name.cap` のような形式にします。CAA コマンドでは、リソース名 (`resource_name`) だけを用いて各リソースを指定します。

プロファイルの各タイプには、必須の属性とオプションの属性があります。オプションのプロファイル属性は、プロファイル内で指定しなくてもかまいません。オプションのプロファイル属性で省略時の値を持つものは、登録時に、そのタイプのテンプレートと共通のテンプレートに保存されている値とマージされます。各リソース・タイプには、`TYPE_resource_type.cap` という名前で `/var/cluster/caa/template` に格納されているテンプレート・ファイルがあり、属性の省略時の値が設定されています。すべてのタイプのリソースで使用する値のための共通のテンプレート・ファイルは、`/var/cluster/caa/template/TYPE_generic.cap` に格納されています。

以降の各項の例に、リソース・プロファイルの構文の例を示します。シャープ記号 (#) で始まる行はコメント行であり、リソース・プロファイルとしては処理されません。行末のバックスラッシュ (\) は、次の行に続くことを示します。プロファイルの構文のさらに詳しい説明は、`caa(4)` を参照してください。

2.2.1 リソース・プロファイルの作成

アプリケーションの高可用性を実現する最初のステップは、リソース・プロファイルを作成することです。次のいずれかの方法で、リソース・プロファイルを作成することができます。

- `caa_profile` コマンドを使用する。
- `SysMan (/usr/sbin/sysman caa)` にアクセスする。この方法では、スケジューリングされたアプリケーションの再分散やフェイルバックの設定はサポートされない。
- 既存のリソース・プロファイルを `/var/cluster/caa/profile` にコピーし、そのコピーを `emacs`、`vi`、またはその他のテキスト・エディタで編集する。

これらの方法を組み合わせて使用することもできます。たとえば、`caa_profile` コマンドを使用してリソース・プロファイルを作成したのち、テキスト・エディタを使用して手作業でそのプロファイルを編集することができます。

プロファイルの例がいくつか `/var/cluster/caa/examples` ディレクトリにあります。

リソース・プロファイルを作成したら、それを CAA に登録する必要があります。登録すると、リソースを管理、監視することができます。アプリケーションの登録方法については、2.5 節を参照してください。

2.2.2 アプリケーション・リソース・プロファイル

表 2-1 に、アプリケーション・プロファイルの各種属性を示します。それぞれの属性について、その属性が必須かどうかと省略時の値を示し、その属性について説明します。

表 2-1: アプリケーション・プロファイルの属性

属性	必須	省略時の値	説明
TYPE	はい	なし	リソースのタイプ。アプリケーション・リソースでは、 <code>application</code> タイプを使用。
NAME	はい	なし	リソースの名前。英字 <code>a~z</code> 、 <code>A~Z</code> 、数字 <code>0~9</code> 、下線 (<code>_</code>)、またはピリオド (<code>.</code>) からなる文字列。ピリオドで始まるリソース名は使用できない。
DESCRIPTION	いいえ	リソースの名前	リソースの説明

表 2-1: アプリケーション・プロファイルの属性 (続き)

属性	必須	省略時の値	説明
FAILURE_THRESHOLD	いいえ	0	CAA が FAILURE_INTERVAL の間に検出する失敗の数。この値を超えると、CAA はリソースを利用不可としてマークし監視しなくなる。アプリケーションのチェック・スクリプトがこの回数のために失敗すると、アプリケーション・リソースは停止されてオフラインになる。値がゼロ (0) の場合は、失敗の追跡が無効になる。最大値は 20 である。
FAILURE_INTERVAL	いいえ	0	CAA が失敗のしきい値を適用する期間。秒単位で指定。値がゼロ (0) の場合は、失敗の追跡が無効になる。
REQUIRED_RESOURCES	いいえ	なし	このリソースが依存するリソース名の順序付けされたリスト。空白で区切られている。このプロファイルで必須リソースとして使うそれぞれのリソースは、CAA に登録しておかなければならない。登録しておかなければ、プロファイルの登録に失敗する。詳細は、2.2.2.1 項を参照。
OPTIONAL_RESOURCES	いいえ	なし	このリソースの配置の決定時に使用する、順序付けされたオプション・リソースのリスト。空白で区切られている。オプション・リソースは 58 までリストできる。詳細は、2.2.2.3 項を参照。
PLACEMENT	いいえ	balanced	配置ポリシー (balanced, favored, または restricted) は、CAA がリソースを起動するクラスタ・メンバを選択する方法を指定する。
HOSTING_MEMBERS	ときどき	なし	リソースを提供できるクラスタ・メンバのリスト。順番に、空白で区切られてリストされている。この属性は、PLACEMENT が favored または restricted の場合にのみ必須で、balanced の場合には空にしなければならない。
RESTART_ATTEMPTS	いいえ	1	CAA が、1 つのクラスタ・メンバ上でリソースを再起動しようと試みる回数。CAA は、この試行回数後、アプリケーションを再配置しようとする。値が 1 の場合、CAA は 1 つのメンバ上で 1 回だけアプリケーションの再起動を試みる。2 回目の失敗で、アプリケーションを再配置しようとする。

表 2-1: アプリケーション・プロファイルの属性 (続き)

属性	必須	省略時の値	説明
FAILOVER_DELAY	いいえ	0	CAA が、リソースを再起動しようとする前に、またはフェイルオーバーを行おうとする前に待つ時間。秒単位で指定。
AUTO_START	いいえ	0	クラスタのリブートの前にリソースが動作していたかどうかにかかわらず、リブート後に CAA が自動的にリソースを起動するかどうかを示すフラグ。0 の場合、CAA は、アプリケーション・リソースがリブート前に動作していた場合だけこのアプリケーションを起動する。1 の場合、CAA は、リブートの後に無条件にアプリケーションを起動する。
ACTION_SCRIPT	はい	なし	リソースを起動、停止、およびチェックする、リソース固有のスクリプト。処理スクリプト・ファイルのフル・パスを指定する。指定しない場合は、 <code>/var/cluster/caa/script</code> とみなされる。開始点として、この省略時のパスで相対パスを指定してもよい。
ACTIVE_PLACEMENT	いいえ	0	1 に設定すると、CAA は、クラスタ・メンバが追加または再起動されるたびにアプリケーションの配置を再評価する。
SCRIPT_TIMEOUT	いいえ	60	処理スクリプトの実行が完了するまでにかかる時間の最大値。この時間以内に実行が完了しなければ、エラーが返される。秒単位で指定。
CHECK_INTERVAL	いいえ	60	リソースの処理スクリプトのチェック・エントリ・ポイントを繰り返し実行する時間間隔。秒単位で指定。
REBALANCE	いいえ	なし	最適に配置するためにアプリケーションが自動的に再評価される時間。フィールドには、 <code>t:day:hour:min</code> の形式で指定する必要がある。このフィールドには、再評価が発生する日時、 <code>day</code> には曜日 (0 ~ 6)、 <code>hour</code> には時間 (0 ~ 23)、 <code>min</code> には分 (0 ~ 59) をそれぞれ指定する。毎日または毎時を指定する場合には、ワイルドカードとしてアスタリスク (*) を使用できる。

`caa_profile` を使用して、CAA でアプリケーション・リソースを作成する例を次に示します。

```
# /usr/sbin/caa_profile -create clock -t application -B /usr/bin/X11/xclock \
```

2-6 シングル・インスタンス・アプリケーションの高可用性実現のための CAA の使用


```
-d "Clock Application" -r network1 -l application2 \  
-a clock.scr -o ci=5,ft=2,fi=12,ra=2,bt=*:12:00
```

前述の例で作成されたリソース・プロファイル・ファイルの内容を次に示します。

```
NAME=clock  
TYPE=application  
ACTION_SCRIPT=clock.scr  
ACTIVE_PLACEMENT=0  
AUTO_START=0  
CHECK_INTERVAL=5  
DESCRIPTION=Clock Application  
FAILOVER_DELAY=0  
FAILURE_INTERVAL=12  
FAILURE_THRESHOLD=2  
REBALANCE=t:*:12:00  
HOSTING_MEMBERS=  
OPTIONAL_RESOURCES=application2  
PLACEMENT=balanced  
REQUIRED_RESOURCES=network1  
RESTART_ATTEMPTS=2  
SCRIPT_TIMEOUT=60
```

アプリケーション・リソース・プロファイルの構文の詳細は、
caa_profile(8) および caa(4) を参照してください。

2.2.2.1 必須リソース

CAA は、アプリケーション・リソースを提供できるクラスタ・メンバがどれかを判断するために、必須リソース・リストを、配置ポリシーやホスト・メンバ・リストとともに使用します。必須リソースは、アプリケーションが動作しているかまたは起動されたメンバ上で ONLINE 状態でなければなりません。アプリケーション・リソースのみが必須リソースを持つことができ、どのタイプのリソースでも、必須リソース・リストに定義することができます。

ホスト・メンバ上の必須リソースに障害が発生すると、CAA は、アプリケーションのフェイルオーバを開始するか、RESTART_ATTEMPTS が 0 でない場合は現在のメンバで再起動を試みます。これにより CAA は、アプリケーション・リソースを、必須リソースが使える他のメンバへフェイルオーバさせます。適切なメンバがない場合は、アプリケーションを停止させます。後者の場合、CAA は引き続き必須リソースを監視し、適切なクラスタ・メンバ上でリソースが再び利用可能になるとアプリケーションを再起動します。

必須リソースのリストは、コマンド `caa_start`、`caa_stop`、`caa_relocate` を、強制 (-f) オプションを指定して実行する場合に、相互に依存しているアプリケーション・リソースのグループを、起動、停止、再配置するのにも利用できます。

2.2.2.2 アプリケーション・リソース配置ポリシー

配置ポリシーでは、CAA がリソースを起動したり、障害の後にリソースを再配置したりするクラスタ・メンバを選択する方法を指定します。

注意

そのアプリケーションに関して配置を決定する際にはいつも、すべての必須リソース (アプリケーション・リソースのプロファイルにリストされている) が利用できるクラスタ・メンバだけが候補として考慮されます。

次の配置ポリシーがサポートされています。

- **balanced** — CAA は、現在動作しているアプリケーション・リソースが最も少ないメンバ上でのアプリケーション・リソースの起動、または再起動を優先します。オプション・リソース・リストによる配置をまず考えます。2.2.2.3 項を参照してください。次に、実行しているアプリケーション・リソースが最も少ないホストが選択されます。この基準に従っても優先するクラスタ・メンバがない場合には、メンバをランダムに選択します。
- **favored** — CAA は、リソース・プロファイルの `HOSTING_MEMBERS` 属性のメンバのリストを参照します。このリストにあり、必須リソースを満足するクラスタ・メンバだけが配置の候補として考慮されます。オプション・リソース・リストによる配置をまず考えます。2.2.2.3 項を参照してください。オプション・リソースに基づいてメンバが選択できない場合は、ホスト・メンバの順番により、アプリケーション・リソースを実行するメンバが決定されます。ホスト・メンバ・リストにあるメンバのいずれも利用できない場合、CAA は利用できる任意のメンバ上にアプリケーション・リソースを配置します。このメンバは、`HOSTING_MEMBERS` リストに含まれているときも、含まれていないときもあります。
- **restricted** — **favored** とほとんど同じですが、ホスト・リストのどのメンバも利用できない場合、CAA は、アプリケーション・リソースの起動や再起動を行いません。**restricted** 配置ポリシーを使用することによって、リストにないメンバ上でリソースが実行されないようにできます。そのメンバに、手動で再配置しても同様です。

配置ポリシーの `avored` または `restricted` を使用するためには、`HOSTING_MEMBERS` 属性で、ホスト・メンバを指定する必要があります。配置ポリシーの `balanced` を使用する場合には、`HOSTING_MEMBERS` にホスト・メンバを指定してはなりません。指定した場合、リソースは無効になり、登録できません

`ACTIVE_PLACEMENT` を 1 に設定した場合、クラスタ・メンバがクラスタに追加されるか再起動されるたびにアプリケーションの配置が再評価されます。このようにすると、メンバが障害から復旧したときに、クラスタ内の適切なメンバへアプリケーションを再配置することができます。

クラスタ・メンバがクラスタに再参加するとき以外に、アプリケーションを適切なメンバに再配置するには、`REBALANCE` 属性を使用して、配置が再評価される時刻を指定します。

2.2.2.3 配置決定の際のオプション・リソース

CAA はオプション・リソース・リストを使用して、各ホスト・メンバ上での `ONLINE` 状態のオプション・リソースの数に基づいてホスト・メンバを選択します。どのメンバの `ONLINE` 状態のオプション・リソースの数も同じ場合、CAA は、次のようにしてオプション・リソースの順番を決めます。

CAA は、リソース・リストの最初から順に、各メンバのオプション・リソースの状態を比較します。リスト内のリソースそれぞれに対して、あるメンバでそのリソースが `ONLINE` の場合、`ONLINE` のリソースを持たないメンバは対象から外されます。リストの各リソースは、リソースを使用するメンバが 1 つになるまでこの方法で評価されます。オプション・リソースの最大個数は 58 です。

このアルゴリズムで複数の優先メンバが選択された場合、その配置ポリシーに従って選択された、これらのメンバのうちの 1 つにアプリケーションが配置されます。

2.2.3 ネットワーク・リソース・プロファイル

表 2-2 では、ネットワーク・プロファイルの属性について説明します。各属性について、必須かどうかと省略時の値、および説明を示します。

表 2-2: ネットワーク・プロファイルの属性

属性	必須	省略時の値	説明
TYPE	はい	なし	リソースのタイプ。ネットワーク・リソースのタイプは <code>network</code> 。
NAME	はい	なし	リソースの名前。英字 <code>a ~ z</code> , <code>A ~ Z</code> , 数字 <code>0 ~ 9</code> , 下線 (<code>_</code>), またはピリオド (<code>.</code>) からなる文字列で指定。ピリオドで始まるリソース名は使用できない。
DESCRIPTION	いいえ	なし	リソースの説明。
SUBNET	はい	なし	<code>nnn.nnn.nnn.nnn</code> 形式のネットワーク・リソースのサブネット・アドレス (たとえば, <code>16.140.112.0</code>)。SUBNET 値は, IP アドレスとネットマスクの各ビットの論理積 (AND) をとった値。IP アドレスが <code>16.69.225.12</code> で, ネットマスクが <code>255.255.255.0</code> の場合, サブネット値は <code>16.69.225.0</code> となる。
FAILURE_THRESHOLD	いいえ	0	CAA がリソースを使用不可とマークして監視しなくなる直前の, FAILURE_INTERVAL 間で検出された失敗回数。アプリケーションのチェック・スクリプトがこの回数失敗すると, アプリケーション・リソースは停止しオフラインに設定される。値がゼロ (0) の場合は, 失敗の追跡が無効になる。最大値は 20 である。
FAILURE_INTERVAL	いいえ	0	CAA が障害しきい値と比較する時間間隔 (秒単位)。値がゼロ (0) の場合は, 失敗の追跡が無効になる。

ネットワーク・リソース・プロファイルを作成するコマンドの例を次に示します。

```
# /usr/sbin/caa_profile -create network1 -t network -s "16.69.244.0" \
-d "Network1"
```

前述のコマンドで作成されたファイル `/var/cluster/caa/profile/network1.cap` のプロファイルの内容を次に示します。

```
NAME=network1
TYPE=network
DESCRIPTION=Network1
FAILURE_INTERVAL=0
FAILURE_THRESHOLD=0
SUBNET=16.69.244.0
```

ネットワーク・リソース・プロファイルの構文の詳細は, `caa_profile(8)` および `caa(4)` を参照してください。

2-10 シングル・インスタンス・アプリケーションの高可用性実現のための CAA の使用

ルーティングによって、クラスタ内のすべてのメンバは、どのメンバに接続されたネットワークにも間接的にアクセスできますが、アプリケーションによっては、ネットワークに直接接続されたメンバで動作した場合に得られる高い性能を必要とするものがあります。このため、アプリケーション・リソースでは、ネットワーク・リソースへの依存性(オプションまたは必須)を定義します。CAA は、そのアプリケーション・リソースの配置をネットワーク・リソースの位置に基づいて最適化します。

ネットワーク・リソースを、アプリケーションのオプション・リソース (OPTIONAL_RESOURCES) として定義した場合、そのアプリケーションは、必須リソース、配置ポリシ、およびクラスタの状態に基づいて、サブネットに直接接続されたメンバ上で起動されます。ネットワーク・アダプタが故障しても、そのアプリケーションはルーティングによって、リモートからそのサブネットにアクセスすることができます。

ネットワーク・リソースを必須リソース (REQUIRED_RESOURCES) として定義し、ネットワーク・アダプタが故障した場合、CAA は、アプリケーションを再配置するか、または停止します。すべての適切なホスト・メンバとのネットワーク接続に失敗した場合、CAA は、アプリケーションを停止します。

2.2.4 テープ・リソース・プロファイル

表 2-3 で、テープ・プロファイルの属性について説明します。各属性について、必須かどうかと省略時の値、および説明を示します。

表 2-3: テープ・プロファイルの属性

属性	必須	省略時の値	説明
TYPE	はい	なし	リソースのタイプ。テープ・リソースのタイプは <code>tape</code> 。
NAME	はい	なし	リソースの名前。英字 <code>a ~ z</code> , <code>A ~ Z</code> , 数字 <code>0 ~ 9</code> , 下線 (<code>_</code>), またはピリオド (<code>.</code>) からなる文字列で指定。ピリオドで始まるリソース名は使用できない。
DESCRIPTION	いいえ	なし	リソースの説明。
DEVICE_NAME	はい	なし	テープ・リソースのデバイス名。デバイス・スペシャル・ファイルのフル・パスを使用する (たとえば, <code>/dev/tape/tape1</code>)。

表 2-3: テープ・プロファイルの属性 (続き)

属性	必須	省略時の値	説明
FAILURE_THRESHOLD	いいえ	0	CAA がリソースを使用不可とマークして監視しなくなる直前の、FAILURE_INTERVAL 間で検出された失敗回数。アプリケーションのチェック・スクリプトがこの回数失敗すると、アプリケーション・リソースは停止しオフラインに設定される。値がゼロ (0) の場合は、失敗の追跡が無効になる。最大値は 20 である。
FAILURE_INTERVAL	いいえ	0	CAA 障害しきい値と比較する時間間隔 (秒単位)。値がゼロ (0) の場合は、失敗の追跡が無効になる。

デバイス要求ディスパッチャにより、すべてのクラスタ・メンバは、どのメンバに接続されたテープ装置にも、間接的にアクセスできますが、アプリケーションによっては、テープ装置を直接接続したクラスタ・メンバ上で動作した場合に得られる高い性能を必要とするものがあります。このため、アプリケーション・リソースでは、テープ・リソースへの依存性 (オプションまたは必須) を定義できます。CAA は、そのアプリケーション・リソースの配置を、テープ・リソースの位置に基づいて最適化します。

テープ・リソース・プロファイルを作成する例を次に示します。リソース・プロファイルにテープ・リソースを定義すると、アプリケーション・リソース・プロファイルは、それを必須リソースまたはオプション・リソースとして指定することができます。

```
# /usr/sbin/caa_profile -create tape1 -t tape -n /dev/tape/tape1 -d "Tape Drive"
```

前述のコマンドで作成されたファイル `/var/cluster/caa/profile/tape1.cap` のプロファイルの内容を次に示します。

```
NAME=tape1
TYPE=tape
DESCRIPTION=Tape Drive
DEVICE_NAME=/dev/tape/tape1
FAILURE_INTERVAL=0
FAILURE_THRESHOLD=0
```

2.2.5 メディア・チェンジャ・リソース・プロファイル

メディア・チェンジャ装置はテープ装置に似ていますが、複数のテープ・カートリッジにアクセスできます。

表 2-4 では、メディア・チェンジャ・プロファイルの属性について説明します。各属性について、必須かどうかと省略時の値、および説明を示します。

表 2-4: メディア・チェンジャの属性

属性	必須	省略時の値	説明
TYPE	はい	なし	リソースのタイプ。メディア・チェンジャ・リソースのタイプは <code>changer</code> 。
NAME	はい	なし	リソースの名前。英字 <code>a ~ z</code> , <code>A ~ Z</code> , 数字 <code>0 ~ 9</code> , 下線 (<code>_</code>), またはピリオド (<code>.</code>) からなる文字列で指定。ピリオドで始まるリソース名は使用できない。
DESCRIPTION	いいえ	なし	リソースの説明。
DEVICE_NAME	はい	なし	メディア・チェンジャ・リソースのデバイス名。デバイス・スペシャル・ファイルのフル・パスを使用する (たとえば, <code>/dev/changer/mcl</code>)。
FAILURE_THRESHOLD	いいえ	0	CAA がリソースを使用不可とマークして監視しなくなる直前の, <code>FAILURE_INTERVAL</code> 間で検出された失敗回数。アプリケーションのチェック・スクリプトがこの回数失敗すると, アプリケーション・リソースは停止しオフラインに設定される。値がゼロ (0) の場合は, 失敗の追跡が無効になる。最大値は 20 である。
FAILURE_INTERVAL	いいえ	0	CAA が障害しきい値と比較する時間間隔 (秒単位)。値がゼロ (0) の場合は, 失敗の追跡が無効になる。

デバイス要求ディスパッチャにより、すべてのクラスタ・メンバは、どのメンバに接続されたメディア・チェンジャにも、間接的にアクセスできますが、アプリケーションによっては、メディア・チェンジャに直接接続されたメンバ上で動作したときに得られる高い性能を必要とするものがあります。このため、アプリケーション・リソースでは、メディア・チェンジャ・リソースへの依存性 (オプションまたは必須) を定義できます。CAA は、そのアプリケーション・リソースの配置を、メディア・チェンジャ・リソースの位置に基づいて最適化します。

メディア・チェンジャ・リソース・プロファイルを作成する例を次に示します。メディア・チェンジャ・リソースをリソース・プロファイルで定義す

ると、アプリケーション・リソース・プロファイルでは、それを依存対象として指定することができます。

```
# /usr/sbin/caa_profile -create mchanger1 -t changer -n /dev/changer/mc1 \
-d "Media Changer Drive"
```

前述のコマンドで作成されたファイル `/var/cluster/caa/profile/mchanger1.cap` のプロファイルの内容を次に示します。

```
NAME=mchanger1
TYPE=changer
DESCRIPTION=Media Changer Drive
DEVICE_NAME=/dev/changer/mc1
FAILURE_INTERVAL=0
FAILURE_THRESHOLD=0
```

2.2.6 プロファイルの検証

リソース・プロファイルを登録する前に、構文が正しいかどうかを検証できます。プロファイルが検証に合格しない場合には、登録できません。次のように `caa_profile` コマンドを使用して、プロファイルが正しく作成されたことを確認できます。

```
# /usr/sbin/caa_profile -validate resource
```

リソースに問題があれば、属性が正しくないというメッセージが適切に表示されます。

2.3 処理スクリプトの作成

CAA が管理、監視するアプリケーションを起動、停止、再配置するために、アプリケーション・リソースには処理スクリプトが必要です。

処理スクリプトを使って、次のことを指定することができます。

- アプリケーションの開始方法

CAA は、アプリケーション・リソースを起動または再起動するために、`start` エントリ・ポイントで処理スクリプトを呼び出します。この開始エントリ・ポイントでは、アプリケーションを起動するのに必要なすべてのコマンドを実行し、成功したときには 0 を返し、失敗したときには 0 以外の値を返さなければなりません。

- アプリケーションの停止方法と、アプリケーションのフェイルオーバを発生させる前に行うクリーンアップ処理

CAA は、動作中のアプリケーション・リソースを停止するために、`stop` エントリ・ポイントで処理スクリプトを呼び出します。UNKNOWN

状態のアプリケーション・リソースを停止するときには、呼び出しません (詳細は `caa_stop(8)` を参照)。停止エントリ・ポイントでは、アプリケーションを停止するのに必要なすべてのコマンドを実行し、成功したときには 0 を返し、失敗したときには 0 以外の値を返さなければなりません。停止スクリプトでは、停止するアプリケーションがないと判断したときには 0 を返す必要があります。

- アプリケーションが動作しているかどうかを判断する方法

CAA は、アプリケーションが動作しているかどうかを確認するために、処理スクリプトの `check` エントリ・ポイントを呼び出します。チェック・エントリ・ポイントは、`CHECK_INTERVAL` 秒間隔で繰り返し実行されます。このエントリでは、成功したときには 0 を返し、失敗したときには 0 以外の値を返さなければなりません。

省略時の設定では、処理スクリプトは、クラスタ単位のディレクトリ `/var/cluster/caa/script` に置かれます。処理スクリプトのファイル名は、`name.scr` の形式です。

処理スクリプトを作成する最も簡単な方法は、`caa_profile` コマンドでリソース・プロファイルを作成するときに、自動的に作成されるようにすることです。たとえば、次のように `-B` オプションを指定すると、作成することができます。

```
# caa_profile -create resource_name -t application -B application_path
```

`caa_profile` コマンドで `-B` オプションを使用して、アプリケーションの実行形式ファイルのフル・パス名を `/usr/local/bin/httpd` のように指定します。`-B` オプションを指定すると、`caa_profile` コマンドは、`/var/cluster/caa/script/resource_name.scr` という形式の名前の処理スクリプトを作成します。別の処理スクリプト名を指定したい場合は、`-a` オプションを使います。

アプリケーションによっては、アプリケーションの環境を正しく設定するために、処理スクリプトを編集する必要があります。たとえば、`xclock` のような X アプリケーションでは、処理スクリプトのコマンド行で `DISPLAY` 環境変数を現在のシェルに合わせて設定する必要があります。たとえば、次のようになります。

```
DISPLAY='hostname':0
export DISPLAY
```

アプリケーション・リソースには処理スクリプトが必要なため、`caa_profile -create` コマンドを使用してアプリケーション・リソース・プロファイルを作成する場合は、次のいずれかの条件を満たさなければなりません。

- `caa_profile` にオプション `-B application_executable_pathname` を指定して、処理スクリプトが自動的に作成されるようにします。また、`-a` オプションを使って、作成される処理スクリプトの名前を指定することもできます。

- 省略時のディレクトリ `/var/cluster/caa/script/` に実行可能な処理スクリプトをあらかじめ作成しておかなければなりません。スクリプト名のルートは、作成するリソースと同じ名前であればなりません。

たとえば、処理スクリプトが `/var/cluster/caa/script/up-app-1.scr` という名前の場合、リソース名は `up-app-1` でなければなりません。したがって、`caa_profile` コマンドを使ったリソース・プロファイルを作成する場合、コマンド行に次のように指定します。

```
# caa_profile -create up-app-1 -t application
```

- 実行形式の処理スクリプトがすでにある場合、`caa_profile` のオプション `-a action_script_pathname` を、たとえば次のように指定して、CAA に処理スクリプトのある場所を知らせます。

```
-a /usr/users/smith/caa/scripts/app.scr
```

警告

処理スクリプトはセキュリティ上の理由からルートでだけ作成可能です。

2.3.1 アプリケーション・リソースの処理スクリプトを作成する際のガイドライン

アプリケーション・リソースの処理スクリプトを作成するときには、次のことに注意してください。

- CAA は、アプリケーションの状態を `ONLINE` にするか `OFFLINE` にするかを、処理スクリプトからの終了コードによって判断します。処理スクリプト内のそれぞれのエントリ・ポイントでは、成功したときに

は終了コード 0 を返し、失敗したときには 0 以外の終了コードを返さなければなりません。

- CAA は、stop エントリ・ポイントの処理スクリプトが SCRIPT_TIMEOUT 値 (秒数) の間に終了しないか、ゼロ以外の値を返した場合、アプリケーションの状態を UNKNOWN とします。UNKNOWN 状態は、起動、再配置、または停止を実行するときに発生します。アプリケーションが正常に停止した場合、またはアプリケーションを実行していない場合、stop エントリ・ポイントの処理スクリプトが値 0 で終了するようにしてください。
- デーモンは起動すると、通常はバックグラウンド・プロセスとして動作します。起動時に即座にバックグラウンドに置かれないアプリケーションについては、そのアプリケーションを起動する行の終わりにアンパサンド (&) を追加することによって、そのアプリケーションをバックグラウンドで動作するようにします。アプリケーションをこの方法で使用する場合、起動しようとするときに正常のリターン・コードが戻ります。つまり、省略時のスクリプトには、つまらない理由による失敗 (たとえば、コマンド・パスのつづり間違い) を検出する方法がないということになります。このようなコマンドを使うときは、CAA で使う前にそのスクリプトに使うコマンドを対話的に実行して構文エラーや単純ミスを取り除くことをお勧めします。

CAA の下で実行する X Windows アプリケーションについて、次のようなことを考慮してください。

- クラスタで動作し、CAA が監視しているグラフィカル・アプリケーションでは、処理スクリプト内でクライアント・システムの環境変数 DISPLAY を設定する必要があります。例を次に示します。

```
export DISPLAY=everest:0.0
/usr/bin/my_application &
```

- クライアント・システムでは、許可する X サーバ接続のリストに、省略時のクラスタ別名を追加します。例を次に示します。

```
everest#> xhost +my_cluster
```

- caa_profile または SysMan で作成する CAA の処理スクリプトは、PATH 環境変数を設定しません。処理スクリプトを実行するとき、PATH は省略時の値 /sbin:/usr/sbin:/usr/bin に設定されます。したがって、処理スクリプトで使用するパス名のほとんどは明示的に指定するか、作成された処理スクリプトを修正して PATH を明示的に設定するように

しなければなりません。以前のリリースで自動的に作成された処理スクリプトでは、PATH に現在のディレクトリ (.) が設定されています。この設定には潜在的なセキュリティ上の問題があるので、それらの処理スクリプトについては、パスから現在のディレクトリを削除してください。

2.3.2 処理スクリプトの例

処理スクリプトのテンプレートは、`/var/cluster/caa/template/template.scr` にあります。これは、`caa_profile` コマンドによって作成される処理スクリプトのベースとして使われるものであり、処理スクリプトの個々の要素のモデルとして使用できます。

例として、次のアプリケーション・リソースの処理スクリプトを使用することができます。これらの処理スクリプトの例は `/var/cluster/caa/script` ディレクトリにあります。

- `cluster_lockd.scr`
- `dhcp.scr`
- `named.scr`
- `autofs.scr`

また、2.14 節に示したスクリプトも、処理スクリプトのよい例となります。`/var/cluster/caa/examples_unsupported` ディレクトリには、これらの例を含むスクリプトが格納されています。ここには、CAA を使用するいくつかのアプリケーションの例があります。このディレクトリにあるスクリプト `sysres_tmpl.scr` には、追加のシステムの性能に関連するコードがあり、システム負荷、スワップ領域使用率、使用可能なディスク・スペースを調べるのに使用できます。これらの機能をスクリプトに組み込む場合は、システムに適した値をこれらの機能に関連付けられた変数に設定します。

2.3.3 環境変数のアクセス

処理スクリプトは、多くの環境変数にアクセスすることができるので、スクリプトを変数に対応させるように作成することができます。

CAA 環境で実行する処理スクリプトがアクセスできる変数には、次のものがあります。

- プロファイル属性

- 理由コード
- ロケール情報
- ユーザ定義属性

CAA 定義のリソース・プロファイルの属性は、その属性に `_CAA_` を付けることにより処理スクリプトの環境変数としてアクセスすることができます。たとえば、`AUTO_START` の値は、`_CAA_AUTO_START` を使用することによって得られます。

理由コードは、処理スクリプトが実行された理由を示します。環境変数 `_CAA_REASON` には、次の理由コードの値のいずれかが設定されます。

<code>user</code>	<code>caa_start</code> , <code>caa_stop</code> , <code>caa_relocate</code> など、ユーザが開始したコマンドによって処理スクリプトが起動された。
<code>failure</code>	障害状態が発生したために処理スクリプトが起動された。この値が設定される状態は、チェック・スクリプトが失敗した場合が一般的である。
<code>dependency</code>	依存関係にある他のリソースが障害になったために処理スクリプトが起動された。
<code>boot</code>	処理スクリプトが最初のクラスタ・ブートの結果として起動された (リソースはシステムの起動前に実行されていた)。
<code>autostart</code>	リソースが自動起動された。 <code>AUTOSTART</code> プロファイルの属性が 1 に設定されている場合に、最後のシャットダウンの前に、リソースがあらかじめオフラインになっているときにはクラスタのブート時にリソースは自動起動される。
<code>system</code>	処理スクリプトが通常の保守のためにシステムによって起動された (たとえば、チェック・スクリプトによる再配置の開始)。

unknown

スクリプトが起動されたが、その状態は内部的には認識されていない。この値が表示された場合には、クラスタとアプリケーションの状態を記録し、サポート担当に連絡する。

CAA コマンドが処理スクリプトを起動する環境のロケールは、`_CAA_CLIENT_LOCALE` 環境変数の情報が使用可能です。この変数には、以下のロケール情報が含まれていて、文字列の値の各情報はスペースで区切られています (`LC_ALL`, `LC_CTYPE`, `LC_MONETARY`, `LC_NUMERIC`, `LC_TIME`, `LC_MESSAGES`)。処理スクリプトは必要に応じて、この情報を使用して処理スクリプト環境にロケールを設定できます。

ロケールについての詳細は、`setlocale(3)` および `locale(1)` を参照してください。

処理スクリプトでの理由コードの使用例を次に示します。

```
if [ "$_CAA_REASON" = "user" ]; then
    echo "Action invoked by User"
    :
fi
```

2.3.4 処理スクリプトの出力先

処理スクリプトからの出力がリダイレクトでき、`caa_start`, `caa_stop`, `caa_relocate` の実行時にその結果を表示できます。出力の各行には、オプションでクラスタ・メンバとリソース名からなるプリフィクスを付けることができます。

省略時には、出力はリダイレクトされません。

CAA で処理スクリプトの出力をリダイレクトできるようにするには、`caa_start`, `caa_stop`, または `caa_relocate` を実行する環境では、以下のように環境変数 `_CAA_UI_FMT` を `v` または `vs` のいずれかに設定する必要があります。

```
# export _CAA_UI_FMT=v
# caa_start db_2 ...
nodex:db_2:output text ...
nodex:db_2:output text ...
nodex:db_2:output text ...
nodex:db_2:output text ...
```

修飾子 `s` を使用すると、出力のプリフィクス部分のノード：リソースは表示されません。

```
# export _CAA_UI_FMT=vs
# caa_start db_2
output text ...
output text ...
```

2.4 ユーザ定義属性の作成

アプリケーション・リソース・プロファイルでは、ユーザ定義属性を指定して拡張することができます。これらのユーザ定義属性は、リソースの処理スクリプト内で環境変数としてアクセスでき、すべてのアプリケーション・リソースに適用できます。

ユーザ定義属性は、最初に `/var/cluster/caa/template/application.tdf` にあるアプリケーション・リソース・タイプ定義ファイルで定義する必要があります。定義する必要がある値は次のとおりです。

attribute	ユーザが値を指定できる属性を定義する。この属性は、すべてのアプリケーション・リソースの処理スクリプトでアクセス可能な環境変数に変換される。
type	この属性で使用可能な値の種類を定義する。種類としては、 <code>boolean</code> 、 <code>string</code> 、 <code>name_list</code> 、 <code>name_string</code> 、 <code>positive_integer</code> 、 <code>internet_address</code> 、 <code>file</code> がある。
switch	プロファイルの値を指定するために、 <code>caa_profile</code> コマンドで使用するスイッチを定義する。
default	この属性をプロファイルに指定しない場合の省略時の値を定義する。
required	スイッチがプロファイルに指定されているかいないかを定義する。

ユーザ定義属性は、プロファイルだけでなく `caa_start`、`caa_relocate`、`caa_stop` のコマンド行にも指定することができます。コマンド行に指定された値は、プロファイルで指定された値より優先されます。詳細については、`caa_start(8)`、`caa_relocate(8)`、または `caa_stop(8)` を参照してください。

で始まるタイプ定義ファイルの各行は、コメントとして解釈されます。

タイプ定義ファイルのエントリの例を次に示します。

```
#!/=====
attribute: USR_DEBUG
type: boolean
switch: -o d
default: 0
required: no
```

2.5 リソースの登録

どのリソースにもプロファイルが必要です。リソースは CAA によって管理されるため、CAA に登録する必要があります。リソースの登録は、`caa_register` コマンドで行います。たとえば、`clock` アプリケーションを登録する場合、次のコマンドを実行します。

```
# /usr/sbin/caa_register clock
```

リソースを登録すると、プロファイル内の情報はバイナリの CAA レジストリに保存されます。プロファイルを変更したときには、`caa_register -u` コマンドを使ってデータベースを更新する必要があります。

詳細は、`caa_register(8)` を参照してください。

2.6 アプリケーション・リソースの起動

CAA に登録したアプリケーションを起動するには、`caa_start` コマンドを実行します。アプリケーション・リソースの名前は、アプリケーションと同じ名前でも違う名前でもかまいません。たとえば、次のようにアプリケーションを起動します。

```
# /usr/sbin/caa_start clock
```

次は、このコマンドの出力結果の例です。

```
Attempting to start 'clock' on member 'polishham'
Start of 'clock' on member 'polishham' succeeded.
```

このアプリケーションは、`polishham` という名前のシステムで動作しています。

このコマンドは、処理スクリプトを呼び出すたびに、処理スクリプトから返される成功したかどうかの通知を `SCRIPT_TIMEOUT` 値だけ待ちます。

アプリケーション・リソースと非アプリケーション・リソースが障害しきい値を超えたために停止した場合にも、アプリケーション・リソースを起動す

ることができ、非アプリケーション・リソースは再起動することができます (非アプリケーション・リソースの再起動については、『クラスタ管理ガイド』を参照してください)。起動する前に、`caa_register` でこのリソースを登録する必要があります。

注意

リソースを起動/停止するときには、必ず `caa_start` および `caa_stop`、または同等の SysMan 機能を使用します。アプリケーションを起動したり停止したりするために、コマンド行を使用して手作業で行ったり、処理スクリプトを実行したりしてはなりません。CAA の管理外で、手作業による起動または停止を行うと、リソースの状態が不適切になります。

別のクラスタ・メンバで `ONLINE` 状態になっている必須リソースのあるリソースを起動すると、起動に失敗します。すべての必須リソースは、リソースが起動されるメンバで `OFFLINE` 状態または `ONLINE` の状態になっている必要があります。

`OFFLINE` 状態の必須リソースのあるリソースでは、`caa_start -f resource_name` コマンドを使用すると、対象リソースが起動されると同時に、現在 `ONLINE` 状態でないすべての必須リソースも起動されます。

アプリケーション・リソースでは、`caa_start` コマンドを実行することによってのみ、実際にリソースのターゲット値が `ONLINE` 状態になります。ターゲット値は、CAA がリソースに設定しようとする状態を定義します。

CAA は、ターゲットに一致するように状態を変更しようとし、処理スクリプトの開始エントリ・ポイントを実行してアプリケーションを起動しようとします。アプリケーションが動作している場合は、ターゲットの状態と現在の状態が両方とも `ONLINE` 状態です。ターゲットと状態のフィールドおよびそのリソースの状態の詳細は、『クラスタ管理ガイド』を参照してください。

注意

システムがクラッシュしているクラスタ・メンバ上でアプリケーションを起動しようとすると、`caa_start` はあいまいな結果を表示します。このような場合には、処理スクリプトの開始部分は実行されますが、開始の通知がコマンド行に表示される前に

そのクラスタ・メンバはクラッシュします。caa_start は、Remote start for [resource_name] failed on member [member_name] というエラーを表示して障害を通知します。アプリケーション・リソースは実際には ONLINE で、別のメンバにフェイルオーバーし、あたかもアプリケーションが間違ったメンバ上で起動したかのように見えます。

アプリケーション・リソースを起動している間にそのクラスタ・メンバに障害が発生した場合には、そのリソースの状態を調べるため、caa_stat でクラスタ上のリソースの状態をチェックしてください。

詳細については、caa_start(8) を参照してください。

2.7 アプリケーション・リソースの再配置

アプリケーション・リソースを再配置するには、caa_relocate コマンドを使用します。ネットワーク、テープ、およびメディア・チェンジャは再配置できません。

アプリケーション・リソースを、利用可能なクラスタ・メンバ、または指定したクラスタ・メンバに再配置するには、caa_relocate コマンドを使用します。たとえば、clock アプリケーションをメンバ provolone に再配置するには、次のコマンドを実行します。

```
# /usr/sbin/caa_relocate clock -c provolone
```

次に、このコマンドの出力結果の例を示します。

```
Attempting to stop 'clock' on member 'polishham'
Stop of 'clock' on member 'polishham' succeeded.
Attempting to start 'clock' on member 'provolone'
Start of 'clock' on member 'provolone' succeeded.
```

アプリケーションのリソース・プロファイルに定義されている配置ポリシーを使って、clock アプリケーションを別のメンバに再配置するには、次のコマンドを実行します。

```
# /usr/sbin/caa_relocate clock
```

次に、このコマンドの出力結果の例を示します。

```
Attempting to stop 'clock' on member 'pepicelli'
Stop of 'clock' on member 'pepicelli' succeeded.
Attempting to start 'clock' on member 'polishham'
Start of 'clock' on member 'polishham' succeeded.
```

次に、スクリプトがゼロ以外の値を返したか、スクリプトのタイムアウトのために、アプリケーションをうまく再配置できなかった場合の、出力結果の例を示します。

```
Attempting to stop 'clock' on member 'pepicelli'
Stop of 'clock' on member 'pepicelli' succeeded.
Attempting to start 'clock' on member 'provolone'
Start of 'clock' on member 'provolone' failed.
No more members to consider
Attempting to restart 'clock' on member 'pepicelli'
Could not relocate resource clock.
```

caa_relocate コマンドは、処理スクリプトを呼び出すたびに、処理スクリプトから返される成功したかどうかの通知を SCRIPT_TIMEOUT 値だけ待ちます。

次の場合は、再配置しようとしても失敗します。

- そのリソースに ONLINE 状態の必須リソースがある。
- 特定リソースを必要とするリソースが ONLINE 状態である。

ONLINE 状態の必須リソースを持っているか、リソースが ONLINE であることを必要とするリソースで、caa_relocate -f resource_name コマンドを使用すると、そのリソースは再配置され、それが ONLINE であることを必要とするすべてのリソースは再配置されます。指定したリソースで必要とされるすべてのリソースは再配置され、それらの状態とは関係なく、起動されます。

詳細は、caa_relocate(8) を参照してください。

2.8 アプリケーション・リソースの分散

アプリケーション・リソースの分散とは、クラスタ上のリソースの現在の状態とリソースの配置規則に基づいて、アプリケーション・リソースの配置を再評価することです。アプリケーションの分散は、クラスタ単位、メンバ単位、または指定したリソースで行うことができます。分散は、CAA の標準の配置決定メカニズムを使用して決定されますが、負荷を考慮しません。

caa_balance を使用すると、アプリケーション・リソースのみを分散できます。ネットワーク、テープ、またはメディア・チェンジャのリソースを分散することはできません。

2.2.2.2 項で説明しているように、クラスタ単位で分散を行うと、クラスタ上にあるすべての ONLINE アプリケーション・リソースを再評価し、配置決

定メカニズムにより選択されたクラスタ・メンバで実行されていないそれぞれのリソースを再配置します。

クラスタ上のすべてのアプリケーションを分散させるには、次のコマンドを入力します。

```
# /usr/sbin/caa_balance -all
```

2つのアプリケーション `test` と `test2` だけが ONLINE で、balanced 配置ポリシーによりメンバ `rye` 上で実行されているとします。その場合、次のようなテキストが表示されます。

```
Attempting to stop 'test' on member 'rye'
Stop of 'test' on member 'rye' succeeded.
Attempting to start 'test' on member 'swiss'
Start of 'test' on member 'swiss' succeeded.
Resource test2 is already well placed
test2 is placed optimally. No relocation is needed.
```

クラスタ内で3つ以上のアプリケーションが ONLINE の場合、出力にはそれぞれのアプリケーション・リソースに対してとられた処理が表示されます。

クラスタ・メンバ `rye` 上で実行されているアプリケーションの配置を再評価するには、次のコマンドを入力します。

```
# /usr/sbin/caa_balance -s rye
```

2つのアプリケーションだけが ONLINE で、balanced 配置ポリシーのもとでメンバ `rye` 上で実行されているとします。その場合、次のようなテキストが表示されます。

```
Attempting to stop 'test' on member 'rye'
Stop of 'test' on member 'rye' succeeded.
Attempting to start 'test' on member 'swiss'
Start of 'test' on member 'swiss' succeeded.
Resource test2 is already well placed
test2 is placed optimally. No relocation is needed.
```

3つ以上のアプリケーションがクラスタ・メンバ上で ONLINE の場合、出力にはそれぞれのアプリケーション・リソースに対してとられた処理が表示されます。

指定したアプリケーションだけを分散させるには、次のコマンドを入力します。

```
# /usr/sbin/caa_balance test test2
```

2つのアプリケーション `test` と `test2` が balanced 配置ポリシーのもとでメンバ `rye` 上で実行されているとします。その場合、次のテキストが表示されます。

```
Attempting to stop 'test' on member 'rye'
Stop of 'test' on member 'rye' succeeded.
Attempting to start 'test' on member 'swiss'
Start of 'test' on member 'swiss' succeeded.
Resource test2 is already well placed
test2 is placed optimally. No relocation is needed.
```

プロファイル内の時間値は、`t:day:hour:min` というフォーマットで指定する必要があります。ここで、`day` は曜日 (0 ~ 6) を示し、`hour` はその曜日の時間 (0 ~ 23) を示し、`min` は分 (0 ~ 59) を示して、再評価が行われる時間を指定します。毎日または毎時を指定する場合には、ワイルドカードとしてアスタリスク (*) を使用できます。

たとえば、アプリケーションを毎日曜日午前 3 時に再分散する場合は、次のようになります。

```
REBALANCE=t:0:3:0
```

また、アプリケーションを毎日午前 2:30 に再分散する場合は、次のようになります。

```
REBALANCE=t:*:2:30
```

`caa_profile` コマンドを使用してこれを指定すると、次のようになります。

```
# /usr/sbin/caa_profile -create testapp -t application
-B /usr/bin/true -o bt=*:2:30
```

その結果、作成されるプロファイルは次のようになります。

```
NAME=testapp
TYPE=application
ACTION_SCRIPT=testapp.scr
ACTIVE_PLACEMENT=0
AUTO_START=0
CHECK_INTERVAL=60
DESCRIPTION=testapp
FAILOVER_DELAY=0
FAILURE_INTERVAL=0
FAILURE_THRESHOLD=0
HOSTING_MEMBERS=
OPTIONAL_RESOURCES=
PLACEMENT=balanced
REBALANCE=t:*:2:30
REQUIRED_RESOURCES=
RESTART_ATTEMPTS=1
SCRIPT_TIMEOUT=60
```

詳細については、`caa_balance(8)` を参照してください。

2.9 アプリケーション・リソースの停止

クラスタ環境で動作しているアプリケーションを停止するには、`caa_stop` コマンドを使用します。`caa_stop` コマンドを実行すると、直ちにターゲットが `OFFLINE` に設定されます。CAA は常にリソースの状態をターゲットに一致させようとするので、CAA サブシステムはアプリケーションを停止します。停止できるリソースは、アプリケーション・リソースだけです。ネットワーク、テープ、およびメディア・チェンジャ・タイプのリソースは、停止できません。

次の例では、`clock` アプリケーション・リソースを停止します。

```
# /usr/sbin/caa_stop clock
```

次は、このコマンドの出力例です。

```
Attempting to stop 'clock' on member 'polishham'  
Stop of 'clock' on member 'polishham' succeeded.
```

別の `ONLINE` アプリケーションに対して、そのアプリケーションが必須リソースである場合には、そのアプリケーションを停止することはできません。

他の `ONLINE` 状態のリソースが必要とするリソースで `caa_stop -f resource_name` コマンドを使用すると、そのリソースは停止して、それを必要とする `ONLINE` 状態の他のリソースも停止します。

詳細は、`caa_stop(8)` を参照してください。

2.10 アプリケーション・リソースの登録抹消

アプリケーション・リソースの登録を抹消するには、`caa_unregister` コマンドを使用します。

`ONLINE` 状態にあるかまたは他のリソースに必要なアプリケーションは登録を抹消できません。次の例では、`clock` アプリケーションの登録を抹消します。

```
# /usr/sbin/caa_unregister clock
```

詳細は、`caa_unregister(8)` を参照してください。

2.11 CAA の状態情報の表示

クラスタ・メンバ上のリソースに関する状態情報を表示するには、`caa_stat` コマンドを使用します。

次に、clock リソースの状態情報の表示例を示します。

```
# /usr/bin/caa_stat clock

NAME=clock
TYPE=application
TARGET=ONLINE
STATE=ONLINE on provolone
```

すべてのリソースの情報を表示するには、次のコマンドを実行します。

```
# /usr/bin/caa_stat

NAME=clock
TYPE=application
TARGET=ONLINE
STATE=ONLINE on provolone

NAME=dhcp
TYPE=application
TARGET=ONLINE
STATE=ONLINE on polishham

NAME=named
TYPE=application
TARGET=ONLINE
STATE=ONLINE on polishham

NAME=network1
TYPE=network
TARGET=ONLINE on provolone
TARGET=ONLINE on polishham
STATE=ONLINE on provolone
STATE=ONLINE on polishham
```

すべてのリソースの情報を表形式で表示するには、次のコマンドを実行します。

```
# /usr/bin/caa_stat -t
```

Name	Type	Target	State	Host
cluster_lockd	application	ONLINE	ONLINE	provolone
dhcp	application	OFFLINE	OFFLINE	
network1	network	ONLINE	ONLINE	provolone
network1	network	ONLINE	ONLINE	polishham

リソースを再起動した回数またはリソース障害間隔内に障害が発生した回数、リソースを再起動できる回数または障害に対応できる回数、アプリ

ケーションのターゲット状態，および通常の状態情報を表示するには，次のコマンドを実行します。

```
# /usr/bin/caa_stat -v

NAME=cluster_lockd
TYPE=application
RESTART_ATTEMPTS=30
RESTART_COUNT=0
FAILURE_THRESHOLD=0
FAILURE_COUNT=0
TARGET=ONLINE
STATE=ONLINE on provolone

NAME=dhcp
TYPE=application
RESTART_ATTEMPTS=1
RESTART_COUNT=0
FAILURE_THRESHOLD=3
FAILURE_COUNT=1
TARGET=ONLINE
STATE=OFFLINE

NAME=network1
TYPE=network
FAILURE_THRESHOLD=0
FAILURE_COUNT=0 on polishham
FAILURE_COUNT=0 on polishham
TARGET=ONLINE on provolone
TARGET=ONLINE on polishham
STATE=ONLINE on provolone
STATE=OFFLINE on polishham
```

詳細情報を表形式で表示したい場合は，次のコマンドを実行します。

```
# /usr/bin/caa_stat -v -t
```

Name	Type	R/RA	F/FT	Target	State	Host
cluster_lockd	application	0/30	0/0	ONLINE	ONLINE	provolone
dhcp	application	0/1	0/0	OFFLINE	OFFLINE	
named	application	0/1	0/0	OFFLINE	OFFLINE	
network1	network		0/5	ONLINE	ONLINE	provolone
network1	network		0/5	ONLINE	ONLINE	polishham

データベースに保存されているプロファイル情報を表示するには，次のコマンドを実行します。

```
# /usr/bin/caa_stat -p

NAME=cluster_lockd
TYPE=application
ACTION_SCRIPT=cluster_lockd.scr
```



```

ACTIVE_PLACEMENT=0
AUTO_START=1
CHECK_INTERVAL=5
DESCRIPTION=Cluster lockd/statd
FAILOVER_DELAY=30
FAILURE_INTERVAL=60
FAILURE_THRESHOLD=1
REBALANCE=
HOSTING_MEMBERS=
OPTIONAL_RESOURCES=
PLACEMENT=balanced
REQUIRED_RESOURCES=
RESTART_ATTEMPTS=2
SCRIPT_TIMEOUT=60
:

```

詳細は、『クラスタ管理ガイド』および `caa_stat(1)` を参照してください。

2.12 グラフィカル・ユーザ・インタフェース

以降の各項では、CAA を操作するために、SysMan および SysMan Station グラフィカル・ユーザ・インタフェース (GUI) を使用する方法を説明します。

2.12.1 SysMan Menu による CAA の管理

SysMan Menu は、コマンド行に `/usr/sbin/sysman` と入力することにより、起動することができます。CAA の各種ツールにアクセスするには、[TruCluster Specific] ブランチの下に [Cluster Application Availability (CAA) Management] を選択します。

```

:
+ TruCluster Specific
  |Cluster Application Availability (CAA) Management

```

[Cluster Application Availability (CAA) Management] タスクだけを起動する場合は、`/usr/sbin/sysman caa` を使用します。

SysMan Menu のアクセス方法の詳細は、Tru64 UNIX の『システム管理ガイド』を参照してください。

SysMan Menu を使用して、次のことを行うことができます。

- リソース・プロファイルの管理
- CAA リソースの監視

- リソースの登録
- リソースの起動
- リソースの再配置
- リソースの停止
- リソースの登録抹消

CAA GUI では、EVM (イベント・マネージャ) と CAA デーモンからのイベント・レポートをベースにして、グラフィカルにクラスタを管理します。

2.12.2 SysMan Station による CAA の管理と監視

SysMan Station は、クラスタの包括的情報をグラフィカルに表示します。SysMan Station では、クラスタ全体の CAA リソースの現在の状態を表示して、リソースを管理することができます。また SysMan Station では、管理ツール SysMan Menu で個々の CAA リソースを管理することができます。SysMan Station にアクセスする方法についての詳細は、Tru64 UNIX の『システム管理ガイド』を参照してください。

SysMan Station で、CAA SysMan Menu の各種ツールにアクセスするには、次の手順に従います。

1. [Views] の下の、たとえば [CAA_Applications_(active)] や [CAA_Applications_(all)] などのビューのいずれかを選択します。
2. [Views] ウィンドウ (たとえば [CAA_Applications_(active) View] や [CAA_Applications_(all) View] など) の下のクラスタ名を選択します。
3. [Tools]メニューから [SysMan Menu] を選択します。[Cluster Application Availability (CAA) Management] タスクは、[TruCluster Specific] ブランチの下にあります。

SysMan Menu と SysMan Station についての詳細は、オンライン・ヘルプまたは Tru64 UNIX の『システム管理ガイド』を参照してください。

2.13 CAA のチュートリアル

この CAA のチュートリアルでは、CAA を使ってアプリケーションの高可用性を迅速に実現するために必要な基本的な手順を紹介します。個々のコマンドについての詳細は、CAA の各種コマンドについて説明したドキュメントを読む必要があります。

- 前提条件 (2.13.1 項)
- 準備作業 (2.13.2 項)
- dtcalc の処理スクリプトの例 (2.13.3 項)
- ステップ 1: アプリケーション・リソース・プロファイルの作成 (2.13.4 項)
- ステップ 2: アプリケーション・リソース・プロファイルの検証 (2.13.5 項)
- ステップ 3: アプリケーションの登録 (2.13.6 項)
- ステップ 4: アプリケーションの起動 (2.13.7 項)
- ステップ 5: アプリケーションの再配置 (2.13.8 項)
- ステップ 6: アプリケーションの停止 (2.13.9 項)
- ステップ 7: アプリケーションの登録抹消 (2.13.10 項)

このチュートリアルでは、サンプルのクラスタは、メンバ `provolone` , `polishham` , `pepicelli` を含んでいます。コマンド行で使用するメンバ名は、ご使用のクラスタのメンバ名に置き換えてください。

2.13.1 前提条件

2 メンバで構成する TruCluster Server クラスタに root ユーザとしてアクセスできる必要があります。

このチュートリアルでは、CAA を使って、Tru64 UNIX アプリケーションの dtcalc に高可用性を実現します。テスト・アプリケーション `/usr/dt/bin/dtcalc` があることを確認してください。

この例では、X ベースのアプリケーションを使用していますが、特にこのアプリケーションに限定する必要はありません。ただ、X ベースのアプリケーションであれば、起動、停止、および再配置の結果を即座に見ることができます。この種の高可用性アプリケーションは、あまり使用されないかもしれません。

2.13.2 準備作業

CAA を使ってグラフィカル・インタフェースを持つアプリケーションの高可用性を実現するには、`ActionScript.scr` ファイルで、

DISPLAY 変数を正しく設定する必要があります。DISPLAY 変数を修正したら、ファイル *ActionScript.scr* をスクリプト・ディレクトリ */var/cluster/caa/script* にコピーします。

アプリケーションを表示したいホストで、クラスタから X アプリケーションを表示できることを確認します。アクセスを変更する必要がある場合は、アプリケーションを表示するマシンで、次のようなコマンドを実行します。

```
# xhost + clustername
```

各メンバの実際の名前がわからない場合は、システム上の */etc/hosts* ファイルを見て確認してください。また、*clu_get_info* コマンドを使って各クラスタ・メンバの情報 (ホスト名など) を得ることもできます。

次に、*clu_get_info* コマンドとその実行結果の例を示します。

```
# clu_get_info
```

```
Cluster information for cluster deli

Number of members configured in this cluster = 3
memberid for this member = 3
Quorum disk = dsk10h
Quorum disk votes = 1

Information on each cluster member

Cluster memberid = 1
Hostname = polishham.zk4.com
Cluster interconnect IP name = polishham=ics0
Member state = UP

Cluster memberid = 2
Hostname = provolone.zk4.com
Cluster interconnect IP name = provolone=ics0
Member state = UP

Cluster memberid = 3
Hostname = pepicelli.zk4.com
Cluster interconnect I name = pepicelli=ics0
Member state = UP
```

2.13.3 dtcalc の処理スクリプトの例

次に、dtcalc のチュートリアルで使用する処理スクリプトの例を示します。caa_profile コマンドで作成されるもっと複雑な処理スクリプトを使用することもできます。

```
#!/usr/bin/ksh -p
#
# This action script will be used to launch dtcalc.
#
export DISPLAY='hostname':0
PATH=/sbin:/usr/sbin:/usr/bin
export PATH
CAATMPDIR=/tmp

CMDPATH=/usr/dt/bin

APPLICATION=${CMDPATH}/dtcalc

CMD='basename $APPLICATION'

case $1 in
    'start') ❶ if [ -f $APPLICATION ]; then
                $APPLICATION & exit 0
                echo "Found exit1" >/dev/console exit 1
            fi ;;
    'stop') ❷ PIDLIST='ps ax | grep $APPLICATION | grep -v 'caa_' \
                | grep -v 'grep' | awk '{print $1}'
                if [ -n "$PIDLIST" ]; then
                    kill -9 $PIDLIST
                    exit 0
                fi
                exit 0
            ;;
    'check') ❸ PIDLIST='ps ax | grep $CMDPATH | grep -v 'grep' | awk '{print $1}'
                if [ -z "$PIDLIST" ]; then
                    PIDLIST='ps ax | grep $CMD | grep -v 'grep'
                    | awk '{print $1}'
                fi
                if [ -n "$PIDLIST" ]; then
                    exit 0
                else
                    echo "Error: CAA could not find $CMD." >/dev/console
                    exit 1
                fi
            ;;
esac
```

- ❶ 起動エントリ・ポイントは、アプリケーションを起動するときに実行されます。
- ❷ 停止エントリ・ポイントは、アプリケーションを停止するときに実行されます。
- ❸ チェック・エントリ・ポイントは、CHECK_INTERVAL 秒ごとに実行されます。

2.13.4 ステップ 1: アプリケーション・リソース・プロファイルの作成

dtcalc のリソース・プロファイルを作成するために、`caa_profile` コマンドに次のオプションを指定して実行します。

```
# /usr/sbin/caa_profile -create dtcalc -t application -B /usr/dt/bin/dtcalc \
-d "dtcalc application" -p balanced
```

`/var/cluster/caa/profile/` にある `dtcalc.cap` ファイルを調べると、次のような内容になっているはずです。

```
# cat dtcalc.cap

NAME=dtcalc
TYPE=application
ACTION_SCRIPT=dtcalc.scr
ACTIVE_PLACEMENT=0
AUTO_START=0
CHECK_INTERVAL=60
DESCRIPTION=dtcalc application
FAILOVER_DELAY=0
FAILURE_INTERVAL=0
FAILURE_THRESHOLD=0
HOSTING_MEMBERS=
OPTIONAL_RESOURCES=
PLACEMENT=balanced
REQUIRED_RESOURCES=
RESTART_ATTEMPTS=1
SCRIPT_TIMEOUT=60
```

2.13.5 ステップ 2: アプリケーション・リソース・プロファイルの検証

リソース・プロファイルの構文を検証するために、次のコマンドを実行します。

```
# caa_profile -validate dtcalc
```

プロファイルに構文エラーがあると、`caa_profile` は、プロファイルが検証をパスしなかったことを示すメッセージを表示します。

2.13.6 ステップ 3: アプリケーションの登録

アプリケーションを登録するために、次のコマンドを実行します。

```
# /usr/sbin/caa_register dtcalc
```

プロファイルを登録できない場合は、その理由を示すメッセージが表示されます。

アプリケーションが登録されたことを確認するには、次のコマンドを実行します。

```
# /usr/bin/caa_stat dtcalc

NAME=dtcalc
TYPE=application
TARGET=OFFLINE
STATE=OFFLINE
```

2.13.7 ステップ 4: アプリケーションの起動

アプリケーションを起動するために、次のコマンドを実行します。

```
# /usr/bin/caa_start dtcalc
```

次のメッセージが表示されます。

```
Attempting to start 'dtcalc' on member 'provolone'
Start of 'dtcalc' on member 'provolone' succeeded.
```

/usr/bin/caa_stat dtcalc コマンドを実行して、dtcalc 処理スクリプトの起動エントリ・ポイントが正しく実行され、dtcalc が起動されたことを確認できます。次に、例を示します。

```
# /usr/bin/caa_stat dtcalc

NAME=dtcalc
TYPE=application
TARGET=ONLINE
STATE=ONLINE on provolone
```

DISPLAY 変数がスクリプト内で正しく設定されていれば、dtcalc が画面に表示されます。

2.13.8 ステップ 5: アプリケーションの再配置

アプリケーションを再配置するために、次のコマンドを実行します。

```
# /usr/bin/caa_relocate dtcalc -c polishham
```

コマンド /usr/bin/caa_stat dtcalc を実行して、dtcalc が正常に起動されたことを確認します。次に例を示します。

```
# /usr/bin/caa_stat dtcalc

NAME=dtcalc
TYPE=application
TARGET=ONLINE
STATE=ONLINE on polishham
```

STATE 属性にクラスタ・メンバがリストされます。

2.13.9 ステップ 6: アプリケーションの停止

アプリケーションを停止するために、次のコマンドを実行します。

```
# /usr/bin/caa_stop dtcalc
```

次のメッセージが表示されます。

```
Attempting to stop 'dtcalc' on member 'provolone'  
Stop of 'dtcalc' on member 'provolone' succeeded.
```

/usr/bin/caa_stat dtcalc コマンドを次のように実行して、dtcalc 処理スクリプトの停止エントリ・ポイントが正常に実行され、dtcalc が停止することを確認します。

```
# /usr/bin/caa_stat dtcalc
```

```
NAME=dtcalc  
TYPE=application  
TARGET=OFFLINE  
STATE=OFFLINE
```

2.13.10 ステップ 7: アプリケーションの登録抹消

アプリケーションの登録を抹消するために、次のコマンドを実行します。

```
# /usr/sbin/caa_unregister dtcalc
```

2.14 CAA で管理するアプリケーションの例

以降の各項では、CAA で管理する高可用性シングル・インスタンス・アプリケーションの例を示します。

2.14.1 OpenLDAP ディレクトリ・サーバ

OpenLDAP (Lightweight Directory Access Protocol) ディレクトリ・サーバは、HP によって開発された管理ツールに統合された人気の高いインターネット・ソフトウェアの集まりである、Internet Express for Tru64 UNIX 製品スイートの一部です (Internet Express は、すべての HP Tru64 UNIX AlphaServer システムといっしょに出荷されています。Internet Express は、次の URL から入手できます。 http://www.tru64unix.com-paq.com/docs/pub_page/iass_docs.html)。このスイート内の製品はクラスタ対応で、クラスタ内で高可用性を持つように構成可能です。

システム認証用 LDAP モジュールにより，LDAP サーバに格納されたユーザ認識情報と認証情報は以下を含め，すべてのアプリケーションで 사용할 ことができます。

- ログイン認証 (rlogin , ftp , および telnet)
- POP および IMAP 認証
- libc ライブラリにある getpw*() と getgr*() に対する，透過的な LDAP データベースへのアクセス

TruCluster Server 環境で高可用性の OpenLDAP ディレクトリ・サーバを作成するには，次の処理を行います。

Internet Express Installation グラフィカル・ユーザ・インタフェース (GUI) を使って Internet Express キットをインストールします。Internet Express Administration Utility とインストールする OpenLDAP のサブセットを選択します。

このインストレーション・プロシージャにより，OpenLDAP アプリケーション・リソース用に /var/cluster/caa/profile の中に CAA リソース・プロファイルが作成されます。

```
TYPE = application
NAME = openldap
DESCRIPTION = OpenLDAP Directory Server
CHECK_INTERVAL = 60
FAILURE_THRESHOLD = 0
FAILURE_INTERVAL = 0
REQUIRED_RESOURCES =
OPTIONAL_RESOURCES =
HOSTING_MEMBERS =
PLACEMENT = balanced
RESTART_ATTEMPTS = 1
FAILOVER_DELAY = 0
AUTO_START = 0
ACTION_SCRIPT = openldap.scr
```

また，/var/cluster/caa/script ディレクトリにこのリソースの処理スクリプトが作成されます。

```
#!/sbin/sh

#
# Start/stop the OpenLDAP Directory Server.
#

OLPIDFILE=/data/openldap/var/openldap_slapd.pid
OPENLDAP_CAA=1
export OPENLDAP_CAA

case "$1" in
'start')
/sbin/init.d/openldap start
```

```

        ;;
'stop')
    /sbin/init.d/openldap stop
    ;;
'check')
    # return non-zero if the service is stopped
    if [ -f "$OLPIDFILE" ]
    then
        MYPID=`cat $OLPIDFILE`
        RUNNING=`/usr/bin/ps -e -p $MYPID -o command | grep slapd`
    fi

    if [ -z "$RUNNING" ]
    then
        exit 1
    else
        exit 0
    fi

    ;;
*)
    echo "usage: $0 {start|stop|check}"
    ;;
esac

```

次の init.d スクリプトは、適切な CAA コマンドを呼び出すことによってクラスタ内の OpenLDAP サービスを起動したり停止したりします。

```

#!/sbin/sh
#
# Start the OpenLDAP Directory Server daemon.
#
NAME="OpenLDAP Directory Server"
HOME=/usr/internet/openldap

OLPIDFILE=/data/openldap/var/openldap_slapd.pid
MYPID=
RUNNING=

if [ -x /usr/sbin/clu_get_info ] && /usr/sbin/clu_get_info -q
then
    CLUSTER="YES"
fi

check_running()
{
    if [ -f "$OLPIDFILE" ]
    then
        MYPID=`cat $OLPIDFILE`
        RUNNING=`/usr/bin/ps -e -p $MYPID -o command | grep slapd`
    fi

    if [ ! -z "$RUNNING" ]
    then
        return 1
    else
        return 0
    fi
}

case "$1" in
'start')

```

```

if [ "$CLUSTER" = "YES" -a "$OPENLDAP_CAA" != "1" ]
then
    /usr/sbin/caa_start -q openldap
else
    check_running
    checkres=$?
    if [ $checkres = 1 ]
    then
        echo "$NAME already running"
    else
        $HOME/libexec/slapd -f $HOME/etc/slapd.conf
    fi
fi

;;
'stop')

if [ "$CLUSTER" = "YES" -a "$OPENLDAP_CAA" != "1" ]
then
    exit 1
else
    check_running
    checkres=$?

    if [ $checkres = 1 ]
    then
        kill -TERM $MYPID
    fi
fi

;;
*)
    echo "usage: $0 {start|stop}"
    ;;
esac

```

さらに、`/etc/clua_services` ファイルに次の行を追加します。

```
openldap 389/tcp in_single,out_alias
```

2.14.2 CAA を使ってシングル・インスタンスの高可用性 Apache HTTP サーバを作成する

フェイルオーバー機能のあるシングル・インスタンスの Apache HTTP サーバを作成するには、次の手順に従います。

1. Web サイト (www.apache.org) から最新の標準 Apache 配布ソフトウェアをクラスタにダウンロードし、そのサイトの指示に従って、Apache を構築し `/usr/local/apache` ディレクトリにインストールします。
2. 次のコマンドを使って、省略時の CAA アプリケーション・リソース・プロファイルと処理スクリプトを作成します。

```
# caa_profile -create httpd -t application -B /usr/local/apache/bin/httpd
```

省略時のプロファイルに使用されているフェイルオーバー・ポリシーでは、`httpd` サービスを提供しているメンバがクラスタから離れると他

のメンバがそのサービスを引き継ぐようになっています。また、アクティブなクラスタ・メンバであれば、どのメンバでも `httpd` サービスを提供できるようになっています。フェイルオーバー・ポリシー、配置ポリシー、およびリソースの依存関係を省略時とは別のものにしたい場合は、プロファイルを編集します。

省略時の処理スクリプトには、`httpd` サービスを起動する起動エントリ・ポイントと `httpd` サービスを停止する停止エントリ・ポイントがあります。

3. メンバの 1 つで次のコマンドを実行して、プロファイルを CAA に登録します。

```
# caa_register httpd
```

4. メンバの 1 つで次のコマンドを実行して、CAA から `httpd` サービスを起動します。

```
# caa_start httpd
```

2.14.3 CAA を使ってシングル・インスタンスの Oracle8i サーバを作成する

CAA を使って、フェイルオーバー機構を持つシングル・インスタンスの Oracle8i バージョン 8.1.7 のデータベース・サーバを作成するには、次の手順に従います。

1. Oracle8i のドキュメントに記述されている指示に従って、Oracle8 バージョン 8.1.7 をインストールして構成します。

Oracle では、特定のカーネル属性に特定の値を設定していること、特定の UNIX グループ (`dba`, `oinstall`) を作成していること、特別な環境変数を初期化していることを必要とします。

2. Oracle8i シングル・サーバの CAA サービスをセットアップする前に、クライアント・アプリケーションにどの方法でサービスへアクセスさせるかを決める必要があります。これには、TruCluster Server のクラスタ別名機能を使う方法と、インタフェース (IP) 別名を使う方法があります。クラスタ別名を使う場合は、Oracle8i サーバであるクラスタ・メンバごとにクラスタ別名を作成して、各別名のルーティングおよびスケジューリング属性を個別に最適化します (クラスタ別名の作成方法についての詳細は、`cluamgr(8)` を参照してください)。

クラスタ別名を使用するときは、`/etc/hosts` ファイルに IP アドレスとクラスタ別名の名前を追加します。

次の行を `/etc/clua_services` ファイルに追加して、Oracle8i リスナが使用するポートの特性をセットアップします。

```
listener 1521/tcp in_single
```

`in_single` 属性を設定すると、クラスタ別名サブシステムが、省略時のクラスタ別名宛の接続要求を別名の 1 メンバに配信するようになります。そのメンバが利用できなくなると、クラスタ別名サブシステムは、省略時のクラスタ別名の別のメンバを選択して、そのメンバがすべての要求を受け取るようにします。

サービスの定義を再ロードするために、すべてのメンバで次のコマンドを実行します。

```
# cluamgr -f
```

3. クライアントから出す Oracle8i サービス要求の宛先としてインタフェース・アドレスを使用する場合は、IP アドレスとクラスタ別名の名前を `/etc/hosts` に追加します。
4. `listener.ora` と `tnsnames.ora` ファイルにある `HOST` フィールドをたとえば次のように編集して、クライアントがサービスへアクセスするときに使用する各クラスタ別名を設定します。

```
.  
. .  
(ADDRESS = (PROTOCOL = TCP) (HOST = alias1) (PORT = 1521))  
. .  
.
```

5. Oracle CAA スクリプトの例が `/var/cluster/caa/examples/Database/oracle.scr` にあります。このスクリプトを `/var/cluster/caa/script/oracle.scr` にコピーして、電子メール・アカウント、ログ・ファイルの出力先、別名の優先順位などを環境に合わせて編集します。スクリプト内にはファイル・システムの参照を含めないでください。
6. スクリプトの初期テストを行います。次の例に示すように、このテストでは、最初に、CAA の外で起動エントリ・ポイントおよび停止エントリ・ポイントを実行します。

```
# cd /var/cluster/caa/script  
# ./oracle.scr start
```

7. SysMan Station を使用するかまたは次のコマンドを実行して、CAA アプリケーション・リソース・プロファイルを作成します。

```
# caa_profile -create oracle -t application \  
-d "ORACLE Single-Instance Service" -p restricted -h "member1 member2"
```

Oracle CAA リソース・プロファイルが `/var/cluster/caa/examples/DataBase/oracle.cap` にあるプロファイル例のようになっていることを確認してください。

8. メンバの 1 つで SysMan Station を使用するかまたは次のコマンドを実行して、`oracle` プロファイルを CAA に登録します。

```
# caa_register oracle
```

9. メンバの 1 つで SysMan Station を使用するかまたは次のコマンドを実行して、`oracle` サービスを起動します。

```
# caa_start oracle
```

2.14.4 CAA を使ってシングル・インスタンスの Informix サーバを作成する

CAA を使って、フェイルオーバー機構を持つシングル・インスタンスの Informix サーバを作成するには、次の手順に従います。

1. Informix のドキュメントにある指示に従って、Informix をインストールして構成します。

Informix では、特定の UNIX グループ (`dba` , `informix`) を作成することが必要です。

2. Informix シングル・サーバの CAA サービスをセットアップする前に、クライアント・アプリケーションにどの方法でサービスへアクセスさせるか決める必要があります。これには、TruCluster Server 製品のクラスタ別名機能を使う方法と、インタフェース (IP) 別名を使う方法があります。クラスタ別名を使う場合は、Informix サーバであるクラスタ・メンバごとにクラスタ別名を作成して、各別名のルーティングおよびスケジューリング属性を個別に最適化します (クラスタ別名の作成方法についての詳細は、`cluamgr(8)` を参照してください)。

クラスタ別名を使用するときは、`/etc/hosts` ファイルに IP アドレスとクラスタ別名の名前を追加します。

次の行を `/etc/clua_services` ファイルに追加して、Informix リスナが使用するポートの特性をセットアップします。

```
informix 8888/tcp in_single
```

`in_single` 属性を設定すると、クラスタ別名サブシステムは、クラスタ別名宛の接続要求を別名の 1 メンバに配信するようになります。そのメンバが利用できなくなると、クラスタ別名サブシステムは、クラスタ別名の別のメンバを選択して、そのメンバがすべての要求を受け取るようにします。

サービスの定義を再ロードするために、すべてのメンバで次のコマンドを実行します。

```
# cluamgr -f
```

3. クライアントから出す Informix サービス要求の宛先としてインタフェース・アドレスを使用する場合は、IP アドレスとクラスタ別名の名前を `/etc/hosts` に追加します。
4. Informix CAA スクリプトの例が `/var/cluster/caa/examples/DataBase/informix.scr` にあります。このスクリプトを `/var/cluster/caa/script/informix.scr` にコピーして、電子メール・アカウント、ログ・ファイルの出力先や別名の優先順位などを環境に合わせて編集します。スクリプト内にはファイル・システムの参照を含めないでください。
5. スクリプトの初期テストを行います。次の例に示すように、このテストでは、最初に、CAA の外で起動エントリ・ポイントおよび停止エントリ・ポイントを実行します。

```
# cd /var/cluster/caa/script
# ./informix.scr start
```

6. SysMan Station を使用するかまたは次のコマンドを実行して、CAA アプリケーション・リソース・プロファイルを作成します。

```
# caa_profile -create informix -t application \
-d "INFORMIX Single-Instance Service" -p restricted -h "member1 member2"
```

Informix CAA リソース・プロファイルが `/var/cluster/caa/examples/DataBase/informix.cap` にあるプロファイル例のようになっていることを確認してください。

7. メンバの 1 つで SysMan Station を使用するかまたは次のコマンドを実行して、`informix` プロファイルを CAA に登録します。

```
# caa_register informix
```

8. メンバの 1 つで SysMan Station を使用するかまたは次のコマンドを実行して、informix サービスを起動します。

```
# caa_start informix
```

マルチ・インスタンス・アプリケーション でのクラスタ別名の使用

クラスタ別名は、クラスタ内の一部またはすべてのシステムを、クライアントにとって個々のシステムとしてではなく、1つのシステムとして見せる IP アドレスです。クラスタは、複数のクラスタ別名を持つことができます。省略時のクラスタ別名には、クラスタのすべてのメンバが含まれていて、すべてのメンバがその別名宛のパケットを受信することができます。

この章では、省略時のクラスタ別名を使って要求をすべてのクラスタ・メンバに分散させる、マルチ・インスタンス・アプリケーションの例を示します。別名属性を修正する方法については、『クラスタ管理ガイド』を参照してください。

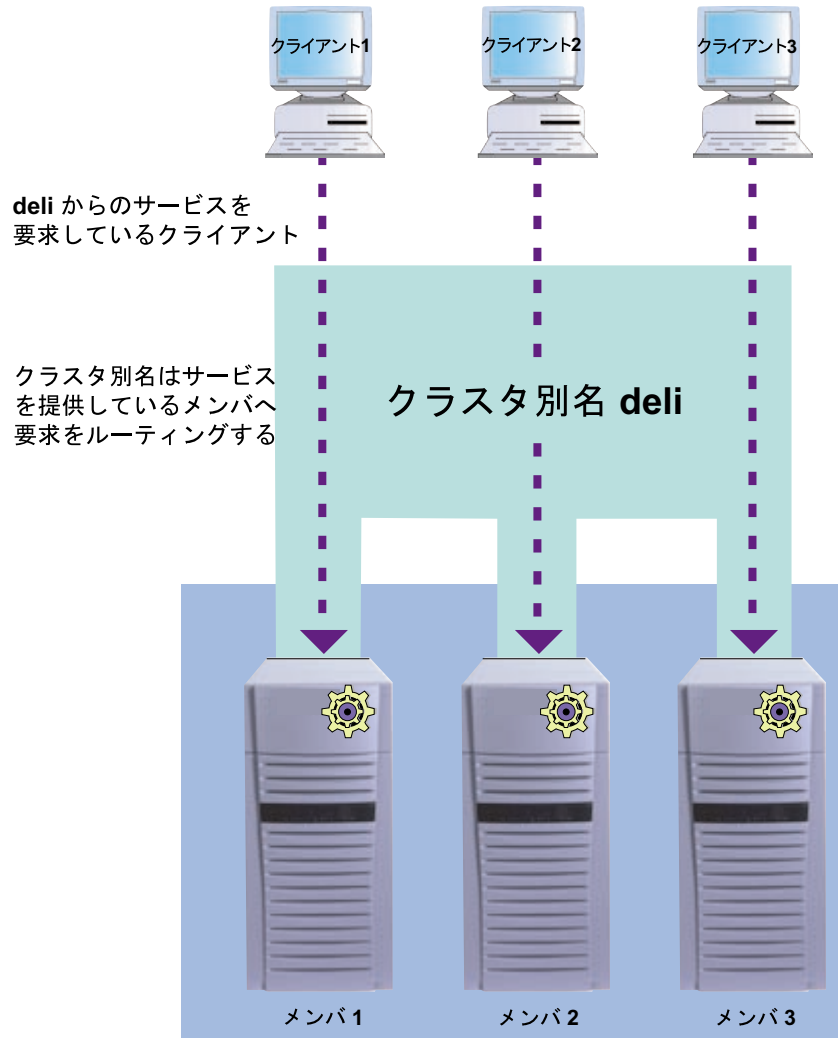
この章で扱う内容は次のとおりです。


- いつクラスタ別名を使うか (3.1 節)
- アプリケーションの例 (3.2 節)

3.1 いつクラスタ別名を使うか

クラスタ別名を使うと、要求やパケットをその別名のメンバに配信することができるため、複数のクラスタ・メンバ上で動作するアプリケーションの場合に最も効果があります。受信パケットや接続要求は、クラスタ別名のすべてのメンバに分散されます。その別名に属するメンバのいずれかで障害が発生した場合、クラスタ別名のソフトウェアは、アプリケーションに関連するトラフィックを、そのクラスタ別名の残りのメンバに透過的にルーティングします。これ以降の新しい要求は、管理者が用意したメトリクスに従ってメンバに配信されるようになります (クラスタ別名の機能概要は、『クラスタ概要』を参照してください)。図 3-1 に、クラスタ別名サブシステムがクライアント要求を分散する方法を示します。

図 3-1: マルチ・インスタンス・アプリケーションへのクラスタ別名を介したアクセス



 マルチ・インスタンス・アプリケーション

ZK-1693U-AI

シングル・インスタンス・アプリケーションでは、アプリケーション制御とフェイルオーバーのために、CAA (Cluster Application Availability) 機能を使います。CAA を使用した場合でも、クラスタ別名サブシステムは、別名宛のパケットをルーティングしますが、CAA では、アプリケーションが 1 つのメンバ上だけで動作するため、クラスタ別名サブシステムは、要求を常にその

3-2 マルチ・インスタンス・アプリケーションでのクラスタ別名の使用

メンバにルーティングします (そのメンバがアプリケーションを実行しているとともに、使用するクラスタ別名のメンバでもあることが必要です)。クラスタ別名サブシステムと CAA の違いの概要については『クラスタ管理ガイド』を参照してください。

3.2 省略時のクラスタ別名を使用してマルチ・インスタンス Apache HTTP サーバにアクセスする

省略時のクラスタ別名を使用し、マルチ・インスタンスの高可用性 Apache HTTP サーバにアクセスして、クラスタのすべてのメンバに要求を分散させる場合は、次の手順に従います。

1. 最新の標準 Apache の配布ソフトウェアを `www.apache.org` からダウンロードします。このサイトの指示に従って、Apache を構築し `/usr/local/apache` ディレクトリにインストールします。
2. `apache/conf/http.conf` 構成ファイルを編集して、KeepAlive パラメータを `off` に設定します。

```
KeepAlive Off
```

KeepAlive パラメータが `on` に設定されている場合、Apache サーバ・デーモン (`httpd`) は、既存の TCP 接続をオープンしたままにして、`KeepAliveTimeout` で何秒定義されていても (省略時の設定では 15 秒)、その間それを再使用します。要求の負荷を要求ごとに分散させたい場合は、これらの接続をオープンしたままにする KeepAlive タイマをオフにする必要があります。

3. Apache ログ・ファイル用のメンバごとのサブディレクトリを、`/usr/local/apache` ディレクトリの下に作成します。
4. そのログ・ディレクトリのコンテキスト依存シンボリック・リンク (CDSL) を作成します。

```
# mkdir -p /usr/local/apache/member1/logs
# mkdir -p /usr/local/apache/member2/logs
```

```
# mkcdsl /usr/local/apache/{memb}/logs /usr/local/apache/logs
```

注意

C シェルを使っている場合は、文字列 `{memb}` の中カッコをエスケープする必要があります。つまり、`\{memb\}` のように記述します。こうしなければ、シェルが中カッコを取り除

いてしまいます。Bourne シェルまたは Korn シェルを使っている場合は、中カッコをエスケープする必要はありません。

5. クラスタ別名サービス・ファイル `/etc/clua_services` に次のエントリを追加します。

```
http 80/tcp in_multi,out_alias
```

このエントリでは、ポート 80 に `in_multi` 属性を設定していますが、これは、クラスタ別名サブシステムが省略時のクラスタ別名宛の接続要求を、その別名のすべてのメンバに分散させることを意味します。

6. サービス定義を再ロードするために、すべてのメンバ上で次のコマンドを実行します。

```
# cluamgr -f
```

7. 各メンバ上で、Apache サーバ・デーモンを起動します。

```
# /usr/local/apache/bin/httpd
```

Part 2

TruCluster Server へのアプリケーション の移行



アプリケーション移行の一般的な問題

この章では、すべてのアプリケーションのタイプにあてはまる移行の問題を採り上げます。表 4-1 に、移行上の問題点、それが発生するアプリケーション、および詳細情報のある節を示します。

表 4-1: アプリケーション移行での検討項目

検討項目	影響を受けるアプリケーション・タイプ	参照する節
クラスタ単位のファイルとメンバ固有のファイル	シングル・インスタンス マルチ・インスタンス 分散型	4.1 節
デバイスの命名規則	シングル・インスタンス マルチ・インスタンス 分散型	4.2 節
プロセス間通信	マルチ・インスタンス 分散型	4.3 節
共用データへの同期アクセス	マルチ・インスタンス 分散型	4.4 節
メンバ固有のリソース	シングル・インスタンス	4.5 節
拡張プロセス ID (PID)	マルチ・インスタンス 分散型	4.6 節
削除された分散ロック・マネージャ (DLM) パラメータ	マルチ・インスタンス 分散型	4.7 節
ライセンス	シングル・インスタンス マルチ・インスタンス 分散型	4.8 節
ブロッキング・レイヤー・プロダクト	シングル・インスタンス マルチ・インスタンス 分散型	4.9 節

4.1 クラスタ単位のファイルとメンバ固有のファイル

クラスタには、次の 2 つの構成データのセットがあります。

- クラスタ単位のリータ

クラスタ単位のリータは、クラスタ内のすべてのメンバで共用することのできるファイルとデータです。たとえば、2つのシステムからなるクラスタの場合、それらのシステムは両方のシステムのユーザの権限についての情報が入った `/etc/passwd` ファイルを共用します。

構成データや管理データを共用することで、ファイル管理が簡素化されます。たとえば、Apache と Netscape それぞれの構成ファイルを共用して、クラスタのすべてのノードのアプリケーションを管理することができます。

- メンバ固有のリータ

メンバ固有のリータは、メンバ固有のリータ・ファイルに格納されています。これらのファイルは、すべてのクラスタ・メンバによって共用することができません。メンバ固有のリータには、個々のシステムだけに該当するハードウェアに関するデータなどがあります。たとえば、1つのクラスタ・メンバに接続された特定のプリンタのレイヤード・プロダクト・ドライバに関するデータなどがこれにあたります。

クラスタ・ファイル・システム (CFS) では、すべてのファイルがすべてのクラスタ・メンバから見えてアクセスできるため、クラスタ単位の構成データが必要なアプリケーションでは、簡単に構成ファイルに書き込むことができ、すべてのメンバでその内容を見ることができます。しかし、メンバ固有の構成情報を使用し保守しなければならないアプリケーションでは、ファイルへの上書きを防ぐために何らかの対処をする必要があります。

ファイルへの上書きを防ぐためには、次の方式の1つを採用することを検討してください。

方式	利点	欠点
単一ファイル	管理しやすい。	アプリケーション側で、単一ファイル上のメンバ固有データにどのようにアクセスするかを意識する必要があります。

方式	利点	欠点
複数ファイル	構成情報をクラスタ単位のファイルのセットに保持する。	ファイルの複数のコピーを保守する必要がある。アプリケーション側で、メンバ固有のファイルにどのようにアクセスするかを意識する必要がある。
コンテキスト依存シンボリック・リンク (CDSL)	構成情報をメンバ固有の領域に保持する。CDSL は、アプリケーションでは意識する必要がなく、シンボリック・リンクのように見える。	ファイルを移動したり名前を変更するとシンボリック・リンクが壊れる。アプリケーションでは、どのように CDSL を扱うか意識する必要がある。CDSL を使用すると、アプリケーションでは、そのクラスタ内の別のインスタンスについての情報を見つけ出すのが難しくなる。

アプリケーションの要件に最もよく合う方法を決定する必要があります。以降の各項では、それぞれの選択肢について詳しく説明していきます。

4.1.1 単一ファイル方式の使用

一意の名前のファイルを 1 つ使い、アプリケーションの構成情報を 1 つのクラスタ単位のファイルに保持します。このファイルには、個々のノードのデータが独立したレコードの形で保存されています。アプリケーションでは、このファイル内の正しいレコードを読み書きします。単一ファイルの管理は、すべてのデータが中央に置かれるため簡単です。

たとえば、クラスタ内の `/etc/printcap` ファイルに、個々のプリンタのエントリを置きます。次のパラメータを指定して、クラスタのどのノードがそのプリンタ・キューのスプーラを実行できるかを示します。

```
:on=nodename1,nodename2,nodename3,...:
```

最初のノードが動作している場合は、そのノードがスプーラを実行します。そのノードが停止すると、次のノードが動作していれば、そのノードがスプーラを実行します。

4.1.2 複数ファイル方式の使用

一意な名前のクラスタ単位のファイルのセットによって、構成情報を保持する方法です。たとえば、各クラスタ・メンバは、それぞれのメンバ固有の

gated 構成ファイルを /etc に保持します。それぞれのメンバ固有の一意のファイル名は、そのメンバ ID を利用した命名規則によって作成します。メンバ固有のファイルを、コンテキスト依存シンボリック・リンク (CDSL) を使用することにより共通の名前によって参照する方法とは異なります。たとえば、次のようなファイル名になります。

```
# ls -l /etc/gated.conf.member*
-rw-r--r-- 1 root system 466 Jun 21 17:37 /etc/gated.conf.member1
-rw-r--r-- 1 root system 466 Jun 21 17:37 /etc/gated.conf.member2
-rw-r--r-- 1 root system 466 Jun 21 13:28 /etc/gated.conf.member3
```

この方式では、ファイルの複数のコピーを保守しなければならないため管理に手間がかかります。たとえば、あるクラスタ・メンバのメンバ ID に変更があった場合、そのメンバにかかわるすべてのメンバ固有のファイルを探し出して、名前を変更する必要があります。また、アプリケーションがメンバ固有のファイルのアクセス方法を意識していない場合は、意識するように構成する必要があります。

4.1.3 CDSL の使用

Tru64 UNIX バージョン 5.0 では、コンテキスト依存シンボリック・リンク (CDSL) と呼ぶ、特殊な形式のシンボリック・リンクを採用していました。TruCluster Server では、これを使用して、各メンバの適切なファイルを指し示します。CDSL は、複数のアプリケーション・インスタンスが、複数のクラスタ・メンバ上で、異なるデータのセットに対して動作している場合に便利です。

CDSL を使用する場合、構成情報はメンバ固有の領域に保存されますが、データは、CDSL を通して参照することができます。各メンバは、共通のファイル名のファイルを読みますが、それは透過的に構成ファイルのメンバ固有のコピーにリンクされています。CDSL は、複数ファイル方式を使用するようにアプリケーションを簡単に変更できない場合に、メンバ固有のファイルを保持するために使う選択肢です。

次の例は、/etc/rc.config ファイルの CDSL 構造です。

```
/etc/rc.config -> ../cluster/members/{memb}/etc/rc.config
```

たとえば、クラスタ・メンバのメンバ ID が 3 の場合、パス名 /cluster/members/{memb}/etc/rc.config は、/cluster/members/member3/etc/rc.config になります。

Tru64 UNIX では `mkcdsl` コマンドを提供しており、システム管理者はこれを使って、CDSL を作成し、CDSL インベントリ・ファイルを更新することができます。このコマンドについての詳細は、TruCluster Server の『クラスタ管理ガイド』および `mkcdsl(8)` を参照してください。CDSL の詳しい作成方法については、Tru64 UNIX の『システム管理ガイド』、`hier(5)`、`ln(1)`、および `symlink(2)` を参照してください。

4.2 デバイスの命名規則

Tru64 UNIX バージョン 5.0 では、新しいデバイス命名規則を採用していました。この命名規則は、デバイスの意味の分かる名前とインスタンス番号からなります。この 2 つの要素が、デバイスのベースネームとなります。この例を、次の表に示します。

<code>/dev</code> の下の場所	デバイス名	インスタンス番号	ベースネーム
<code>./disk</code>	dsk	0	dsk0
<code>./disk</code>	cdrom	1	cdrom1
<code>./tape</code>	tape	0	tape0

ディスクの物理的な接続位置を移動しても、そのディスクのデバイス名は変わりません。したがってシステム管理者は、システム上にデバイスを柔軟に構成できます。詳細は、Tru64 UNIX の『システム管理ガイド』を参照してください。このデバイス命名規則のモデルの説明があります。

Tru64 UNIX では、古いスタイル (`rz`) のデバイスと新しいスタイル (`dsk`) のデバイスの両方のデバイス名を認識しますが、TruCluster Server では、新しいスタイルのデバイス名しか認識しません。古いスタイルのデバイス名に依存するアプリケーションや `/dev` ディレクトリ構造は、新しい命名規則を使用するように変更する必要があります。

ハードウェア管理用の汎用ユーティリティ `hwmgr` を使うと、Tru64 UNIX バージョン 5.1B のインストール後に、デバイス名をそれぞれのバス、ターゲット、および LUN 位置にマップすることができます。たとえば、次のようにこのコマンドを実行してデバイスを表示します。

```
# hwmgr -view devices

HWID: Device Name      Mfg      Model      Location
-----
45: /dev/disk/floppy0c   DEC      3.5in floppy fdi0-unit-0
54: /dev/disk/cdrom0c   DEC      RRD47 (C) DEC bus-0-targ-5-lun-0
55: /dev/disk/dsk0c     COMPAQ   BB00911CA0  bus-1-targ-0-lun-0
```

```

56: /dev/disk/dsk1c    COMPAQ BB00911CA0    bus-1-targ-1-lun-0
57: /dev/disk/dsk2c    DEC    HSG80          IDENTIFIER=7
.
.
.

```

クラスタ単位のデバイスを参照するには、次のコマンドを使用します。

```
# hwmgr -view devices -cluster
```

HWID:	Device Name	Mfg	Model	Hostname	Location
45:	/dev/disk/floppy0c		3.5in floppy	swiss	fdi0-unit-0
54:	/dev/disk/cdrom0c	DEC	RRD47 (C) DEC	swiss	bus-0-targ-5-lun-0
55:	/dev/disk/dsk0c	COMPAQ	BB00911CA0	swiss	bus-1-targ-0-lun-0
56:	/dev/disk/dsk1c	COMPAQ	BB00911CA0	swiss	bus-1-targ-1-lun-0
57:	/dev/disk/dsk2c	DEC	HSG80	swiss	IDENTIFIER=7
.					
.					
.					

このコマンドの使用方法についての詳細は、hwmgr(8) および 『クラスタ管理ガイド』 を参照してください。

新しいスタイルのデバイス命名規則を使用するようにアプリケーションを変更する際には、次の項目を探してください。

- AdvFS (Advanced File System) ドメインに含まれるディスク
- raw ディスク・デバイス
- LSM (Logical Storage Manager) のボリュームにカプセル化されているディスク、またはディスク・グループに属するディスク
- スクリプト内のディスク名
- データ・ファイル (Oracle OPS および Informix XPS) 内のディスク名
- SCSI バスの番号のふり直し

注意

ASE で、SCSI バスの番号をふり直していた場合は、TruCluster Server にアップグレードする際に、物理デバイスとバス番号の対応を注意深く調査する必要があります。詳細は、『クラスタ・インストール・ガイド』を参照してください。

4.3 プロセス間通信

クラスタ単位でサポートされているプロセス間通信 (IPC) メカニズムは、次のとおりです。

- ソケットを使用した TCP/IP 接続
- バッファ I/O またはメモリ・マップ・ファイル
- UNIX ファイル・ロック
- 分散ロック・マネージャ (DLM) ロック
- クラスタ単位のキル・シグナル
- Memory Channel アプリケーション・プログラミング・インタフェース (API) ライブラリ (メモリ・ウィンドウ, 低レベル・ロック, およびシグナル)

クラスタ単位でサポートされていない IPC メカニズムは、次のとおりです。

- UNIX ドメイン・ソケット
- 名前付きパイプ (FIFO 特殊ファイル)
- シグナル
- System V IPC (メッセージ, 共用メモリ, およびセマフォ)

アプリケーションがこれらの IPC 方式のいずれかを使う場合は、シングル・インスタンス・アプリケーションとしての動作に制限されます。

4.4 共用データへの同期アクセス

クラスタ内で実行されている複数のアプリケーション・インスタンスは、お互いに同期をとらなければなりません。これは、マルチプロセス、マルチスレッドのアプリケーションがスタンドアロンのシステム上で同期をとる必要があるのと同じ理由です。しかし、メモリ・ベースの同期メカニズム (クリティカル・セクション, 相互排他, 単純なロック, および複雑なロックなど) は、ローカル・システムでのみ機能し、クラスタ単位では機能しません。共用ファイルのデータでは、同期アクセスができなければなりません。これができない場合、クラスタ全体でインスタンスの実行の同期をとるために、ファイルを使用しなければなりません。

クラスタ・ファイル・システム (CFS) は完全に POSIX に準拠しているため、アプリケーションは `flock()` システム・コールを使ってインスタンス間

で共用するファイルの同期をとることができます。また、分散ロック・マネージャ (DLM) API ライブラリを使うことで、より高性能なロック機能 (追加のロック・モード、ロック変換、デッドロック検出など) を利用することもできます。DLM API ライブラリは TruCluster Server 製品のみで提供されているため、その関数を使用するコードのうち、非クラスタ・システムでも実行する予定のあるコードは、`clu_is_member()` への呼び出しを行うすべての DLM 関数の前に置くようにしてください。`clu_is_member()` 関数は、システムが実際にクラスタ・システムであるかどうかを検証します。このコマンドについての詳細は、`clu_is_member(3)` を参照してください。DLM API についての詳細は、第 9 章を参照してください。

4.5 メンバ固有のリソース

複数のアプリケーション・インスタンスを複数のクラスタ・メンバで同時に起動した場合、アプリケーションのインスタンスが正しく動作しないことがあります。これは、大規模な CPU 処理能力や大容量物理メモリなど、特定のメンバだけが利用できるリソースに依存する場合があるためです。これによって、アプリケーションはクラスタ内でシングル・インスタンスとしてしか動作できなくなる場合があります。アプリケーションのこのような特性を変更するだけで、クラスタ内でマルチ・インスタンスとして実行できる場合があります。すなわち、複数のメンバにリソースが存在する場合は、これらのメンバ上でアプリケーションを実行するだけです。

4.6 拡張 PID

TruCluster Server では、プロセス識別子 (PID) が完全な 32 ビット値に拡張されました。データ・タイプ `PID_MAX` は、2147483647 (0x7fffffff) に拡大されました。このため、`PID <= PID_MAX` という条件判定をしているアプリケーションは、再コンパイルする必要があります。

PID はクラスタ全体で一意的な番号です。各クラスタ・メンバの PID は、メンバ ID をベースとしており、そのメンバにとって一意で一定範囲の番号が割り当てられます。クラスタ内で利用できる PID は次の式に従います。

$$\text{PID} = (\text{memberid} * (2^{19})) + 2$$

通常、最初の 2 つの値は `kernel idle` プロセスと `/sbin/init` 用に予約されています。たとえば、PID 524,288 と 524,289 は `memberid` が 1 のクラスタ・メンバ上にある `kernel idle` プロセスと `init` にそれぞれ割り当てられます。

ログ・ファイルと一時ファイルを一意に識別するのに PID を使います。アプリケーションでファイルに PID を保存する場合は、そのファイルをメンバー固有のファイルとする必要があります。

4.7 削除された DLM パラメータ

分散ロック・マネージャ (DLM) の不変リソース、リソース・グループ、およびトランザクション ID は、TruCluster Available Server、TruCluster Production Server Version 1.6、および TruCluster Server バージョン 5.0 以降では、省略時の設定により有効にされるため、`dml_disable_rd` および `dml_disable_grptx` 属性は必要でなく、DLM カーネル・サブシステムから削除されました。

4.8 ライセンス

この節では、ライセンスの制約と問題点について説明します。

4.8.1 TruCluster Server のクラスタ単位でのライセンスはサポートされない

TruCluster Server バージョン 5.1B は、クラスタ単位のライセンスをサポートしていません。クラスタにメンバを追加するたびに、そのメンバ上で実行するアプリケーションのすべての必要なライセンスをそのメンバに登録する必要があります。

4.8.2 レイヤード・プロダクトのライセンスとネットワーク・アダプタのフェイルオーバー

NetRAIN (Redundant Array of Independent Network Adapter) および NIFF (Network Interface Failure Finder) は、ネットワーク・フェイルオーバーを容易に行うためのメカニズムを提供し、TruCluster Available Server および Production Server 製品で採用されていたネットワーク・インタフェース監視機能はこれに置き換わります。

NetRAIN では、複数アダプタ構成での透過的なネットワーク・アダプタ・フェイルオーバーを提供します。NetRAIN は、自身のネットワーク・インタフェースの状態を NIFF を使って監視します。NIFF は、ネットワークの障害を検出して報告します。NIFF を使うことにより、複合 NetRAIN デバイスなどのネットワーク・デバイスに障害が起こったときにイベントを生成することができます。そうしたイベントを監視すれば、障害が起こったときに

適切な処置をとることができます。NetRAIN と NIFF についての詳細は、Tru64 UNIX の『ネットワーク管理ガイド：接続編』を参照してください。

クラスタ内では、アプリケーションはフェイルオーバー機構によって他のメンバで再起動することができます。再起動する際にアプリケーションがライセンス・チェックを行う場合、特定のメンバの IP アドレス、またはアダプタのメディア・アクセス制御 (MAC) アドレスをチェックするため、チェックに失敗することがあります。

ネットワーク・アダプタの MAC アドレスを使って、マシンを一意に識別するライセンス機構では、NetRAIN がどのように MAC アドレスを変換するかに左右されます。すべてのネットワーク・ドライバは、MAC アドレスをインタフェースから取り出す `SIOCRPHYSADDR ioctl` をサポートしています。この `ioctl` は、次の 2 つのアドレスを配列内に返します。

- 省略時のハードウェア・アドレス — それぞれの LAN アダプタに実装されている小さな PROM から取り出したパーマネント・アドレス
- 現在の物理アドレス — ネットワーク・インタフェースが回線上の要求に応答するアドレス

MAC アドレスに基づくライセンス機構では、`SIOCRPHYSADDR ioctl` で返される省略時のハードウェア・アドレスを使う必要があります。現在の物理アドレスは、NetRAIN がそのときに応じて変更しているので使用してはなりません。`SIOCRPHYSADDR ioctl` のサンプル・プログラムが、使用しているネットワーク・アダプタのリファレンス・ページ (たとえば、`tu(7)`) にあるので参照してください。

4.9 ブロッキング・レイヤード・プロダクト

移行したいアプリケーションがブロッキング・レイヤード・プロダクトかどうか確認してください。ブロッキング・レイヤード・プロダクトは、コマンド `installupdate` を実行して、TruCluster Server バージョン 5.1B のアップデート・インストレーションを行う際に弊害となる製品です。コマンド `installupdate` を使用してローリング・アップグレードを開始する前に、ブロッキング・レイヤード・プロダクトをクラスタから削除する必要があります。

レイヤード・プロダクトのドキュメントで、最初にローリングするメンバ上にプロダクトの新バージョンをインストールできること、および複数のバージョンが混在するクラスタ内でレイヤード・プロダクトが動作できることが

特記されていない場合は、新しいレイヤード・プロダクトまたは現在インストール済みのレイヤード・プロダクトの新バージョンのどちらもローリング・アップグレード中にインストールしないでください。

TruCluster Server バージョン 5.1B でアップデート・インストールを中断することが分かっているレイヤード・プロダクトは、TruCluster Server 『クラスタ・インストール・ガイド』 にリストされています。



ASE アプリケーションの TruCluster Server への移行

この章では、ASE (Available Server Environment) アプリケーションを TruCluster Server バージョン 5.1B に移行する方法を説明します。

シングル・インスタンス・アプリケーションの可用性とフェイルオーバー機構を維持するために、TruCluster Server では、CAA (Cluster Application Availability) サブシステムを提供します。TruCluster Server では、以前の TruCluster ソフトウェア製品でアプリケーションの高可用性機能を実現するために提供されていた ASE (Available Server Environment) ではなく、CAA を使用します。ただし、ASE の場合と違って、TruCluster Server のクラスタではストレージ・リソースを明示的に管理したり、高可用性アプリケーションに代わってファイル・システムをマウントしたりしてはなりません。クラスタ・ファイル・システム (CFS) とデバイス要求ディスクパッチャで、ファイルとディスク・ストレージがクラスタ単位で利用できるようにします。

ASE の各種サービスを TruCluster Server に移行する前に、CAA について理解しておく必要があります。CAA の使用方法の詳細は、第 2 章を参照してください。

この章では、次の項目について説明します。

- ASE と CAA の違い (5.1 節)
- ASE サービスから TruCluster Server への移行の準備 (5.2 節)
- ASE スクリプトの見直し (5.3 節)
- IP 別名の使用とネットワーク・サービス (5.4 節)
- ファイル・システムのパーティショニング (5.5 節)

5.1 ASE と CAA の違い

CAA には、リソースの監視機能とアプリケーションの再起動機能があります。CAA では、TruCluster Available Server Software および TruCluster

Production Server Software 製品のユーザ定義サービスと同じ種類のアプリケーション可用性が得られます。表 5-1 に、ASE サービスとそれに対応する TruCluster Server 製品の機能を対比して示します。

表 5-1: ASE サービスとそれに対応する TruCluster Server の機能

ASE サービス	ASE の説明	TruCluster Server の対応する機能
ディスク・サービス (5.1.1 項)	1 つ以上の高可用性ファイル・システム, AdvFS (Advanced File System) ファイルセット, または LSM (Logical Storage Manager) ボリューム。ディスク・ベースのアプリケーションも含まれる。	クラスタ・ファイル・システム (CFS), デバイス要求ディスパッチャ, および CAA。
ネットワーク・ファイル・システム (NFS) サービス (5.1.2 項)	1 つ以上の高可用性ファイル・システム, AdvFS ファイルセット, またはエクスポートされた LSM ボリューム。高可用性アプリケーションも含まれる。	CFS と省略時のクラスタ名により, エクスポートされたファイル・システムに対し自動的に提供される。サービス定義は必要ない。
ユーザ定義サービス (5.1.3 項)	処理スクリプトを使ってフェイルオーバーさせるアプリケーション。	CAA
DRD (Distributed Raw Disk) サービス (5.1.4 項)	raw 物理ディスクへのクラスタ単位のアクセスによって, ディスク・ベースのユーザ・レベル・アプリケーションをクラスタ内で動作させる機能。	デバイス要求ディスパッチャによって自動的に提供される。サービス定義は必要ない。
テープ・サービス (5.1.5 項)	NetWorker サーバや他のサーバのフェイルオーバーを構成するのに, 1 つ以上のテープ装置のセットに依存する。	CFS, デバイス要求ディスパッチャ, および CAA

以降の各項では、これらの ASE サービスについて説明し、これらを TruCluster Server 環境でどのように扱うかを示します。

5.1.1 ディスク・サービス

ASE

ASE ディスク・サービスには、1 つ以上の高可用性ファイル・システム, AdvFS (Advanced File System) ファイルセット, または LSM (Logical

5-2 ASE アプリケーションの TruCluster Server への移行

Storage Manager) ポリリュームが含まれます。ディスク・サービスにディスク・ベースのアプリケーションを含めて、ASE 内で管理することもできます。

TruCluster Server

TruCluster Server には、明示的なディスク・サービスはありません。クラスタ・ファイル・システム (CFS) を使用すると、すべてのファイル・ストレージをすべてのクラスタ・メンバから利用できます。また、デバイス要求ディスクパッチャにより、ディスク・ストレージがクラスタ単位で利用できるようになります。ファイル・システムとディスクがクラスタ全体で利用可能になるので、処理スクリプトの中で明示的にそれらをマウントしたりフェイルオーバーさせる必要はありません。CFS の使用方法についての詳細は、『クラスタ管理ガイド』および `cfsmgr(8)` を参照してください。

CAA を使用して、ディスク・サービスの再配置ポリシーと依存関係を定義します。CAA に慣れていない場合は、第 2 章を参照してください。

ディスク・サービスでは、クライアントからのアクセスのために、クラスタ別名と IP 別名のいずれかを使用するように定義することができます。

5.1.2 NFS サービス

ASE

ASE のネットワーク・ファイル・システム (NFS) サービスには、1 つ以上の高可用性ファイル・システム、AdvFS ファイルセット、またはデータの高可用性を実現するためにメンバ・システムがクライアントにエクスポートした LSM ポリリュームが含まれます。また、NFS サーバには高可用性アプリケーションを含めることもできます。

TruCluster Server

TruCluster Server には明示的な NFS サービスはありません。NFS サーバとして構成すると、TruCluster Server のクラスタは、エクスポートしたファイル・システムに対して高可用性アクセスを提供します。CFS では、すべてのファイル・ストレージはすべてのクラスタ・メンバから利用できます。ファイル・システムを、処理スクリプトの中でマウントする必要はありません。スタンドアロン・サーバで定義するのと同様に、サービスされる NFS ファイル・システムを `/etc/exports` ファイル内に定義します。

リモート・クライアントでは、クラスタからエクスポートされた NFS ファイル・システムを、省略時のクラスタ別名または代替クラスタ別名を使ってマウントすることができます。exports.aliases(4) を参照してください。

5.1.3 ユーザ定義サービス

ASE

ASE のユーザ定義サービスは、処理スクリプトを使ってフェイルオーバーさせたいアプリケーションだけからなります。ユーザ定義サービス内のアプリケーションでは、ディスクを使うことができません。

TruCluster Server

ASE では、高可用性インターネット・ログイン・サービスは、ifconfig を発行するユーザ定義の起動スクリプトと停止スクリプトを設定することにより実現していました。TruCluster Server では、ログイン・サービスを作成する必要はありません。クライアントは、省略時のクラスタ別名を使ってクラスタにログインすることができます。CFS により、ディスク・アクセスがすべてのクラスタ・メンバで利用できるようになります。

CAA を使用して、ユーザ定義サービスのフェイルオーバー・ポリシー、再配置ポリシーおよび依存関係を定義します。CAA に慣れていない場合は、第 2 章を参照してください。

5.1.4 DRD サービス

ASE

ASE の DRD (Distributed Raw Disk) サービスは、raw 物理ディスクへのクラスタ単位のアクセスを提供します。ディスク・ベースのユーザ・レベル・アプリケーションは、依存している物理ストレージがクラスタ内のどこにあるかに関係なく、クラスタ内で動作することができます。DRD サービスを使用すると、データベースやトランザクション処理 (TP) 監視システムなどのアプリケーションが、複数のクラスタ・メンバからストレージ媒体に並列にアクセスできるようになります。DRD サービスを作成するには、このサービスをクラスタ単位で利用できるようにする物理媒体を指定する必要があります。

TruCluster Server

TruCluster Server には明示的な DRD サービスはありません。デバイス要求ディスク・サブシステムを使用すると、すべてのディスク・ストレージとテープ・ストレージは、物理ストレージがどこにあってもすべてのクラスタ・メンバから利用できます。アプリケーションを別のメンバにフェイルオーバーさせるときに、ディスクを明示的にフェイルオーバーさせる必要はありません。

Tru64 UNIX バージョン 5.0 より前では、TruCluster Production Server 環境に、DRD 用のネームスペースが別に用意されていました。DRD サービスが追加されるたびに、`asemgr` ユーティリティが DRD スペシャル・ファイル名を以下の形式で順に割り当てていました。

```
/dev/rdrd/drd1  
/dev/rdrd/drd2  
/dev/rdrd/drd3  
⋮
```

TruCluster Server クラスタでは、Tru64 UNIX バージョン 5.0 以降のスタンドアロン・システムで行うのと同じ方法で、TruCluster Server 構成の raw ディスク・デバイスのパーティションにアクセスします。つまり、`/dev/rdisk` ディレクトリ内の次の例のようなデバイス・スペシャル・ファイル名を使ってアクセスします。

```
/dev/rdisk/dsk2c
```

5.1.5 テープ・サービス

ASE

ASE のテープ・サービスは、1 つ以上のテープ装置のセットに依存します。これには、メディア・チェンジャ装置やファイル・システムも含まれます。テープ・サービスを使用すると、Legato NetWorker サーバや他のクライアント/サーバ・ベースのアプリケーションのサーバを、フェイルオーバーするように構成することができます。テープ装置、メディア・チェンジャおよびファイル・システムは、1 つのユニットとしてフェイルオーバーされます。

TruCluster Server

TruCluster Server には明示的なテープ・サービスはありません。CFS によって、すべてのファイル・ストレージがすべてのクラスタ・メンバから利用できるようになります。デバイス要求ディスクパッチャは、ディスク・ストレージとテープ・ストレージをクラスタ単位で利用できるようにします。ファイル・システム、ディスク、およびテープがクラスタ全体で利用可能になるので、処理スクリプト内で明示的にそれらをマウントしたりフェイルオーバーさせる必要はありません。

CAA を使用して、テープ・リソースのフェイルオーバー・ポリシー、再配置ポリシーおよび依存関係を定義します。CAA に慣れていない場合は、第 2 章を参照してください。

テープ装置やメディア・チェンジャにアクセスするアプリケーションは、クライアントからアクセスできるように、クラスタ別名または IP 別名を使用するように定義することができます。

5.2 ASE サービスから TruCluster Server への移行の準備

TruCluster Server バージョン 5.1B には、ASE (Available Server Environment) から新しいクラスタヘストレージを移動するために使用できる次のスクリプトが用意されています。

- `clu_migrate_check`
- `clu_migrate_save`
- `clu_migrate_configure`

スクリプトおよび関連するユーティリティ・プログラムは、「Tru64 UNIX Associated Products Volume 2 CD-ROM」の TruCluster Server バージョン 5.1B ディレクトリにある TCRMIGRATE540 サブセットから入手できます。スクリプトおよびインストール手順についての説明は『クラスタ・インストール・ガイド』を参照してください。

一般に、そして可能な場合には、『クラスタ・インストール・ガイド』で説明されている手順およびスクリプトを使用することを推奨します。ただし、ストレージ・トポロジ、システム構成、またはサイトのポリシーにより使用できない場合は、手作業で ASE ストレージ情報を収集して構成することができます。古いスタイル (rz*) のデバイス名を新しいスタイル (dsk*) デバイス名にマッピングするのはユーザの責任です。デバイス

およびストレージ情報を手作業で収集し、新しい Tru64 UNIX システム上でストレージを構成する方法については、『クラスタ・インストール・ガイド』を参照してください。

ストレージ情報を手作業で収集し、新しい Tru64 UNIX システム上でストレージを構成することを決めた場合には、ASE クラスタをシャットダウンする前に、`var/ase/config/asecdb` データベースとそのテキスト・コピーの両方を保存しておきます。ASE データベースの内容を利用できるようにしておくことで、TruCluster Server でのアプリケーションの設定が簡単になります。

ASE データベースの内容がどのように保存されるかは、TruCluster Available Server と TruCluster Production Server のバージョンによって異なります。以降の各項では、バージョン 1.5 以降のシステムおよびバージョン 1.4 以前のシステムで、ASE データベースの内容を保存する方法を説明します。

5.2.1 TruCluster Available Server および Production Server のバージョン 1.5 以降での ASE データベースの内容の保存

`/var/ase/config/asecdb` データベースおよびそのテキスト・コピーの両方を保存するには、次のコマンドを実行します。

```
# cp /var/ase/config/asecdb asecdm.copy
# asemgr -d -C > asecdm.txt
```

サンプルの ASE データベースで保存される次の情報は、CAA プロファイルを作成する際に役に立ちます。

```
!! ASE service configuration for netscape

@startService netscape
Service name: netscape
Service type: DISK
Relocate on boot of favored member: no
Placement policy: balanced
:
```

サンプルの ASE データベースで保存される次の情報は、TruCluster Server 上でアプリケーションをインストールして構成する際に役に立ちます。

```
IP address: 16.141.8.239
Device: cludemo#netscape
  cludemo#netscape mount point: /clumig/Netscape
  cludemo#netscape filesystem type: advfs
  cludemo#netscape mount options: rw
  cludemo#netscape mount point group owner: staff
Device: cludemo#cludemo
  cludemo#cludemo mount point: /clumig/cludemo
  cludemo#cludemo filesystem type: advfs
  cludemo#cludemo mount options: rw
```

```
cludemo#cludemo mount point group owner: staff
AdvFS domain: cludemo
cludemo volumes: /dev/rz12c
:
```

5.2.2 TruCluster Available Server および Production Server のバージョン 1.4 以前での ASE データベースの内容の保存

TruCluster Available Server または Production Server バージョン 1.4 以前のシステムでは、すべての ASE サービスの情報を保存するのに `asemgr` コマンドを使用することはできません。`asemgr` コマンドでは、ASE スクリプトの情報は取得できません。すべての情報を保存したい場合は、`asemgr` ユーティリティを使用する必要があります。

スクリプトのデータを保存するには、次の手順を実行します。

1. `asemgr` ユーティリティを起動します。
2. [ASE Main] メニューで、[Managing ASE Services] を選択します。
3. [Managing ASE Services] メニューで、[Service Configuration] を選択します。
4. [Service Configuration] メニューで、[Modify a Service] を選択します。
5. このメニューからサービスを選択します。
6. [General] サービス情報を選択します。
7. [User-defined Service Modification] メニューから [User-defined] の処理スクリプトを選択します。
8. このメニューから [Start] アクションを選択します。スクリプトの引数とスクリプトのタイムアウトの値を記録します。
9. このメニューから起動処理スクリプトの [Edit] を選択します。
内部スクリプトを、削除されないストレージ上のファイルに書き込みます。

必要に応じて、上記の手順をすべての、停止、追加、削除スクリプトについても実行します。ユーザ定義サービスでは、チェック・スクリプトも保存します。

ASE データベースの内容とその他の ASE サービス情報 (配置ポリシ , サービス名など) を保存するには , 次のコマンドを実行します。

```
# asemgr -dv > ase.services.txt
# asemgr -dv {ServiceName}
```

サービス名 *ServiceName* には , asemgr -dv で出力されたものを使います。各サービスについて , asemgr -dv {*ServiceName*} を実行します。

注意

バージョン 1.5 より前の TruCluster Available Server Software または TruCluster Production Server Software については , Tru64 UNIX バージョン 5.1B および TruCluster Server バージョン 5.1B のフル・インストールを実行する必要があります。

5.3 ASE スクリプトの検討項目

ASE スクリプトを慎重に見直してください。TruCluster Server 上でスクリプトを正しく動作させるためには , 次のことについて検討する必要があります。

- ASE コマンドを CAA (Cluster Application Availability) コマンドに置き換える (5.3.1 項)。
- 起動スクリプトと停止スクリプトを結合する (5.3.2 項)。
- スクリプトの出力をリダイレクトする (5.3.3 項)。
- nfs_config を ifconfig に置き換えるか , クラスタ別名を作成する (5.3.4 項)。
- エラーを適切に処理する (5.3.5 項)。
- 処理スクリプトからストレージ管理情報を削除する (5.3.6 項)。
- デバイス名を変換する (5.3.7 項)。
- ASE 固有の環境変数の参照を削除する (5.3.8 項)。
- 終了コードについて (5.3.9 項)。
- EVM (イベント・マネージャ) を使ってイベントをポストする (5.3.10 項)。

5.3.1 ASE コマンドの CAA コマンドへの置き換え

TruCluster Server バージョン 5.1B では、`asemgr` コマンドは、いくつかの CAA コマンドに置き換えられます。次の表に、ASE コマンドと CAA コマンドを対比して示します。

ASE コマンド	CAA コマンド	説明
<code>asemgr -d</code>	<code>caa_stat</code>	CAA リソースのクラスタ単位の状態を表示する。
<code>asemgr -m</code>	<code>caa_relocate</code>	あるクラスタ・メンバから別のメンバにアプリケーション・リソースを再配置する。
<code>asemgr -s</code>	<code>caa_start</code>	アプリケーション・リソースを起動する。
<code>asemgr -x</code>	<code>caa_stop</code>	アプリケーション・リソースを停止する。
	<code>caa_profile</code>	CAA リソース・プロファイルを作成、検証、削除、更新する。
	<code>caa_register</code>	CAA にリソースを登録する。
	<code>caa_unregister</code>	CAA へのリソースの登録を抹消する。
	<code>caa_balance</code>	リソースの状態に基づいてアプリケーションを最適に再配置する。
	<code>caa_report</code>	アプリケーション・リソースの可用性に関する統計をレポートする。

`caa_profile`、`caa_register`、`caa_unregister`、`caa_balance`、および `caa_report` の各コマンドは、TruCluster Server 製品のみで提供されている機能です。これらの CAA コマンドの使用方法は、第 2 章を参照してください。

5.3.2 起動スクリプトと停止スクリプトの結合

CAA では、アプリケーションの起動と停止に別々のスクリプトを呼び出しません。アプリケーションの起動と停止に別々のスクリプトを使っている場合は、1 つに結合してください。使用例については、`/var/cluster/caa/template/template.scr` を参照してください。

5.3.3 スクリプトの出力のリダイレクト

CAA スクリプトは、実行時に標準出力および標準エラー・ストリームを `/dev/null` にリダイレクトします。これらのストリームをキャプチャしたい場合には、次のいずれかの方法を使用してください。次に説明する方法は、推奨順に記述しています。

1. EVM (イベント・マネージャ) の使用 (テンプレート・スクリプト `/var/cluster/caa/template/template.scr` で示すとおり)。これは、EVM による出力管理であるため、推奨する方法です。
EVM を使用して出力をリダイレクトする例については、`/var/cluster/caa/examples` にある CAA スクリプト例を参照してください。
2. `logger` コマンドを使用して、出力をシステム・ログ・ファイル (`syslog`) へリダイレクト。詳細については、`logger(1)` を参照してください。この方法は、EVM ほど柔軟に使用することはできません。たとえば、`syslog` に格納されたメッセージはシンプル・テキストで、EVM の高度なフォーマット機能や検索機能を利用することはできません。
3. 出力を `/dev/console` へリダイレクト。この方法では、記録としては何も残らず、メッセージがコンソールに表示されるだけです。
4. 出力をファイルへリダイレクト。この方法の場合は、ログ・ファイルのサイズに注意して、ファイル・スペースを適切に管理してください。

5.3.4 `nfs_ifconfig` スクリプトの置き換え

TruCluster Server には、TruCluster ASE のような `nfs_ifconfig` スクリプトは含まれていません。CAA の処理スクリプト内で `nfs_ifconfig` を `ifconfig alias/-alias` 文に置き換えるか、クラスタ別名を使用してください。

CAA スクリプトでインタフェース別名を使う方法についての詳細は、5.4.1 項を参照してください。CAA シングル・インスタンス・アプリケーションでクラスタ別名を使う方法については、2.14 節の例を参照してください。

5.3.5 適切なエラー処理

TruCluster Server 内のスクリプトでエラーが適切に処理されることを確認してください。「`filesystem busy`」というメッセージは、返されなくなり

ました。したがって、アプリケーションのプロセスの一部が別のメンバ上でアクティブであっても、そのアプリケーションがもう一度起動されることがあります。

アプリケーションが別のノードで起動するのを防ぐには、停止スクリプトですべてのプロセスが停止できることを確認するか、または `fuser(8)` を使用してアプリケーション・プロセスを停止してください。

次の例は、アプリケーションの処理スクリプトに追加された、`fuser` ユーティリティを使用するシェル・ルーチンを示しています。このシェル・ルーチンは、アプリケーション・ディレクトリ `/AppDir`、`/AppDir2`、および `/AppDir3` のオープンされているすべてのファイルをクローズしようとし、ファイルをクローズできない場合、ルーチンはエラーを指定して戻り、スクリプトはエラーで終了し、ユーザによる介入が必要であることを示すシグナルを送信します。

```
FUSER="/usr/sbin/fuser" # Command to use for closing
ADVFS_DIRS="/AppDir /AppDir2 /AppDir3" # Application directories
#
# Close open files on shared disks
#
closefiles () {
    echo "Killing processes"
    for i in ${ADVFS_DIRS}
    do
        echo "Killing processes on $i"
        $FUSER -ck $i
        $FUSER -uv $i > /dev/null 2>&1
        if [ $? -ne 0 ]; then
            echo "Retrying to close files on ${i} ..."
            $FUSER -ck $i
            $FUSER -uv $i > /dev/null 2>&1
            if [ $? -ne 0 ]; then
                echo "Failed to close files on ${i} aborting"
                $FUSER -uv $i
                return 2
            fi
        fi
    done
    echo "Processes on ${ADVFS_DIRS} stopped"
}
```

5.3.6 ストレージ管理情報の削除

ASE サービスのストレージは次の条件を満たす必要があります。

- すべてのクラスタ・メンバで共用されているバス上にある。
- `asemgr` ユーティリティを使用するサービスの一部として定義されている。
- サービス・スクリプトで管理されている。

TruCluster Server のクラスタ・ファイル・システム (CFS) では、すべてのファイル・ストレージを、すべてのクラスタ・メンバが使うことができます (ストレージへのアクセスは、クラスタのアーキテクチャに組み込まれています)。このため、処理スクリプト内で、ファイル・システムのマウントやフェイルオーバを扱う必要はありません。

TruCluster Server では、すべてのストレージ管理情報をスクリプトから削除することができます。たとえば、SAP R/3 のスクリプトでは、スクリプト内でファイル・システムをマウントするように設定されていますが、それらのマウント・ポイントを削除することができます。

5.3.7 デバイス名の変換

4.2 節で説明したとおり、古いスタイルのデバイス名を使用しているスクリプトは、Tru64 UNIX バージョン 5.0 で導入された新しいスタイルのデバイス名を使用するように修正しなければなりません。

アップグレード中、`clu_migrate_save` および `clu_migrate_configure` スクリプトは、デバイス名をマッピングし、新しいシステム上でストレージを構成するための情報を収集します。

`clu_migrate_save` および `clu_migrate_configure` スクリプトを使用していない場合には、古いスタイル (`rz*`) のデバイス名を新しいスタイル (`dsk*`) のデバイス名に手動でマッピングする必要があります。クラスタのアップグレード時に手動でストレージを構成する方法については、『クラスタ・インストール・ガイド』を参照してください。

バスの番号を振り直すために `ase_fix_config` コマンドを使っていた場合は、アップグレード時にそのコマンドの出力を保存し、物理装置とバス番号の対応を検証する必要があります。

5.3.8 ASE 変数の置き換えまたは削除

ASE スクリプトでは、次の ASE 環境変数が使用されている場合があります。

- `MEMBER_STATE`

ASE では、`MEMBER_STATE` 変数を停止スクリプト内に置き、そのスクリプトが、動作中のシステムと、ブート中のシステムのどちらで実行されているかを判断していました。`MEMBER_STATE` 変数の値は、次のどちらかです。

- RUNNING

- BOOTING

TruCluster Server では、システムの起動時に、処理スクリプトの停止セクションを実行するオプションはありません。参照やログ・ファイルなどをクリーンアップするためには、これらのクリーンアップ処理を処理スクリプトの起動セクションに移動してください。

- ASEROUTING

ASEROUTING 変数は存在しません。この変数は、TruCluster Server のクラスタ別名サブシステムの機能に置き換えられました。この変数は、TruCluster Server アプリケーションの処理スクリプトから削除してください。

- ASE_PARTIAL_MIRRORING

ASE_PARTIAL_MIRRORING 変数は、TruCluster Server にはありません。この変数は、TruCluster Server アプリケーションの処理スクリプトから削除してください。

5.3.9 終了コード

すべてのスクリプトの ASE 終了コードでは、成功で 0、失敗で 0 以外を返していました。

CAA スクリプトの各エントリ・ポイントでは、成功で 0、失敗で 0 以外の終了コードを返します (caa_profile コマンドでスクリプト・テンプレートから生成したスクリプトは、失敗のときに 2 を返します)。CAA スクリプトの check セクションでは、終了コードの 0 は、アプリケーションが動作していることを示します。

5.3.10 イベントのポスト

EVM (イベント・マネージャ) では、複数のチャネル (ログ・ファイルなど) に対して単一ポイントのフォーカスを提供します。システムの構成要素は、この単一ポイントを経由してイベントおよび状態情報をレポートします。EVM は、これらのイベントを単一のイベント・ストリームに結合し、システム管理者がリアル・タイムでモニタしたり、ストレージから検索する過去のイベントとして参照したりするようにします。evmpost(1) コマンドを使って、処理スクリプトから EVM にイベントをポストします。

注意

`/var/cluster/caa/examples` ディレクトリにある CAA のスクリプト例はすべて、EVM を使ってイベントをポストしています。いずれかの例を参照してください。

5.4 ネットワークの検討項目

以降の各項では、ASE サービスを TruCluster Server に移行する時期について、ネットワークの検討項目を示します。

- 別名の使用 (5.4.1 項)
- ネットワーク・サービス (5.4.2 項)

5.4.1 別名の使用

アプリケーションで別名を使う必要がある場合は、インタフェース別名とクラスタ別名のいずれかを使用します。

クラスタ別名の使用は次の場合にもっとも適しています。

- TCP (Transport Control Protocol) ベースまたは UDP (User Datagram Protocol) ベースのアプリケーションにおいて、複数のメンバ・システムを 1 つのシステムであるかのようにクライアントに見せる。アプリケーションの複数のインスタンスが、いくつかのクラスタ・メンバでアクティブになっていることがよくある。そのような場合にクラスタ別名を使えば、そのアプリケーションを動作させているそれらのメンバに、簡単に信頼性が高く、しかも透過的な方法で、クライアントを接続することができる。
- クラスタ別名の機能を利用して、クライアント・ネットワークの可用性を透過的に処理する。

インタフェース別名の使用は次の場合に適してきます。

- サービスをシングル・インスタンスで動作させ、常に 1 つのメンバがすべてのクライアント要求を処理するようにする。
- 性能条件が厳しい場合。つまり、クライアント要求のサービスを 1 つのメンバに行わせ、余分なルーティング・ホップが発生しないようにしたい場合。

- NetRAIN (Redundant Array of Independent Network Adapters) インタフェースを使い、アプリケーションの CAA プロファイルにあるクライアント・ネットワーク・リソースの依存関係を設定することにより、サービスを提供できるそれぞれのクラスタ・メンバ上でクライアント・ネットワークの可用性を実現できる場合。

5.4.1.1 クラスタ別名

クラスタ別名を使って、マルチ・インスタンスのネットワーク・サービスへのクライアント・アクセスを提供する必要があります。TruCluster Server のクラスタ別名サブシステムでは、クラスタ単位でクラスタ別名を作成して管理するために、`ifconfig` コマンドでインタフェース別名を明示的に確立したり削除したりする必要はありません。マルチ・インスタンス・アプリケーションでクラスタ別名を使用する方法については、第 3 章を参照してください。クラスタ別名の一般的な使用法については、『クラスタ管理ガイド』を参照してください。

クラスタ別名を使うことにより、クライアントのトラフィックを、Oracle8i シングル・サーバのようなシングル・インスタンス・アプリケーションが動作している特定のクラスタ・メンバへ集中させることができます。クラスタ別名の下でサービスを構成し `in_single` クラスタ別名属性を設定すると、その別名に宛てたクライアントからのすべての要求は、別名サブシステムによって、対応するサービスの動作しているクラスタ・メンバへ送られるようになります (そのクラスタ・メンバが利用可能であることが条件です)。ただし、シングル・インスタンス・アプリケーションについては、CAA を使ってアプリケーションのフェイルオーバーをより制御しやすくすることと柔軟性を向上させることを検討してください。CAA の使用方法については、第 2 章を参照してください。

TruCluster Server クラスタでは、クラスタでエクスポートされた NFS ファイル・システムにクライアントがアクセスできるように、クラスタで NFS サービスを定義する必要はありませんが、クライアントが ASE 環境で提供される IP アドレスにより NFS サービスを認識していることに対処する必要があります。クラスタでサービスされている NFS ファイル・システムにクライアントがアクセスする場合は、省略時のクラスタ別名か、または `/etc/exports.alias` ファイルにリストされているクラスタ別名を使用しなければなりません。

5.4.1.2 インタフェース別名

シングル・インスタンス・アプリケーションで簡単にクラスタ別名を使うことができない場合は、既存の `nfs_ifconfig` エントリを `ifconfig(8)` の呼び出しに変更するか、CAA 処理スクリプトに `ifconfig` の呼び出しを追加して、インタフェース別名を継続して使うことができます。

CAA の処理スクリプトを変更するときには、`ifconfig alias` を呼び出して、別名をインタフェースに割り当てます。アプリケーションを起動する前に、次のコマンドを実行するようにしてください。これを行わない場合には、アプリケーションを別名アドレスにバインドすることはできません。

```
ifconfig interface_id alias alias_address netmask mask
```

別名のインタフェースへの割り当てを解除するには、すべてのアプリケーションやプロセスを停止した後に、`ifconfig -alias` を呼び出します。このようにしなければ、アプリケーションやプロセスは、インタフェース別名で通信できなくなります。

次の例は、スクリプト例の抜粋です。

```
:
:
# Assign an IP alias to a given interface
IFCNFG_ALIAS_ADD="/sbin/ifconfig tu0 alias 16.141.8.118 netmask 255.255.255.0"
#
#Deassign an IP alias to an interface
IFCNFG_ALIAS_DEL="/sbin/ifconfig tu0 -alias 16.141.8.118 netmask 255.255.255.0"
:
:
```

5.4.2 ネットワーク・サービス

TruCluster Available Server Software と TruCluster Production Server Software 製品では、`asemgr` ユーティリティで、クライアント・ネットワークを監視するメカニズムが提供されていました。

Tru64 UNIX バージョン 5.0 以降では、クライアント・ネットワークの監視は、ベース・オペレーティング・システムの機能になっています。NetRAIN インタフェースを使うことにより、さまざまなネットワークの接続障害からシステムを保護することができます。この機能を補足する機能として NIFF (Network Interface Failure Finder) があります。NIFF はネットワーク・インタフェースの状態を監視して、ネットワークに障害が発生するとそれを報告します。TruCluster Server クラスタで動作するアプリケーション

ンでは、NetRAIN および NIFF の機能と、Tru64 UNIX の EVM (イベント・マネージャ) を併用して、クライアント・ネットワークの状態が正常かどうかを監視できます。

NetRAIN と NIFF についての詳細は、Tru64 UNIX 『ネットワーク管理ガイド：接続編』、`niffd(8)`、`niff(7)`、および `nr(7)` を参照してください。

5.5 ファイル・システムのパーティショニング

CFS は、すべてのファイルをすべてのクラスタ・メンバにアクセス可能にします。ファイルがすべてのクラスタ・メンバに接続された装置上に格納されているか、単一メンバ専用の装置上に格納されているかにかかわらず、各クラスタ・メンバはファイルにアクセスできます。ただし、CFS では単一のクラスタ・メンバからだけアクセスできるように AdvFS ファイル・システムをマウントすることができます。これをファイル・システムのパーティショニングといいます。

ASE は、ファイル・システムのパーティショニング機能と同じ機能を提供していました。バージョン 5.1B では、ASE からの移行を容易にするためにファイル・システムのパーティショニングを提供しています。パーティショニングされたファイル・システムのマウント方法および制限事項については、TruCluster Server 『クラスタ管理ガイド』を参照してください。

分散型アプリケーションの TruCluster Server への移行

分散型アプリケーションは、クラスタ内で使用するように修正されているアプリケーションです。クラスタに対応している、つまり、クラスタ内で動作することを意識しているアプリケーションです。アプリケーションをクラスタ内で実行しただけでは、アプリケーションはクラスタ対応にはなりません。分散型アプリケーションの構成要素は、しばしば、TruCluster Server 製品にバンドルされているアプリケーション・プログラミング・インタフェース (API) ライブラリを使い、メンバ・システム間で情報を交換し、協調して共用データにアクセスします。

次のサブシステム API は、TruCluster の以前の製品で提供されていた API と完全に互換性があります。

- クラスタ別名 (clua_*.3)
- 分散ロック・マネージャ (DLM) (d1m_*.3)
- Memory Channel (imc_*.3)

クラスタ別名、DLM、および Memory Channel の各 API の使用方法についての詳細は、それぞれ第 8 章、第 9 章、および第 10 章を参照してください。

この章では、次の項目について説明します。

- 分散型アプリケーションの TruCluster Server への移行の準備 (6.1 節)
- TruCluster Server での Oracle Parallel Server の作成 (6.2 節)
- Oracle Parallel Server の TruCluster Server への移行 (6.3 節)

6.1 分散型アプリケーションの TruCluster Server への移行の準備

分散型アプリケーションを TruCluster Server に移行する準備をする際には、次の点に注意します。

- デバイス名が、データ・ファイルの名前に埋め込まれている場合があります。データ・ファイルの名前を変更するか、シンボリック・リンクを作成するか、またはデータベースを作成し直します。シンボリック・リンクを作成する方法が、管理が最も簡単になる方法です。raw デバイスへのシンボリック・リンクを作成して、許可を更新します。

注意

OPS のデータ・ファイル名は変更できますが、Informix XPS のデータ・ファイル名は変更できません。これについてはシンボリック・リンクを使用してください。詳細は、『*Informix Installation Guide*』を参照してください。

- コントロール・ファイルには編集を加えず、オリジナルを保存しておいてください。
- 単一メンバの TruCluster Server クラスタを作成した後に、Oracle のバイナリ・コードを再度リンクします。これは、接続マネージャ・ライブラリに TruCluster Server 用の新しいエントリ・ポイントがあるためです。

6.2 TruCluster Server 上での Oracle Parallel Server の実行

この節では、Oracle 8i リリース 3 (8.1.7) の OPS (Oracle Parallel Server) オプションを立ち上げて TruCluster Server バージョン 5.1 以降のクラスタで動作させる方法について説明します。Oracle 8i 8.1.7 は TruCluster Server バージョン 5.1B 以降で実現されたダイレクト入出力機能を利用しているので、TruCluster Server バージョン 5.1B 以降のクラスタ・ファイル・システム上に OPS を構成することができます。Oracle の以前のバージョンでは、クラスタ内にある raw ディスクのパーティションまたはボリュームを使う必要がありました。クラスタにおけるダイレクト入出力についての説明は、TruCluster Server 『クラスタ管理ガイド』を参照してください。

注意

Tru64 UNIX および TruCluster Server バージョン 5.1B 上で Oracle9i Real Application Cluster (RAC) を実行するには、次のサイトで提供されている『*Installation Guide*』を参照してください。 http://otn.oracle.com/products/oracle9i/pdf/Oracle_9i_on_Tru64_UNIX.pdf

このドキュメントでは Oracle9i RAC について説明するとともに、クラスタ環境で Tru64 UNIX、TruCluster Server、および Oracle をインストールして構成する方法について説明しています。

OPS を TruCluster Server 上で実行するためには、次の手順に従います。

1. Oracle8i リリース 3 (8.1.7) のドキュメントに記載されている指示に従って、Oracle8i リリース 3 (8.1.7) をインストールして構成します。Oracle8i は 1 つのクラスタ・メンバにインストールするだけで十分です。

Oracle には特別な要件があります。この要件には、特定のカーネル属性に特定の値を設定していること、特定の UNIX グループ (dba, oinstall) を作成していること、特別な環境変数を初期化していることなどがあります。

Oracle8i リリース 3 (8.1.7) のドキュメントに記載されている指示に従って、Oracle Parallel Server オプションを構成します。

2. クライアント要求の負荷を分散するためにクラスタ別名を使用するよう、Net8 リスナを構成します。また、OPS の MTS (Multi-Threaded Server) 機能を使用することでも、クライアント要求の負荷を分散することができます。Oracle8i のドキュメントを参照してください。

クラスタ別名を使う場合は、次の行を /etc/clua_services ファイルに追加して、Oracle8i リスナの使用するポートの特性を設定します。

```
listener          1521/tcp          in_multi
```

ポート 1521 に in_multi 属性を設定すると、クラスタ別名サブシステムはそのクラスタ別名宛の接続要求を、その別名のすべてのメンバに分散させるようになります。

3. 各クラスタ・メンバで次のコマンドを実行し、クラスタ別名のサービス定義を再ロードします。

```
# cluamgr -f
```

4. OPS をクラスタにセットアップし、ローカルからリモートからもアクセスできることを確認したら、各メンバがブート時にそれぞれのデータベース・インスタンスを起動し、シャットダウン時にそのデータベース・インスタンスを停止することを確認する必要があります。このために、スクリプトを /sbin/init.d に置く方法をお勧めします。

CAA (Cluster Application Availability) の処理スクリプトを作成して、データベース・インスタンスを自動的に起動し停止させることも可能です。ただし、その場合は OPS を 1 つのクラスタ・メンバに制限する必要があります。OPS の起動と停止についての詳細は、Oracle8i リリース 3 (8.1.7) のドキュメントを参照してください。

6.3 Oracle Parallel Server の TruCluster Server への移行

OPS を TruCluster Server に移行する際には次の点に注意してください。

- TruCluster Software Products バージョン 1.6 以前から移行する場合は、デバイス名に注意してください。DRD で管理されたストレージのスペシャル・デバイス名はなくなります。データベースから `/dev/drđ` または `/dev/rđrd` を参照している場合は、6.2 節のステップ 3 での説明に従って、新しいデバイス名へのシンボリック・リンクを作成してください。
- OPS ではデータ・ファイルの名前を変更することができます。データ・ファイルへのシンボリック・リンクを作成しないで名前を変更したい場合は、次の手順に従います。
 1. オリジナルのコントロール・ファイルを保存します。これらのコントロール・ファイルは、編集しないでください。
 2. `nomount` オプションを使ってデータベースを起動します。
 3. 次のコマンドで、データ・ファイルの名前を一般的なファイル名に変更します。

```
SVRMGR> alter database rename file ...
```
 4. データ・ファイルの名前を変更したら、次のコマンドでデータベースを確認します。これによって、データベースの整合性が検証され、シンボリック・リンクが不適切であれば検出されます。

```
SVRMGR> analyze table validate structure cascade
```


Part 3

クラスタ対応のアプリケーションの作成



プログラミングの検討項目

この章では、アプリケーションをクラスタ内で実行できるようにするために、アプリケーションのソース・コードへの変更について説明します。実際に変更を行うには、アプリケーションのソース・ファイルにアクセスできなければなりません。

この章では、次の項目について説明します。

- リモート・プロシージャ・コール (RPC) プログラムに必要な変更 (7.1 節)
- 移植性のあるアプリケーション — クラスタ・システムやスタンドアロン・システムで動作するアプリケーションの開発 (7.2 節)
- CLSM (Cluster Logical Storage Manager) のサポート (7.3 節)
- 診断ユーティリティのサポート (7.4 節)
- CDFS (Compact Disc-Read Only Memory File System) ファイル・システムの制約 (7.5 節)
- `/cluster/admin/run` ディレクトリから呼ばれるスクリプト (7.6 節)
- ローリング・アップグレード中のクラスタ・メンバの状態 (7.7 節)
- クラスタにおけるファイル・アクセスの回復力 (7.8 節)

7.1 RPC プログラムに必要な変更

既存の、カプセル化されていないリモート・プロシージャ・コール (RPC) プログラムを次のように変更して、クラスタ環境で実行できるようにします。

- クラスタ環境で実行されているときに、`bind()` への呼び出しから `clusvc_getcommport()` または `clusvc_getresvcommport()` への呼び出しに切り替えられるように、コードに条件判定を組み込みます。これらの関数は、RPC アプリケーションを複数のクラスタ・メンバ上で実行し、クラスタ別名を使ってこれにアクセスできるようにしたい場合にだけ使用します。これらの関数では、RPC アプリケーションのそれぞれのインスタンスが同じ共用ポートを使用することが保証さ

れ、また、ポート・マップにそのアプリケーションがマルチ・インスタンスの別名アプリケーションであることが伝えられます。詳細は、`clusvc_getcommport(3)` および `clusvc_getresvcommport(3)` を参照してください。

- `svc_register()` を呼び出さないサービスは、`clua_registerservice()` を呼び出して、到着したクラスター別名接続要求を、そのサービスが受け取れるようにしなければなりません。詳細は、`clua_registerservice(3)` を参照してください (`clusvc_getcommport()` と `clusvc_getresvcommport()` は、自動的に `clua_registerservice()` を呼び出します)。

7.2 移植性のあるアプリケーション: スタンドアロンとクラスター

Tru64 UNIX バージョン 5.0 以降では、クラスター・システムとスタンドアロン・システムのどちらでも実行できるアプリケーションを簡単に開発できる、次のような組み込み機能が提供されています。

- Tru64 UNIX バージョン 5.0 以降のスタブ・ライブラリによって、TruCluster Server で提供される `libclu.so` API ライブラリの関数を呼び出すアプリケーションを作成することができます。
- `libc` で用意されている `clu_is_member()` 関数は、ローカル・システムがクラスター・メンバかどうかを判断します。この関数は、ローカル・システムがクラスター・メンバの場合は `TRUE` を返し、クラスター・メンバでない場合は `FALSE` を返します。
- `libc` で用意されている `clu_is_ready()` 関数も、ローカル・システムがクラスター・メンバとして構成されているかどうか (つまり、TruCluster Server Software がインストールされ、システムがクラスター対応のカーネルを実行しているかどうか) を判断します。この関数は、ローカル・システムがクラスター内で動作している場合は `TRUE` を返し、動作していない場合は `FALSE` を返します。`clu_is_ready()` 関数は、接続マネージャがクラスター・メンバシップを確立する前に、ブート・パスで実行されるコードの中で一番有用な関数です。
- `clu_info()` 関数と `clu_get_info()` コマンドは、構成に関する情報、またはそのシステムがクラスター内で構成されていないことを示す値を返します。

詳細は、`clu_info(3)`、`clu_is_member(3)` および `clu_get_info(8)` を参照してください。

7.3 CLSM のサポート

CLSM (Cluster Logical Storage Manager) は、2 つの異なるノード間でミラーリングされているボリュームに対する、通常の入出力のインターロックをサポートしていません。CLSM では、異なるノードから同じボリュームを同時にオープンするアプリケーションは、同じブロックに同時に書き込まないようにするための必要なロックをすでに行なっていると仮定しています。つまり、クラスタ対応のアプリケーションの統制がとれていなくて、CLSM ミラー・ボリューム上の同じブロックに対して異なるノードから書き込みが行われた場合には、データの整合性が失われます。

OPS (Oracle Parallel Server) では、このような事態にはなりません。OPS が分散ロック・マネージャ (DLM) を使って、このような状況を回避しているからです。クラスタ・ファイル・システム (CFS) の場合も、このような事態にはなりません。ファイル・システムは一度に 1 つのノードしかマウントできないからです。

CLSM は、ミラー・ボリュームへの定常状態の入出力では、異なるノード間のインターロックを行いませんが、ミラーの回復とブレッスの組み込み用には、ノード間のインターロック機能を用意しています。

7.4 診断ユーティリティのサポート

アプリケーションやサブシステムの診断ユーティリティがすでにある場合または作成する場合、TruCluster Server の `clu_check_config` コマンドは、その診断ユーティリティを呼び出して実行するとともに、ログ・ファイルを保持することができます。

クラスタ環境に診断ユーティリティを追加する方法と、`clu_check_config` コマンドでそれ呼び出す方法については、`clu_check_config(8)` を参照してください。

7.5 CDFS ファイル・システムの制約

TruCluster Server 環境では、CDFS (Compact Disc-Read Only Memory File System) ファイル・システムをクラスタ内で管理する際に制約があります。

クラスタ・システムとスタンドアロン・システムで、異なる動きをするコマンドやライブラリ関数があります。

表 7-1 に、CDFS ライブラリ関数と TruCluster Server 環境での予想される動作を示します。

表 7-1: CDFS ライブラリ関数

ライブラリ関数	サーバでの予想される結果	クライアントでの予想される結果
cd_drec	成功	サポートされていない
cd_ptrec	成功	サポートされていない
cd_pvd	成功	サポートされていない
cd_suf	成功	サポートされていない
cd_type	成功	サポートされていない
cd_xar	成功	サポートされていない
cd_nmconv CD_GETNMCONV	成功	成功
cd_nmconv CD_SETNMCONV	成功	成功
cd_getdevmap	マップなし	サポートされていない
cd_setdevmap	サポートされていない	サポートされていない
cd_idmap CD_GETUMAP CD_GETGMAP	成功	サポートされていない
cd_idmap CD_SETUMAP CD_SETGMAP	成功	成功
cd_defs CD_GETDEFS	成功	成功
cd_defs CD_SETDEFS	成功	成功

クラスタの CDFS ファイル・システムを管理する方法についての詳細は、TruCluster Server の『クラスタ管理ガイド』を参照してください。

7.6 /cluster/admin/run ディレクトリから呼び出されるスクリプト

クラスタが作成されるとき、メンバが追加されるとき、また削除されるときに、特定の処理を行う必要があるアプリケーションでは、/cluster/admin/run ディレクトリにスクリプトを置くことができます。このスクリプトは、初期クラスタ・メンバの最初のブート時に `clu_create` の実行に続いて呼び出されます。`clu_add_member` に続いては、クラスタの各メンバ (最も新しいものも含めて) について呼び出され、`clu_delete_member` に続いては、残りのすべてのクラスタ・メンバについて呼び出されます。

/cluster/admin/run 内のスクリプトには、次のエントリ・ポイントを記述する必要があります。

- `-c`
`clu_create` の実行時にとるべき処理
- `-a`
`clu_add_member` の実行時にとるべき処理
- `-d memberid`
`clu_delete_member` の実行時にとるべき処理

`root` が実行できるファイルやファイルへのシンボリック・リンクだけを、/cluster/admin/run ディレクトリに置きます。できるだけ、次のファイル命名規則に従うようにしてください。

- 実行可能ファイルは、大文字 `C` で始める。
- 次の 2 つの文字は /sbin/rc3.d 内で使われる順番を示す続き番号にする。番号が小さいスクリプトから先に実行される。これらのスクリプトはシングル・ユーザ・モードで実行されるため、すべてのデーモンおよびサービスは利用できない。
- 残りの文字は、スクリプトに対応する名前にする。

この命名規則を使用したファイル名の例を次に示します。

```
/cluster/admin/run/C40sendmail
```

clu_create, clu_add_member, および clu_delete_member コマンドでは、必要な it(8) ファイルとリンクが作成されます。これによって、スクリプトが適切なタイミングで実行されることが保証されます。

7.7 ローリング・アップグレード中のクラスタ・メンバの状態のテスト

次のプログラム例に、クラスタがローリング・アップグレード中であるかどうかと、クラスタ・メンバがローリングされたかどうかを調べる方法の例を示します。

```
#include <stdio.h>
#include <sys/clu.h> /* compile with -lclu */

#define DEBUG 1

main()
{
    struct clu_gen_info *clu_gen_ptr = NULL;
    char cmd[256];

    if (clu_is_member()) {
        if (clu_get_info(&clu_gen_ptr) == 0) {
            if (system("/usr/sbin/clu_upgrade -q status") == 0) {
                sprintf(cmd, "/usr/sbin/clu_upgrade -q check roll %d",
                        clu_gen_ptr->my_memberid);
                if (system(cmd) == 0) {
                    if (DEBUG) printf("member has rolled\n");
                }
                else if (DEBUG) printf("member has not rolled\n");
            }
            else if (DEBUG) printf("no rolling upgrade in progress\n");
        }
        else if (DEBUG) printf("nonzero return from clu_get_info(\n");
    }
    else if (DEBUG) printf("not a member of a cluster\n");
}
```

7.8 クラスタにおけるファイル・アクセスの障害許容度

クラスタのアプリケーションがファイルを転送している最中にそのアプリケーションを実行しているメンバがシャットダウンされるか障害が発生すると、そのファイルの読み取り/書き込み操作は失敗します。そのような場合、アプリケーションのクライアントにはサーバとの間の接続が切れたように見えるのが普通です。アプリケーションが消失した接続をどのように処理するかに注意してください。アプリケーションが消失した接続を処理する方法には次のようなものがあります。

- クライアントのアプリケーションが単に異常終了する（たとえば、エラーをログに出力して、アプリケーションが終了する）。

- クライアントのアプリケーションはコネクションの問題を知ると自動的に読み取り/書き込み操作を再試行する。
- クライアントのアプリケーションはコネクションの問題を知ると、ウィンドウにそのことを表示して、操作を中止するか、再試行するか、またはキャンセルするかの決定をユーザに任せる。

サーバ・アプリケーションとの間の接続が消失した場合に（それがシングル・システムまたはクラスタのどちらで動作しているかに関係なく）クライアントのアプリケーションが異常終了するようであれば、次に示す方法を検討してください。

- ファイルを更新するとき、最初にそのアップデートを新しい一時ファイルに書く。書き込みが成功したら、一時ファイルの内容を元のファイルに上書きする。書き込みが失敗しても、元のファイルはそのまま残るので、一時ファイルを消して再び処理を開始するだけでよい。
- ファイルを読み取る場合は、アプリケーションが読み取り操作のエラーを処理してそのエラーから復帰できるようになっていることを確認する（たとえば、操作の再試行）。



クラスタ別名アプリケーション・プログラミング・インタフェース

この章では、次の項目について説明します。

- クラスタ別名ポートの用語 (8.1 節)
- クラスタ別名関数 (8.2 節)
- クラスタ・ポート空間 (8.3 節)
- 予約ポートへバインドするマルチ・インスタンス・サービスの情報 (8.4 節)
- クラスタ別名 `setsockopt()` のオプション (8.5 節)
- ポート属性: `/etc/clua_services`, `clua_registerservice()` および `setsockopt()` クラスタ別名ソケット・オプションで設定されるポート属性間の関係 (8.6 節)
- UDP アプリケーションとソース・アドレス (8.7 節)

8.1 クラスタ別名ポートの用語

この章では、ポートに関連した以下の用語を使用します。

既知のポート	クライアントとサーバの両方が番号を知っているポート。たとえば、番号が <code>/etc/services</code> にリストされているポート。クライアントがこのポートの番号を知っていて、そのポートを介してサーバへ接続しようとするため、サーバは、この既知のポートを明示的にバインドしリッスンします。
動的ポート	OS が割り当てる空きポート。アプリケーションでは、明示的にポート番号を指定しません。オペレーティング・システムが見つけた最初の空きポートが割り当てられます。

一時ポート	一時ポート空間内の動的ポート (1024を超える番号を持つポート。厳密には、IPPORT_RESERVED と IPPORT_USERRESERVED の間のポート)。
ロック・ポート	特定のクラスタ・メンバが専用に使い、クラスタ単位で使用できないポート。ポートがロックされている場合、アプリケーションがそのポートをバインドしようとする と EADDRINUSE で失敗します。ただし、そのアプリケーションがソケットに SO_REUSEALIASPORT オプションを設定している場合はこの限りではありません。
予約ポート	クラスタ別名サブシステムで、ポート番号が 512 から IPPORT_RESERVED (1024) までのポート。省略時のクラスタの動作では、予約ポートにバインドしようとする と自動的にそのポートはロックされます (番号が 512 以下のポートはどんな場合もロックされません)。

8.2 クラスタ別名関数

クラスタ別名ライブラリ libclua.a および libclua.so には、クラスタ別名サブシステムの属性を表示する関数と設定する関数が用意されています。表 8-1 に、クラスタ別名ライブラリの関数を示します。詳細は、セクション 3 の個々のリファレンス・ページを参照してください。

表 8-1: クラスタ別名ライブラリ関数

関数	説明
clua_error	クラスタ別名メッセージ ID を、それに対応する印字可能な文字列に変換し、呼び出し元にその文字列を返す。
clua_getaliasaddress	ローカル・ノードが知っているクラスタ別名中の 1 つの IP アドレスを取得する。
clua_getaliasinfo	クラスタ別名の 1 つとそのメンバの情報を取得する。
clua_getdefaultalias	省略時のクラスタ別名の IP アドレスを取得する。

表 8-1: クラスタ別名ライブラリ関数 (続き)

関数	説明
clua_isalias	IP アドレスがクラスタ別名の IP アドレスかどうかを判断する。
clua_registerservice	動的ポートを，着信接続要求の受け付け用として登録する。
clua_unregisterservice	ポートを解放する。
clusvc_getcommport	クラスタ内の共用ポートにバインドする。
clusvc_getresvcommport	クラスタ内の予約ポートにバインドする。
print_clua_liberror	クラスタ別名メッセージ ID を，それに対応する印字可能な文字列に変換し，stderr にその文字列を返す。

クラスタ別名関数を使用するプログラムの作成およびコンパイルを行う際には，次の `#include` ファイルとライブラリを使用します。

- `clua` 関数を使用するプログラムは，`#include <clua/clua.h>` を記述し `-lclua` でコンパイルする。
- 次の関数を使用するプログラムは，`-lclua -lcfg` でコンパイルする。
 - `clua_getaliasaddress()`
 - `clua_getaliasinfo()`
 - `clua_getdefaultalias()`
 - `clua_isalias()`
 - `clua_registerservice()`
 - `clua_unregisterservice()`
- `clusvc` 関数を使用するプログラムは，`#include <netinet/in.h>` を記述し，`-lclu` でコンパイルする。

次に，これらの関数をさらに詳しく説明します。

`clua_error()` と `print_clua_liberror()`

`clua_error()` および `print_clua_liberror()` 関数は，クラスタ別名メッセージ ID をそれに対応する印字可能な文字列に変換します。`clua_error()` 関数は，呼び出し元にその文字列を返します。

`print_clua_liberror()` 関数は、`stderr` にその文字列を出力します。詳細は、`clua_error(3)` を参照してください。

`clua_getaliasaddress()` と `clua_getaliasinfo()`

`clua_getaliasaddress` 関数を呼び出すことで、ローカル・ノードが知っているクラスタ別名の 1 つの IP アドレスを得ることができます。続けて呼び出すたびに別の別名の IP アドレスが返されます。そのノードが知っている別名のリストが尽きると、この関数は、`CLUA_NOMOREALIASES` を返します。

`sockaddr` 構造体の形で別名のアドレスが渡されると、`clua_getaliasinfo()` は、`clu_info` 構造体にその別名の情報を格納します。

プログラムでは、通常、`clua_getaliasaddress()` を対話的に呼び出して、別名アドレスを 1 つずつ `clua_getaliasinfo()` に渡し、それぞれの別名についての情報を取得します。以下の呼び出しシーケンスは、この対話型ループの部分です (`printf()` を数回呼び出す短い `main()` プログラムを内部に含んでいます)。

```
/* compile with -lclua -lcfg */
#include <sys/socket.h> /* AF_INET */
#include <clua/clua.h> /* includes <netinet/in.h> */
#include <netdb.h> /* gethostbyaddr() */
#include <arpa/inet.h> /* inet_ntoa() */

main ()
{
    int context = 0;
    struct sockaddr addr;
    struct clu_info outbuf, *pout;
    clua_status_t result1, result2;
    struct hostent *hp;
    pout=&outbuf;

    while ((result1=clua_getaliasaddress(&addr,
                                        &context)) == CLUA_SUCCESS)
    {
        if ((result2=clua_getaliasinfo(&addr, pout)) == CLUA_SUCCESS) {
            hp = gethostbyaddr((const void *)&pout->addr,
                              sizeof (struct in_addr), AF_INET);
            printf ("\nCluster alias name:\t\t %s\n", hp->h_name);
            printf ("Cluster alias IP address:\t %s\n",
                    inet_ntoa(pout->addr));
            printf ("Cluster alias ID (aliasid):\t %d\n", pout->aliasid);
            printf ("Connections rcvd from net:\t %d\n",
                    pout->count_cnx_rcv_net);
            printf ("Connections forwarded:\t\t %d\n",
                    pout->count_cnx_fwd_clu);
            printf ("Connections rcvd within cluster: %d\n",
                    pout->count_cnx_rcv_clu);
        } else {
            print_clua_liberror(result2);
        }
    }
}
```

```

        break;
    }
}
if (result1 != CLUA_SUCCESS && result1 != CLUA_NOMOREALIASES)
    print_clua_liberror(result1);
}

```

詳細は、`clua_getaliasaddress(3)` および `clua_getaliasinfo(3)` を参照してください。

`clua_getdefaultalias()`

`clua_getaliasaddress()` 関数では、ローカル・ノードが知っているすべてのクラスタ別名の IP アドレスが対話的に返されますが、`clua_getdefaultalias()` 関数では、省略時のクラスタ別名アドレスしか返されません。詳細は、`clua_getdefaultalias(3)` を参照してください。

`clua_isalias()`

`clua_isalias()` 関数は、IP アドレスを渡されると、そのアドレスがクラスタ別名のアドレスかどうかを判断します。詳細は、`clua_isalias(3)` を参照してください。

`clua_registerservice()` と `clua_unregisterservice()`

`clua_registerservice()` 関数は、動的ポートを、着信接続要求の受け付け用として登録します。512 ~ 1024 の範囲にあるポートでは、`CLUSRVC_STATIC` オプションを指定します。これを指定しなければ、そのポートをバインドする最初のノードがクラスタ単位でそのポートを予約するため、残りのクラスタ・メンバはそのポートにバインドできなくなります。

`clua_unregisterservice()` 関数は、ポートを解放します。

詳細は、`clua_registerservice(3)` を参照してください。

`clusvc_getcommport()` と `clusvc_getresvcommport()`

RPC を使用するプログラムでは、`clusvc_getcommport()` および `clusvc_getresvcommport()` 関数を呼び出して、クラスタ内の共用ポートにバインドすることができます。予約 (特権) 共用ポート (ポート番号が 0 ~ 1024) にバインドするときには、`clusvc_getresvcommport()` を使用します。予約ポートにバインドする方法の詳細は、8.4 節を参照してください。以下に典型的な呼び出し

シーケンスの例を示します (printf() を数回呼び出す短い main() プログラムを内部に含んでいます)。

```
/* compile with -lclu */
#include <rpc/rpc.h> /* includes <netinet/in.h> */
#include <syslog.h> /* LOG_ERR */
#include <unistd.h> /* gethostname() */
#include <sys/param.h> /* MAXHOSTLEN */

main () {
    int s, i, namelen;
    int cluster = 0;
    uint prog = 100999; /* replace with real program number */
    struct sockaddr_in addr;
    int len = sizeof(struct sockaddr_in);
    char local_host[MAXHOSTNAMELEN + 1];

    gethostname (local_host, sizeof (local_host) - 1);
    cluster = clu_is_member();
    printf ("\nSystem %s %s a cluster member\n",
            local_host, cluster?"is":"is not");

    if ((s = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0) {
        syslog(LOG_ERR, "socket: %m");
        exit(1);
    }

    bzero(&addr, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY;

    if (cluster) {
        if (clusvc_getcommpport(s, prog, IPPROTO_UDP, &addr) < 0) {
            syslog(LOG_ERR, "clusvc_getcommpport: %m");
            exit(1);
        }
    } else {
        addr.sin_port = 0;
        if (bind(s, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
            syslog(LOG_ERR, "bind: %m");
            exit(1);
        }
        if (getsockname(s, (struct sockaddr *)&addr, &len) != 0) {
            syslog(LOG_ERR, "getsockname: %m");
            (void) close(s);
            exit(1);
        }
    }
    printf ("  addr.sin_family: %d\n", addr.sin_family);
    printf ("  addr.sin_port:   %u\n", addr.sin_port);
    printf ("  addr.sin_addr:   %x\n", addr.sin_addr);
    printf ("  addr.sin_zero:   ");
    for (i = 0; i < 8; i++)
        printf("%d ", (int)addr.sin_zero[i]);
    putchar('\n');
}
```

8.3 クラスタ・ポート空間

クラスタでは、クラスタ全体で同じポート空間を使用することにより、シングル・システムのポートのセマンティクスをエミュレートします。以下

8-6 クラスタ別名アプリケーション・プログラミング・インタフェース

の項では、クラスタ別名サブシステムでどのようにポート空間を扱うかを簡単に説明します。

- ポート番号 512 未満のポート ($0 < (\text{IPPORT_RESERVED}^1 / 2)$): これらの既知のポートはロックされません。シングル・システムでは、`SO_REUSEPORT` ソケット・オプションを使わない限り、複数のプロセスが同じポートにバインドすることはできません。クラスタ・システムでは、番号が 512 未満のポートは、各クラスタ・メンバの 1 つのプロセスが、そのポートにバインドできるようにするために、決してロックされません。
- 512 ~ 1024 の範囲にあるポート ($(\text{IPPORT_RESERVED} / 2) \sim \text{IPPORT_RESERVED}$): ポートが明示的にバインドされている場合、`static` としてマークされていない限り、予約ポートはロックされません。予約ポートのバインドについては、8.4 節を参照してください。
- 1024 を超える番号のポート ($\text{IPPORT_RESERVED} \sim \text{IPPORT_USERRESERVED}$): 一時ポートは、`sin_port=0` の場合、ロックされます。一時ポートは、ポートに明示的にバインドした場合、ロックされません。

ポート空間には、シングル・インスタンス・アプリケーションとマルチ・インスタンス・アプリケーションで、それぞれ一般的に次のような意味があります。

- シングル・インスタンス・アプリケーションでは、1 つのインスタンスだけがネットワーク要求を処理するので、ロック・ポートを使用するようにします。したがって、次の方法のいずれかを使用することができます。
 - 予約ポート (512 ~ 1024) に明示的にバインドする。
 - 一時ポート (`sin_port=0`) を使用する。
これによりポートはロックされますが、何らかの方法でクライアントにポート番号を通知しなければならないという問題があります。
 - CAA を使って、アプリケーションの 1 つのインスタンスだけがそのクラスタ内で動作することを保証する。
- マルチ・インスタンス・アプリケーションでは、アプリケーションの最初のインスタンスだけが、ポートにバインドするように制限するべきではありません。そこで、次のような方法をとります。

¹ `IPPORT_RESERVED` および `IPPORT_USERRESERVED` は `<netinet/in.h>` で定義されています。

- 512 未満の番号のポートを使用する。
- 予約ポート (512 ~ 1024) を使用し、SO_REUSEALIASPORT を設定する。
- 一時ポートに明示的にバインドする。
- 一時ポート (sin_port=0) を使用して、SO_REUSEALIASPORT を設定する。

これにより、ポートはロックされなくなりますが、何らかの方法でクライアントにそのポート番号を通知しなければならないという問題が残ります。

クラスタ別名ポート空間についての詳細は、『クラスタ概要』のクラスタ別名の章を参照してください。

8.4 予約ポートへのバインド (512 ~ 1024)

次にマルチ・インスタンス・サービスを予約ポートにバインドする場合に役立つ、背景となる情報について説明します。

予約ポートは、既知のポートとしても動的ポートとしても使用できるので、特別に扱われます。省略時の設定では、プロセスが明示的に 512 ~ 1024 の範囲のポートにバインドした場合、そのポートは動的ポートと見なされて、クラスタ単位で予約され (ロックされ)、他のメンバ上でそのポートにバインドすると失敗します。このようになっている主な理由は、多くのプログラムでは、ポートが使用中であれば失敗するという前提で、この範囲にある空きポートを探して bind() を呼び出すためです。

クラスタ別名サブシステムでは、次に示す理由で、予約ポートをこれとは異なる方法で扱います。

- 動的予約ポートを探すアプリケーションと、使用する既知のポートが分かっているアプリケーションを区別する方法がない。
- 動的ポートは、しばしば出力接続のローカル・ポートとして使われ、この接続は、ポート番号とアドレスの組み合わせでのみ識別される。

動的ポートが出力接続のために使われ、そのアドレスがクラスタ別名のものであった場合、別名サブシステムは、複数のノードに同じポートを持たせることはできません。これを許すと、クラスタ別名サブシステムは、クラスタ内の複数のノードからの接続、たとえば 1000 と別名

のペアを区別できません。したがって、別名サブシステムでは、予約ポートが `static` (クラスタ単位で使用) であることが分からない限り、そのポートをロックしなければなりません。

番号 1024 を超えるポートには、このような問題はありません。これは、動的にポートを割り当てる標準的な方法があるためです。つまり、ポート 0 にバインド (`sin_port=0`) すると、システムは一時ポート空間からポートを選択します。したがって、クラスタ別名サブシステムは、明示的に 512 未満または 1024 を超える番号にバインドしているプロセスは何をしているかを知っていると想定します。つまり、既知のポートを取得しようとしていると見なします。

512 ~ 1024 の範囲の既知のポートを入力接続に使用するマルチ・インスタンス・サービスでは、そのポートを `static` としてクラスタ別名サブシステムに登録するようにします。ポートを `static` として登録するためには、`/etc/clua_services` にそのポートのエントリを置くか、`CLUASRV_STATIC` オプションを指定して `clua_registerservice` を呼び出すようにプログラムを変更します。これらの関数の詳細は、`clua_registerservice(3)` と `clua_services(4)` を参照してください (`static` オプションは、1024 番を超えるポートでは、別の目的に使用されません。つまり、`static` として指定したポートを、一時ポート空間から除外するようにシステムに指示します)。

注意

`static` 属性は、クラスタ・メンバがブートされるたびに起動されるマルチ・インスタンス・サービスのためにだけ、`/etc/clua_services` の中で指定してください。たとえば、`/etc/inetd.conf` にエントリのあるサービスに指定します。これに従わなければ、異なるクラスタ・メンバ上の、動的ポートを探す異なるアプリケーションが、同じポートにバインドしてしまいます。

8.5 setsockopt() のオプション

`setsockopt()` および `getsockopt()` システム・コールは、以下のクラスタ別名のソケット・オプションをサポートしています。

SO_CLUA_DEFAULT_SRC

ローカル・アドレスが `bind()` 呼び出しを通してまだ設定されていない場合、ソケットは、省略時のクラスタ別名をソース・アドレスとして使用します。

SO_CLUA_IN_NOALIAS

ソケットをクラスタ別名アドレスにバインドしようとする、失敗します。クラスタ別名を使ってアクセスさせたくないサービスに、このオプションを指定してください。

動的ポート (`IPPORT_RESERVED` 以上 `IPPORT_USERRESERVED` 未満のポート) にバインドしても、そのポートはロックされません。

ワイルドカード・アドレス (`INADDR_ANY` または `IN6ADDR_ANY`) を使って予約ポートにバインドしても、そのポートはロックされません。

外方向の UDP 送信や TCP 接続要求のソース・アドレスは、ローカル・ホスト・アドレスになります (クラスタ別名アドレスではない)。

`SO_CLUA_IN_NOLOCAL` と `SO_CLUA_IN_NOALIAS` オプションは、一緒に使用することはできません。

SO_CLUA_IN_NOLOCAL

ソケットは、クラスタ別名宛のパケットを受信しなければならず、クラスタ別名宛でないパケットはすべて廃棄します。このオプションは、クラスタ別名でのみアクセスさせたいサービスに使用します。

`SO_CLUA_IN_NOLOCAL` と `SO_CLUA_IN_NOALIAS` オプションは、一緒に使用することはできません。

SO_RESVPORT

ソケットを予約ポートの範囲 (512 ~ 1024) にバインドしようすると、ポートが、`/etc/clua_services` の `static` エントリか、`CLUASRV_STATIC` オプションを指定した `clua_registerservice()` 呼び出しによって `static` としてマークされている場合に失敗します。`bind()` の呼び出しでは、`EADDRINUSE` が返されます。

SO_REUSEALIASPORT

ソケットは、ロックされたクラスタ別名ポートを再利用することができます。このオプションを設定すると、`bind()` はクラスタ全体に分散さ

れます。分散型アプリケーションでは、この副作用を利用してポートが使用中かどうかを判断できます。

8.6 ポート属性: `/etc/clua_services` , `clua_registerservice()` , および `setsockopt()`

`/etc/clua_services` ファイルで使用する文字列は、`clua_registerservice()` で使用する `CLUASRV_*` オプションと 1 対 1 に対応しています。この文字列とオプションの一部は、さらにクラスタ別名 `setsockopt()` のオプションとも対応しています。表 8-2 に、これらの関係を示します。

表 8-2: クラスタ別名ポート属性間の関係

<code>clua_services</code>	<code>clua_registerservice()</code>	<code>setsockopt()</code>
<code>in_multi</code>	<code>CLUASRV_MULTI</code>	
<code>in_single</code>	<code>CLUASRV_SINGLE</code>	
<code>in_noalias</code>	<code>CLUASRV_IN_NOALIAS</code>	<code>SO_CLUA_IN_NOALIAS</code>
<code>out_alias</code>	<code>CLUASRV_OUT</code> ^a	<code>SO_CLUA_DEFAULT_SRC</code> ^b
<code>in_nolocal</code>	<code>CLUASRV_IN_NOLOCAL</code>	<code>SO_CLUA_IN_NOLOCAL</code>
<code>static</code>	<code>CLUASRV_STATIC</code>	
		<code>SO_REUSEALIASPORT</code>
		<code>SO_RESVPORT</code>

^a`CLUASRV_OUT` オプションは、`clua_services` ファイル内で、サービスのデスティネーション・ポートに対して `out_alias` 属性が設定されている場合だけ、省略時のクラスタ別名をソース・アドレスとして強制します。

^b`SO_CLUA_DEFAULT_SRC` オプションでは、デスティネーション・ポートに関連する属性は検査されません (`CLUASRV_OUT` と `SO_CLUA_DEFAULT_SRC` のどちらも手動でのアドレス設定より優先されません。これらの 2 つのオプションでは、アドレスが設定されていない場合だけ、ソース・アドレスとして省略時のクラスタ別名が設定されます)。

8.7 UDP アプリケーションとソース・アドレス

UDP (User Datagram Protocol) ベースのアプリケーションでは、サーバからクライアントへ送られるメッセージは、そのクライアントが UDP 要求を発行するときに使用したサーバのアドレスと同じアドレスを使用している必要があります。クラスタ内のシステムでは、通常、外方向 UDP メッセージのソース・アドレスとしてクラスタ別名を使用しないので、これらのアプリケーションがクラスタ別名を使うと、問題が生じる場合があります。

この節では、クラスタ内の UDP ベースのサーバ・アプリケーションが、着信メッセージで使用されていたアドレス (クラスタ別名またはローカル・アドレスのいずれか) を使って応答するようにする方法を説明します。

- ソケットを個々のローカル IP アドレスにバインドし、これに加えて、別のソケットを省略時のクラスタ別名アドレスにバインドするのが最善の方法です (別名のアドレスを得るには、`clua_getdefaultalias` を使用します)。要求を受け取ったのと同じソケットを使って要求に応答します。アドレスは、自動的にクライアントが使ったアドレスになります。この方法は、ワイルドカードに対して `listen` を 1 回実行する方法を置き換えます。
- これより少し単純な方法として、2 つのソケットを使う方法があります。1 つはワイルドカードに対してリッスンし、1 つはクラスタ別名に対してリッスンします (上記の例と同じように、入力ソケットを出力パケットとして再利用します)。同じサブネットに複数のローカル・アドレスがある場合は、この方法では、常に正しいローカル・アドレスを得ることはできませんが、省略時のクラスタ別名アドレスは正しく扱われます。複数のクラスタ別名があってアプリケーションが正しい別名 (入力方向で使われたもの) に応答するようにしたい場合は、複数のソケットの方法で、それぞれのクラスタ別名について `listen()` を行ってください。`clua_getaliasaddress()` を使って、定義されたクラスタ別名のリストを作成します。

注意

クラスタ別名でのみアプリケーションにアクセスできるようにして、アプリケーションが常に省略時のクラスタ別名を使って応答するようにしたい場合は、1 つのソケットでワイルドカードに対してリッスンするようにしますが、そのソケットは `SO_CLUA_DEFAULT_SRC` オプションで設定します。アプリケーションは、外方向トラフィックに常に省略時のクラスタ別名アドレスを使うようになります。

分散ロック・マネージャ

この章では、分散ロック・マネージャ (DLM) を使用して、クラスタ内の共有リソースへのアクセスの同期をとる方法について説明します。クラスタ・ファイル・システム (CFS) は、POSIX に完全に準拠しているので、アプリケーションでは `flock` システム・コールを使って、複数のインスタンスによる共有ファイルへのアクセスの同期をとることができます。また、アプリケーションではこの目的に、クラスタ DLM が提供する関数を使うこともできます。

DLM には、これまでの POSIX のファイル・ロックにはない機能があります。次のような機能です。

- 排他ロックや共有ロック以外の豊富なロック・モードのセット。
- 他のプロセスへのロック許可が阻止 (ブロック) されていることの、ロックを取得しているプロセスへの非同期通知とコールバック。
- ロック要求が許可されたことの、ロックの許可を待つプロセスへの非同期通知とコールバック。
- 既存のロックを上位あるいは下位のロック・モードに変換する要求を出す仕組みと変換を待つ仕組み。
- ロック用の独立したネームスペース。これらのネームスペースへのアクセスのさまざまな保護メカニズム。
- ロック構造体内の値ブロック。リソースを共有しているプロセスは、これを使って互いの動作を伝達し調整することができる。
- 高度なデッドロック検出メカニズム。

注意

DLM のアプリケーション・プログラミング・インタフェース (API) ライブラリは、TruCluster Server のクラスタ・ソフトウェアとして提供され、TruCluster Server のベース・オペレーティング・システムには添付されていません。このため、スタ

ンドアロン・システムとクラスタの両方で動作することを意図し、DLM 関数を呼び出すコードでは、`clu_is_member` を使ってアプリケーションがクラスタ内で動作しているかどうかを調べる必要があります。

この章では、基本的なDLM 操作の例についても記載しています。
`/usr/examples/cluster/` ディレクトリにこれらのコーディング例があります。

この章では、次の項目について説明しています。

- 特定のリソースに複数のプロセスがアクセスした場合、DLM はどのようにして同期をとるか (9.1 節)。
- リソース、リソースの細分性 (granularity)、ネームスペース、リソース名、およびロック・グループについての概念 (9.2 節)。
- ロック、ロック・モード、ロックの共存性、ロック管理のキュー、ロック変換、およびデッドロックの検出の概念 (9.3 節)。
- ロック要求をキューから外す `dml_unlock` 関数の使用方法 (9.4 節)。
- ロック変換要求を取り消す `dml_cancel` 関数の使用方法 (9.5 節)。
- ロック要求の完了通知、ロック要求の高速処理、ブロック通知、ロック変換、親ロックとサブロック、およびロック値ブロックといった高度なロック技術 (9.6 節)。
- ローカル・バッファ・キャッシングをアプリケーションで行う方法 (9.7 節)。
- DLM の基本的な操作を示すコーディング例 (9.8 節)。

9.1 DLM の概要

分散ロック・マネージャ (DLM) が提供する関数を使うと、1 つのクラスタ内で連携動作するプロセスによる、`raw` ディスク・デバイス、ファイル、プログラムなどの共用リソースへのアクセスの同期をとることができます。DLM で、共用リソースへのアクセスの同期を効果的にとるためには、クラスタ内のプロセスが特定のリソースを共用する場合は必ず、DLM の関数を使ってリソースへのアクセスを制御しなければなりません。

DLM の関数では、次のことができます。

- リソースに対するロックを要求する。
- ロック，あるいはロック・グループを解放する。
- 既存のロックのモードを変換する。
- ロック変換要求を取り消す。
- ロック要求が許可されるのを待つ，または処理を続行して，要求の完了通知を非同期に受け取る。
- 許可されたロックによって別のロック要求がブロックされる場合に，このことを非同期に通知してもらう。

表 9-1 は，DLM が提供する関数を表にしたものです。アプリケーションでは，libdlm ライブラリからこれらの関数を利用することができます。

表 9-1: 分散ロック・マネージャの関数

関数	説明
dml_cancel	ロック変換要求を取り消す。
dml_cvt	現在のロックを，同期をとって新しいモードに変換する。
dml_detach	すべてのネームスペースからプロセスをデタッチする。
dml_get_lkinfo	指定したプロセスに関連するロック要求の情報を取得する。
dml_get_rsbinfo	DLM が管理しているリソースのロック情報を取得する。
dml_glc_attach	既存のプロセス・ロック・グループ・コンテナにプロセスをアタッチする。
dml_glc_create	グループ・ロックのコンテナを作成する。
dml_glc_destroy	グループ・ロックのコンテナを削除する。
dml_glc_detach	プロセス・ロック・グループからデタッチする。
dml_lock	指定したリソースのロックを要求する (同期式)。
dml_locktp	グループ・ロックまたはトランザクション ID を使って，指定したリソースのロックを要求する (同期式)。
dml_notify	処理待ちの完了通知やブロック通知の配信を要求する。
dml_nsjoin	指定したネームスペースにプロセスを接続する。
dml_nsleave	指定したネームスペースからプロセスを切断する。

表 9-1: 分散ロック・マネージャの関数 (続き)

関数	説明
dml_perrno	指定された DLM メッセージ ID に対応するメッセージを出力する。
dml_perror	指定された DLM メッセージ ID に対応するメッセージと、呼び出し側で指定したメッセージ文字列を出力する。
dml_quecvt	既存のロックを、非同期に新しいモードに変換する。
dml_quelock	指定したリソースへのロックを要求する (非同期)。
dml_quelocktp	グループ・ロックまたはトランザクション ID を使って、指定したリソースのロックを要求する (非同期)。
dml_rd_attach	プロセスまたはプロセス・ロック・グループを回復ドメインにアタッチする。
dml_rd_collect	指定した回復ドメインのリソースのロックで、不正なロック値ブロックを持つものを収集し、そのドメインの回復手順を開始する。
dml_rd_detach	プロセスまたはプロセス・ロック・グループを回復ドメインからデタッチする。
dml_rd_validate	指定した回復ドメインで収集したリソースを有効にし、回復手順を終了する。
dml_set_signal	完了通知およびブロック通知で使用するシグナルを指定する。
dml_sperrno	指定された DLM メッセージ ID に対応する文字列を取得し、変数に格納する。
dml_unlock	ロックを解放する。

DLM 自体は、リソースへのアクセスが正しく行われているかどうかを保証しません。リソースにアクセスしているプロセス間の合意によって、リソースへのアクセスが許可されます。この動作は、ロック・マネージャの使用規則に従って、DLM の関数を使用することで実現できます。この規則は次のようになっています。

- すべてのプロセスが常に同じ名前でもリソースを参照しなければならない。この名前は、1 つのネームスペース内で一意でなければならない。
- ユーザ ID とグループ ID で修飾されたネームスペースでは、ネームスペース内で使われる保護および所有権 (つまり、ユーザ ID とグループ ID) は、クラスタ全体で矛盾しないようにしなければならない。パブ

リック・ネームスペースには、このようなアクセス制限はない。詳細は、9.2.2 項を参照。

- すべてのプロセスは、リソースにアクセスする前にロック要求をキューに入れ、そのリソースのロックを取得しなければならない。これには、`dlm_lock`、`dlm_locktp`、`dlm_quelock`、および `dlm_quelocktp` 関数を使用する。

ロックはプロセスが所有するので、DLM を使用するアプリケーションでは次の点を考慮しなければなりません。

- DLM は、シグナル、完了通知、およびブロック通知をプロセスに送信するので、スレッドを使用しているアプリケーションでは DLM の API 関数を使用しないでください。ほとんどのシグナルは、マルチスレッド・プロセス内の任意のスレッドに配信されます。(`dlm_set_signal` を使用して) DLM 通知シグナルとして定義されているシグナルはすべて、ロック許可またはブロック通知を待機していないスレッドに送信されます。このため、待機中のスレッドには送信されません。
- プロセスが子プロセスを生成 (fork) した場合、子プロセスは親プロセスが所有しているロックまたはアタッチしているネームスペースを受け継ぎません。子プロセスは、共用リソースにアクセスする前に、必要なリソースがあるネームスペースにアタッチし、必要なロックを取得しなければなりません。
- DLM は、ブロック・ルーチンおよび完了ルーチンのプロセス空間アドレスといった、プロセス固有の情報を保持しているため、プロセスが `exec` ルーチンを呼び出して、この情報が無効になると、予期しない動作が発生します。プロセスは `exec` ルーチンを呼び出す前に、`dlm_unlock` および `dlm_detach` 関数を使用して、ロックを解放しなければなりません。プロセスがこれらの関数を呼び出していない場合は、DLM が `exec` ルーチンの呼び出しを失敗に終わらせます。

9.2 リソース

クラスタ内のすべてのものが、リソースとなり得ます。たとえば、ファイル、データ構造、raw ディスク・デバイス、データベース、実行可能プログラムなどがリソースになります。複数のプロセスが同じリソースに同時にアクセスする場合、これらのアクセスの同期をとらなければ、正しい結果が得られません。

ロック管理の関数を使うと、プロセスは、名前あるいはバイナリ・データとリソースを関連付けて、そのリソースへのアクセスの同期をとることができます。同期をとらなければ、あるプロセスが新しいデータを書き込んでいる間に、別のプロセスがそのリソースを読み取った場合、読み取ったデータが、書き込みによってまったく違ったものになっている可能性があります。

DLM 側から見るとリソースは、プロセス (あるいは DLM のプロセス・グループの代表プロセス) から、そのリソースの名前に対して最初にロックが要求されたときに作成されます。このとき DLM は、リソースのロック・キューやロック値ブロックなどを含む、構造体を作成します。

リソースをロックしているプロセスが 1 つでもある間は、そのリソースは存在し続けます。リソースへの最後のロックが解除されると、DLM はリソースを削除することができます。通常は、`dlm_unlock` 関数を呼び出すことでロックを解除しますが、プロセスが突然終了したときには、通常の手続きによらずに、ロック、そしてリソースが解放されることがあります。

9.2.1 リソースの細分性

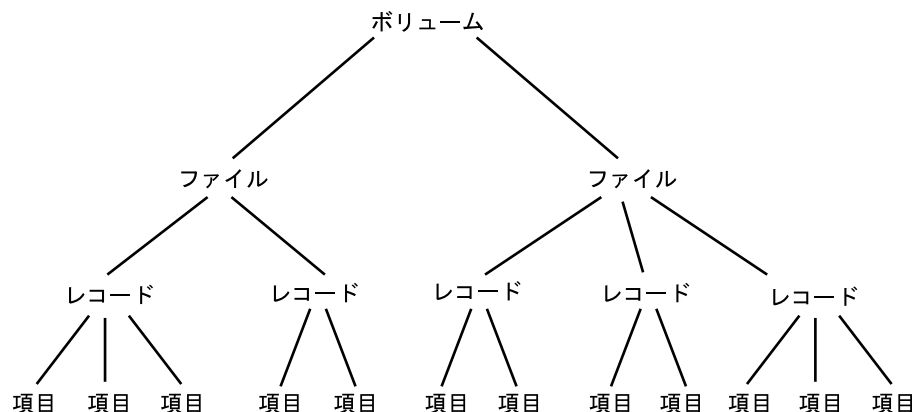
多くのリソースはさらに細かい部品に分割することができます。リソースの部品は、リソース名で識別できる限りロックすることができます。

図 9-1 は、データベースのモデルです。このデータベースはボリュームに分割され、各ボリュームはさらにファイルに分割されています。ファイルはさらにレコードに分割され、レコードはさらに項目に分割されています。

図 9-1 に示したデータベースにロック要求を出すプロセスは、データベース全体をロックすること、個々のボリューム、ファイル、レコード、あるいは項目単位でロックすることもできます。データベース全体をロックすることを、粗い細分性でのロックといいます。単体の項目をロックすることを、細かい細分性でのロックといいます。

親ロックとサブロックとは、プロセスがさまざまな細分性のレベルでロックをかけることができるように、DLM で採用している方式のことです。親ロックとサブロックについての詳細は、9.6.5 項を参照してください。

図 9-1: データベースのモデル



ZK-1099U-AI

9.2.2 ネームスペース

ネームスペースは、リソース名のコンテナと考えることができます。セキュリティ上の理由、あるいはモジュール性を持たせる目的で、関連しないアプリケーションを分離するために複数のネームスペースが存在します。

ネームスペースは、実効ユーザ ID または実効グループ ID で修飾する場合と、パブリック・ネームスペースにして、ユーザ ID およびグループ ID に関係なく、すべてのプロセスがアクセスできるようにする場合があります。

ユーザ ID ベースのネームスペースには、そのユーザ ID を持つプロセスだけがアクセスできます。グループ ID ベースのネームスペースには、そのグループのメンバだけがアクセスできます。パブリック・ネームスペースの場合は、すべてのプロセスがそのネームスペースに加わり、ロック操作を行うことができます。

ユーザ ID またはグループ ID で修飾されたネームスペースの場合、セキュリティは、このネームスペースへのアクセス権を、プロセスが持つ実効ユーザ ID または実効グループ ID に基づいて制限することで保護されます。したがって、ユーザ ID およびグループ ID のネームスペースは、クラスタ全体で一貫性がなければなりません。そのネームスペースへのアクセスがプロセスに許可されれば、このネームスペース内では、個々のロック・オペレーションは制限されません。

アプリケーションが、パブリック・ネームスペースを使用する場合は、セキュリティやアクセス・メカニズムは、アプリケーション側で用意しなければなりません。パブリック・ネームスペース内でロックへのアクセスを制御し保護する方法としては、ロックを保持しているプロセスが、保護読み取り (PR) モード以上のモードを維持し、すべてのブロック通知に応答しないようにするものがあります。こうすると、ロックを保持しているプロセスが実行している間は、他のプロセスがロック値ブロックを変更することはできません。

連携して動作するプロセスは、特定のリソースへのロックを調整できるように、同じネームスペースを使用しなければなりません。プロセスが、`dlm_lock` 関数、`dlm_locktp` 関数、`dlm_quelock` 関数、または `dlm_quelocktp` 関数を呼び出してリソースのロックを取得するためには、必要なリソースがあるネームスペースに加わらなければなりません。プロセスが、ユーザ ID またはグループ ID で修飾されたネームスペースに対して `dlm_nsjoin` 関数を呼び出すと、DLM はそのプロセスのグループ ID またはユーザ ID によって、そのプロセスがそのネームスペースへのアクセスを許可されていることを確かめます。プロセスがこのチェックにパスすると、DLM は、ネームスペースへのハンドルを返します。プロセスが、パブリック・ネームスペースに対して `dlm_nsjoin` 関数を呼び出した場合は、DLM はすぐにそのネームスペースへのハンドルを返します。

プロセスは、次に DLM 関数を呼び出してルート・ロックを取得するときに、このハンドルを指定しなければなりません。ルート・ロックとは、ネームスペース内の特定のリソースをロックするための、元となる親ロックのことです。ルート・ロックの下位には、ネームスペースへのアクセス権のチェックを受けることなく、サブロックを追加することができます。

プロセスは、最大 `DLM_NSPROC`MAX 個のネームスペースのメンバになることができます。

9.2.3 リソースの識別

DLM は以下のパラメータで、リソースを識別します。

- ネームスペース (nsp)— `dlm_nsjoin` 関数を使ってネームスペース・ハンドルを取得した後に、`dlm_lock` 関数、`dlm_locktp` 関数、`dlm_quelock` 関数、または `dlm_quelocktp` 関数を呼び出して、ネームスペースのトップ・レベル (ルート) のロックを取得します。ルート・ロックには親がありません。

- プロセスが指定するリソース名 (resnam) — プロセスが指定する名前は、そのプロセスによってロックされるリソースを示します。他のプロセスで、そのリソースにアクセスする必要がある場合は、同じ名前を使ってそのリソースを参照しなければなりません。名前とリソースの相互関係は、連携動作するプロセス間で同意された規則で決まります。
- リソース名の長さ (resnlen)
- 要求時に指定した親ロックの識別子 (parid) — 親ロック ID に 0 を指定するロック要求がキューに入れられた場合、ロック・マネージャはこれを、そのリソースへのルート・ロックの要求と見なします。ロック要求で、親ロック ID に 0 以外が指定された場合は、そのリソースへのサブロックの要求と見なします。この場合、DLM はルート・ロックが許可されている場合のみ、この要求を受け付けます。この方式では、プロセスは複数のレベルの細分性でリソースをロックすることができ、これによってロック・ツリーが構築されます。

たとえば、次の 2 つのパラメータ・セットは同じリソースを表します。

パラメータ	nsp	resnam	resnlen	parid
リソース 1	14	disk1	5	80
リソース 1	14	disk1	5	80

次の 2 つのパラメータ・セットも同じリソースを表します。

パラメータ	nsp	resnam	resnlen	parid
リソース 1	14	disk1	5	40
リソース 1	14	disk12345	5	40

次の 2 つのパラメータ・セットは異なるリソースを表します。

パラメータ	nsp	resnam	resnlen	parid
リソース 1	0	disk1	5	80
リソース 2	0	disk1	5	40

9.3 ロックの使用

分散ロック・マネージャ (DLM) の関数を使うには、プロセスは `dlm_lock` 関数、`dlm_locktp` 関数、`dlm_quelock` 関数、または `dlm_quelocktp` 関数を使って、リソースへのアクセス (つまりロック) を要求しなければなりません。この要求では、以下のようなパラメータを指定します。

- あらかじめ `dlm_nsjoin` 関数を呼び出して取得したネームスペース・ハンドル — 実効ユーザ ID またはグループ ID で修飾されたネームスペースの場合、DLM はまず、そのプロセスがそのネームスペースへのアクセス権を持っていることを検証してから、そのネームスペース内のリソースへのロックを与え、ロックを操作できるようにします。パブリック・ネームスペースは、すべてのプロセスがアクセスすることができます。ネームスペースについての詳細は、9.2.2 項を参照してください。
- リソースを表すリソース名 — リソース名の意味は、アプリケーション・プログラム側で定義します。DLM は、このリソース名を、複数のプロセスからのロック要求を対応付けるためのメカニズムとして利用します。リソース名はネームスペース内に存在します。同じ名前のリソースが異なるネームスペースにあった場合、DLM ではこれらを異なるリソース名と見なします。
- リソース名の長さ — リソース名の長さは、1 ~ `DLM_RESNAMELEN` バイトです。
- 親ロック ID — 親 ID には、ルート・ロックを要求する場合は 0 を指定し、その親のサブロックを要求する場合は、0 以外の値の親 ID を指定します。詳細は、9.2.2 項を参照してください。
- DLM がロック ID を返すメモリ・アドレス — `dlm_lock` 関数、`dlm_locktp` 関数、`dlm_quelock` 関数、および `dlm_quelocktp` 関数は、要求が許可されると、ロック ID を返します。アプリケーションでは、これ以降の操作で、つまり `dlm_cvt` 関数、`dlm_quecvt` 関数、および `dlm_unlock` 関数などの呼び出しで、このロックを参照するために、このロック ID を使います。
- ロックの要求モード — DLM の関数は、新規に要求されたロックのモードと、これと同じリソース名を持つ他のロックのモードを比較します。ロック・モードについての詳細は、9.3.1 項を参照してください。

ヌル・モードのロック (9.3.1 項を参照) は、他のすべてのロック・モードと共存でき、いつでもすぐに許可されます。

次の場合は、新規のロックはすぐに許可されます。

- 該当するリソースを他のプロセスがロックしていない場合。
- 他のプロセスが該当するリソースをロックしていて、新しく要求したロックのモードが現在のロックと共存でき、なおかつ、変換待ちキュー、または待ちキューで待たされているロックがない場合。ロック・モードの共存性についての詳細は、9.3.2 項を参照してください。

次の場合は、新規のロックは許可されません。

- 他のプロセスが既にそのリソースをロックしていて、新規要求のモードが現在のロック・モードと共存できない場合。新規の要求は、先入れ先出し (FIFO) のキューに置かれます。新規のロック要求は、リソースに現在許可されているロック・モード (リソースのグループ許可モード) が、新規のロック要求と共存できる状態になるまでここで待たされます。

プロセスは、`d1m_cvt` 関数および `d1m_quecvt` 関数を使ってロック・モードを変更することもできます。これをロック変換と呼びます。詳細は、9.3.4 項を参照してください。

9.3.1 ロック・モード

他のロック要求とリソースを共用できるかどうかは、ロックのモードで決まります。表 9-2 では、6 つのロック・モードについて説明します。

表 9-2: ロック・モード

モード	説明
ヌル (DLM_NLMODE)	リソースへのアクセスは許可しない。ヌル・モードは、後でロック変換を行うための準備として使用する。またはリソースのロックがすべて解放されても、そのリソースとリソースのコンテキストが保持されるようにするために使用する。
同時読み取り (DLM_CRMODE)	リソースへの読み取りアクセスを許可し、他の読み取りや書き込みプロセスとのリソースの共用を許す。引き続き、より細かい細分性でサブロックを実行する場合や、保護していない状態で、つまり書き込みを許したままリソースからデータを読み取る場合には、通常、同時読み取りモードが使われる。

表 9-2: ロック・モード (続き)

モード	説明
同時書き込み (DLM_CWMODE)	リソースへの書き込みアクセスを許可し、他の書き込みプロセスとのリソースの共用を許す。引き続き、より細かい細分性でロックを実行する場合や、保護していない状態でリソースに書き込む場合には、通常、同時書き込みモードが使われる。
保護読み取り (DLM_PRMODE)	リソースへの読み取りアクセスを許可し、他の読み取りプロセスとのリソースの共用を許す。リソースへの書き込みアクセスは許可しない。一般的にいう、共用ロック。
保護書き込み (DLM_PWMODE)	リソースへの書き込みアクセスを許可し、同時読み取りモードの読み取りプロセスとのリソースの共用を許す。他の書き込みプロセスによる書き込みアクセスは許さない。一般的にいう、更新ロック。
排他 (DLM_EXMODE)	リソースへの書き込みアクセスを許可し、他の読み取りプロセスや書き込みプロセスとのリソースの共用は許さない。一般的にいう、排他ロック。

9.3.2 ロックのレベルと共存性

複数のプロセスでリソースを共用できるロックを低レベルのロックと呼び、プロセスがほぼ排他的にリソースにアクセスできるロックを高レベルのロックと呼びます。ヌル・モードおよび同時読み取りモードのロックは低レベルのロックです。保護書き込みモードおよび排他モードのロックは高レベルのロックです。アクセス・モードをレベルの低い順に並べると、次のようになります。

- 1. ヌル (NL)
- 2. 同時読み取り (CR)
- 3. 同時書き込み (CW) および 保護読み取り (PR)
- 4. 保護書き込み (PW)
- 5. 排他 (EX)

同時書き込み (CW) と保護読み取り (PR) モードは、同じレベルと見なされます。

リソースに対して許可されている他のロックと共用できるロック (つまり、リソースのグループ許可モード) は、共存できるロック・モードを持っていると

いわれます。高レベルのロック・モードは、低レベルのロック・モードに比べ、他のロック・モードとの共存性が低くなります。

表 9-3 では、ロック・モードの共存性を示します。

表 9-3: ロック・モードの共存性

要求されたロック のモード	リソースのグループ許可モード					
	NL	CR	CW	PR	PW	EX
ヌル (NL)	可能	可能	可能	可能	可能	可能
同時読み取り (CR)	可能	可能	可能	可能	可能	不可能
同時書き込み (CW)	可能	可能	可能	不可能	不可能	不可能
保護読み取り (PR)	可能	可能	不可能	可能	不可能	不可能
保護書き込み (PW)	可能	可能	不可能	不可能	不可能	不可能
排他 (EX)	可能	不可能	不可能	不可能	不可能	不可能

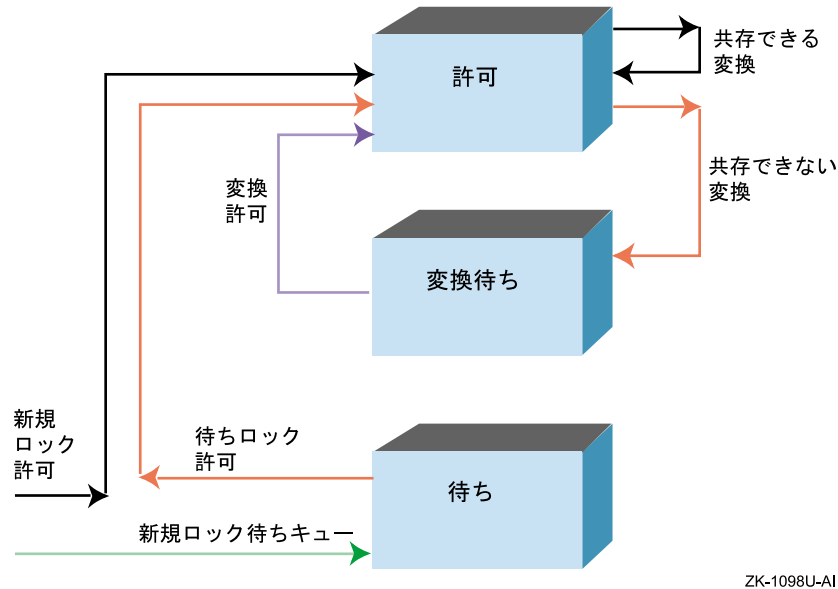
9.3.3 ロック管理のキュー

リソースに対するロックは、次の 3 種類の状態のいずれかになります。

- 許可 — ロック要求が許可された状態。
- 変換待ち — あるモードでロックが許可され、リソースの現在のグループ許可モードと共存できるモードへの変換要求が、許可されるのを待っている状態。
- 待ち — 新規のロック要求が許可されるのを待っている状態。

図 9-2 で示すように、3 種類のそれぞれの状態に対応したキューがあります。

図 9-2: 3 つのロック・キュー



既存のリソースに対して、新規にロックを要求すると、DLM は次のようにして、変換待ちキューまたは待ちキューで待っているロックが他にあるかどうかを確認します。

- いずれかのキューで他のロックが待たされている場合、新規のロック要求は待ちキューの最後に置かれます。ただし、ヌル・モードのロックを要求した場合は、キューには置かれず、その要求はすぐに許可されます。
- 変換待ちキューと待ちキューが両方とも空の場合、ロック・マネージャは新規のロックが、他に許可されているロックと共存できるかどうかを確認します。共存できる場合は、ロックが許可されます。共存できない場合は、待ちキューに置かれます。dml_lock, dml_locktp, dml_quelock, dml_quelocktp, dml_cvt, dml_quecvt のいずれかを呼び出す際に、DLM_NOQUEUE フラグを指定すると、すぐに許可されない場合はロック要求をキューにいれないように、DLM に指示することができます。この場合、リソースのグループ許可モードと共存できる場合は、ロック要求が許可され、共存できない場合は、DLM_NOTQUEUED エラーで拒否されます。

9.3.4 ロック変換

ロック変換を行うことにより、プロセスはロックのモードを変更することができます。たとえば、プロセスはリソースに対して低レベルのロックを保持しておいて、リソースへのアクセスを制限しようとした段階で、ロック変換を行うことができます。

ロック変換を行うには、変換したい、許可されているロックのロック ID を指定して、`d1m_cvt` 関数、または `d1m_quecvt` 関数を呼び出します。要求したロック・モードが現在許可されているロックと共存できる場合、変換要求はすぐに許可されます。要求したロック・モードが、許可キューにある既存のロックと共存できない場合、要求は変換待ちキューの最後に置かれます。変換要求が許可されるまでは、ロックは現在許可されているモードのままです。

DLM は、ある変換要求を許可すると、変換待ちキューに並んでいる順に共存可能な要求を許可していきます。DLM は、変換待ちキューが空になるまで、または共存できないロックに出会うまで、順番に要求を許可していきます。

変換待ちキューが空の場合、DLM は待ちキューを調べます。待ちキューの先頭のロック要求が、現在許可されているロックと共存できれば、要求が許可されます。DLM は、待ちキューが空になるまで、または共存できないロックに出会うまで、順番に要求を許可していきます。

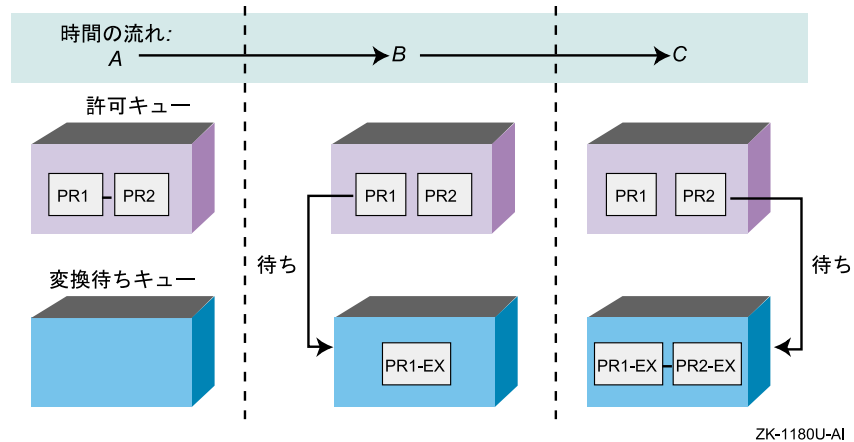
9.3.5 デッドロックの検出

DLM は次の 2 種類の形態のデッドロックを検出することができます。

- 変換デッドロック — 変換デッドロックは、変換要求が持っている許可モードが、変換待ちキューの 1 つ前の変換要求で要求されたモードと共存できない場合に起こります。これを、図 9-3 の例を使って説明します。1 つのリソースに対して、PR モードのロックが 2 つ許可されています。つまり、そのリソースの許可モードは PR です。PR モードのロックの 1 つがモードを EX モードに変換しようとし、その結果変換キューで待つこととなりました。次に、2 つ目の PR モードのロックも同様に、モードを EX に変換しようとする、これは、変換待ちキューの 1 つ目のロック要求の後で待たなければなりません。しかし、1 つ目のロックが要求するモード (EX) は、2 つ目のロックに現在許可されている PR モードとは共存できません。このため、1 つ目のロック要求が許可されることはありません。2 つ目のロック要求

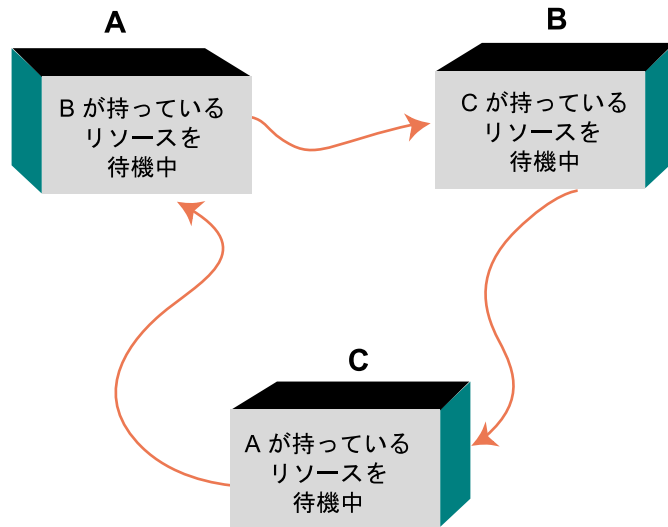
は、変換待ちキューの中で、1 つ目のロック要求の後ろで待っているため、この要求が許可されることもありません。

図 9-3: 変換デッドロック



- 複数リソース・デッドロック — 複数リソース・デッドロックは、一連のプロセスが互いに待ち合い、待ち循環に陥った場合に起こります。これを、図 9-4 の例を使って説明します。3 つのプロセスの要求が、それぞれのリソースのキューに入っています。それぞれの現在のロックが解放されるまで、またはより低いモードに変換されるまで、これらのリソースにはアクセスできません。それぞれのプロセスが、別のプロセスがロック要求を解放するのを互いに待っています。

図 9-4: 複数リソース・デッドロック



ZK-1097U-AI

DLM は、変換デッドロック、または複数リソース・デッドロックが発生したことを検出すると、犠牲にするロックを選び、これを使ってデッドロックを解消します。犠牲にするロックは任意に選択されますが、選ばれたロックは、変換待ちキュー、または待ちキューのどちらかに置かれることが保証されます（つまり、許可キューには置かれません）。

DLM は、`dln_lock` 関数、`dln_locktp` 関数、`dln_cvt` 関数のいずれか呼び出し、犠牲にされたプロセスに、最終的な終了状態コードとして `DLM_DEADLOCK` を返します。あるいは、`dln_quelock` 関数、`dln_quelocktp` 関数、`dln_quecvt` 関数のいずれか呼び出す際に指定された完了ルーチンへのパラメータ `completion_status` として、この状態を渡します。許可されたロックが取り消されることはありません。変換待ちキューおよび待ちキューのロック要求だけが、`DLM_DEADLOCK` 状態コードを受け取る可能性があります。

注意

DLM がデッドロックを解消する際に、どのロックを選択するかを想定しないでください。また、セマフォやファイル・ロックのような他のサービスを DLM と一緒に使用した場合は、検出できないデッドロックが発生する可能性があります。DLM で検

出できるデッドロックは、DLM のロックに関連して発生したデッドロックのみです。

9.4 ロックのキューからの削除

プロセスは、リソースをロックする必要がなくなった場合、`dlm_unlock`関数を呼び出して、ロックを解放することができます。

ロックが解放されると、そのロック要求は、どのキューにあっても削除されます。ロックは、許可キュー、待ちキュー、変換待ちキューのいずれのキューからも外されます。あるリソースの最後のロックがキューから外されると、そのリソースは DLM のデータベースから削除されます。

`dlm_unlock` 関数では、`valb_p` パラメータと `DLM_VALB` フラグを指定することで、リソースのロック値ブロックへ書き込んだり、内容を無効にすることができます。キューから外されるロックに PW モードまたは EX モードが許可されている場合、プロセスの値ブロックの内容が、リソースの値ブロックに格納されます。キューから外されるロックが上記以外のモードの場合、ロック値ブロックは使用されません。`DLM_INVVALBLK` フラグを指定した場合は、リソースのロック値ブロックに、無効であることを示す印が付けられます。

`dlm_unlock` 関数では以下のフラグを使用できます。

- `DLM_DEQALL` フラグは、表 9-4 で示す `lkid_p` パラメータの値に従って、プロセスが保持するすべてのロックをキューから外すか、あるいはロックのサブ・ツリーをキューから外すことを指示します。

表 9-4: `dlm_unlock` 関数の呼び出しでの `DLM_DEQALL` フラグの使用

<code>lkid_p</code>	<code>DLM_DEQALL</code>	結果
<code>≠ 0</code>	設定しない	<code>lkid_p</code> で指定されたロックのみが解放される。
<code>≠ 0</code>	設定	指定されたロックのすべてのサブロックが解放される。 <code>lkid_p</code> で指定されたロックは解放されない。
<code>= 0</code>	設定しない	無効ロック ID の条件値 (<code>DLM_IVLOCKID</code>) が返される。
<code>= 0</code>	設定	プロセスが保持しているすべてのロックが解放される。

- `DLM_INVVALBLK` フラグを指定すると、DLM は、許可キューまたは変換待ちキューにある PW モードまたは EX モードのロックの、リソースのロック値ブロックを無効にします。リソースのロック値ブロックは、再度書き込まれるまで、無効の印が付いたままになります。ロック値ブロックについての詳細は、9.6.6 項を参照してください。
- `DLM_VALB` フラグを指定すると、DLM は、許可キューまたは変換待ちキューにある PW モードまたは EX モードのロックについて、リソースのロック値ブロックへ書き込みます。

`DLM_VALB` フラグと `DLM_INVVALBLK` フラグの両方を、同じ要求で指定することはできません。

9.5 変換要求の取り消し

`dml_cancel` は、ロック変換を取り消す関数です。プロセスは、ロック要求がまだ許可されていない場合だけ、つまり、ロック要求が変換待ちキューにある場合だけ、ロック変換を取り消すことができます。取り消しを実行すると、変換待ちキュー内のロックは、変換要求を出す前に許可されていたモードに戻ります。ロックの値 `blkrtm` および `notprm` も元の値に戻ります。DLM は、変換要求で指定されている完了ルーチン呼び出しで、要求が取り消されたことを伝えます。返される状態コードは、`DLM_CANCEL` です。

9.6 高度なロック技術

ここまでの節では、すべてのアプリケーションに役立つロック技術および概念について説明してきました。これ以降の各項では、分散ロック・マネージャ (DLM) のより特化された機能について説明します。

9.6.1 ロック要求の非同期完了

`dml_lock` 関数、`dml_locktp` 関数、および `dml_cvt` 関数は、ロック要求が許可されるか失敗したときに完了し、結果を示す状態コードが返されます。

ロック要求が完了するまでアプリケーションを待たせたくない場合は、`dml_quelock` 関数、`dml_quelocktp` 関数、および `dml_quecvt` 関数を使用するようにしてください。これらの関数は、ロック要求をキューに入れた後に、呼び出し元のプログラムに制御を返します。これらの関数が返す状態値は、要求が正常にキューに入れられたか、または拒否されたかを知らせるも

のです。要求がキューに入れられた後、呼び出し元のプログラムは、要求が許可されるまではリソースにアクセスすることはできません。

`dlm_quelock` 関数、`dlm_quelocktp` 関数、および `dlm_quecvt` 関数の呼び出しでは、完了ルーチンのアドレスを指定しなければなりません。ロック要求に成功した場合、または失敗した場合に、完了ルーチンが実行されます。DLM は、ロック要求が成功したか、または失敗したかを示す状態情報を完了ルーチンに渡します。

注意

DLM からの完了通知を受けたいアプリケーションでは、この通知が必要な最初のロック要求を出す前に一度、`dlm_set_signal` 関数を呼び出さなければなりません。これ以外に、アプリケーションが `dlm_notify` 関数を定期的に呼び出す方法もあります。`dlm_notify` 関数を使うと、プロセスは、保留になっている通知をポーリングし、`dlm_set_signal` 関数を呼び出さなくても、完了通知の配信を要求することができます。ただし、このポーリングによる方法は、お勧めしません。

9.6.2 同期完了の通知

DLM には、ロック要求がその場で許可されたかどうかをプロセスが確認する機能があります。つまり、そのロックが変換待ちキュー、または待ちキューに入れられなかったかどうかを確認できます。この機能を使うと、シグナルの配信と、その結果の完了ルーチンの実行によるオーバーヘッドが回避されるため、ほとんどのロックがその場で許可される（これが一般的）アプリケーションでは、性能を改善することができます。アプリケーションでは、要求を処理する際にこの機能を使って、衝突するロックがないかテストすることもできます。

この機能のメカニズムは、次のようになっています。

- `dlm_lock` 関数、`dlm_locktp` 関数、`dlm_cvt` 関数、`dlm_quelock` 関数、`dlm_quelocktp` 関数、`dlm_quecvt` 関数のいずれかを呼び出す際に、`DLM_SYNCSTS` フラグを設定し、その場でロックが許可されると、これらの関数は、呼び出し元に状態値 `DLM_SYNC` を返します。`dlm_quelock` 関数、`dlm_quelocktp` 関数、および `dlm_quecvt` 関数の場合、DLM は完了通知を配信しません。

- `dml_quelock` 関数, `dml_quelocktp` 関数, および `dml_quecvt` 関数を呼び出して開始したロック要求が, その場で完了しなかった場合は, これらの関数から状態値 `DLM_SUCCESS` が返されます。この値は, 要求がキューに入れられたことを示します。ロックが正常に許可されるか, またはロックに失敗すると, DLM は完了通知を配信します。

9.6.3 ブロック通知

DLM 関数を使用するアプリケーションの中には, 他のプロセスがリソースをロックするのを防いでいるかどうかを, プロセスが意識しなければならないものがあります。DLM では, ブロック通知を使って, プロセスにこのことを知らせます。ブロック通知を使うには, ロック要求の `blkrttn` パラメータにブロック通知ルーチンのアドレスを指定しなければなりません。これにより, このロックのために, 別のロックの許可が防げられたときに, ブロック通知が配信され, ブロック通知ルーチンが実行されます。

DLM は, ブロック通知ルーチンに以下のパラメータを渡します。

<code>notprm</code>	ブロックされたロックのコンテキスト・パラメータ。このパラメータは, ブロックされたロックのロック要求で, <code>dml_lock</code> 関数, <code>dml_locktp</code> 関数, <code>dml_quelock</code> 関数, <code>dml_quelocktp</code> 関数, <code>dml_cvt</code> 関数, <code>dml_quecvt</code> 関数のいずれかの関数の呼び出し元プロセスが指定したものです。
<code>blocked_hint</code>	最初にブロックされたロックの <code>hint</code> パラメータ。このパラメータは, 最初にブロックされたロックのロック要求で, <code>dml_lock</code> 関数, <code>dml_locktp</code> 関数, <code>dml_quelock</code> 関数, <code>dml_quelocktp</code> 関数, <code>dml_cvt</code> 関数, <code>dml_quecvt</code> 関数のいずれかの関数の呼び出し元プロセスが指定したものです。
<code>lkid_p</code>	ブロックされたロックのロック ID へのポインタ。
<code>blocked_mode</code>	最初にブロックされたロックで要求されたモード。

通知が配信されるまでに, 次のような状況になっている可能性があります。

- ロックはブロックされたまま。

- ブロックされたロックがアプリケーションによって解放され、実際にはブロックされているロックがない。
- ブロックされたロックが、デッドロックを解除するための犠牲にされ、要求が失敗した。
- ブロックされたロックがアプリケーションによって解放され、別のロックがキューに入れられブロックされている。したがって、まったく別のロックが実際にはブロックされている。
- 最初にブロックされたロックの後に他のロックがたまっている、つまりブロックされたロックがキューの後に入れられている。

こうした状況が発生する可能性があるため、DLM では、ブロック・ルーチンの実行時の、*blocked_hint* パラメータ、および *blocked_mode* パラメータの有効性を保証することはできません。

注意

DLM からブロック通知の配信を受けたいアプリケーションでは、ブロック通知を必要とする最初のロック要求を発行する前に、*dln_set_signal* 関数を (1 度) 呼び出さなければなりません。

また、*dln_set_signal* の呼び出し時に指定したシグナルがブロックされた場合、このシグナルのブロックが解除されるまで、ブロック通知は配信されません。これ以外に、*dln_notify* 関数を定期的に呼び出す方法があります。*dln_notify* 関数を使うと、プロセスは、保留になっている通知をポーリングし、これらの通知を配信するように要求することができます。ただし、このポーリングによる方法は、お勧めしません。

9.6.4 ロック変換

ロック変換には、次のような機能があります。

- ロック・モードの変換 (上方向または下方向) — DLM では、低レベルでロックし、それを必要に応じて高レベルのモードに変換することができます。通常、プログラムでは、データの書き込み中は、排他 (EX) モードまたは保護書き込み (PW) モードのロックが必要です。しかし、そのプログラムでの書き込み処理が常に必要なわけではないので、そのプログラムの開始から終了までリソースを排他的にロックしておきたくないこ

ともあります。EX モード，または PW モードのロックがかかっていると，他のプロセスがそのリソースにアクセスできません。ロック変換機能を使うことにより，プロセスは，最初は低レベルのロックを要求し，データの書き込みが必要になったときだけロックのモードを上位レベルのロック・モード（たとえば PW モード）に変換することができます。しかし，プロセスはリソースへの書き込みを常時必要とするわけではないので，書き込みが終了したら，下位レベルのロック・モードにロックを変換してください。

注意

別のタイプのアプリケーション，たとえば，書き込みアクセスの共用がそれほど重要でないアプリケーションでも，DLM のロック変換機能を使うことができます。たとえば，アプリケーションのあるインスタンスがマスタになり，同じアプリケーションの別のインスタンスがスレーブ・インスタンス，または 2 次インスタンスになるような場合，ロックおよびロック変換を使って，マスタ・インスタンスが異常終了したときの回復手順を高速化することができます。

- リソースのロック値ブロックに格納されている値の維持 — ロック値ブロックは，同じリソースをロックしているプロセス間で共用する小さなメモリ領域です。アプリケーションによっては，ロック値ブロックを使う必要があります。バージョン番号やその他のデータをロック値ブロックに格納している場合，値ブロックが消されないように，リソースに対して少なくとも 1 つのロックをかけておく必要があります。この場合プロセスは，そのリソースへのアクセスが完了したときに，ロックをキューから外すのではなく，ヌル・ロックに変換します。ロック値ブロックの使用についての詳細は，9.6.6 項を参照してください。
- 一部のアプリケーションの性能改善 — アプリケーションの中には，性能を改善するために，ロックする可能性のあるリソースすべてを，初期化時にヌル・モードでロックするものがあります。ヌル・モードのロックは，必要に応じて上位レベルのロックに変換することができます。多くの場合，変換の要求の方が，新規ロックの要求よりも高速に処理されます。これは，変換の要求では必要なデータ構造が既に構築されているためです。ただし，プログラムを実行している間，ロックをか

けたままにしておくと、システムの動的メモリが消費されます。したがって、必要なロックすべてを NL モードで作成し、必要に応じてこれらを変換するテクニックを用いれば、メモリ使用量は増えますが、性能を改善することができます。

9.6.4.1 ロック変換のキューイング

ロック変換を行う場合、`d1m_cvt` 関数または `d1m_quecvt` 関数を呼び出します。変換するロックは、`lkid_p` パラメータで指定します。ロック変換を要求するときには、そのロックは既に許可されていなければなりません。

9.6.4.2 変換待ちキューへの強制的配置

特定のリソースへのアクセスがより公平に行われるようにするために、許可できる変換要求を、強制的に変換待ちキューに入れることができます。DLM_QUECVT フラグを設定して変換要求を行うと、その要求は、既にキューに入っている変換要求の後ろに強制的に置かれます。このように、DLM_QUECVT フラグを指定して、他のロックが許可されるチャンスをつくることができます。ただし、キューに入っている変換要求がない場合は、その変換要求はすぐに許可されます。

DLM_QUECVT は、可能な変換要求のサブセットに対してだけ有効です。表 9-5 は、DLM_QUECVT フラグを指定した場合に許される、変換要求の組み合わせを示しています。無効な変換要求は、DLM_BADPARAM 状態が返されて、失敗します。

表 9-5: DLM_QUECVT フラグを指定した場合に許される変換

取得しているロック のモード	変換後のロックのモード					
	NL	CR	CW	PR	PW	EX
ヌル (NL)	—	可	可	可	可	可
同時読み取り (CR)	—	—	可	可	可	可
同時書き込み (CW)	—	—	—	—	可	可
保護読み取り (PR)	—	—	—	—	可	可
保護書き込み (PW)	—	—	—	—	—	—
排他 (EX)	—	—	—	—	—	—

9.6.5 親ロック

プロセスが `d1m_lock` 関数、`d1m_locktp` 関数、`d1m_quelock` 関数、`d1m_quelocktp` 関数のいずれかを呼び出してロック要求を出す際に、*parid* パラメータに親ロック ID を指定すると、新規のロックに対して親ロックを宣言することができます。親ロックを持つロックを、サブロックと呼びます。親ロックが許可された後でなければ、その親に属するサブロックは許可されません。モードは、親と同じモードでも、別のモードでもかまいません。

親ロックとサブロックを使うメリットは、細かい細分性のリソースに、高レベルのサブロック (保護書き込みモード、または排他モード) をかけ、一方で粗い細分性のリソースに、低レベルのロック (同時読み取りモード、または同時書き込みモード) をかけられることです。たとえば、低レベルのロックで、あるファイル全体へのアクセスを制御し、それと同時に、それより上位のレベルで、そのファイル内の個々のレコードやデータ項目にサブロックをかけることができます。

データベースへのアクセスが必要なプロセスがたくさんある場合を考えます。データベースを、ファイルのロックとレコードのロックの 2 つのレベルでロックできるとします。ファイル内のすべてのレコードを更新する場合、ファイル全体をロックして、他のロックは行わずにレコードを更新した方が、高速で効率もよくなります。しかし、特定のレコードを更新する場合は、必要に応じてレコードをロックする方が効率的です。

親ロックをこのように使う場合、すべてのプロセスの要求で、ファイルをロックします。すべてのレコードを更新するプロセスは、該当するファイルに対して、保護書き込み (PW) モード、または排他 (EX) モードのロックを要求しなければなりません。個々のレコードを更新するプロセスは、ファイルに対して同時書き込み (CW) モードのロックを要求し、その後、個々のレコードに対して、PW モードまたは EX モードのサブロックをかけます。

このように、すべてのレコードにアクセスする必要があるプロセスはファイルをロックすることでアクセスでき、一方で、ファイルを共用するプロセスは、個々のレコードをロックすることができます。これにより、多くのプロセスが、ファイル・レベルのロックを同時書き込みモードで共用する一方で、サブロックを使って特定のレコードを更新することができます。

9.6.6 ロック値ブロック

ロック値ブロックは `DLM_VALBLKSIZE` のサイズの符号なしロングワードの構造体です。この構造体は、プロセスが DLM の関数を呼び出す際に `valb_p` パラメータおよび `DLM_VALB` オプションを指定することで、リソースに対応付けます。ロック・マネージャがリソースを作成する際に、このリソースのロック値ブロックも作成されます。DLM は、リソースへのロックがなくなるまで、リソースのロック値ブロックを維持します。

プロセスが新しいロックを要求するときに、`DLM_VALB` オプションを指定し、`valb_p` パラメータに有効なアドレスを指定して、これが許可された場合、リソースのロック値ブロックの内容は、プロセスのロック値ブロックにコピーされます。

プロセスが、PW モードまたは EX モードから、同等レベルまたは下位レベルのモードへのロック変換で、`valb_p` パラメータと `DLM_VALB` オプションを指定した場合、プロセスのロック値ブロックの内容が、リソースのロック値ブロックに格納されます。

このようにしてプロセスは、リソースの所有権とともに、ロック値ブロックの値を渡し、更新することができます。表 9-6 では、ロック変換によって、プロセスおよびリソースのロック値ブロックの内容が、どのように変化するかを示します。

表 9-6: ロック変換によるロック値ブロックへの影響

取得しているロック のモード	ロック変換後のモード					
	NL	CR	CW	PR	PW	EX
ヌル (NL)	読取り	読取り	読取り	読取り	読取り	読取り
同時読み取り (CR)	—	読取り	読取り	読取り	読取り	読取り
同時書き込み (CW)	—	—	読取り	—	読取り	読取り
保護読み取り (PR)	—	—	—	読取り	読取り	読取り
保護書き込み (PW)	書込み	書込み	書込み	書込み	書込み	読取り
排他 (EX)	書込み	書込み	書込み	書込み	書込み	書込み

PW モードまたは EX モードで許可されているロックが `dml_unlock` 関数を使って解放されて、ロック値ブロックのアドレスが `valb_p` パラメータに指

定され、`DLM_VALB` オプションが指定されていると、プロセスのロック値ブロックの内容がリソースのロック値ブロックに書き込まれます。解放されるロックがこれ以外のモードの場合は、ロック値ブロックは使用されません。

リソースのロック値ブロックが無効になる場合があります。このような場合、`DLM` は `valb_p` パラメータを指定して関数を呼び出したプロセスに対し、`DLM_SUCCVALNOTVALID` または `DLM_SYNCVALNOTVALID` の終了状態を返して警告します。次のような場合に、リソースのロック値ブロックが無効になることがあります。

- PW モードまたは EX モードでリソースをロックしているプロセスが異常終了した場合。
- ロックに関与しているノードが停止し、そのノード上のプロセスが PW モードまたは EX モードでリソースをロックしていた場合、あるいはロックしていた可能性がある場合。
- PW モードまたは EX モードでリソースにロックをかけているプロセスが、ロックをキューから外すために `dml_unlock` 関数を呼び出し、`flags` パラメータに `DLM_INVVALBLK` フラグを指定している場合。

9.7 DLM 関数を使ったローカル・バッファ・キャッシング

アプリケーションは、分散ロック・マネージャ (DLM) を使って、ローカル・バッファ・キャッシング (分散型バッファ管理とも呼ぶ) を行うことができます。ローカル・バッファ・キャッシングを行うと、多くのプロセスが、データ (たとえばディスク・ブロックなど) のコピーを、各プロセスのローカル・バッファに持つことができます。他のプロセスによる変更で、バッファ内のデータが有効でなくなった場合は、各プロセスに通知されます。変更の頻度が低いアプリケーションでは、バッファのコピーをローカルに持つことにより、入出力動作を大幅に削減することができます。このため、ローカル・バッファ・キャッシングあるいは分散型バッファ管理という名前で呼ばれています。ロック値ブロック、またはブロック通知 (あるいはその両方) を使って、バッファ・キャッシングを行うことができます。

9.7.1 ロック値ブロックを使う方法

ロック値ブロックを使ってローカル・バッファ・キャッシングを行うため、バッファをキャッシュしている各プロセスは、各バッファの現在の内容に対応するリソースに、ヌル (NL) モードのロックをかけます。この説明では、

バッファにディスク・ブロックが格納されていることを前提とします。各リソースに対応するロック値ブロックは、ディスク・ブロックのバージョン番号を入れておくために使います。あるディスク・ブロックに初めてロックがかけられたとき、アプリケーションは、プロセスのロック値ブロックにある、そのディスク・ブロックの現在のバージョン番号を返します。

バッファの内容をキャッシュすると、このバージョン番号はバッファの内容とともに保存されます。バッファの内容を再利用するには、バッファを読み取るのか書き込むのかによって、NL モードのロックを、保護読み取り (PR) モードまたは排他 (EX) モードに変換しなければなりません。この変換で、ディスク・ブロックの最新のバージョン番号が返されます。アプリケーションでは、ディスク・ブロックのバージョン番号と、保存していたバージョン番号を比較します。これらが等しい場合、キャッシュされているコピーは有効です。これらが等しくない場合には、アプリケーションはそのディスク・ブロックのコピーを、ディスクから新しく読み取らなければなりません。

プログラムでバッファを変更する場合は必ず、変更したバッファをディスクに書き込み、バージョン番号をインクリメントして、対応するロックを NL モードに変換します。このようにすることで、次のプロセスが同じバッファのローカル・コピーを使おうとした場合、バージョン番号が異なることに気づき、キャッシュしてあるバッファ (無効になった) を使わずに、ディスクから最新のコピーを読み取ります。

9.7.2 ブロック通知を使う方法

ブロック通知は、現在ロックが許可されているプロセスに、他のプロセスが共存できないモードで同じリソースにアクセスしようとしてキューに置かれたことを通知するために使用します。

ローカル・バッファ・キャッシングを行う際に、ブロック通知を使う方法は 2 つあります。1 つは、バッファへの書き込みを遅らせる方法で、もう 1 つは、ロック値ブロックを使わずにローカル・バッファ・キャッシングを代替する方法です。

9.7.2.1 バッファへの書き込みの延期

ローカル・バッファ・キャッシングを行っているときには、EX モードのロックを解放する前に、バッファへの変更をディスクに書き込まなければなりません。特に短時間に多くの変更が見込まれる場合は、変更処理を行って

いる間 EX モードでロックしておき、バッファへの書き込みを 1 回だけにすると、ディスクへの入出力を減らすことができます。

しかし、これでは、その間は他のプロセスが同じディスク・ブロックを使えなくなります。EX モードでロックしているプロセスに、ブロック通知ができれば、このような事態は避けられます。この通知では、ロックしているプロセスに、他のプロセスで同じディスク・ブロックを使う必要があることを知らせます。この通知を受け取った場合、EX モードでロックしているプロセスは、バッファをディスクに書き込んで、ロック・モードを NL モードに変換することができます。これにより、他のプロセスがそのディスク・ブロックにアクセスできるようになります。ただし、同じディスク・ブロックにアクセスしたいプロセスが他にない場合は、最初のプロセスは変更を何度も行ってから、書き込みを 1 回だけで済ませることができます。

注意

ブロック通知を受け取ったプロセスは、次のブロック通知を受け取れるように、ロックを変換しなければなりません。

9.7.2.2 バッファのキャッシング

ブロック通知を使ってローカル・バッファ・キャッシングを実行する場合、バッファへの処理が終わったプロセスは、PR モードまたは EX モードのロックを NL モードに変換しません。代わりに、他のプロセスが共存できないモードで同じリソースをロックしようとするたびに、ブロック通知を受け取るようにします。この方式では、次にそのプロセスがバッファを使おうとしたときではなく、別の書き込みプロセスでこのバッファが必要になったときに、このプロセスでキャッシュしているバッファが無効であることが通知されます。

9.7.3 バッファのキャッシング方式の選択

ローカル・バッファ・キャッシングを実行する場合、バージョン番号を使う方法と、ブロック通知のどちらを使えばよいかは、アプリケーションの性格に依存します。バージョン番号を使用するアプリケーションでは、ロック変換が増え、ブロック通知を使用するアプリケーションでは、通知の配信が増えます。これらの方式は共存することに注意してください。一方の方式を使っているプロセスがあって、同時に、もう一方の方式を使っているプロセ

スがあるということが可能です。一般的には、ブロック通知は、衝突が少ない環境で選択し、バージョン番号は衝突が多い環境で選択します。組み合わせて使ったり、必要に応じて切り替えて使うこともできます。

組み合わせて使うときに、プロセスが短時間にバッファの内容を再利用すると予測される場合は、ブロック通知を使い、すぐにバッファの内容を再利用すると予測できない場合は、バージョン番号を使います。

必要に応じて切り替えて使う場合、アプリケーションはブロック通知と変換の発生頻度を評価します。ブロック通知が頻繁に届く場合、バージョン番号を使う方式に切り替え、変換がたくさん発生し、キャッシュされたコピーが長時間変更されず有効な場合は、ブロック通知を使う方式に切り替えます。

例として、あるプロセスがデータベースの状態を絶えず表示し続け、他のプロセスが時々このデータベースをアップデートする場合を考えてみます。バージョン番号を使った場合、表示するプロセスはデータベースのコピーが有効かどうか、ロック変換を行って常に検証しなければなりません。これに対し、ブロック通知を使った場合、表示するプロセスは、データベースがアップデートされるたびに通知を受けます。ただし、アップデートが頻繁に行われる場合は、ブロック通知を頻繁に配信するよりは、バージョン番号を使う方式の方が効率的です。

9.8 分散ロック・マネージャの関数のコーディング例

次のプログラム例では、ネームスペースに加わり、そのネームスペースのリソースの初期ロックを取得するためにアプリケーションが使用する基本的な手法について説明します。また、ロック変換、ロック値ブロックの使用方法、ブロック通知ルーチンの使用方法といった DLM の主要な概念についても、この中で具体的に説明します。

例 9-1 に示すプログラム `api_ex_master.c` および `api_ex_client.c` は、ディレクトリ `/usr/examples/cluster` にあり、`TCRMANxxx` サブセットがインストールされている場合は、同じクラスタ・メンバ上、または異なるクラスタ・メンバ上で、並列に実行することができます。どちらのプログラムも、同じユーザ ID (UID) を持つアカウントから実行しなければなりません。また、プログラム `api_ex_master.c` を先に実行しなければなりません。このプログラムの出力結果は、次のようになります。

```
% api_ex_master &
api_ex_master: grab a EX mode lock
api_ex_master: value block read
api_ex_master: expected empty value block got <>
```

```

api_ex_master: start client and wait for the blocking notification to
                continue
% api_ex_client &
    api_ex_client: grab a NL mode lock
    api_ex_client: value block read
    api_ex_client: expected empty value block got <>
    api_ex_client: converting to NL→EX to get the value block.
    api_ex_client: should see blocking routine run on master
api_ex_master: blk_and_go hold the lock for a couple of seconds
api_ex_master: blk_and_go sleeping
api_ex_master: blk_and_go sleeping
api_ex_master: blk_and_go sleeping
api_ex_master: blk_and_go setting done
api_ex_master: now convert (EX→EX) to write the value block <abc>
api_ex_master: blk_rtn: down convert to NL
api_ex_master: waiting for blocking notification
api_ex_master: trying to get the lock back as PR to read value block
    api_ex_client: blk_rtn: dequeue EX lock to write value block <>
    api_ex_client: hold the lock for a couple of seconds
    api_ex_client: sleeping
    api_ex_client: sleeping
    api_ex_client: sleeping
    api_ex_client: value block read
    api_ex_client: expected <abc> got <abc>
    api_ex_client: sleeping waiting for blocking notification
    api_ex_client: done
api_ex_master: value block read
api_ex_master: expected <efg> got <efg>
api_ex_master done

```

例 9-1: ロック , ロック値ブロック , およびロック変換

```

/*****
 *
 *          api_ex_master.c
 *
 *****/

/* cc -g -o api_ex_master api_ex_master.c -ldlm */

#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <signal.h>

#include <sys/dlm.h>

char *resnam = "dist shared resource";
char *prog;
int done = 0;

#ifdef DLM_DEBUG
int dlm_debug = 2;
#define Cprintf if(dlm_debug)printf
#define Dprintf if(dlm_debug >= 2 )printf
#else /* DLM_DEBUG */
#define Cprintf ;
#define Dprintf ;
#endif /* DLM_DEBUG */

void
error(dlm_lkid_t *lk, dlm_status_t stat)
{

```

例 9-1: ロック , ロック値ブロック , およびロック変換 (続き)

```
        printf("%s: lock error %s on lkid 0x%x\n",
               prog, dlm_sperrno(stat), lk);
        abort();
    }

    void
    blk_and_go(callback_arg_t x, callback_arg_t y,
               dlm_lkid_t *lk, dlm_lkmode_t blkmode)
    {
        int i;

        printf("%s: blk_and_go hold the lock for a couple of seconds\n",
               prog);
        for (i = 0; i < 3; i++) {
            printf("%s: blk_and_go sleeping\n", prog);
            sleep(1);
        }
        printf("%s: blk_and_go setting done\n", prog);
        /* done waiting */
        done = 1; [13]
    }

    void
    blkrtm(callback_arg_t x, callback_arg_t y,
           dlm_lkid_t *lk, dlm_lkmode_t blkmode)
    {
        dlm_status_t stat;

        Cprintf("%s: blkrtm: x 0x%x y 0x%x lkid 0x%x blkmode %d\n",
                prog, x, y, *lk, blkmode);
        printf("%s: blkrtm: down convert to NL\n", prog);
        if ((stat = dlm_cvt(lk, DLM_NLMODE, 0, 0, 0, 0, 0)) != DLM_SUCCESS)
            error(lk, stat); [16]
        /* let waiters know we're done */
        done = 1;
    }

    main(int argc, char *argv[])
    {
        int resnlen, i;
        dlm_lkid_t lkid;
        dlm_status_t stat;
        dlm_valb_t vb;
        dlm_nsp_t nsp;

        /* this prog must be run first */

        /* first we need to join a namespace */
        if ((stat = dlm_nsjoin(getuid(), &nsp, DLM_USER)) != DLM_SUCCESS) { [1]
            printf("%s: can't join namespace\n", argv[0]);
            error(0, stat);
        }

        prog = argv[0];

        /* now let DLM know what signal to use for blocking routines */
        dlm_set_signal(SIGIO, &i); [2]
        Cprintf("%s: dlm_set_signal: i %d\n", prog, i);

        resnlen = strlen(resnam); [3]
        Cprintf("%s: dlm_set_signal: i %d\n", prog, i);

        printf("%s: grab a EX mode lock\n", prog);
        stat = dlm_lock(nsp, (uchar_t *)resnam, resnlen, 0, &lkid,
                       DLM_EXMODE, &vb, (DLM_VALB | DLM_SYNCSTS), 0, 0,
                       blk_and_go, 0); [4]

        /*
         * since we're the only one running it
         * had better be granted SYNC status
         */
        if (stat != DLM_SYNC) { [5]
            printf("%s: dlm_lock failed\n", prog);
        }
    }
}
```

例 9-1: ロック , ロック値ブロック , およびロック変換 (続き)

```
        error(&lkid, stat);
    }
    printf("%s: value block read\n", prog);
    printf("%s: expected empty value block got <%s>\n",
        prog, (char *)vb.valblk);
    if (strlen((char *)vb.valblk)) { 6
        printf("%s: lock: value block not empty\n", prog);
        error(&lkid, stat);
    }
    printf("%s: start client and wait for the blocking
        notification to continue\n",
        prog);
    while (!done)
        sleep(1); 7

    done = 0;
    /* put a known string into the value block */
    (void) strcat((char *)vb.valblk, "abc"); 14
    printf("%s: now convert (EX→EX) to write the value block <%s>\n",
        prog, (char *)vb.valblk);
    /* use a new blocking routine */
    stat = dlm_cvt(&lkid, DLM_EXMODE, &vb, (DLM_VALB | DLM_SYNCSTS),
        0, 0, blktrn, 0); 15
    /*
     * since we own (EX) the resource the
     * convert had better be granted SYNC
     */
    if (stat != DLM_SYNC) {
        printf("%s: convert failed\n", prog);
        error(&lkid, stat);
    }

    printf("%s: waiting for blocking notification\n", prog);
    while (!done)
        sleep(1);
    printf("%s: trying to get the lock back as PR to read value block\n",
        prog);
    stat = dlm_cvt(&lkid, DLM_PRMODE, &vb, DLM_VALB, 0, 0, 0, 0); 19
    if (stat != DLM_SUCCESS) {
        printf("%s: error on conversion lock\n", prog);
        error(&lkid, stat);
    }
    printf("%s: value block read\n", prog);
    printf("%s: expected <efg> got <%s>\n", prog, (char *)vb.valblk);
    /* compare to the other known string */
    if (strcmp((char *)vb.valblk, "efg")) {
        printf("%s: main: value block mismatch <%s>\n",
            prog, (char *)vb.valblk);
        error(&lkid, stat); 23
    }
    printf("%s done\n", prog); 24
    exit(0);
}

/*****
 *
 *          api_ex_client.c
 *
 *****/

/* cc -g -o api_ex_client api_ex_client.c -ldlm */

#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <signal.h>

#include <sys/dlm.h>

char *resnam = "dist shared resource";
```

例 9-1: ロック , ロック値ブロック , およびロック変換 (続き)

```
char *prog;
int done = 0;

#ifdef DLM_DEBUG
int dlm_debug = 2;
#define Cprintf if(dlm_debug)printf
#define Dprintf if(dlm_debug >= 2 )printf
#else /* DLM_DEBUG */
#define Cprintf ;
#define Dprintf ;
#endif /* DLM_DEBUG */

void
error(dlm_lkid_t *lk, dlm_status_t stat)
{
    printf("\t%s: lock error %s on lkid 0x%lx\n",
           prog, dlm_sperrno(stat), *lk);
    abort();
}

/*
 * blocking routine that will release the lock and in doing so will
 * write the resource value block.
 */
void
blkrtm(callback_arg_t x, callback_arg_t y, dlm_lkid_t *lk, dlm_lkmode_t blkmode\
)
{
    dlm_status_t stat;
    dlm_valb_t vb;
    int i;

    Cprintf("\t%s: blkrtm: x 0x%lx y 0x%lx lkid 0x%lx blkmode %d\n",
            prog, x, y, *lk, blkmode);
    printf("\t%s: blkrtm: dequeue EX lock to write value block < %s>\n",
            prog, (char *)vb.valblk);
    printf("\t%s: hold the lock for a couple of seconds\n",
            prog);
    for (i = 0; i < 3; i++) {
        printf("\t%s: sleeping\n", prog);
        sleep(1);
    }
    /* make sure its clean */
    bzero(vb.valblk, DLM_VALBLKSIZE);
    /* write something different */
    (void) strcat((char *)vb.valblk, "efg"); 20
    if((stat = dlm_unlock(lk, &vb, DLM_VALB)) != DLM_SUCCESS)
        error(lk, stat); 21
    /* let waiters know we're done */
    done = 1;
}

main(int argc, char *argv[])
{
    int resnlen, i;
    dlm_lkid_t lkid;
    dlm_status_t stat;
    dlm_nsp_t nsp;
    dlm_valb_t vb;

    /* first we need to join a namespace */
    if ((stat = dlm_nsjoin(getuid(), &nsp, DLM_USER)) != DLM_SUCCESS) {
        printf("\t%s: can't join namespace\n", argv[0]);
        error(0, stat); 8
    }

    prog = argv[0];

    /* now let DLM know what signal to use for blocking routines */
    dlm_set_signal(SIGIO, &i);
    Cprintf("\n\t%s: dlm_set_signal: i %d\n", prog, i); 9
```


例 9-1: ロック , ロック値ブロック , およびロック変換 (続き)

```
resnlen = strlen(resnam);
Cprintf("\t%s: resnam %s\n", prog, resnam);

printf("\n\t%s: grab a NL mode lock\n", prog);
stat = dlm_lock(nsp, (uchar_t *)resnam, resnlen, 0, &lkid,
                DLM_NLMODE, &vb, (DLM_VALB | DLM_SYNCSTS), 0, 0, 0, 0);
/* NL mode better be granted SYNC status */
if(stat != DLM_SYNC) { 10
    printf("\t%s: dlm_lock failed\n", prog);
    error(&lkid, stat);
}
/* should be nulls since master hasn't written anything yet */
printf("\t%s: value block read\n", prog);
printf("\t%s: expected empty value block got < %s>\n", prog, (char *)vb.\
valblk);
if (strlen((char *)vb.valblk) { 11
    printf("\t%s: value block not empty\n", prog);
    error(&lkid, stat);
}

done = 0;
printf("\t%s: converting to NL->EX to get the value block.\n", prog);
printf("\t%s: should see blocking routine run on master\n", prog);
stat = dlm_cvt(&lkid, DLM_EXMODE, &vb, DLM_VALB, 0, 0, blktrn, 0); 12
if(stat != DLM_SUCCESS) {
    printf("\t%s: dlm_cvt failed\n", prog);
    error(&lkid, stat);
}
/* should have read what master wrote, "abc" */
printf("\t%s: value block read\n", prog);
printf("\t%s: expected <abc> got < %s>\n", prog, (char *)vb.valblk);
if (strcmp((char *)vb.valblk, "abc")) { 17
    printf("\t%s: main: value block mismatch < %s>\n",
        prog, (char *)vb.valblk);
    error(&lkid, stat);
}
/* now wait for blocking from master */
printf("\t%s: sleeping waiting for blocking notification\n", prog);
while (!done)
    sleep(1); 18
printf("\t%s: done\n", prog); 22
exit(0);
}
```

- 1 プログラム `api_ex_master.c` は、`dlm_nsjoin` 関数を呼び出して、ロックを要求するリソースのネームスペースに加わります。このネームスペースは、現在のプロセスの UID です。これは `getuid` システム・コールで取得しています。このネームスペースは、`DLM_USER` パラメータで示されるとおり、実効 UID がリソースの所有者の UID と同じプロセスにアクセスを許すネームスペースです。正常終了の場合、この関数は、`nsp` パラメータで示されるメモリ位置に、ネームスペース・ハンドルを返します。

- ② プログラム `api_ex_master.c` は、`dml_set_signal` 関数を呼び出して、DLM が、このプロセスに完了通知およびブロック通知を送る際に、SIGIO シグナルを使うように指定しています。
- ③ プログラム `api_ex_master.c` は、次に呼び出す `dml_lock` 関数に渡すために、リソース名の長さを取得します。リソースの名前は `dist shared resource` です。
- ④ プログラム `api_ex_master.c` は、ネームスペース `uid` のリソース `dist shared resource` を、排他モード (DLM_EXMODE) でロックするために、`dml_lock` 関数を呼び出します。ネームスペース・ハンドル、リソース名、およびリソース名の長さは、すべて必須パラメータとして渡されます。

DLM_SYNCSTS フラグで指示しておく、ロック要求がすぐに許可された場合、DLM は DLM_SYNC 状態を返します。関数の呼び出しに成功すると、DLM は、`lkid` パラメータで指定されたメモリ位置に、排他モード (EX) ロックのロック ID を返します。

この関数には DLM_VALB フラグ、およびリソースのロック値ブロックの内容を書き込むまたは読み取るメモリ位置を指定します。DLM は、`dml_lock` 関数を呼び出して要求したロックが許可された場合に、リソースのロック値を、このメモリ位置にコピーします。最後に、ブロック通知ルーチン `blk_and_go` を指定します。DLM は、ロックが許可された後、このリソースに対する他のロック要求がブロックされたときに、このルーチンを呼び出します。

- ⑤ プログラム `api_ex_master.c` は、`dml_lock` 関数から返された状態値を調べます。状態値が DLM_SYNC 状態 (`dml_lock` 関数を呼び出す際に DLM_SYNCSTS フラグを指定した場合の、成功の条件値) でない場合は、ロック要求が許可されるのを待たなければなりません。今回は、このロックを要求しているプログラムが他に実行されていないので、このような状況は発生しません。
- ⑥ プログラム `api_ex_master.c` は、DLM が `vb` パラメータで指定されたメモリ位置に書き込んだ、値ブロックの内容が空であることを検証します。
- ⑦ プログラム `api_ex_master.c` は、ユーザがプログラム `api_ex_client.c` を起動するのを待ちます。このプログラムは、`dist shared resource` をロックしている排他モード (DLM_EXMODE)

のロックのために、プログラム `api_ex_client.c` で出された同じリソースへのロック要求がブロックされたというブロック通知を受け取ったときに再開されます。

- ⑧ プログラム `api_ex_client.c` を起動すると、`uid` ネームスペースに加わるために `dln_nsjoin` 関数を呼び出します。このネームスペースは、プログラム `api_ex_master.c` を実行しているプロセスが、前に加わっていたのと同じものです。
- ⑨ プログラム `api_ex_client.c` では、プログラム `api_ex_master.c` と同じように、`dln_set_signal` 関数を呼び出して、DLM が、完了通知およびブロック通知をこのプロセスに送る際に、SIGIO シグナルを使うように指示します。
- ⑩ プログラム `api_ex_client.c` では、`dln_lock` 関数を呼び出して、`api_ex_master.c` を実行しているプロセスが既に排他モードでロックしているリソースに、ヌル・モード (DLM_NLMODE) のロックをかけます。DLM_SYNCSTS フラグで指示しておく、と、ロック要求がすぐに許可された場合、DLM は DLM_SYNCH 状態を返します。ヌル・モード (NL) のロックは、この前に許可されている排他モードのロックと共存できるため、このロック要求はすぐに許可されます。この関数ではまた、DLM_VALB フラグとロック値ブロックへのポインタも指定しています。DLM は `dln_lock` で要求したロックが許可された場合、リソースのロック値をこのメモリ位置にコピーします。
- ⑪ プログラム `api_ex_client.c` は、DLM が、`vb` パラメータで指定されたメモリ位置に書き込んだ値ブロックの内容を調べます。値ブロックは、プログラム `api_ex_master.c` がまだ書き込んでいないため、空でなければなりません。
- ⑫ プログラム `api_ex_client.c` は、`dln_cvt` 関数を呼び出して、このプロセスがリソースにかけているヌル・モードのロックを、排他モードに変換します。ここで、ブロック通知ルーチン `blk_rtn` を指定します。プログラム `api_ex_master.c` を実行しているプロセスが、既にこのリソースに排他ロックをかけているので、そのプロセスは、プログラム `api_ex_client.c` のロック変換要求をブロックします。しかし、プログラム `api_ex_master.c` がかけている排他モードのロックで、ブロック通知ルーチンを指定しているため、DLM は、SIGIO シグナルを使って、プログラム `api_ex_master.c` を実行しているプロセスにブロック通知を送り、ブロック通知ルーチン (`blk_and_go`) をトリガします。

- [13] blk_and_go ルーチンは 3 秒間スリープし、その後、done フラグを立てます。これにより、プログラム api_ex_master.c が再開されます。
- [14] プログラム api_ex_master.c は、そのリソースの値ブロックのローカル・コピーに、文字列 abc を書き込みます。
- [15] プログラム api_ex_master.c は、ロック値ブロックに書き込むために dlm_cvt 関数を呼び出します。これを行うために、リソースにかけている排他モードのロックを、排他モード (DLM_EXMODE) に変換します。このとき、関数にはパラメータとして、ロック ID、値ブロックのコピーのあるメモリ位置、および DLM_VALB フラグを指定します。DLM_SYNCSTS フラグで指示しておく、ロック要求がすぐに許可されたとき、DLM は DLM_SYNCH 状態を返します。このプロセスは既に排他モードでリソースをロックしているので、このロック変換要求はすぐに許可されるはずです。
- dlm_cvt 関数には、ブロック通知ルーチンとして blkrttn ルーチンも指定します。リソースにかけている、この排他モードのロックで、プログラム api_ex_client.c から出されるロック変換要求がブロックされるので、DLM はこのブロック通知ルーチンをすぐに呼び出します。
- [16] プログラム api_ex_master.c の blkrttn ルーチンが起動され、すぐに dlm_cvt 関数を呼び出して、リソースにかけているロックを、排他モードからヌル・モードに下位変換します。この呼び出しは、すぐに正常終了します。
- [17] この変換が実行されるとすぐに、プログラム api_ex_client.c のロック変換要求が正常終了します。これは、プログラム api_ex_master.c を実行しているプロセスがかけたヌル・モードのロックが、プログラム api_ex_client.c を実行しているプロセスが今取得した排他モードのロックと共存できるためです。ヌル・モードのロックを排他モードに上位変換する際、DLM は、プログラム api_ex_client.c を実行しているプロセスに、リソースのロック値ブロックをコピーします。このとき、プログラム api_ex_client.c は、プログラム api_ex_master.c が前にリソースのロック値ブロックに書き込んだ文字列 abc を受け取ります。
- [18] プログラム api_ex_client.c は、スリープ状態に入ってブロック通知を待ちます。

- [19] プログラム `api_ex_master.c` は、リソース `dist shared resource` のロックを下位変換してから、スリープ状態にありましたが、ここで `dln_cvt` 関数を呼び出して、リソースにかけているヌル・モードのロックを保護読み取りモード (`DLM_PMODE`) に変換します。プログラム `api_ex_client.c` を実行しているプロセスが、既に排他モードでこのリソースをロックしているので、プログラム `api_ex_master.c` のロック変換要求はこのロックによってブロックされることになります。つまり、排他モードのロックと保護読み取りモードのロックは共存できません。ただし、プログラム `api_ex_client.c` がかけている排他モードのロックでブロック通知ルーチンが指定されているため、DLM は、SIGIO シグナルを送信して、プロセスにブロック通知を配信し、そのプロセスのブロック通知ルーチン (`blkrtm`) をトリガします。
- [20] プログラム `api_ex_client.c` のブロック通知ルーチン `blkrtm` は、数秒間スリープ状態になり、その後、そのリソースの値ブロックのローカル・コピーに文字列 `efg` を書き込みます。
- [21] プログラム `api_ex_client.c` のブロック通知ルーチン `blkrtm` は、`dln_unlock` 関数を呼び出して、このリソースのロックを解放します。リソースのロック値ブロックのローカル・コピーのアドレス、および `DLM_VALB` フラグを指定して、DLM に対し、許可されるモードが保護書き込み (`DLM_PMODE`) または排他 (`DLM_EXMODE`) の場合は、値ブロックのローカル・コピーを、リソースの値ブロックに書き込むよう要求します。このプログラムでは、許可されるモードは `DLM_EXMODE` なので、値ブロックのローカル・コピーがリソースのロック値ブロックに書き込まれます。
- [22] プログラム `api_ex_client.c` が完了し終了します。
- [23] プログラム `api_ex_client.c` を実行しているプロセスがリソースのロックを解放すると、すぐにプログラム `api_ex_master.c` のロック変換要求が正常終了します。ヌル・モードのロックを保護読み取りモードに上位変換する際に、DLM は、リソースのロック値ブロックを、プログラム `api_ex_master.c` を実行しているプロセスにコピーします。このとき、プログラム `api_ex_master.c` は、前にプログラム `api_ex_client.c` がリソースのロック値ブロックに書き込んだ文字列 `efg` を受け取ります。
- [24] プログラム `api_ex_master.c` が完了し終了します。



Memory Channel API ライブラリ

Memory Channel アプリケーション・プログラミング・インタフェース (API) を使うと、自動エラー処理、ロック、および UNIX スタイルの保護を使用し、Memory Channel API クラスタ・メンバ間で効率のよいメモリ共有が実現できます。この章では、Memory Channel API ライブラリをベースにしたアプリケーションの開発に役立つ情報を説明します。また、Memory Channel のアドレス空間と、一般的な共有メモリとの違いを説明し、トランスポート層の機構として、Memory Channel を使うプログラミングと、共有メモリを使うプログラミングでは、どのように異なるかを説明します。

注意

Memory Channel API ライブラリは、Memory Channel ハードウェアで構成されたクラスタでだけサポートされます。Memory Channel ハードウェアが実装されていないクラスタでは、アプリケーションはクラスタ単位でのキル・メカニズム (kill(1) と kill(2) を参照) を使用することがあります。このメカニズムは Memory Channel API の 1 つの機能の代わりとして使うことができます。

この章では、プログラムで Memory Channel API ライブラリの関数を使用する方法についても、例を使って説明します。このプログラムのソース・ファイルは、`/usr/examples/cluster/` ディレクトリにあります。各ファイルには、コンパイルの指示が記載されています。

この章では、次のトピックについて説明します。

- Memory Channel のマルチレール・モデルの理解 (10.1 節)
- Memory Channel API ライブラリの初期化 (10.2 節)
- ユーザ・プログラムでの Memory Channel API ライブラリの初期化 (10.3 節)

- Memory Channel の構成のチューニング (10.4 節)
- トラブルシューティング (10.5 節)
- Memory Channel のアドレス空間へのアクセス (10.6 節)
- クラスタ単位のロックの使用 (10.7 節)
- クラスタ・シグナルの使用 (10.8 節)
- クラスタ情報へのアクセス (10.9 節)
- 共用メモリ・モデルとメッセージ渡しモデルの比較 (10.10 節)
- Memory Channel API を使って TruCluster Server システム用のプログラムを開発しているプログラマからの質問と、それに対する回答 (10.11 節)

10.1 Memory Channel マルチレール・モデル

Memory Channel マルチレール・モデルには、物理レールと論理レールの概念があります。物理レールとは、ケーブルおよび Memory Channel アダプタが接続された Memory Channel ハブ (物理または仮想) と、各ノードのアダプタ用の Memory Channel ドライバのことをいいます。論理レールは、1 つまたは 2 つの物理レールで構成されます。

1 つのクラスタは、1 つ以上、最大 4 つまでの論理レールを持つことができます。論理レールは、次のスタイルで構成することができます。

- シングルレール (10.1.1 項)
- フェイルオーバー・ペア (10.1.2 項)

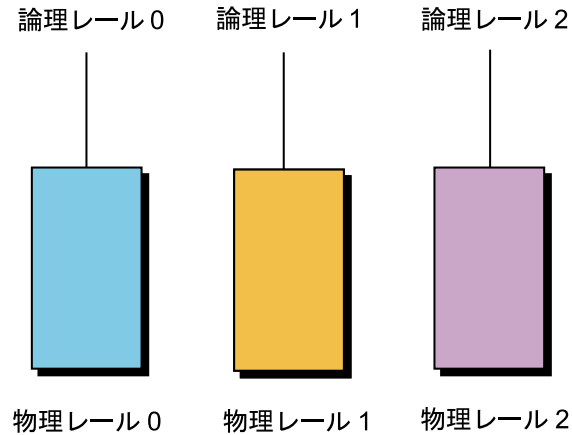
10.1.1 シングルレール・スタイル

クラスタがシングルレール・スタイルで構成されている場合、物理レールと論理レールは 1 対 1 に対応します。この構成では、フェイルオーバーには対応できません。つまり、物理レールに障害が起きた場合、論理レールにも障害が発生することになります。

シングルレール構成を使う利点は、アプリケーションがすべての論理レールのアドレス空間の集合体にアクセスでき、集合体の帯域幅を利用して最大限の性能を実現できることです。

図 10-1 は、3 つの論理レールからなるシングルレールの Memory Channel の構成を示します。各論理レールには、それぞれ物理レールが対応しています。

図 10-1: シングルレール Memory Channel の構成



ZK-1653U-AI

10.1.2 フェイルオーバー・ペア・スタイル

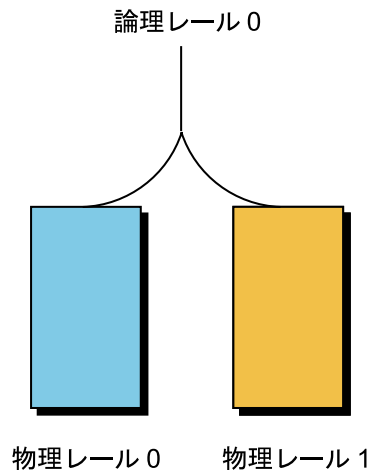
クラスタがフェイルオーバー・ペア・スタイルで構成されている場合、1つの論理レールは2つの物理レールからなっており、一方がアクティブ状態で、もう一方は非アクティブ状態になっています。アクティブ状態の物理レールが故障した場合、フェイルオーバーが起こり、非アクティブ状態だった物理レールが使われます。これによって、フェイルオーバー後も、論理レールはアクティブ状態を保てます。このフェイルオーバーは、ユーザからは見えません。

フェイルオーバー・ペア・スタイルは、Memory Channel が2つの物理レールで構成されている場合にのみ、使うことができます。

フェイルオーバー・ペア構成では、2番目の物理レールを予備として使えるため、物理レールに障害が発生した場合の可用性が高くなります。ただし、一度に利用できるアドレス空間と帯域幅は、どちらか一方の物理レールのものだけです。

図 10-2 は、フェイルオーバー・ペア・スタイルのマルチレール Memory Channel の構成を示しています。この図の構成では、2つの物理レールから1つの論理レールができています。

図 10-2: フェイルオーバー・ペア Memory Channel の構成



ZK-1654U-AI

10.1.3 Memory Channel のマルチレール・モデルの構成

Memory Channel のマルチレール・モデルを実現する場合，クラスタ内のすべてのノードは，同じ数の物理レールで同じ数の論理レールを構成し，それぞれが同じフェイルオーバー・スタイルでなければなりません。

ファイル `/etc/sysconfigtab` 内のシステム構成パラメータ `rm_rail_style` を使って，マルチレール・スタイルを設定します。`rm_rail_style` パラメータには，次の値のどちらかが設定されます。

- シングルレール・スタイルの場合は 0。
- フェイルオーバー・ペア・スタイルの場合は 1。

`rm_rail_style` パラメータの省略時の値は 1 です。

`rm_rail_style` パラメータは，クラスタ内のすべてのノードで同じ値でなければなりません。値が異なっていると，構成エラーが発生することがあります。

`rm_rail_style` パラメータの値を 0 に変更してシングルレール・スタイルにするには，`/etc/sysconfigtab` ファイルを変更して，`rm` サブシステムのスタンザを，次のように追加または変更してください。

```
rm:  rm_rail_style=0
```

注意

/etc/sysconfigtab ファイル内のスタンザを修正または追加する場合は、sysconfigdb(8) を使用することをお奨めします。

rm_rail_style パラメータを変更した場合は、クラスタ全体を停止し、各メンバ・システムをリブートしなければなりません。

Memory Channel のマルチレール・モデルのエラー処理は、指定した論理レールに対して実行されます。Memory Channel API ライブラリのエラー処理関数の説明とコーディング例は、10.6.6 項を参照してください。

注意

Memory Channel のマルチレール・モデルを使うと、ハブや Memory Channel アダプタの追加といったクラスタの再構成が難しくなります。このような再構成を行う場合は、クラスタを完全にシャットダウンしなければなりません。

10.2 Memory Channel API ライブラリの初期化

Memory Channel API ライブラリをベースにしたアプリケーションを実行するには、Memory Channel API クラスタの各ホストでライブラリを初期化しなければなりません。imc_init コマンドで Memory Channel API ライブラリを初期化し、アプリケーションが API を使用できるようにします。Memory Channel API ライブラリの初期化は、システムのブート時に imc_init コマンドを自動的に実行して行う場合と、システムのブート後に、システム管理者がコマンド行からこのコマンドを実行して行う場合があります。

システムのブート時に Memory Channel API ライブラリを初期化するかどうかは、/etc/rc.config ファイル内の IMC_AUTO_INIT 変数で制御します。この変数が 1 に設定されている場合は、システムのブート時に imc_init コマンドが呼び出されます。Memory Channel API ライブラリがブート時に初期化される場合、-a maxalloc フラグおよび -r maxrecv フラグの値には、/etc/rc.config ファイル内の変数 IMC_MAX_ALLOC および IMC_MAX_RECV に指定された値が設定されます。maxalloc パラメータおよび maxrecv パラメータの省略時の値は 10 MB です。

IMC_AUTO_INIT 変数が 0 に設定されている場合、Memory Channel API ライブラリは、システムのブート時に初期化されません。システム管理者は、`imc_init` コマンドを実行してライブラリを初期化しなければなりません。`imc_init` コマンドを手動で実行した場合は、`/etc/rc.config` ファイルのパラメータ値は使用されません。

`imc_init` コマンドは、システムのブート時と、システムのブート後のどちらで実行されたかにかかわらず、最初に実行されたときに Memory Channel API ライブラリを初期化します。`-a maxalloc` フラグの値は、Memory Channel API クラスタ内では、すべてのホストで同じでなければなりません。異なる値が指定されている場合は、指定されている値のうち、クラスタ内で最も大きな値がすべてのホストに適用されます。

Memory Channel API ライブラリが現在のホストで初期化された後、システム管理者は `imc_init` コマンドを再び実行して、リソースの限界値 `maxalloc` および `maxrecv` を再構成することができます。このとき、リブートする必要はありません。システム管理者はどちらの限界値も増減させることができますが、新しい限界値を現在のリソース使用量よりも小さくすることはできません。コマンド行からクラスタを再構成しても、`/etc/rc.config` ファイルに指定されている値を読み取ることや、この値を変更することはできません。システム管理者は `rcmgr(8)` コマンドを使って、このパラメータを変更することができます。変更した値は、システムをリブートしたときに有効になります。

`imc_init` コマンドを実行するには、ルート特権が必要です。

10.3 ユーザ・プログラムでの Memory Channel API ライブラリの初期化

`imc_api_init` 関数は、ユーザ・プログラム内で Memory Channel API ライブラリを初期化するために使用します。プロセスで、`imc_api_init` 関数を呼び出した後にこれ以外の Memory Channel API の関数を呼び出します。プロセスが子プロセスを生成 (fork) する場合、子プロセスでは、`imc_api_init` 関数を呼び出した後にこれ以外の Memory Channel API の関数を呼び出さなければなりません。この順番で呼び出さなければ、予期しない動作をします。

10.4 Memory Channel 構成のチューニング

`imc_init` コマンドは、省略時の設定で指定されるリソースで Memory Channel API ライブラリを初期化します。アプリケーションによっては、省略時の設定より多くのリソースが必要になるかもしれません。場合によっては、Memory Channel のパラメータと仮想メモリのリソース・パラメータの値を変更して、これらの制限を超えることができます。以降の各項では、これらのパラメータについて説明し、これらの変更方法も示します。

10.4.1 Memory Channel のアドレス空間を拡張する

Memory Channel API ライブラリで利用できる Memory Channel のアドレス空間の総容量は、`imc_init` コマンドの `maxalloc` パラメータで指定します。ホストが受信用にアタッチできる Memory Channel のアドレス空間の最大容量は、`imc_init` コマンドの `maxrecv` パラメータで指定します。省略時の制限は、それぞれ 10 MB です。10.2 節で、`imc_init` コマンドを使って Memory Channel API ライブラリを初期化する方法を説明しています。

自動的に初期化するときに使用される値を変更したい場合は、`rcmgr(8)` コマンドを使って、変数 `IMC_MAX_ALLOC` および `IMC_MAX_RECV` の値を設定することにより行うことができます。たとえば、Memory Channel API ライブラリのクラスタ単位で利用できる Memory Channel のアドレス空間として、全部で 80 MB を割り当て、このうち 60 MB を、現在のホストが受信用にアタッチする領域として割り当てるには、次のようにこの変数を設定します。

```
rcmgr set IMC_MAX_ALLOC 80
rcmgr set IMC_MAX_RECV 60
```

`rcmgr(8)` コマンドを使って新しい制限を設定した場合、変更した値は、システムをリブートしたときに有効になります。

Memory Channel API ライブラリが初期化された後に、利用可能な Memory Channel のアドレス空間の総容量、および受信用にアタッチできる Memory Channel のアドレス空間の最大容量はどちらも、Memory Channel API ライブラリの初期化コマンド `imc_init` を使って変更することができます。たとえば、クラスタ単位で利用可能な Memory Channel のアドレス空間として全部で 80 MB を割り当て、現在のホストで受信用にアタッチする Memory Channel のアドレス空間に 60 MB を割り当てる場合は、次のコマンドを実行します。

```
imc_init -a 80 -r 60
```

ただし、`imc_init` コマンドを使って新しい限界値を設定した場合、設定した値はシステムのリブート時に消滅し、変数 `IMC_MAX_ALLOC` および `IMC_MAX_RECV` の値が限界値として再び使われるようになります。

10.4.2 固定メモリを増やす

受信用にアタッチする Memory Channel のアドレス空間の各ページは、システムの物理メモリのページに対応していなければなりません。このメモリは、ページング不可のメモリ、つまり固定メモリです。ホスト上の固定メモリは、無限に増やすことはできません。システム構成パラメータ `vm_syswiredpercent` で制限されます。`/etc/sysconfigtab` ファイル内の `vm_syswiredpercent` パラメータを変更することができます。

たとえば、`vm_syswiredpercent` パラメータに 80 を設定する場合、`/etc/sysconfigtab` ファイルの `vm` スタンザに、次のようなエントリを記述する必要があります。

```
vm:  vm_syswiredpercent=80
```

`vm_syswiredpercent` パラメータを変更した場合は、システムをリブートしなければなりません。

注意

大半の運用形態では、省略時の固定メモリの容量で十分に対応できます。この限界値を変更する場合は十分に注意するようにしてください。

10.5 トラブルシューティング

以降の各項では、Memory Channel API ライブラリの関数を使った場合に発生するエラー状況について説明し、解決策を示します。

10.5.1 IMC_NOTINIT リターン・コード

`IMC_NOTINIT` 状態は `imc_init` コマンドがまだ実行されていない場合、または `imc_init` コマンドが正常に終了していない場合に返されます。

Memory Channel API ライブラリの関数を使用する前に、`imc_init` コマンドを Memory Channel API クラスタ内の各ホスト上で実行しなければなら

せん。10.2 節で、`imc_init` コマンドを使って Memory Channel API ライブラリを初期化する方法を説明しています。

`imc_init` コマンドが正常に終了しなかった場合、考えられる解決方法については、10.5.2 項を参照してください。

10.5.2 Memory Channel API ライブラリの初期化の失敗

ホスト上で Memory Channel API ライブラリの初期化に失敗することがあります。このような場合は、エラー・メッセージがコンソールに表示されるとともに、`/usr/adm` ディレクトリの `messages` ログ・ファイルに書き込まれます。次に示す一連のエラー・メッセージとその解決方法を参考にし、エラーを取り除いてください。

- Memory Channel is not initialized for user access

このエラー・メッセージは、現在のホストが Memory Channel API を使うように初期化されていないことを示します。

この問題を解決するには、Memory Channel のすべてのケーブルが、このホストの Memory Channel のアダプタに正しく接続されていることを確認してください。

- Memory Channel API - insufficient wired memory

このエラー・メッセージは、`/etc/rc.config` ファイルの `IMC_MAX_RECV` 変数の値または `imc_init` コマンドで指定した `-r` オプションの値が、構成パラメータ `vm_syswiredpercent` で指定されている固定メモリの限界値よりも大きいことを示しています。

この問題を解決するには、`maxrecv` パラメータにもっと小さい値を指定して、`imc_init` コマンドを呼び出すか、10.4.2 項で説明しているようにして、システムの固定メモリの限界値を大きくします。

10.5.3 Memory Channel の致命的なエラー

Memory Channel API を初期化する際に、Memory Channel の物理的な構成やインターコネクトが原因で障害が発生することがあります。こうした状況で、コンソールに表示されるエラー・メッセージでは、Memory Channel API のことには触れていません。次の項で、このような障害が発生する典型的な原因を挙げます。

10.5.3.1 レールの初期化の失敗

ノードのレールの構成が、他のクラスタ・メンバのものと同じでない場合、そのクラスタ内のいくつかのホストで、システム・パニックが発生し、コンソールに次の形式のエラー・メッセージが表示されます。

```
rm_slave_init
rail configuration does not match cluster expectations for rail 0
rail 0 has failed initialization
rm_delete_context: lcsr = 0x2a80078, mcerr = 0x20001, mcport =
0x72400001
panic (cpu 0): rm_delete_context: fatal MC error
```

このエラーは、構成パラメータ `rm_rail_style` がすべてのノードで同じでない場合に発生します。

この問題を解決するには、次の手順を実行してください。

1. システムを停止します。
2. `/genvmunix` をブートします。
3. 10.1.3 項の説明に従い、`/etc/sysconfigtab` ファイルを修正します。
4. Memory Channel API クラスタ・サポート (`/vmunix`) を使ってカーネルをリブートします。

10.5.4 IMC_MCFULL リターン・コード

ある操作を実行するのに Memory Channel のアドレス空間が不足していると、`IMC_MCFULL` 状態が返されます。

Memory Channel API ライブラリで利用できる Memory Channel のアドレス空間の総容量は、10.5.2 項で説明しているように、`imc_init` コマンドの `maxalloc` パラメータで指定します。

`rcmgr(8)` コマンド、または Memory Channel API ライブラリの初期化コマンド `imc_init` を使って、ライブラリがクラスタ単位で利用できる Memory Channel のアドレス空間を増やすこともできます。詳細は、10.4.1 項を参照してください。

クラスタ内の Memory Channel のアドレス空間が十分でない場合、ブートしているノードはクラスタへの結合に問題が生じていることがあります。その場合には、1 つ以上のメンバが妥当性チェックの失敗でパニックになっている (ICS MCT Assertion Failed) か、またはブートしているメンバがブート初期にハングしている場合があります。

Memory Channel リソースは、新しいメンバがクラスタのメンバになるときに動的に割り当てられます。Memory Channel アプリケーション・プログラミング・インタフェース (API) のライブラリ関数を呼び出すアプリケーションが実行されていると、必要な Memory Channel リソースを消費し、メンバはクラスタのメンバになるのに必要なリソースを取得できなくなる場合があります。この問題を解決するには、Memory Channel API ライブラリ関数を呼び出すアプリケーションを開始する前にすべてのクラスタ・メンバをブートしてください。

10.5.5 IMC_RXFULL リターン・コード

ある領域を受信用にアタッチしようとしたときに、受信用のマッピング空間を使いきっていた場合、`imc_asattach` 関数は `IMC_RXFULL` 状態を返します。

注意

現在のホスト上での受信用空間の省略時の総容量は、10 MB です。

各ホスト上で受信用にアタッチできる Memory Channel のアドレス空間の最大容量は、10.2 節で説明しているように、`imc_init` コマンドの `maxrecv` パラメータを使って指定します。

`rcmgr(8)` コマンド、または Memory Channel API ライブラリの初期化コマンド `imc_init` を使って、ホスト上で受信用にアタッチできる Memory Channel のアドレス空間の最大容量を拡張することができます。詳細は、10.4.1 項を参照してください。

10.5.6 IMC_WIRED_LIMIT リターン・コード

リターン値 `IMC_WIRED_LIMIT` は、固定メモリの最大容量を超える操作が行われたことを示します。

固定メモリの限界値は、システム構成パラメータ `vm_syswiredpercent` で指定します。この限界値の変更方法についての詳細は、10.4.2 項を参照してください。

10.5.7 IMC_MAPENTRIES リターン・コード

リターン値 `IMC_MAPENTRIES` は、現在のプロセスの仮想メモリのマップ・エントリの数が、最大値を超えたことを示します。

10.5.8 IMC_NOMEM リターン・コード

リターン状態 `IMC_NOMEM` は、`malloc` 関数が Memory Channel API の関数を呼び出した際に失敗したことを示します。

これは、プロセスの仮想メモリの限界値を超えた場合に発生し、プロセスの仮想メモリの上限を増やす通常の方法で解決することができます。つまり、C シェルの場合は `limit` コマンドおよび `unlimit` コマンドを使い、Bourne シェル、および Korn シェルの場合は `ulimit` コマンドを使って解決します。

10.5.9 IMC_NORESOURCES リターン・コード

`IMC_NORESOURCES` リターン値は、要求された動作を実行するのに利用できる Memory Channel のデータ構造体が不足していることを示します。ただし、利用可能な Memory Channel のデータ構造体の大きさは固定であり、パラメータを変更して増やすことはできません。この問題を解決するには、使用する領域やロックを減らすようにアプリケーションを修正する必要があります。

10.6 Memory Channel のアドレス空間へのアクセス

Memory Channel インターコネクトは、Memory Channel API クラスタ・メンバ間でのメモリ共有の 1 つの形態です。Memory Channel API ライブラリを使ってメモリ共有をセットアップすると、クラスタ内の各メンバ上のプロセスは、その仮想アドレス空間のアドレスに直接読み取り/書き込み操作を実行するだけで、互いのデータを交換することができます。Memory Channel API ライブラリを使ってメモリ共有をセットアップすると、これらの直接の読み取り/書き込み操作は、オペレーティング・システムや Memory Channel API ライブラリ・ソフトウェアの関数を介さずに、ハードウェアの速度で実行されます。

システムが Memory Channel を使うように構成されている場合、システムの物理アドレス空間の一部が Memory Channel のアドレス空間に割り当てられます。Memory Channel のアドレス空間の大きさは、`imc_init` コマンドで指定します。プロセスは、Memory Channel API を使って、Memory Channel のアドレス空間の領域をプロセスの仮想アドレス空間にマップすることによって、この Memory Channel のアドレス空間にアクセスします。

別のクラスタ・メンバ上の Memory Channel のアドレス空間にアクセスするアプリケーションでは、`imc_asalloc` 関数を呼び出して、アドレス空間の

一部を特定の目的のために割り当てることができます。 *key* パラメータで、クラスタ単位のキーと領域を関連付けます。同じ領域を割り当てる他のプロセスは、これと同じキーを指定します。こうすることで、複数のプロセスがこの領域に連携してアクセスできるようになります。

プロセスは、Memory Channel のアドレス空間の割り当てられた領域を使うために、 *imc_asattach* 関数または *imc_asattach_ptp* 関数を使って、その領域をそのプロセスの仮想アドレス空間にマップします。プロセスが Memory Channel の領域にアタッチすると、アタッチした領域と同じ大きさの領域が、プロセスの仮想アドレス空間に追加されます。領域をアタッチする際に、プロセスはその領域をデータ送信用にマップするか、データ受信用にマップするかを次のように指定します。

- 送信 — その領域が Memory Channel を介したデータ送信に使われることを示します。プロセスがこの仮想アドレス領域のアドレスに書き込みを行うと、データは Memory Channel インターコネクトを通して Memory Channel API クラスタの他のメンバに送信されます。

送信用に領域をマップするには、 *imc_asattach* 関数の *dir* パラメータに、値 *IMC_TRANSMIT* を指定します。

- 受信 — その領域が Memory Channel からのデータの受信に使われることを示します。この場合、プロセスの仮想アドレス空間にマップされるアドレス空間は、実際にはそのシステム上の物理メモリの領域に対応します。データが Memory Channel 上で送信されると、そのデータは、受信用としてその領域をマップしているすべてのホストの物理メモリに書き込まれます。これにより、そのシステム上の複数のプロセスが、物理メモリの同じ領域からこのデータを読み取れるようになります。プロセスは、この領域をマップする前に送信されたデータを受信することはできません。

受信用に領域をマップするには、 *imc_asattach* 関数の *dir* パラメータに、値 *IMC_RECEIVE* を指定します。

Memory Channel インターコネクトを使ってメモリを共用する方式は、接続を確立すれば、仮想アドレス空間にアクセスするだけで、2つの異なるプロセスがデータを共用できるという点では、通常の共用メモリと似ています。ただし、これらのメモリ共用方式には、次のような考慮しなければならぬ違いが2つあります。

- 通常の共用メモリでは、作成したときに仮想アドレスが割り当てられます。C プログラムの用語では、メモリへのポインタがあるということです。この 1 つのポインタは、その共用メモリの読み取りにも書き込みにも使うことができます。一方、Memory Channel の領域には、2 つの異なる仮想アドレス、つまり送信用仮想アドレスと受信用仮想アドレスを割り当てることができます。C プログラムの用語で言えば、2 つの異なるポインタを管理するということです。1 つのポインタは、書き込み操作でのみ使用し、もう一方は読み取り操作で使います。
- 通常の共用メモリでは、書き込み操作は直接メモリに対して実行されるため、その結果は同じメモリを読み取っている他のプロセスにすぐに反映されます。しかし、Memory Channel の領域に書き込み操作を行う場合は、書き込みは直接メモリに対して行われるのではなく、入出力システムと Memory Channel のハードウェアに対して行われます。したがって、受信側のシステム上のメモリにデータが届くまでに遅れが生じます。これについては、10.6.5 項で詳しく説明します。

10.6.1 Memory Channel のアドレス空間へのアタッチ

以降の各項では、プロセスに Memory Channel のアドレス空間をアタッチさせる方法を説明します。プロセスに Memory Channel のアドレス空間をアタッチさせる方法は、次のとおりです。

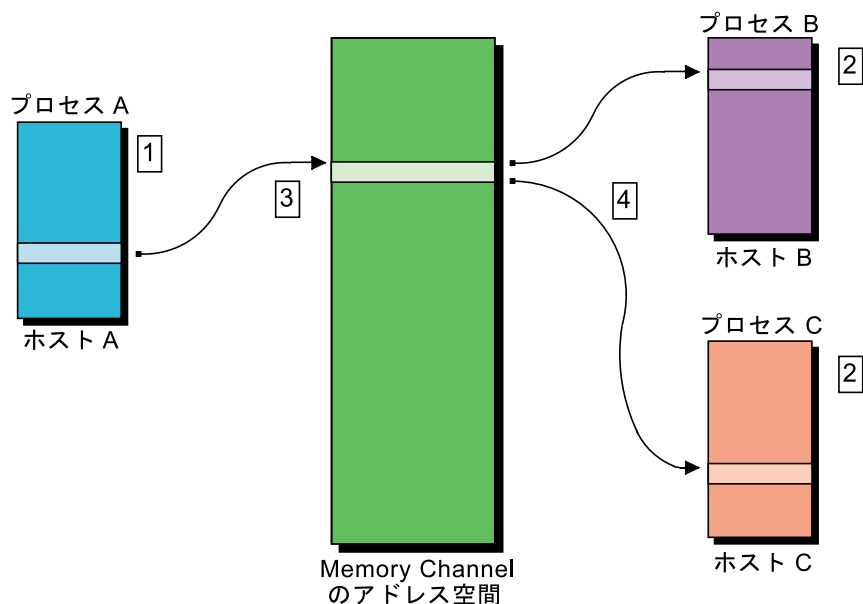
- ブロードキャスト・アタッチ (10.6.1.1 項)
- ポイント・ツー・ポイント・アタッチ (10.6.1.2 項)
- ループバック・アタッチ (10.6.1.3 項)

この節では、初期状態の一貫性、Memory Channel 領域の読み取り/書き込み、遅延と一貫性、およびエラー管理についても説明し、コーディング例をいくつか示します。

10.6.1.1 ブロードキャスト・アタッチ

あるプロセスが送信用にマップした領域を、別のプロセスが受信用にマップした場合、送信側プロセスがその領域に書き込んだデータは Memory Channel を通して、もう一方のプロセスの受信メモリに伝送されます。図 10-3 は、ホストが 3 つある Memory Channel システムで、アドレス空間がどのように対応するかを示します。

図 10-3: ブロードキャスト・アドレス空間のマッピング



ZK-1650U-AI

上記の図 10-3 のようにアドレス空間をマッピングすると、次のようになります。

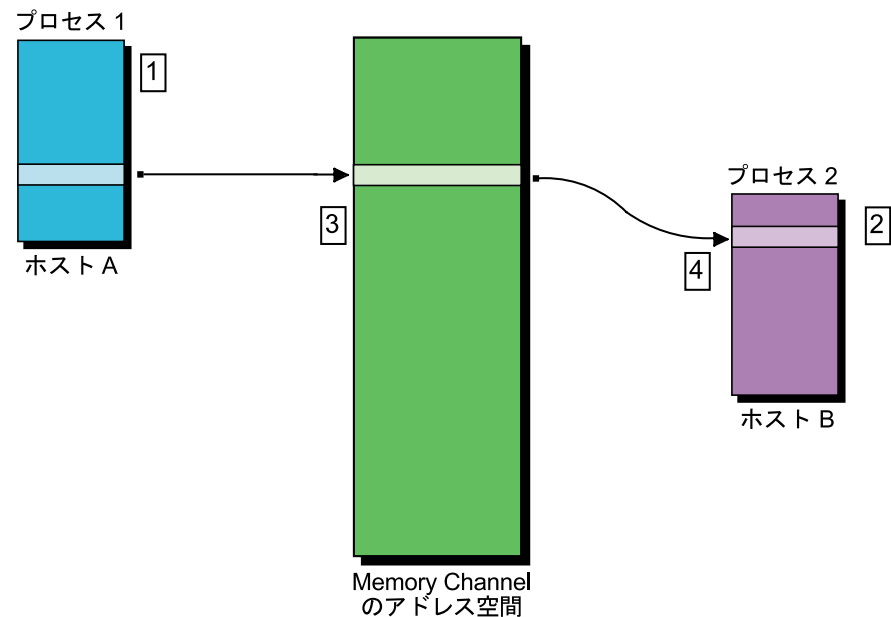
1. プロセス A は、Memory Channel のアドレス空間のある領域を割り当てます。次に、このプロセス A は、`imc_asattach` 関数を使ってこの領域を送信用にアタッチして、この領域を自分の仮想アドレス空間にマップします。
2. プロセス B と C は、どちらもプロセス A と同じ Memory Channel のアドレス空間の領域を割り当てます。ただし、プロセス A とは異なり、プロセス B と C は、データを受信するためにこの領域にアタッチします。
3. データがプロセス A の仮想アドレス空間に書き込まれると、そのデータは Memory Channel 上で伝送されます。
4. プロセス A からのデータが Memory Channel に送られると、このデータは、プロセス B および C の、データ受信用に割り当てられた仮想アドレス空間に対応する物理メモリに書き込まれます。

10.6.1.2 ポイント・ツー・ポイント・アタッチ

割り当てた Memory Channel のアドレス空間の領域は、別のノード上の特定のプロセスの仮想アドレス空間に、送信用としてポイント・ツー・ポイント・モードでアタッチすることができます。これは、パラメータでホストを指定し、`imc_asattach_ptp` 関数を呼び出すことで実行できます。これにより、この領域に書き込んだデータは、クラスタ内のすべてのホストに送られるのではなく、パラメータで指定したホストにのみ送られるようになります。

`imc_asattach_ptp` 関数を使ってアタッチする領域は、常に送信モードでアタッチし、書き込みのみが可能です。図 10-4 は、ホストが 2 つある Memory Channel システムで、ポイント・ツー・ポイントのアドレス空間がどのように対応するかを示します。

図 10-4: ポイント・ツー・ポイント・アドレス空間のマッピング



ZK-1652U-AI

上記の図 10-4 のようにアドレス空間をマッピングすると、次のようになります。

1. プロセス 1 は、Memory Channel のアドレス空間の領域を割り当てます。次に、`imc_asattach_ptp` 関数を使って、この領域をホスト B と

のポイント・ツー・ポイント用としてアタッチすることで、この領域を自分の仮想アドレス空間にマップします。

2. プロセス 2 は、この領域を割り当て、`imc_asattach` 関数を使って、受信用としてこの領域にアタッチします。
3. データがプロセス 1 の仮想アドレス空間に書き込まれると、このデータは Memory Channel 上で伝送されます。
4. プロセス 1 からのデータが Memory Channel に送られると、このデータは、ホスト B 上のプロセス 2 の仮想アドレス空間に対応する物理メモリに書き込まれます。

10.6.1.3 ループバック・アタッチ

Memory Channel の領域は、1 つのホスト上のプロセスから送信と受信の両方でアタッチすることができます。あるホストが書き込んだデータは、その領域を受信用にアタッチしている他のホストに書き込まれます。しかし、省略時の設定では、あるホストが書き込んだデータは、そのホストの受信用メモリには書き込まれないようになっています。データは、他のホストにのみ書き込まれます。書き込んだデータを、書き込んだホスト自身で参照できるようにするには、送信用にアタッチする際に、`imc_asattach()` 関数に `IMC_LOOPBACK` フラグを指定しなければなりません。

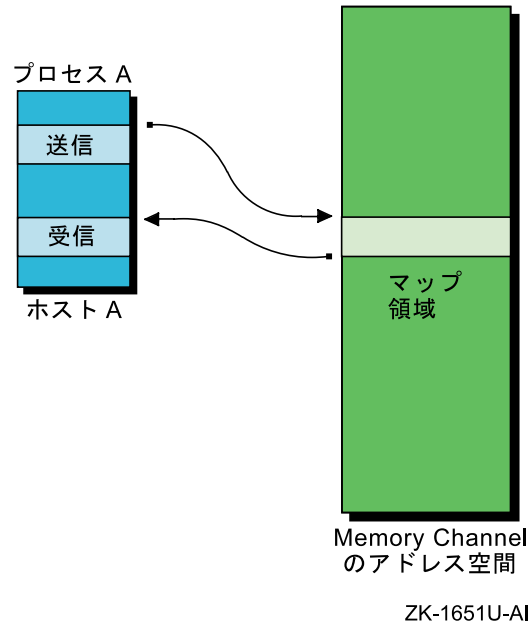
領域のループバック属性はホスト単位でセットアップし、そのホストでの最初の送信用アタッチで渡す `flag` パラメータの値で判断されます。

`flag` パラメータに値 `IMC_LOOPBACK` を指定すると、書き込みのたびに 2 つの Memory Channel トランザクションが発生します。1 つはデータの書き込み用で、もう 1 つはデータのループバック用です。

ポイント・ツー・ポイント・アタッチ方式では、性質上、ループバックによる書き込みは許されません。

図 10-5 では、Memory Channel のアドレス空間の領域を、ループバック付きの送信と受信の両方でアタッチする場合の構成を示します。

図 10-5: ループバック・アドレス空間のマッピング



10.6.2 初期状態の一貫性

Memory Channel の領域に受信用にアタッチした場合、初期状態は不定です。このような状況は、Memory Channel のある領域を送信用にマップしているプロセスが、他のプロセスがその領域を受信用にマップする前に、その領域の内容をアップデートした場合に発生します。これは、初期状態の一貫性の問題と呼ばれます。この問題に対処する方法は 2 つあります。

- アプリケーションを作成する際に、受信側のすべてのプロセスが領域にアタッチした後で、送信側のプロセスが領域に書き込むようにする。
- `imc_asalloc` 関数を使って領域を割り当てる際に、`IMC_COHERENT` フラグを指定して、その領域の一貫性を保つことを明示する (一貫性のある領域)。これによって、その領域へのアップデート内容は、プロセスがその領域にアタッチするタイミングに関係なく、すべてのプロセスに反映されるようになる。

一貫性のある領域では、ループバック機能を使っています。したがって、書き込みを 1 回行うたびに、データ書き込みとデータのループバックのための 2 つの Memory Channel トランザクションが発生します。

このため、一貫性のある領域では、利用可能な帯域幅が、一貫性のない領域よりも小さくなります。

10.6.3 Memory Channel 領域の読み取りと書き込み

Memory Channel のアドレス空間の領域をアタッチするプロセスは、送信ポインタにのみ書き込むことができ、受信ポインタからのみ読み取ることができます。送信ポインタから読み取ろうとすると、セグメント違反になります。

Memory Channel の送信ポインタからの明らかな読み取り操作以外に、コンパイラが読み取り、変更、書き込みのサイクルの命令を生成するような操作でも、セグメント違反が発生します。たとえば、次のような操作です。

- 後置インクリメントおよび後置デクリメント演算。
- 前置インクリメントおよび前置デクリメント演算。
- 4 バイトの整数倍バイトでない単純データ型への代入。
- `bcopy(3)` ライブラリ関数を使用する際に、`length` パラメータが 8 バイトの整数倍でない場合、または、コピー元またはコピー先として渡す引数が 8 バイトの境界に整列されていない場合。
- クオードワード (quadword) に整列されていない構造体、つまり `sizeof` 関数でのリターン値が 8 の整数倍でない構造体への代入。これは、構造体全体を 1 単位として代入する場合にのみあてはまる。たとえば、`mystruct1 = mystruct2` のような場合。

10.6.4 アドレス空間の例

例 10-1 では、Memory Channel のアドレス空間の領域の初期化、割り当て、およびこれへのアタッチの方法を具体的に示します。また、次の 2 つの点について、Memory Channel のアドレス空間と通常の共用メモリの違いを示します。

- 初期状態の一貫性 (10.6.2 項を参照)
- 受信領域と送信領域の非対称性 (10.6.3 項を参照)

例 10-1 に示すサンプル・プログラムは、コマンド行のパラメータで、マスタ・モードまたはスレーブ・モードを指定して実行します。マスタ・モードでは、Memory Channel のグローバルなアドレス空間のデータ構造体に、プログラムのプロセス識別子 (PID) を書き込みます。スレーブ・モードでは、

Memory Channel のアドレス空間のデータ構造体をポーリングして、マスタ・プロセスの PID を確認します。

注意

個々のアプリケーションでは、キーに対する命名スキームを定義する必要があります。プログラムでは、キーの衝突による問題に対応できるように、キーを柔軟に使用する必要があります。アプリケーション固有の、意味のあるキーを使用してください。

アドレス空間の例は、`/usr/examples/cluster/mc_ex1.c` にあります。コンパイル後に、1 つのクラスタ・メンバ上でスレーブ (`mc_ex1 0`) を起動し、次に別のクラスタ・メンバ上でマスタ (`mc_ex1 1`) を起動します。

例 10-1: Memory Channel のアドレス空間の領域へのアクセス

```
/*
 * To compile: cc -g -o mc_ex1 mc_ex1.c -limc
 */

#include <c_asm.h>
#include <sys/types.h>
#include <sys/imc.h>

#define mb() asm("mb")
#define VALID 756

main (int argc, char *argv[])
{
    imc_asid_t    glob_id;
    typedef struct {
        pid_t      pid;
        volatile int valid; [1]
    } clust_pid;

    clust_pid     *global_record;
    caddr_t       add_rx_ptr = 0, add_tx_ptr = 0;
    int           status;
    int           master;
    int           logical_rail=0;
    char          *prog;

    prog = argv[0];

    /* check for correct number of arguments */

    if (argc != 2) {
        printf("usage: %s 0|1\n", prog);
        exit(-1);
    }

    /* test if process is master or slave */
```

例 10-1: Memory Channel のアドレス空間の領域へのアクセス (続き)

```
master = atoi(argv[1]); [2]

/* initialize Memory Channel API library */
status = imc_api_init(NULL); [3]

if (status < 0) {
    imc_perror("imc_api_init",status); [4]
    exit(-2);
}

imc_asalloc(123, 8192, IMC_URW, 0, &glob_id,
           logical_rail); [5]

if (master) {
    imc_asattach(glob_id, IMC_TRANSMIT, IMC_SHARED,
                 0, &add_tx_ptr); [6]

    global_record = (clust_pid*)add_tx_ptr; [7]
    global_record->pid = getpid();
    mb(); [8]
    global_record->valid = VALID;
    mb();
}

else { /* secondary process */

    imc_asattach(glob_id, IMC_RECEIVE, IMC_SHARED,
                 0, &add_rx_ptr); [9]

    (char*)global_record = add_rx_ptr;

    while ( global_record->valid != VALID)
        ; /* continue polling */ [10]

    printf("pid of master process is %d\n",
           global_record->pid);
}

imc_asdetach(glob_id);
imc_asdealloc(glob_id); [11]
}
```

- [1] valid フラグを volatile として定義することで、コンパイラの最適化によって、アップデートした PID の値がメモリから読み取れなくなるのを防いでいます。
- [2] コマンド行の最初の引数で、プロセスがマスタ・プロセス (引数は 1) か、スレーブ・プロセス (引数は 0) かを指定します。

- ③ `imc_api_init` 関数は、Memory Channel API ライブラリを初期化します。この関数は、Memory Channel API ライブラリの他の関数を呼び出す前に呼び出さなければなりません。
- ④ すべての Memory Channel API ライブラリの関数は、正常に終了した場合、リターン状態として 0 を返します。`imc_perror` 関数は、エラー状態値を解釈します。この例では簡潔にするため、`imc_api_init` 関数を除く、他のすべての関数からの状態を無視しています。
- ⑤ `imc_asalloc` 関数は、次のような特性を持つ Memory Channel のアドレス空間の領域を割り当てます。
- `key=123` — この値で Memory Channel のアドレス空間の領域を識別します。この領域にアタッチする他のアプリケーションは同じキー値を使います。
 - `size=8192` — この領域のサイズは 8192 バイトです。
 - `perm=IMC_URW` — この領域のアクセス許可は、ユーザによる読み取り/書き込みが可能です。
 - `id=glob_id` — `imc_asalloc` 関数がこの変数に値を返します。この値は、割り当てられた領域を一意に識別します。プログラムでは、続いて Memory Channel の他の関数を呼び出す際に、この値を使います。
 - `logical_rail=0` — この領域は、Memory Channel の論理レール 0 を使って割り当てられます。
- ⑥ マスタ・プロセスでは、`imc_asattach` 関数を呼び出し、`glob_id` 識別子を指定して、送信用にこの領域をアタッチします。`glob_id` 識別子は、`imc_asalloc` 関数を呼び出して返された値です。`imc_asattach` 関数は、`add_tx_ptr` を返します。これは、プロセスの仮想アドレス空間の領域のアドレスへのポインタです。値 `IMC_SHARED` は、この領域が共用できることを示します。つまり、このホスト上の他のプロセスも、この領域にアタッチできます。
- ⑦ このプログラムは、グローバル・レコードのポインタ変数に、プロセスの仮想アドレス空間にある、仮想メモリの領域 (これは、Memory Channel の領域に対応している) へのポインタを代入し、このグローバル・レコードの `pid` フィールドに、そのプロセス ID を書き込みます。マスタ・プロセスは、この領域を送信用にアタッチしていることに注意

してください。したがって、実行できるのは、フィールドへのデータの書き込みだけです。たとえば、次のようにしてこのフィールドを読み取ろうとすると、セグメント違反になります。

```
(pid_t)x = global_record->pid;
```

- 8 VALID フラグを設定する前に、メモリ・バリア命令を使って、pid フィールドを、Alpha CPU の書き込みバッファから強制的に書き出します。
- 9 スレーブ・プロセスが、glob_id 識別子を指定して imc_asattach 関数を呼び出して、受信用にこの領域にアタッチします。この識別子は、imc_asalloc 関数を呼び出した際に返された値です。imc_asattach 関数は add_rx_ptr を返します。これは、プロセスの仮想アドレス空間の領域のアドレスへのポインタです。マッピングした時点では、この領域をマッピングしたすべてのプロセスで、その領域の内容に一貫性がない可能性があります。そこで、マスタ・プロセスより先にスレーブ・プロセスを起動して、マスタ・プロセスによるすべての書き込みが、スレーブ・プロセスの仮想アドレス空間に反映されるようにします。
- 10 スレーブ・プロセスは、この領域にグローバル・レコードの構造体を当てはめ、valid フラグをポールします。このフラグは最初に volatile として定義されているため、コンパイラによる最適化に影響されず、フィールドの内容がレジスタに格納されます。これにより、このループ内の命令を実行するたびに、メモリから新しい値がロードされ、VALID への遷移が正しく検出されます。
- 11 最後に、マスタ・プロセスとスレーブ・プロセスは、imc_asdetach 関数および imc_asdealloc 関数を呼び出して、明示的にこの領域をデタッチし、割り当てを解除します。異常終了した場合には、プロセスが終了する際に、割り当てられた領域が自動的に解放されます。

10.6.5 遅延と一貫性

10.6.2 項で説明したように、初期状態の一貫性の問題は、同じ領域への受信用のすべてのマッピングが完了した後に、データを再送信することで解決できます。または、割り当てるときに、その領域を一貫性のある領域として指定することで解決できます。ただし、プロセスが送信ポインタに書き込んでから、アップデートされたデータが、受信ポインタに対応する物理メモリに反映されるまで、数マイクロ秒かかる場合があります。この期間に、プロセ

スが受信ポインタを読み取ると、読み取られたデータは正しくありません。このことは、遅延にかかわる一貫性の問題として知られています。

遅延の問題は、通常の共用メモリ・システムでは発生しません。メモリとキャッシュの制御で、格納およびロード命令と、データ伝送との同期が確実にとれるようになっているためです。

例 10-2 は /usr/examples/cluster/mc_ex2.c にあります。この例では、グローバルなプロセス・カウンタの値を減らしていき、カウンタの値が 0 になるのを検証するプログラムの 2 つのバージョンを示します。最初のプログラムでは、System V の共用メモリとプロセス間通信を使用しています。2 番目のプログラムでは、Memory Channel API ライブラリを使用しています。

例 10-2: System V IPC および Memory Channel のコードの比較

```
/*
 * To compile : cc -o mc_ex2 -g mc_ex2.c -limc
 */

#if 0

/***** System V IPC example *****/
/*****

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
main()
{
    typedef struct {
        int    proc_count;
        int    remainder[2047];
    } global_page;
    global_page *mypage;
    int    shmid;

    shmid = shmget(123, 8192, IPC_CREAT |  SHM_R |  SHM_W);

    (caddr_t)mypage = shmat(shmid, 0,  0); /* attach the
                                           global region    */

    mypage->proc_count ++;                /* increment process
                                           count                */

    /* body of program goes here */
    .
    .
    .
    /* clean up */

    mypage->proc_count --;                /* decrement process
                                           count                */

    if (mypage->proc_count == 0 )
        printf("The last process is exiting\n");
    .
}
```

例 10-2: System V IPC および Memory Channel のコードの比較 (続き)

```

    .
    .
}

/*****
**** Memory Channel example ****
*****/

#include <sys/types.h>
#include <sys/imc.h>
main()
{
    typedef struct {
        int    proc_count;
        int    remainder[2047];
    } global_page;
    global_page *mypage_rx, *mypage_tx; [1]
    imc_asid_t   glob_id;
    int         logical_rail=0;
    int         temp;

    imc_api_init(NULL);

    imc_asalloc(123, 8192, IMC_URW | IMC_GRW, 0, &glob_id,
               logical_rail); [2]

    imc_asattach(glob_id, IMC_TRANSMIT, IMC_SHARED,
                 IMC_LOOPBACK, &(caddr_t)mypage_tx); [3]

    imc_asattach(glob_id, IMC_RECEIVE, IMC_SHARED,
                 0, &(caddr_t)mypage_rx); [4]

    /* increment process count */

    mypage_tx->proc_count = mypage_rx->proc_count + 1; [5]

    /* body of program goes here */
    :
    /* clean up */

    /* decrement process count */

    temp = mypage_rx->proc_count - 1; [6]
    mypage_tx->proc_count = temp;

    /* wait for MEMORY CHANNEL update to occur */

    while (mypage_rx->proc_count != temp)
        ;

    if (mypage_rx->proc_count == 0 )
        printf("The last process is exiting\n");
}

```

- ① このプロセスは、Memory Channel のグローバル・アドレス空間に書き込んだデータを読み取ることができなければなりません。そのため、送信用と受信用に 2 つのアドレスを宣言します。
- ② `imc_asalloc` 関数は Memory Channel のアドレス空間の領域を割り当てます。この領域の特性は次のとおりです。
 - `key=123` — この値で Memory Channel のアドレス空間の領域を識別します。この領域にアタッチする他のアプリケーションは同じキー値を使います。
 - `size=8192` — この領域のサイズは 8192 バイトです。
 - `perm=IMC_URW | IMC_GRW` — この領域のアクセス許可は、ユーザおよびグループによる読み取り/書き込み可能として割り当てられます。
 - `id=glob_id` — `imc_asalloc` 関数がこの変数に値を返します。この値は、割り当てられた領域を一意に識別します。プログラムでは、続いて Memory Channel の他の関数を呼び出す際に、この値を使います。
 - `logical_rail=0` — この領域は、Memory Channel の論理レール 0 を使って割り当てられます。
- ③ ここで `imc_asattach` 関数を呼び出して、`mypage_tx` 変数が指すアドレスに、送信用に領域をアタッチします。`flag` パラメータの値として `IMC_LOOPBACK` を指定し、プロセスがこの領域に書き込みを行うたびに、受信メモリにそのデータがループバックされるようにします。
- ④ ここで `imc_asattach` 関数を呼び出して、`mypage_rx` 変数が指すアドレスに、受信用に領域をアタッチします。
- ⑤ 受信ポインタが指す値に 1 を加え、その結果を送信ポインタが指すカウンタに代入することで、グローバル・プロセス・カウンタをインクリメントします。送信ポインタに書き込みを行う際、プログラムでは、書き込み命令が完了するのを待ちません。
- ⑥ プログラムの本体が完了した後、このプログラムはプロセスのカウンタをデクリメントし、デクリメントした値がクラスタ内の他のホストに伝送されたかどうかをテストします。デクリメントしたカウントであるこ

と(過渡的な値ではなく)を検証するために、このカウンタをローカル変数 `temp` に格納します。デクリメントした値を送信領域に書き込み、受信領域に到着する値が `temp` の値と一致するまで待ちます。一致したことで、プログラムは、デクリメントしたプロセス・カウンタが Memory Channel のアドレス空間に書き込まれたことが分かります。

この例では、ローカル変数を使うことで、プログラムが、受信側のメモリの値と送信された値を比較していることが保証されます。受信側メモリの値がアップデートされたことを確認する前にその値を使うと、誤ったデータを読み込んでしまう可能性があります。

10.6.6 エラー管理

共用メモリ・システムでは、メモリへの読み取りおよび書き込み処理で、エラーは発生しないと考えられています。Memory Channel システムでのエラーの発生頻度は1年に3回程度です。これは通常のネットワークおよび入出力サブシステムのエラー発生頻度に比べ、格段に低い数値です。

Memory Channel のハードウェアは、検出されたエラーを Memory Channel のソフトウェアに報告します。Memory Channel のハードウェアでは、アプリケーションでエラーに対処できるように、次の2つの保証をしています。

- 不正なデータをホスト・システムに書き込まない。
- Memory Channel のハードウェアにデータが書き込まれた順番で、ホスト・システムにデータを送る。

これらが保証されていることで、信頼性の高い、効果的なメッセージ・システムの開発が容易になります。

Memory Channel API ライブラリには、アプリケーションでのエラー処理を作成する際に役立つ、次の関数があります。

- `imc_ckerrcnt_mr` — `imc_ckerrcnt_mr` 関数は、Memory Channel のホスト上の指定した論理ルールでエラーが発生していないか探します。これにより、送信側プロセスは、メッセージを送信するときに、エラーが発生しているかどうかを知ることができます。
- `imc_rderrcnt_mr` — `imc_rderrcnt_mr` 関数は、指定した論理ルールの、クラスタ単位のエラー・カウントを読み取り、呼び出し元のプログラムにその値を返します。これにより、受信側のプロセスは、受け取ったメッセージのエラー状態を学習することができます。

オペレーティング・システムでは、クラスタ内で発生したエラーの数をカウントしています。システムは、クラスタ内の Memory Channel のハードウェアでエラーが検出されるたびに、また、ホストがクラスタのメンバになったりクラスタから外れたりするたびに、この値をインクリメントします。

エラーの検出および処理には、時間が少し (0 ではない) かかります。クラスタ内の別のホストでちょうど今発生したエラーがあるかもしれないため、`imc_rderrcnt_mr` 関数から返されるカウントは必ずしも最新ではありません。ローカル・ホストでは、このカウントは常に最新のデータになっています。

`imc_rderrcnt_mr` 関数を使って、単純で、効率的なエラー検出を実現することができます。メッセージを送信する前に、エラー・カウントを読み取って、メッセージに、このエラー・カウントを添付して送信します。受信側のプロセスは、メッセージを受け取った際に、メッセージに添付されているエラー・カウントと、メッセージの到着後に取り出したローカル値を比較します。ローカル値は、最新のデータであることが保証されているので、この値が送信されてきた値と同じであれば、その間にエラーが発生していないことが分かります。例 10-3 で、この方法を実際に示します。この例は `/usr/examples/cluster/mc_ex3.c` にあります。

例 10-3: `imc_rderrcnt_mr` 関数を使ったエラー検出

```
/*
 *
 * To compile : cc -DTRANSMIT -o mc_ex3_tx -g mc_ex3.c -limc
 *              cc -o mc_ex3_rx -g mc_ex3.c -limc
 *
 */

#ifdef TRANSMIT

/***** Transmitting Process *****/

#include <c_asm.h>
#define mb() asm("mb")

#include <sys/imc.h>

main()
{
    typedef struct {
        volatile int    msg_arrived;
        int    send_count;
        int    remainder[2046];
    } global_page;
    global_page *mypage_rx, *mypage_tx;
    imc_asid_t glob_id;
```

例 10-3: imc_rderrcnt_mr 関数を使ったエラー検出 (続き)

```
int          i;
volatile     int err_count;

imc_api_init(NULL);

imc_asalloc (1234, 8192, IMC_URW, 0, &glob_id,0);
imc_asattach (glob_id, IMC_TRANSMIT, IMC_SHARED, IMC_LOOPBACK,
              &(caddr_t)mypage_tx);
imc_asattach (glob_id, IMC_RECEIVE, IMC_SHARED, 0,
              &(caddr_t)mypage_rx);

/* save the error count */
while ( (err_count = imc_rderrcnt_mr(0) ) < 0 )
    ;

mypage_tx->send_count = err_count;

/* store message data */
for (i = 0; i < 2046; i++)
    mypage_tx->remainder[i] = i;

/* now mark as valid */
mb();

do {
    mypage_tx->msg_arrived = 1;
} while (mypage_rx->msg_arrived != 1); /* ensure no error on
                                     valid flag */

}

#else

/*****
***** Receiving Process *****/
*****/

#include <sys/imc.h>
main()
{
    typedef struct {
        volatile     int      msg_arrived;
        int          send_count;
        int          remainder[2046];
    } global_page;
    global_page      *mypage_rx, *mypage_tx;
    imc_asid_t        glob_id;
    int               i;
    volatile          int err_count;

    imc_api_init(NULL);

    imc_asalloc (1234, 8192, IMC_URW, 0, &glob_id,0);
    imc_asattach (glob_id, IMC_RECEIVE, IMC_SHARED, 0,
                  &(caddr_t)mypage_rx);

    /* wait for message arrival */
    while ( mypage_rx->msg_arrived == 0 )
        ;
}
```

例 10-3: imc_rderrcnt_mr 関数を使ったエラー検出 (続き)

```
/* get this systems error count */
while ( (err_count = imc_rderrcnt_mr(0) ) < 0 )
;

if (err_count == mypage_rx->send_count) {
    /* no error, process the body */

}
else {
    /* do error processing */

}
}
#endif
```

例 10-3 に示すとおり，`imc_rderrcnt_mr` 関数をメッセージの受信側で使うと，エラーを確実に検出することができます。ただし，送信側で使った場合は，エラーの検出は保証されません。これは，受信側ホストで発生したエラーが送信側ホストに通知される前に，送信側プロセスがエラー・カウントを読み取る可能性が少しある (0 ではない) ためです。例 10-3 では，送信側ホストにエラーを通知するためには，高レベルのプロトコルを使用しなければなりません。

`imc_ckerrcnt_mr` 関数を使うと，指定した論理レールのエラーを確実に検出することができます。この関数は，ユーザが用意したローカルのエラー・カウントと，論理レールの番号をパラメータとして受け取り，次のような場合にエラーを返します。

- 指定した論理レール上で，未処理のエラーが検出された場合。
- エラー処理が進行中の場合。
- エラー・カウントが，パラメータで指定されたものより大きい場合。

関数が正常に戻ってくれば，ローカルのエラー・カウントが格納されてから，`imc_ckerrcnt_mr()` 関数が呼び出されるまでの間に，エラーが検出されなかったことになります。

`imc_ckerrcnt_mr` 関数は，指定した論理レールの Memory Channel アダプタのハードウェア・エラー状態を読み取ります。これは，数マイクロ秒かかる，ハードウェア操作です。そのため，メモリの内容を読み取るだけの

imc_rderrcnt_mr 関数に比べ、imc_ckerrcnt_mr 関数の方が、実行に時間がかかります。

例 10-4 は、例 10-3 で示した送信手順の改訂版です。例 10-4 では、伝送側プロセスが、エラーの検出を行っています。この例は /usr/examples/cluster/mc_ex4.c にあります。

例 10-4: imc_ckerrcnt_mr 関数を使ったエラー検出

```
/*
 *
 * To compile: cc -o mc_ex4 -g mc_ex4.c -limc
 *
 */

#include <c_asm.h>
#define mb() asm("mb")

#include <sys/imc.h>
main()
{
    typedef struct {
        volatile int msg_arrived;
        int send_count;
        int remainder[2046];
    } global_page;
    global_page *mypage_rx, *mypage_tx;
    imc_asid_t glob_id;
    int i, status;
    volatile int err_count;

    imc_api_init(NULL);

    imc_asalloc (1234, 8192, IMC_URW, 0, &glob_id,0);
    imc_asattach (glob_id, IMC_TRANSMIT, IMC_SHARED, IMC_LOOPBACK,
        &(caddr_t)mypage_tx);
    imc_asattach (glob_id, IMC_RECEIVE, IMC_SHARED, 0,
        &(caddr_t)mypage_rx);

    /* save the error count */
    while ( (err_count = imc_rderrcnt_mr(0) ) < 0 )
        ;

    do {
        mypage_tx->send_count = err_count;

        /* store message data */
        for (i = 0; i < 2046; i++)
            mypage_tx->remainder[i] = i;

        /* now mark as valid */
        mb();

        mypage_tx->msg_arrived = 1;

        /* if error occurs, retransmit */
    } while ( (status = imc_ckerrcnt_mr(&err_count,0)) != IMC_SUCCESS);
```

例 10-4: `imc_ckerrcnt_mr` 関数を使ったエラー検出 (続き)

```
}
```

10.7 クラスタ単位のロック

Memory Channel システムでは、プロセスは Memory Channel のアドレス空間の領域を読み書きして、通信します。前の節では、領域に自由に読み書きするサンプル・プログラムを示しました。しかし実際は、ロック・メカニズムを使って、領域や他のクラスタ単位のリソースへのアクセスを制御する必要があります場合があります。Memory Channel API ライブラリには、ロックのための一連の関数が用意されています。これらの関数を使えば、アプリケーションでリソースへのアクセス制御を実現することができます。

Memory Channel API ライブラリは、Memory Channel のグローバル・アドレス空間にマッピングされたページを使って、ロックを実現しています。効率を考え、ロックは個々にではなく、セットで割り当てられます。`imc_lkalloc` 関数を使うと、ロック・セットを割り当てることができます。たとえば、20 個のロックを使用したい場合は、20 個のロックを 1 セット作成する方が、4 個のロックを 5 セット作成するよりも効率的です。

分散型アプリケーションでの最初の調整を簡単に行いたい場合は、`imc_lkalloc` 関数を使うことで、プロセスはアトミックに (つまり、1 回の操作で) ロック・セットを割り当て、そのセットの最初のロックを取得することができます。この機能を使用すると、プロセスはそれ自身がそのロック・セットを割り当てた最初のプロセスかどうか分かります。最初のプロセスであった場合、そのプロセスはロックへのアクセスが保証され、リソースを確実に初期化することができます。

ロック・セットを割り当てて、最初のロックをアトミックに取得する代わりに、プロセスはまず、`imc_lkalloc` 関数を呼び出し、その後で、`imc_lkacquire` 関数を呼び出すこともできます。この場合、1 つ目の関数を呼び出して 2 つ目の関数を呼び出すまでの間に、別のプロセスがそのロックを取得してしまうという恐れがあります。この場合、最初のプロセスはロックへのアクセスが保証されません。

例 10-5 では、Memory Channel のアドレス空間の領域をロックする最初のプロセスがその領域を初期化し、次にこの領域にアクセスするプロセスが、

プロセス・カウンタをアップデートする単純なプログラムを示します。この例は /usr/examples/cluster/mc_ex5.c にあります。

例 10–5: Memory Channel の領域のロック

```
/*
 *
 * To compile: cc -o mc_ex5 -g mc_ex5.c -limc
 *
 */

#include <sys/types.h>
#include <sys/imc.h>

main ( )
{
    imc_asid_t    glob_id;
    imc_lkid_t    lock_id;
    int          locks = 4;
    int          status;

    typedef struct {
        int      proc_count;
        int      pattern[2047];
    } clust_rec;

    clust_rec *global_record_tx, *global_record_rx; [1]
    caddr_t    add_rx_ptr = 0, add_tx_ptr = 0;
    int        j;

    status = imc_api_init(NULL);

    imc_asalloc(123, 8192, IMC_URW, 0, &glob_id, 0);

    imc_asattach(glob_id, IMC_TRANSMIT, IMC_SHARED,
        IMC_LOOPBACK, &add_tx_ptr);

    imc_asattach(glob_id, IMC_RECEIVE, IMC_SHARED,
        0, &add_rx_ptr);

    global_record_tx = (clust_rec*) add_tx_ptr; [2]
    global_record_rx = (clust_rec*) add_rx_ptr;

    status = imc_lkalloc(456, &locks, IMC_LKU, IMC_CREATOR,
        &lock_id); [3]
    if (status == IMC_SUCCESS)
    {
        /* This is the first process. Initialize the global region */

        global_record_tx->proc_count = 0; [4]
        for (j = 0; j < 2047; j++)
            global_record_tx->pattern[j] = j;

        /* release the lock */
        imc_lkrelease(lock_id, 0); [5]
    }

    /* This is a secondary process */
```

例 10-5: Memory Channel の領域のロック (続き)

```
else if (status == IMC_EXISTS)
{
    imc_lkalloc(456, &locks, IMC_LKU, 0, &lock_id); [6]

    imc_lkacquire(lock_id, 0, 0, IMC_LOCKWAIT); [7]

    /* wait for access to region */

    global_record_tx->proc_count = global_record_rx->proc_count+1; [8]

    /* release the lock */

    imc_lkrelease(lock_id, 0);

}

/* body of program goes here */


/* clean up */

imc_lkdealloc(lock_id); [9]
imc_asdetach(glob_id);
imc_asdealloc(glob_id);
}
```

- [1] このプロセスは、Memory Channel のグローバル・アドレス空間に自身で書き込んだデータを読み取るために、領域を送信用と受信用にマップします。この手順についての詳細は、例 10-2 を参照してください。
- [2] プログラムは、送信ポインタと受信ポインタに、グローバル・レコードの構造体を代入します。
- [3] このプロセスは、4 つのロックからなり、key の値が 456 のロック・セットを作成しようとします。imc_lkalloc 関数の呼び出しでは、IMC_CREATOR フラグも指定します。こうすると、ロック・セットがまだ割り当てられていなかった場合は、この関数が自動的に 0 番のロックを取得します。このロック・セットが既に存在していた場合、imc_lkalloc 関数はロック・セットの割り当てに失敗し、値 IMC_EXISTS を返します。
- [4] ロック・セットを作成してロック番号 0 を取得したプロセスが、グローバル領域を初期化します。

- ⑤ このプロセスは、この領域の初期化が終了すると、`imc_lkrelease` 関数を呼び出して、ロックを解放します。
- ⑥ この領域が初期化された後に実行される 2 番目のプロセスは、初めの `imc_lkalloc` 関数の呼び出しに失敗し、今度は `IMC_CREATOR` フラグを指定せずに、この関数を再度呼び出します。 `key` パラメータの値が同じ (456) なので、この関数呼び出しでは、同じロック・セットが割り当てられます。
- ⑦ 2 番目のプロセスは `imc_lkacquire` 関数を呼び出して、ロック・セットからロック 0 を取得します。
- ⑧ 2 番目のプロセスは、プロセス・カウンタをアップデートし、その値を送信領域に書き込みます。
- ⑨ プログラムの最後で、このプロセスはすべての Memory Channel リソースを解放します。

あるプロセスがロックを取得すると、同じクラスタ上の他のプロセスは、そのロックを取得することができません。

ロックが解放されるのを待つと、プログラムがループして待つことになるため、性能に大きな影響を及ぼします。そのため、システム全体の性能を考え、アプリケーションでは、必要なロックだけを取得し、できるだけ早く解放するようにしてください。

10.8 クラスタ・シグナル

Memory Channel API ライブラリには、プロセスが、クラスタ内のリモート・ホスト上で動作している特定のプロセスにシグナルを送ることができるようにする、`imc_kill` 関数があります。この関数は UNIX の `kill(2)` 関数と似ています。`kill` 関数をクラスタで使用すると、そのプロセスが他のクラスタ・メンバで動作していても、プロセス・グループ番号が PID の絶対値と同じすべてのプロセスへシグナルが送られます。PID はクラスタ全体で一意であることが保証されています。

ただし、`imc_kill` 関数は `kill` 関数と違って、シグナルをクラスタの他のメンバに送れないだけでなく、複数のプロセスに送ることもできません。

10.9 クラスタ情報

以降の各項では、Memory Channel API 関数を使ってクラスタ情報にアクセスする方法と、コマンド行から状態情報にアクセスする方法を説明します。

10.9.1 Memory Channel API の関数を使って Memory Channel API クラスタ情報にアクセスする方法

Memory Channel API ライブラリには、`imc_getclusterinfo` 関数があります。この関数を使うと、プロセスは Memory Channel API クラスタ内のホストに関する情報を取得することができます。この関数は、次の情報を返します。

- クラスタ内のホストの数、およびそれぞれのホスト名。
- クラスタ内の論理レールの数。
- Memory Channel のアクティブな論理レールを示すビットマスク。ビットが 1 のとき、アクティブな論理レールを示す。

Memory Channel API ライブラリが初期化されていないホストでは、この関数を使っても、そのホストの情報は返されません。

Memory Channel API ライブラリには、`imc_wait_cluster_event` 関数があります。この関数は、指定したクラスタのイベントが発生するまで、呼び出し元のスレッドをブロックします。次の Memory Channel API クラスタのイベントが有効です。

- ホストがクラスタのメンバになる、またはメンバでなくなる。
- クラスタの論理レール構成が変化する。

`imc_wait_cluster_event` 関数は、監視対象の Memory Channel API クラスタ構成の項目の現在の状況を調べ、新しい Memory Channel API クラスタ構成を返します。

例 10-6 では、`imc_getclusterinfo` 関数を `imc_wait_cluster_event` 関数と併用して、Memory Channel API のクラスタ・メンバの名前と、Memory Channel のアクティブな論理レールのビットマスクを要求し、どちらかが変化したことを示すイベントを待つ方法を示します。この例は `/usr/examples/cluster/mc_ex6.c` にあります。

例 10-6: Memory Channel API クラスタ情報を要求し , Memory Channel API クラスタ・イベントを待つ方法

```
/*
 *
 * To compile: cc -o mc_ex6 -g mc_ex6.c -limc
 *
 */

#include <sys/imc.h>

main ( )
{
    imc_railinfo    mask;
    imc_hostinfo    hostinfo;

    int             status;
    imc_infoType     items[3];
    imc_eventType    events[3];

    items[0] = IMC_GET_ACTIVERAILS;
    items[1] = IMC_GET_HOSTS;
    items[2] = 0;

    events[0] = IMC_CC_EVENT_RAIL;
    events[1] = IMC_CC_EVENT_HOST;
    events[2] = 0;

    imc_api_init(NULL);

    status = imc_getclusterinfo(items,2,mask,sizeof(imc_railinfo),
                                &hostinfo,sizeof(imc_hostinfo));

    if (status != IMC_SUCCESS)
        imc_perror("imc_getclusterinfo:",status);

    status = imc_wait_cluster_event(events, 2, 0,
                                    mask, sizeof(imc_railinfo),
                                    &hostinfo, sizeof(imc_hostinfo));

    if ((status != IMC_HOST_CHANGE) && (status != IMC_RAIL_CHANGE))
        imc_perror("imc_wait_cluster_event didn't complete:",status);
} /*main*/
```

10.9.2 コマンド行から Memory Channel の状態情報にアクセスする方法

Memory Channel API ライブラリでは , Memory Channel の状態を報告する `imcs` コマンドを用意しています。 `imcs` コマンドは , 現在アクティブな Memory Channel のファシリティに関する情報を , 標準出力に表示します。出力結果には , 領域やロック・セットがリストされ , 次のような情報が示されます。

- 領域やロック・セットを作成した，サブシステムのタイプ (IMC または PVM)
- Memory Channel の領域の識別子
- Memory Channel の領域やロック・セットを示す，アプリケーション固有のキー
- 領域のサイズ (バイト単位)
- 領域やロック・セットのアクセス・モード
- 領域やロック・セットの所有者のユーザ名
- 領域やロック・セットの所有者のグループ
- その領域で使用されている Memory Channel の論理レール
- その領域の一貫性を示すフラグ
- そのロック・セットで利用できるロックの数
- 割り当てられた領域の総数
- Memory Channel API のオーバーヘッド
- Memory Channel のレールの使用状況

10.10 共用メモリ・モデルとメッセージ渡しモデルの比較

Memory Channel API ライブラリをベースにアプリケーションを開発する場合，利用できるモデルが 2 つあります。

- 共用メモリ・モデル
- メッセージ渡しモデル

最初は，共用メモリ・モデルを使ったアプローチの方が，Memory Channel の機能には適しているように見えるかもしれません。しかし，このモデルを使用する開発者は，この章で説明した，遅延の問題，一貫性の問題，およびエラー検出の問題に対処しなければなりません。場合によっては，これらの問題をアプリケーションから隠す，単純なメッセージ渡しのライブラリを開発した方が適していることがあります。このようなライブラリでのデータ伝送関数は，ユーザ空間内で完全に実現することができます。そのため，これらの関数は，共用メモリ・モデルを基にして実現されたものと同じぐらい効率的に動作します。

10.11 よくある質問 (FAQ)

この節では、Memory Channel API を使って TruCluster システムのプログラムを開発しているプログラマから寄せられた質問に対する回答を示します。

10.11.1 IMC_NOMAPPER リターン・コード

質問: `imc_asattach` 関数を使って、一貫性のある領域にアタッチしようとしたが、値 `IMC_NOMAPPER` が返されました。これはどういう意味ですか。

回答: このリターン値は、ご使用の Memory Channel API クラスタ内のシステムに `imc_mapper` プロセスが存在しないことを示しています。

`imc_mapper` プロセスは、次の場合に自動的に起動されます。

- システムの初期化時に、構成変数 `IMC_AUTO_INIT` の値が 1 の場合。
`IMC_AUTO_INIT` 変数についての詳細は、10.2 節を参照。
- `imc_init` コマンドを初めて実行した場合。

この問題を解決するには、`imc_mapper` プロセスが消えているシステムをリポートしてください。

このエラーは、システムをシャットダウンして `init` レベル 3 からシングルユーザ・モードにし、完全なリブートの手順を踏まないで、マルチユーザ・モードに戻した場合に発生する可能性があります。TruCluster Server ソフトウェアが動作しているシステムをリポートする場合は、そのシステムの完全なリブート手順を踏まなければなりません。

10.11.2 効率的なデータ・コピー

質問: Memory Channel の帯域幅を最大限に生かすためには、どのようにデータを Memory Channel の送信領域にコピーするとよいですか。

回答: Memory Channel API の `imc_bcopy` 関数を使用すると、位置合わせされたデータと位置合わせされていないデータのどちらでも効率的に Memory Channel にコピーすることができます。`imc_bcopy` 関数は、AlphaServer CPU のバッファリング機能を最大限に利用するように最適化されています。

`imc_bcopy` 関数を使って、通常のメモリの 2 つのバッファ間で、データを効率的にコピーすることもできます。

10.11.3 Memory Channel の帯域幅の可用性

質問: Memory Channel の一貫性のある領域を使った場合, Memory Channel の帯域幅を最大限に利用できますか。

回答: できません。一貫性のある領域では, 局所的な一貫性を維持するために, ループバック機能が使われます。したがって, それぞれのデータ書き込みサイクルに対応して, データをループバックするサイクルがあるため, 利用可能な帯域幅は半分になります。ループバック機能についての詳細は, 10.6.1.3 項を参照してください。

10.11.4 Memory Channel API クラスタ構成の変更

質問: プログラムでは, Memory Channel API クラスタ構成に変更があったかどうかを, どのようにして確認することができますか。

回答: 新しい `imc_wait_cluster_event` 関数を使うと, ホストが Memory Channel API クラスタのメンバになったり外れたりする状況や, アクティブな論理レールの状態の変化を監視することができます。プログラムでは, 別のスレッドで `imc_wait_cluster_event` 関数を呼び出すようにすることができます。この関数では, 状態が変化するまで, 呼び出し元がブロックされるためです。

10.11.5 バス・エラー・メッセージ

質問: プログラム内で, アタッチした送信領域に値を設定しようとする, 次のようなメッセージが出力されてクラッシュします。

```
Bus error (core dumped)
```

なぜこのようなエラーが発生するのですか。

回答: その値のデータ型が 32 ビットよりも小さい可能性があります。C 言語では, `int` は 32 ビットのデータ項目, `short` は 16 ビットのデータ項目です。HP Alpha プロセッサは, 他の RISC プロセッサと同様に, データの読み取り/書き込みを 64 ビット単位, または 32 ビット単位で行います。32 ビットより小さい値をデータ項目に代入した場合, コンパイラは, 32 ビット単位のデータをロードして, 変更すべき部分のバイトを変更し, その後, 32 ビット単位全体を格納するコードを生成します。このようなデータ項目が, 送信用にアタッチされた Memory Channel の領域にある場合, 代入を実行する

と、アタッチされた領域内で読み取り操作が発生します。送信領域は書き込み専用なので、バス・エラーが発生します。

この問題は、すべてのアクセスが 32 ビットのデータ項目に対して行われるようにすることで回避できます。詳細は、10.6.3 項を参照してください。



A

ASE (Available Server Environment)

(個々のASEエントリを参照)

ASE アプリケーション

- DRD サービス 5-4
- NFS サービス 5-3
- TruCluster Server バージョン 5.1B
への移行 5-1
- TruCluster Server バージョン 5.1B
への移行の準備 5-6
- ディスク・サービス 5-2
- テープ・サービス 5-5
- ユーザ定義サービス 5-4

ASE コマンドの CAA コマンドへの置き換え 5-10

ASE サービスから TruCluster

Server への移行 5-6

ASE サービスから TruCluster

Server への移行の準備 5-6

ASE スクリプト 5-9

- ASE 変数 5-13
- nfs_ifconfig の置き換え 5-11
- イベントのポスト 5-14
- エラーの処理 5-11
- 起動スクリプトと停止スクリプトの
結合 5-10
- 終了コード 5-14

スクリプトの出力のリダイレク

ト 5-11

ストレージ管理 5-12

デバイス名 5-13

ネットワーク・サービス 5-17

別名の使用 5-15

ASE スクリプトの検討項目

ASE コマンドの CAA コマンドへの
置き換え 5-10

ASE データベース

バージョン 1.4 以前での内容の保
存 5-8

バージョン 1.5 以降での内容の保
存 5-7

ASE と CAA の違い 5-1

ASE と CAA の比較 5-1

ASE 変数 5-13

C

CAA

SysMan Menu による管理 ... 2-31

アプリケーション・リソース配置ポ
リシ 2-8

オプション・リソース 2-9

カスタム・アプリケーション・リ
ソースの属性の定義 2-21

状態情報 2-28

処理スクリプトからの出力のリダイ
 レクト..... 2-20
 処理スクリプトからプロファイル属
 性へのアクセス..... 2-19
 処理スクリプトによりアクセス可能
 な環境変数 2-18
 処理スクリプトの作成 2-14
 シングル・インスタンス・アプリ
 ケーションの高可用性実現のため
 の使用..... 2-1
 チュートリアル 2-32
 テープ・リソース・プロファイ
 ル 2-11
 ネットワーク・リソース・プロファ
 イル 2-9
 必須リソース 2-7
 メディア・チェンジャ・リソース・
 プロファイル 2-12
 リソースの登録抹消..... 2-28
 リソース・プロファイル 2-2
 理由コード..... 2-19
CAA 情報へのアクセス..... 2-28
CAA と **ASE** の違い..... 5-1
CDFS ファイル・システムの制約 7-3
CDSL 4-4
clua_error() 関数..... 8-2t, 8-3
clua_getaliasaddress() 関数 .. 8-2t,
 8-4
 例..... 8-4
clua_getaliasinfo() 関数 . 8-2t, 8-4
 例..... 8-4
clua_getdefaultalias() 関数... 8-2t,
 8-5
clua_isalias() 関数 8-3t, 8-5

clua_registerservice() 関数... 8-3t,
 8-5
 ポート属性..... 8-11
clua_services ファイル..... 8-11
clua_unregisterservice() 関
 数..... 8-3t, 8-5
/cluster/admin/run ディレクトリか
 ら呼び出されるスクリプト.... 7-5
clusvc_getcommport() 関数 .. 8-3t,
 8-5
 例..... 8-6
clusvc_getresvcommport() 関
 数..... 8-3t, 8-5
 例..... 8-6

D

DLM

親..... 9-9
 削除されたパラメータ 4-9
 使用規則..... 9-4
 デッドロックの検出..... 9-15
 プログラミング・インタフェー
 ス 9-2
 変換 9-11, 9-15
 ランク付け..... 9-12
 リソース定義 9-5
 ロック 9-10
 ロックが許可される場合 9-10
 ロックの解除 9-18
 ロックの解放 9-18
 ロックのキュー 9-13
 ロックのキューからの削除... 9-18
 ロックの変換 9-22
 ロックの要求モード..... 9-10

ロック・モード 9-11
dlm_cancel 関数 9-19
dlm_cvt 関数 9-11, 9-15, 9-24
dlm_detach 関数 9-5
dlm_locktp 関数 9-10
dlm_lock 関数 9-10
dlm_notify 関数 9-20, 9-22
dlm_nsjoin 関数 9-8, 9-10
dlm_quecvt 関数 9-11,
 9-15, 9-19, 9-24
dlm_quelocktp 関数 9-10,
 9-11, 9-19
dlm_quelock 関数 9-10, 9-19
dlm_set_signal 関数... 9-20, 9-22n
dlm_unlock 関数 .. 9-5, 9-18, 9-26
DRD サービス 5-4

E

/etc/clua_services ファイル ... 8-11
/etc/rc.config ファイル
 IMC_AUTO_INIT 変数 10-5
/etc/sysconfigtab ファイル
 rm_rail_style パラメータの設
 定 10-4
 sysconfigdb(8) による修正 10-4
 vm_syswiredpercent パラメー
 タ 10-8, 10-9

I

imc_api_init 関数 10-6, 10-22
imc_asalloc 関数 10-22, 10-26
 key パラメータ 10-12

imc_asattach_ptp 関数 10-13
imc_asattach 関数 10-22,
 10-23, 10-26, 10-39
 dir パラメータ 10-13
imc_asdealloc 関数 10-23
imc_asdetach 関数 10-23
IMC_AUTO_INIT 変数 10-5, 10-39
imc_bcopy 関数 10-19, 10-39
imc_ckerrcnt_mr 関数 10-27
IMC_EXISTS
 リターン状態 10-34
imc_getclusterinfo 関数 10-36
 例 10-37e
imc_init コマンド 10-5,
 10-10, 10-12, 10-39
 IMC_NOTINIT リターン状態 10-8
 maxalloc パラメータ 10-7
 maxrecv パラメータ 10-7
imc_kill 関数 10-35
imc_lkacquire 関数 . 10-32, 10-35
imc_lkalloc 関数 10-32
imc_lkrelease 関数 10-35
IMC_MAPENTRIES リターン状
 態 10-11
imc_mapper プロセス
 存在しない 10-39
IMC_MAX_ALLOC 変数 10-7
IMC_MAX_RECV 変数 . 10-7, 10-9
IMC_MCFULL リターン状態 10-10
IMC_NOMAPPER リターン状
 態 10-39
IMC_NOMEM リターン状態 . 10-12

IMC_NORESOURCES リターン状態..... 10-12
IMC_NOTINIT リターン状態.. 10-8
imc_perror 関数..... 10-22
imc_rderrent_mr 関数..... 10-27
IMC_RXFULL リターン状態 . 10-7, 10-11
imc_wait_cluster_event 関数..... 10-36, 10-40
 例..... 10-37e
IMC_WIRED_LIMIT リターン状態..... 10-8, 10-11
imcs コマンド 10-37

L

libcfg システム・ライブラリ.... 8-3
libclua システム・ライブラリ . 8-2, 8-3
libdlm システム・ライブラリ... 9-3
Logical Storage Manager
 (LSM を参照)
LSM..... 7-3

M

malloc の失敗 10-12
Memory Channel
 アドレス..... 10-12
 エラーの発生頻度..... 10-27
 構成のチューニング..... 10-7
 チューニング 10-7
 トラブルシューティング 10-8
 マルチレール・モデル 10-2
Memory Channel API ライブラリ

imc_api_init 関数 10-6, 10-22
imc_asalloc 関数 ... 10-12, 10-22
imc_asattach_ptp 関数 10-13
imc_asattach 関数 . 10-13, 10-22, 10-23, 10-26, 10-39
imc_asdealloc 関数..... 10-23
imc_bcopy 関数 10-19, 10-39
imc_ckerrcnt_mr 関数..... 10-27
imc_getclusterinfo 関数 10-36
imc_init コマンド 10-8, 10-12, 10-39
imc_lkacquire 関数. 10-32, 10-35
imc_lkalloc 関数 10-32
imc_lkrelease 関数 10-35
imc_perror 関数..... 10-22
imc_rderrent_mr 関数..... 10-27
imc_wait_cluster_event 関数 10-36, 10-40
imcs コマンド..... 10-37
 アプリケーションの開発 10-1
 初期化 10-6
Memory Channel API ライブラリの
 初期化 10-5
Memory Channel のアドレス空間
 拡張 10-10
Memory Channel のアドレスへのアクセス 10-12
Memory Channel のチューニング 10-7
Memory Channel の領域
 送信用のアタッチ.. 10-22, 10-26
 デタッチ..... 10-23
 割り当て..... 10-22, 10-26

N

- nfs_ifconfig** の置き換え 5-11
- NFS** サービス 5-3

P

- PID**
 - 拡張 4-8
- print_clua_liberror()** 関数 8-3

R

- rc.config** ファイル
 - IMC_AUTO_INIT** 変数 10-5
- rm_rail_style** パラメータ
 - 可能な値 10-4
- rm_slave_init**
 - 致命的なエラー 10-10
- RPC** プログラム
 - 必要なアプリケーションの変更 7-1
- RPC** プログラムに必要なアプリケーションの変更 7-1

S

- setsockopt()** 関数
 - ソケット・オプション 8-9
 - ソケット・オプションとポート属性の関係 8-11
- SO_CLUA_DEFAULT_SRC** ソケット・オプション 8-10
- SO_CLUA_IN_NOALIAS** ソケット・オプション 8-10

- SO_CLUA_IN_NOLOCAL** ソケット・オプション 8-10
- SO_RESVPORT** ソケット・オプション 8-10
- SO_REUSEALIASPORT** ソケット・オプション 8-10
- sysconfigtab** ファイル . 10-4, 10-8
- SysMan Menu** 2-31
- SysMan Menu** による **CAA** の管理 2-31
- SysMan Station** 2-32
- SysMan Station** の使用 2-32

U

- UDP** アプリケーション
 - ソース・アドレスの返却 8-11
- User Datagram Protocol**
 - (**UDP** アプリケーション を参照)
- /usr/var/adm**
 - メッセージ・ファイル 10-9

V

- vm-mapentries** パラメータ . 10-11
- vm_syswiredpercent** パラメータ 10-9, 10-11

あ

- アタッチ
 - ブロードキャスト・モード . 10-14
 - ポイント・ツー・ポイント・モード 10-13, 10-16

ループバック・モード 10-13,
10-17, 10-40
アドレスのマッピング
 実現方法 10-13
 定義 10-12
アプリケーション開発モデル 10-38
アプリケーションの移行
 拡張 PID..... 4-8
 クラスタ単位のファイルとメンバ固
 有のファイル 4-1
 デバイス名..... 4-5
 パッケージとライセンス 4-9
 プロセス間通信 4-7
アプリケーションのタイプ 1-2
アプリケーションのプログラミングの
 検討項目 7-1
アプリケーション・パッケージとライ
 センス 4-9
アプリケーション・リソース
 起動 2-22
 停止 2-28
 配置ポリシ..... 2-8
 プロファイル 2-4
アプリケーション・リソースの起
 動..... 2-22
アプリケーション・リソースの再配
 置..... 2-24
アプリケーション・リソースの処理ス
 クリプト
 ガイドラインの作成..... 2-16
アプリケーション・リソースの処理ス
 クリプトの作成 2-16
アプリケーション・リソースの停
 止..... 2-28

アプリケーション・リソースの分
散..... 2-25

い

移植性のあるアプリケーション.. 7-2
一時ポート 8-1, 8-7
一貫性
 初期状態..... 10-18, 10-19
 遅延にかかわる 10-23
イベント
 ポスト 5-14
イベントのポスト 5-14

え

エラー管理 10-27
 imc_ckerrcnt_mr を使用 ... 10-30
 例 10-31e
 imc_rderrcnt_mr を使用 ... 10-30
 例 10-28e
エラーの処理..... 5-11, 10-27
 imc_ckerrcnt_mr を使用 ... 10-30
 例 10-31e
 imc_rderrcnt_mr を使用 ... 10-30
 例 10-28e
エラー・メッセージ 10-8
 致命的 10-9

お

オプション・リソース..... 2-9
親ロック 9-9, 9-25

か

- 書き込みロックの保護..... 9-11
- 拡張プロセス ID..... 4-8
- 仮想メモリのマップ・エントリ
 - 拡張 10-11
- 環境変数
 - 処理スクリプトからアクセス可能 2-18

き

- 既知のポート..... 8-1, 8-7
- キャッシング
 - ロック値ブロックをローカル・バッファに使用 9-27
- 共用メモリとメッセージ渡し
 - 比較 10-38
- 許可キュー 9-13
- キル
 - クラスタ単位 10-1

く

- クラスタ・アプリケーション 1-2
- クラスタ・アプリケーションの可用性 (CAA を参照)
- クラスタ・アプリケーションのタイプ..... 1-1
- クラスタ・シグナル 10-35
- クラスタ情報の関数 10-36
- クラスタ単位のアドレス空間 . 10-12
- クラスタ単位のファイルとメンバ固有のファイル..... 4-1

クラスタ単位のロック

- シングルスレッドによるアクセス 10-35
- 性能への影響 10-35
- 定義 10-32
- 例..... 10-33e
- 割り当て..... 10-34
- クラスタ内でのアプリケーションの複数インスタンスの実行 1-6
- クラスタにおけるファイル・アクセスの障害許容度 7-6
- クラスタ別名
 - lua_*() 関数を使うプログラムのコンパイル 8-3
- UDP アプリケーションとソース・アドレス 8-11
- アプリケーション・プログラミング・インタフェース 8-1
- ソケット・オプション 8-9
- マルチ・インスタンス・アプリケーションでの使用..... 3-1
- 予約ポートへのバインド 8-8
- クラスタ・ポート空間..... 8-6

こ

- 高レベルのロック 9-12
- 固定メモリの限界値
 - 拡張 10-11
- コンソール・エラー・メッセージ..... 10-9

さ

サブロック 9-25

し

シグナル 10-35

終了コード 5-14

受信用アドレス空間

 拡張 10-11

初期化エラー

 vm_syswiredpercent パラメー

 タ 10-9

初期化の失敗 10-9

初期状態の一貫性 10-18, 10-19

処理スクリプト

 CAA によって呼び出された理由の

 調査 2-19

 アプリケーション・リソース 2-16

 環境変数へのアクセス 2-18

 作成 2-14

 出力のリダイレクト 2-20

 プロファイル属性へのアクセ

 ス 2-19

処理スクリプトの作成 2-14

シングル・インスタンス・アプリケー

 ション 1-2

診断ユーティリティのサポート.. 7-3

す

スクリプト 5-8

 (ASE スクリプト も参照)

 出力のリダイレクト 5-11

 処理 2-14

スクリプトの出力のリダイレク

 ト 5-11

ストレージ管理 5-12

せ

制限

 リソース 10-7, 10-11

セグメント違反

 送信ポインタからの読み取り 10-19

 送信領域からの読み取り ... 10-40

そ

送信ポインタからの読み取り

 セグメント違反 10-19

送信領域からの読み取り

 セグメント違反 10-40

ソケット・オプション 8-9

ち

遅延と一貫性 10-23

致命的なエラー

 rm_slave_init 10-10

て

ディスク・サービス 5-2

低レベルのロック 9-12

デッドロックの検出

 分散ロック・マネージャによる検

 出 9-15

デバイス名 4-5, 5-13

 識別 4-5

デバイス名の識別 4-5

デバイス要求ディスパッチャ	5-4
テープ・サービス	5-5
テープ・リソース・プロファイ ル.....	2-11

と

同期ロック要求	9-20
同期ロック要求の完了.....	9-20
同時書き込みロック	9-11
同時読み取りロック	9-11
動的ポート	8-1

ぬ

ヌル・モード・ロック.....	9-11
-----------------	------

ね

ネットワーク・アダプタのフェイル オーバー	4-9
ネットワーク・サービス	5-17
ネットワーク・ファイル・システム (NFS サービス を参照)	
ネットワーク・リソース・プロファイ ル.....	2-9
ネームスペース	9-7

は

排他ロック	9-11
バッファ・キャッシング (キャッシング を参照)	

ひ

必須リソース.....	2-7
非同期ロック要求	9-19
非同期ロック要求の完了	9-19

ふ

ファイル	
上書きを防ぐ	4-2
ファイル・システムのパーティショニ ング	5-18
ファイルへの上書き	
防ぐ	4-2
ファイルへの上書きを防ぐ方法..	4-2
フェイルオーバー	
ネットワーク・アダプタ	4-9
複数のアプリケーション・インスタン ス間で共用ファイルの同期をとるた めの flock() の使用	4-7
複数のアプリケーション・インスタン ス間での共用データへの同期アクセ ス.....	4-7
複数リソース・デッドロック ...	9-15
プロセス ID (PID を参照)	
プロセス間通信	4-7
プロセスの仮想メモリ	
拡張	10-12
ブロッキング・レイヤード・プロダク ト.....	4-10
ブロック通知ルーチン.....	9-21
バッファの方式の選択	9-29

ローカル・バッファ・キャッシング
に使用 9-28
プロファイル
(CAA を参照)
ブロードキャスト・アタッチ 10-14
分散型アプリケーション 1-8
TruCluster Server バージョン 5.1B
への移行 6-1
分散型アプリケーションの **TruCluster**
Server への移行 6-1
分散型アプリケーションの **TruCluster**
Server への移行の準備 6-1
分散ロック・マネージャ
(DLM を参照)

へ

別名
(クラスタ別名 を参照)
別名の使用 5-15
変換待ちキュー 9-13
変数
ASE 5-13

ほ

ポイント・ツー・ポイント・アタッ
チ 10-16
保護書き込みロック 9-11
保護読み取りロック 9-11
ポート
一時 8-1, 8-7
既知の 8-1
クラスタ・ポート空間 8-6
シングル・インスタンス・アプリ
ケーション 8-7

属性 8-11
動的 8-1
マルチ・インスタンス・アプリケー
ション 8-7
マルチ・インスタンス・アプリケー
ションと予約ポート 8-9
予約 8-2, 8-7
予約ポートへのバインド 8-8
ロック 8-2

ま

待ちキュー 9-13
マルチ・インスタンス・アプリケー
ション
クラスタ内での実行 1-5
クラスタ別名の使用 3-1
マルチレール・モデル 10-2
rm_rail_style パラメータ 10-4
省略時のスタイル 10-3
シングルレール 10-2
フェイルオーバー・ペア 10-3
物理レール 10-2
論理レール 10-2

め

メッセージ
コンソール・エラー 10-9
メッセージ渡しと共用メモリ
比較 10-38
メディア・チェンジャ・リソース・プ
ロファイル 2-12
メンバ固有のファイルとクラスタ単位
のファイル 4-1
CDSL の使用 4-4

メンバ固有のファイルのコンテキスト
依存シンボリック・リンク
(CDSL を参照)
メンバ固有のリソース..... 4-8

ゆ

ユーザ定義サービス 5-4
ユーザ定義属性
CAA アプリケーション・リソース・
プロファイル 2-21

よ

読み取りロックの保護..... 9-11
予約ポート 8-2, 8-7
既知のポートや動的ポートとして使
用 8-8
予約ポートとクラスタ別名へのバイン
ド..... 8-8

ら

ライセンス
アプリケーション..... 4-9

り

リソース 2-1
(CAA も参照)
オプション..... 2-9
細分性 9-6
識別 9-8
ネームスペース 9-7

必須 2-7
分散ロック・マネージャでの定
義 9-5
リソースのグループ許可モード. 9-11
リソースの制限
IMC_MAPENTRIES リターン状
態 10-11
IMC_MCFULL リターン状
態 10-10
IMC_NOMEM リターン状態 10-12
IMC_NORESOURCES リターン状
態 10-12
IMC_RXFULL リターン状態 10-7,
10-11
IMC_WIRED_LIMIT リターン状
態 10-8, 10-11
リソースの登録 2-22
リソースの登録抹消 2-28
リソース・プロファイル 2-2
ネットワーク 2-9
メディア・チェンジャ 2-12
リソース・プロファイルの作成.. 2-3
リモート・プロシージャ・コール
(RPC プログラム を参照)

る

ループバック・アタッチ 10-17,
10-40

れ

レイヤード・プロダクト
ブロッキング 4-10

ルール
(マルチルール・モデルを参照)

ろ

ロック
高レベル 9-12
シングルスレッドによるアクセス 10-35
(DLM および個々のロックの項目も参照)
性能への影響 10-35
定義 10-32
低レベル 9-12
同時 9-11
例 10-33e
割り当て 10-34
ロック値ブロック
dlm_unlock による影響 9-26
使用 9-26
定義済み 9-23
変換による影響 9-26

無効化 9-27
ローカル・バッファ・キャッシング
に使用 9-27
ロックの完了ルーチン 9-20
ロックのキュー 9-13
ロックの状態 9-13
ロック変換 9-22
定義済み 9-11
デッドロック 9-15
取り消し 9-19
ロック・ポート 8-2
ロック・モード
共存性の表 9-13
定義済み 9-11
要約 9-11
ロック要求
同期完了 9-20
非同期完了 9-19
モード 9-10
ローリング・アップグレード中のクラ
スタ・メンバの状態のテスト.. 7-6

マニュアルに対するご意見

TruCluster Server

クラスタ高可用性アプリケーション・ガイド

AA-RM88D-TE

弊社のマニュアルに関して、ご意見、ご要望、または内容の不明確な部分など、お気づきの点がございましたら、下記にご記入の上、弊社社員にお渡しくださるようお願い申し上げます。

マニュアルの採点：

	大変良い	良い	普通	良くない
正確さ (説明どおりに動作するか)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
情報量 (十分か)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
分かり易さ	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
マニュアルの構成	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
図 (役立つか)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
例 (役立つか)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
索引 (項目の検索性)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
ページ・レイアウト (情報の検索性)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

内容の不明確な部分がありましたら、以下にご記入ください：

ペ ー ジ

その他お気づきの点がございましたら、以下にご記入ください：

ご使用のソフトウェアのバージョン： _____

貴社名/部課名 _____

御名前 _____

記入日 _____

(注) 当用紙を受け取った弊社社員は、すみやかに下記にお送りください。

ビジネスクリティカルシステム統括本部 **BCS** 技術本部 **Alpha** ソフトウェア技術部