不揮発主記憶システムにおける
実行状態復元に関する研究

平成15年度

大村 廉

目 次

図 目 次

表 目 次

第1章

序論

1.1 緒論

突然の電源切断に面した時，通常の計算機システムでは，その実行状態は失われることとなる．電源切断前後で利用されるデータは,「ファイル」という形でハードディスクなどの永続記憶装置に保存され，再び読み込まれることで継続的な利用がなされてきた．一方，アプリケーションの実行状態について，突発的な電源切断前後においても継続可能であるシステムは一般的ではない．

例えば，ある人が計算機上でなんらかの作業を行っていたとする．このとき，掃除を行おうとした人が掃除機のコンセントを確保するため，利用されていないと思われたプラグを引き抜いた．もし，そのプラグが勘違いによって計算機のプラグだったとすれば，計算機上の作業中の状態が失われ，騒動となる可能性がある．もしこれがテレビであれば，プラグを抜いた人は一言謝った後もう一度プラグを挿し直せばすぐに元の番組が鑑賞できるようになる．計算機の場合には，最悪の場合数時間かけて編集していた文章や計算結果が跡形もなく失われる可能性さえあり，このような状況において，計算機がテレビのようにプラグを挿し直しただけで元の状態にすぐに復帰するようになれば，計算機の利便性が飛躍的に高まることが想像される．また情報家電と呼ばれ，計算機システムが家電製品に組み込まれるようになり，今後更なるシステムの動作の複雑化が見込まれる現状においてこのような突発的な電源切断への対応は必須であると考えられる．

計算機システムは電力で動作する以上，電力供給が存在しない場所では動作しない．電力供給の問題は計算機にとって最も重要な問題である．これに対し，現在「ユビキタス電源」と呼ばれる取り組みがなされている [河合 他 03]．ユビキタス電源とは，太陽光や風力，人力，摩擦などを利用した発電や，磁界を利用した無線電力搬送など，様々な形で計算機システムに電力を供給しようとする試みの総称である．これらは，いたるところで計算機システムが利用されようとしている将来において，その電力供給問題の解決の足掛かりになると考えられている．しかし，これらの電力は根本的に不安定であり，常に「突然電力供給が失われる」という危険が

付きまとう．このような今後の計算機システムの利用形態を考える上でも，計算機システムに対して「突発的な電源切断」に対する耐性を付与することは必要不可欠である．

そこで本研究では，計算機器の利便性，信頼性の向上のため，突然の電源切断が生じるような環境においても，利便性を損なうことなく利用可能な計算機機器の構築を目指す．より詳細に述べれば，

> 電力が再び供給されたとき，即座に電源切断直前のシステムの実行状態を回復し，ユーザが電源切断時に行っていた作業を継続的に行うことができるシステム

の構築を目指す．

1.2　本研究の目的

前述のようなシステムを実現可能とするため，本論文では突発的な電源切断に面したときにもその実行状態の復元を可能とする手法を提案する．不揮発メモリを主記憶として用いたシステムを対象として，オペレーティングシステムを含めたシステム全体の実行状態の復元手法について述べる．

実行状態の復元を行うにあたり，本研究で考慮する点は以下のとおりである．

- 低オーバヘッドでの実行状態の保存
- 周辺デバイスを含めたシステム全体の実行状態の復元

従来研究においても，主にフォールトトレランスを目的として，プログラムの実行状態の復元が取り扱われてきた．これらの研究では，プログラムの実行に関連する要素の状態を，通常実行中のある時点において保存する．このとき，従来研究では揮発な主記憶を用いたシステムを対象とし，その保存対象としてハードディスクなどの永続記憶装置が用いられてきた．このため，実行状態の保存作業は非常に時間のかかる作業となり，システム全体の実行状態を対象とするにはシステムの性能を大きく低下させるものとなった．一方，現在主記憶に利用可能な不揮発メモリの実用化がなされつつある．不揮発メモリを主記憶として用いることによって，状態保存に費やす時間の大幅な削減を見込むことができる．本研究では主記憶が不揮発であることを利用し，低オーバヘッドでのシステム全体の実行状態の保存を実現する．

また，従来研究ではそのほとんどにおいて，プログラムに関連する要素として主に CPU の状態と主記憶の状態しか考慮されてこなかった．これは，実行状態の復元を行う対象として，主にユーザレベルで動作するアプリケーションプログラムしか考慮されてこなかったためである．アプリケーションプログラムでは通常オペ

レーティングシステムによって抽象化されたデバイスを操作する．このため，周辺デバイスの状態の復元はオペレーティングシステムにまかせられた．しかし，本研究ではオペレーティングシステムやアプリケーションの再起動を必要とせず，電源切断時の状態からユーザはそのまま作業を継続させることができるシステムを目指す．つまり，本研究では周辺デバイスが持つ状態を含めたオペレーティングシステムレベルからの実行状態の復元を行う．そして，CPU や主記憶の状態に加え，周辺デバイスも含めたシステム全体の実行状態を対象とする．

これらの目的に対し，本研究はソフトウェアベースのアプローチで実行状態の復元を試みる．なお，本稿では主記憶が不揮発メモリで構成されたシステムを「不揮発主記憶システム (Non-volatile main memory system)」と呼ぶ．

1.3　本研究での提案

本研究では，以下の 2 つの手法を提案し，不揮発主記憶システムを対象として前述の目的を達成する．

- CPU と主記憶の状態を高速に保存するチェックポインティング手法
- デバイスドライバの実行制御によるデバイス状態の復元手法

本研究ではまず，CPU と主記憶について，高速な実行状態の保存および復元の手法を提案する．この手法では，主記憶が不揮発であることを利用し，保存が必要となる領域のみに保存対象を限定することによって状態保存の高速化をはかる．プログラムにより利用される主記憶領域について，オブジェクトを単位としてその保存を行う．そして，このオブジェクトの管理手法に構造化された管理手法を用いて，対象となるオブジェクトの特定を高速化する．また，主記憶が不揮発であることの弊害として CPU の主記憶に対するストア命令それぞれが主記憶上に保存されてしまうことがある．この手法では，状態保存処理中や復元処理中に電源切断が生じた場合にも適切に実行状態が復元されるようにするため，一つのストア命令で 2 つの独立な事象を切り替える手法を用いる．

次に，周辺デバイスに着目し，周辺デバイスと CPU や主記憶との状態の一貫性を維持した状態の復元手法を提案する．この手法では，デバイスとデバイスドライバの関係に着目し，デバイスドライバに処理を加えることによってこれらの間で復元される状態の一貫性を保つ．そして，実行状態の復元処理において，システム上の各デバイスドライバが対応するデバイスについて適切な状態を復元することによって，システム全体で一貫性の維持された実行状態の復元を行う．デバイスドライバに対して加えるべき処理は，デバイスとデバイスドライバの関係をメッセージパッシングシステムとしてモデル化することによって得る．メッセージパッシング

システムで議論されてきた一貫性やその維持のための方法論を応用し，その一貫性の維持のための方法を確立する．

1.4　本研究の意義

本研究の意義は以下の通りである．

まず，新しいメモリデバイスである不揮発メモリを主記憶として活用し，ソフトウェアベースのアプローチでシステムの状態復元を実現することである．従来研究においても，不揮発な主記憶を前提とした状態復元の研究は存在した．しかし，厳密に行うためには，ハードウェアの変更を必要とした．本研究では，ソフトウェアベースでこれを行うため，主記憶が不揮発であること以外にはほとんどハードウェアに対する前提を持たない．このため，本稿で提案する手法を用いれば，既存の計算機システムの主記憶を不揮発メモリに置き換えるだけで，システムに電源切断への耐性を持たせることが可能となる．

次に，デバイスが保持する状態に積極的に着目し，システム全体の状態を対象とした実行状態の復元手法を提示することである．従来研究では，主にある特定のアプリケーションが実行状態復元の対象とされ，デバイスが保持する状態にまで及んだものはほとんどなかった．しかし，計算機システム上では多種多様のアプリケーションが走行し，多種多様な周辺デバイスが利用される．システム全体の実行状態を対象としその実行状態の復元を行うためには，周辺デバイスの状態復元が必要である．本研究ではシステムに接続される周辺デバイスが保持する状態を積極的に取り上げ，その実行状態の復元を基本的な方法論を示す．

また，このとき分散システムでの議論を単一システム内の各要素に適用しその方法を検討する．一つのシステムを構成する要素をそれぞれ独立に考え，分散システムとしてみなすこの考え方は，分散システムにおける研究成果を単一システムに還元し，今後の単一システムの性能向上を促すものと考えられる．

そして，本研究で提案する手法は，計算機の耐電源切断能力を向上させ，停電などの不可避な電源切断や，冒頭で述べたような人為的なミスによる電源切断，もしくは，不安定であることを前提とする電源の利用など，今後の計算機システムの利用のされ方の幅を広げることに寄与するものである．さらに，不揮発主記憶システムを対象とし，周辺デバイスを含めた実行状態の復元をおこなうことにより得られる知見は，将来不揮発メモリが IC 上に混載され，キャッシュや周辺デバイスの状態が不揮発となった場合にもその実行状態を復元するための方法論の確立を示唆するものである．

1.5　本論文の構成

本論文の構成は以下の通りである．2 章にて，本研究での立場を明確化し，本研究で対象とするシステムについて述べる．3 章では関連研究について述べる．そして，4 章では CPU と主記憶について低オーバヘッドでの実行状態の保存と復元を行う方法を提案する．また，5 章では周辺デバイスに対象を広げ，周辺デバイスの状態を含めたシステム全体の一貫性の維持を実現する実行状態の復元手法を提案する．そして，6 章では本研究で提案する手法について議論を行い，7 章にて本論文をまとめる．

第2章

実行状態の復元

本研究の目的は，不揮発主記憶システムを対象とし，システムが突発的な電源切断に面したときにも電源切断時の実行状態を復元可能とすることである．本章では，まず，従来研究において扱われてきた実行状態の復元について，その概要を述べ，本研究での立場を明確化する．また，本研究で対象とするシステムの構成について述べる．

2.1 実行状態の復元の概要

実行状態の復元は主にフォールトトレランスを目的として行われてきた．図 2.1 にシステムの実行状態の復元を摸式化した図を示す．

各 $E_i(i = 1, 2, 3, ...)$ はシステムを構成する要素を表わす．それぞれの要素 E_i の状態が $s_i^0 \rightarrow s_i^1 \rightarrow s_i^2 \rightarrow ...$ として遷移する最中，ある時点においてそれぞれの状態を永続記憶装置上に保存する．そして，この保存した状態を後に各要素上に再構築して実行を継続する．これが実行状態の復元である．なお，実行状態を保存する作業は一般に「チェックポインティング (checkpointing)」と呼ばれ，状態を保存する時点のことは「チェックポイント (checkpoint)」と呼ばれる．また，実行状態を復元する作業は一般に「リカバリ (recovery)」と呼ばれる．

実行状態の復元に対する取り組みは，単一プロセッサシステム，分散システムともに行われてきた．そのほとんどは，ある特定のユーザプロセスを対象としたものであったが，オペレーティングシステムを含めたシステム全体の状態を対象とすることも考えられる．このことから，実行状態を復元する対象は主に以下の 3 つに分類することができる．

A. ある特定のプロセス

B. 複数のプロセスから構成されるアプリケーション (分散システム)

C. オペレーティングシステムを含むコンピュータシステム全体

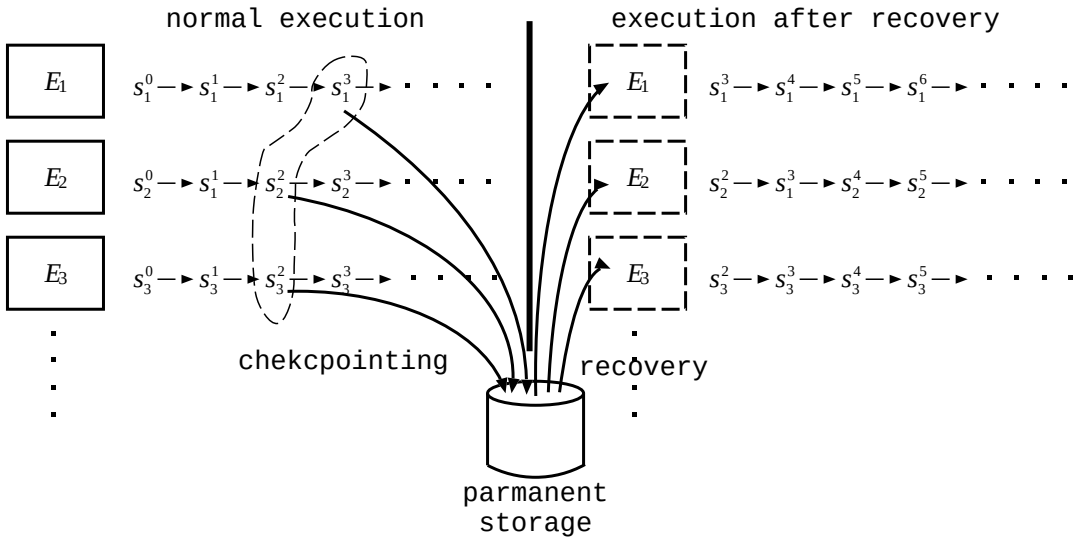

図 2.1: 実行状態復元の模式図

また，実行状態を構成する要素に着目すれば，以下の 3 種類を考えることができる．

I. CPU と主記憶

II. CPU と主記憶と周辺デバイス

III. CPU と主記憶と周辺デバイスと外界 (物理世界) の状態

従来研究において最も多く扱われてきたのは，A. と I. および B. と I. の組み合わせであった．以下，それぞれ A.I.，B.I. と表記する．A.I. はある特定のプロセスについてのみその CPU と主記憶の状態を保存し復元する．B.I. は分散システムにおいて複数のプロセスから構成される 1 つのアプリケーションの状態を復元する．B.I. はアプリケーションの構成要素となる各プロセスそれぞれに対して，A.I. での手法を適用し，一つのアプリケーションの実行状態の復元を行う．

A.I. において，その構成要素である CPU と主記憶について実行状態を復元するための最も基本的な手順は以下のようになる．

まず，チェックポインティングは次のような手順で行われる．

c.1. プログラムの実行を中断する

c.2. 停止時の CPU 状態とプロセス空間内の主記憶状態をディスクに保存する

c.3. プログラムの実行を再開する

また，リカバリの手順は次のようになる．

r.1. チェックポインティングにおいて保存された情報を読み込む

r.2. 保存された主記憶の情報を展開する

r.3. 保存された CPU の状態を復元し，実行を再開する

このような手順の中，従来研究において，実行状態の復元に関し着目されてきた点は主に以下の 2 点である．

- チェックポインティングのオーバヘッドの低減
- 分散システムにおける一貫性の維持

状態を保存する作業は，アプリケーションが本来目的とする処理とは異なるものである．また，前述の手順のように状態の保存作業中は基本的にはプログラム本来の動作は停止させる．このため，その処理にかかる時間はアプリケーションの実行性能を低下させるものとなる．そこで，できる限りこのオーバヘッドを低減するための取り組みがなされてきた．

具体的には，主記憶の保存量を低減することと見掛け上の保存時間を低減することがその主な内容である．従来研究では，CPU や主記憶状態の保存先となる永続記憶装置には，ハードディスクやテープがその対象とされてきた．これらの装置は CPU の実行に比べその入出力性能が低い．このため，一つは，永続記憶装置に対して保存する量をできる限り少なくすることでオーバヘッドの低減がなされてきた．プロセスで利用される主記憶の量はアプリケーション毎に異なるものの，多ければ数ギガバイトにも及ぶ．一方，CPU の状態は主記憶に比べわずかである．よって，特に主記憶の保存量に着目され，保存および復元が必要となる状態についてのみ保存するための手法の研究がなされてきている．また，ハードディスクやテープの入出力は CPU の実行と並列に行うことが可能である場合が多い．このため，永続記憶装置への保存作業とプログラム本来の実行を並行して行い，プログラムの実行が停止している時間をできるだけ短くするための技術が開発されてきた．

また，B.I. は主に分散システムでの取り組みであることについて述べた．分散システムでは 1 アプリケーションを構成する複数のプロセスが複数の計算機上に分散して存在し，それぞれ情報のやり取りを行いながら独立に動作する．このとき，各プロセスについて復元される状態は，一貫性が維持されたものでなければならない．

一貫性とは，簡単に述べれば「矛盾がない」ということである．例えば，あるプロセス A が別のプロセス B の処理結果を用いて動作をしている状況があるとする．このとき，リカバリ処理によってプロセス A はプロセス B の処理結果を反映した状態が復元されるとすれば，プロセス B について復元される状態はその処理結果が得られた後のものでなければならない．逆の場合には，結果が得られていない処理をプロセス A は継続することになり，これは現実には起こり得ない状態となる．つまり，矛盾が生じ，一貫性のない状態となる．

最も単純に一貫性が維持された状態を復元可能とする方法は，ある同一時刻における全プロセスの状態を一斉に保存することである．A.I. のチェックポインティング作業における c.1. の手順 (プログラムの実行の停止) は，CPU と主記憶について一貫性が維持されることを保証するための措置ということができる．しかし，分散システムではそれぞれのノードは独立に動作し，厳密な時刻の同期はほぼ不可能である．このため，それぞれのプロセスの依存関係に着目されて同一時刻とみなせる状態においてそれぞれのプロセスの状態の保存がなされる．そして，この一貫性を保証するための具体的な手法の提案が多くなされている．

2.2　本研究の進め方

2.2.1　本研究の範囲

従来研究において主に対象とされてきたのは 2.1 節で述べた分類のうち A.I. および B.I. であったことについて述べた．本研究では，C.II. の組み合わせをターゲットとする．つまり，オペレーティングシステムを含めたシステム全体の状態について，CPU，主記憶，周辺デバイスの各状態に着目し，実行状態の復元を試みる．このとき，主記憶に不揮発メモリを用いたシステムを対象とする．

従来研究においてはシステム全体を対象とした実行状態の復元はあまりなされてこなかった．なぜなら，前述の通り実行状態の復元を実現するための処理はプログラム本来の実行目的に対してオーバヘッドとなり，システム上全てのアプリケーションやオペレーティングシステムの状態までを保存して復元可能とすることは，システム性能の大幅な低下を招くことになるためである．このため，特定のプロセス，特に科学技術計算やシミュレーションなどの長時間走行するアプリケーションのみが主にその対象とされてきた．

これに対し，本研究では，オペレーティングシステムを含めたシステム全体の状態を対象とする．現在，主記憶として用いることが可能な不揮発メモリの実用化がなされてきており，この不揮発メモリを主記憶として用いることによって従来問題とされた状態保存におけるオーバヘッドの削減が容易になると考えられる．そして，システム全体を対象としてその実行状態の復元を可能としたとしても，現実的なコストでこれが実現可能になると考えられる．

また，従来研究においては周辺デバイスが保持する状態にはあまり考慮がなされてこなかった．この理由には以下の 2 つのことが考えられる．前述のとおり，一つはシステム全体の状態が対象とされてこなかったことである．そしてもう一つは，科学技術計算やシミュレーションなどのアプリケーションが主な対象とされてきたことである．ユーザレベルで動作するアプリケーションではオペレーティングシステムによって抽象化されたデバイスが扱われる．特に科学技術計算やシミュレーションなどでは，デバイスの操作が行われたとしてもファイルの読み込みや書き込

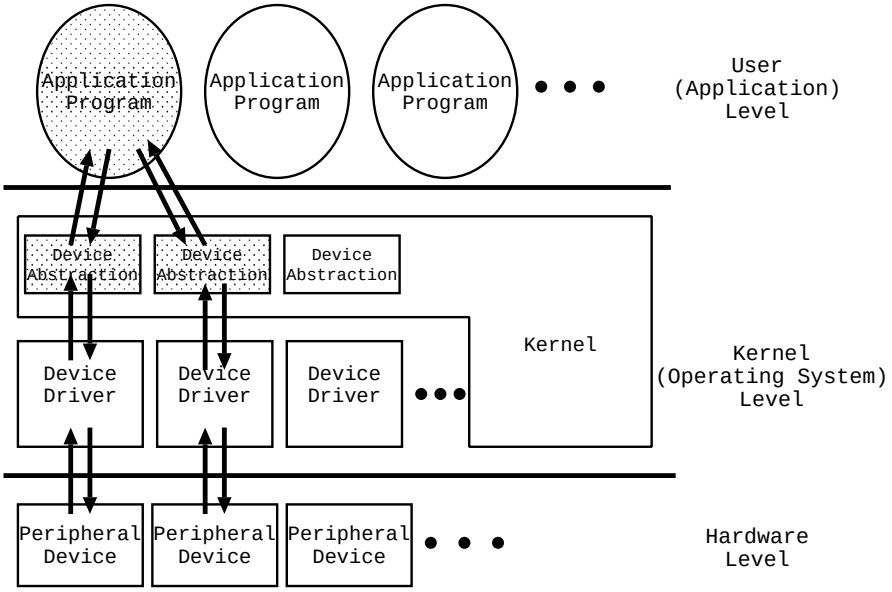

図 2.2: 従来研究における復元対象

み，端末の入出力などその操作が単純なものである場合が多い．このようなデバイスについては，デバイスが利用されることをオペレーティングシステムが再認識し，ファイル内のアクセス位置 (ポインタ) が復元されれば元通り利用が可能であった．

図 2.2 に A.I. に分類される従来研究での実行状態の復元対象を示す．図 2.2 に示すように，アプリケーションはユーザレベルで動作し，カーネルから提供される抽象化されたデバイスのオブジェクトを通してデバイスを操作する．デバイスの初期化や実際のデバイスの操作はカーネルレベルに存在するデバイスドライバによって行われる．この中で，従来研究でその復元対象とされてきたのは図 2.2 中網がけで示した部分，つまりユーザレベルのアプリケーションの状態とカーネル内の一部であった．

一方，本研究では，デバイスが保持する状態もその復元対象とする．オペレーティングシステムの実行はデバイスの状態と密接に関連する．例えばデバイスに対して何らかのコマンドを発行し，その結果を待っている処理の状態がそのまま復元されたとすれば，デバイスはリカバリ後そのコマンドの内容を継続し，結果を返さなければならない．そうでなければ，この処理は永遠に結果を待ち続けることになってしまう．このため，デバイスが持つ状態について，CPU や主記憶と同様にその状態を復元する必要がある．つまり，本研究では図 2.2 内に示される全ての要素を対象とし，その実行状態の復元を行う．

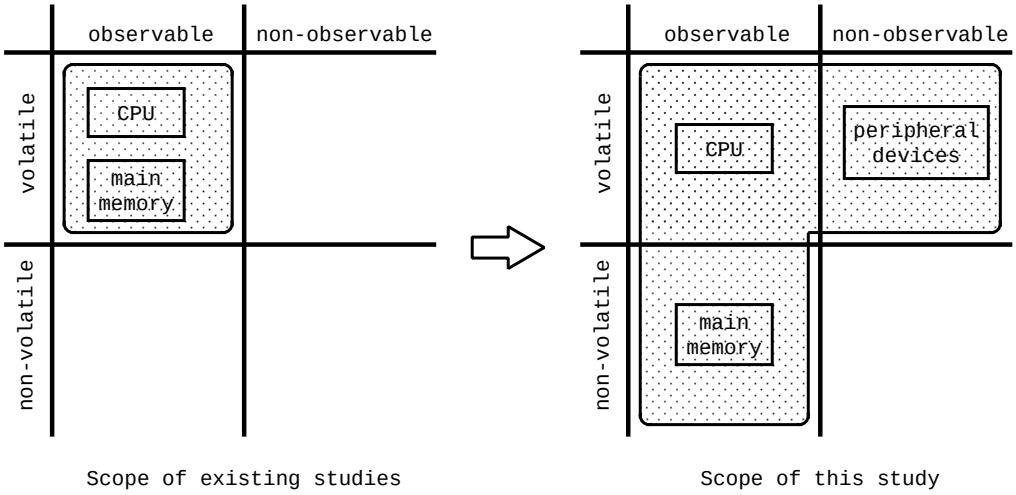

図 2.3: 従来研究と本研究の範囲の比較

2.2.2　対象とする要素の特徴

本研究は，主記憶を不揮発メモリで構成したシステムを対象とする．そしてデバイスの状態をを含め実行状態の復元を試みる．また，ソフトウェアベースのアプローチでこれを行う．

CPU の状態および主記憶の状態は任意の時点で明示的に取得することが可能である．一方デバイスの状態は，特に動作中などその状態を取得できない場合が多い．本稿では，任意の時点でその状態が取得可能な要素を「可観測要素 (observable element)」と呼ぶことにする．一方，状態が取得可能でない要素は「不可観測要素 (non-observable element)」と呼ぶことにする．つまり，CPU および主記憶は可観測要素であり，周辺デバイスは不可観測要素である．

また，CPU の状態や，従来研究における主記憶の状態は電源切断とともにその状態は失われた．一方，本研究で扱う主記憶は不揮発であり，リカバリ時にも電源切断時の状態がそのまま維持される．本稿では，これらを「揮発性要素 (volatile element)」「不揮発性要素 (non-volatile element)」として分類することにする．

以上の基準によって分類すれば，従来研究で対象とされたのは CPU と主記憶に保持されるアプリケーションの状態であり，それぞれは揮発性要素とみなされた．つまり，図 2.3 の左側に示すように，従来研究で扱われたのは可観測かつ揮発な要素についてのみであった．一方，本研究で扱う対象は不揮発主記憶システムであり，周辺デバイスが持つ状態を含める．CPU は可観測かつ揮発，主記憶は可観測かつ不揮発，周辺デバイスは不可観測かつ揮発な要素となるため，本研究で扱う対象は図 2.3 の右側に示されたものとなる．換言すれば，本研究はこれら可観測かつ不揮発である要素や，不可観測かつ揮発である要素に対して，その実行状態の復元方法

を提案するものである．

なお，システムの実行が外部世界の状態と密接に関連する場合もある．例えば，センサやアクチュエータを用い，物理世界に対して影響を及ぼす計算機システムも多く存在する．しかしながら，この問題は時間を含む物理的な状態の復元を取り扱う問題となる．このため，本研究では対象外とする．なお，外部世界の状態は，図 2.3 において，不可観測かつ不揮発 (図 2.3 中右下の枠内) の分類される．デバイスの外部世界に対する影響は 6 章で議論する．

2.2.3　本研究の方針

本研究では以下の手順で研究をすすめる．

1. CPU と主記憶の高速なチェックポインティング／リカバリ手法
2. デバイスアクセスを単位とし，CPU や主記憶との一貫性を維持した周辺デバイスの状態復元方法

まず，可観測な要素である CPU と主記憶について，その実行状態の復元手法を確立する．具体的には，従来研究と同様，ある時点において CPU と主記憶の状態を保存し復元することになるが，このとき従来研究においてなされてきた保存対象となる主記憶量の低減方法を参考にて状態保存 (チェックポインティング) の高速化をはかる．また，チェックポインティング中やリカバリ処理中における電源切断を考慮して，具体的なチェックポインティングの手順やリカバリの手順について検討する．そして，CPU の状態と主記憶の状態の一貫性を常に保ち，実行状態を復元可能とする手法を確立する．

次に，不可観測かつ揮発な要素である周辺デバイスへ対象を拡張し，CPU，主記憶，周辺デバイスの間で実行状態の復元手法を確立する．このとき，デバイスに対するアクセスを単位とし，これらの状態の一貫性の維持をはかる．図 2.2 に示したように，デバイスはデバイスドライバによって制御される．このため，特にデバイスドライバに着目し，デバイスドライバとデバイスとの間の関係に着目する．デバイスとデバイスドライバの関係においてそれぞれは独立に動作するため，これらの関係は一つの分散システムとして捕らえることができる．具体的にはメッセージパッシングシステムにおける一貫性の議論を適用し，デバイスとデバイスドライバの間で一貫性を維持した状態の復元方法を確立する．そして，その方法を各デバイスドライバに適用することによってシステム全体の実行状態の復元を実現する．

つまり，本研究では，まず可観測要素かつ揮発および可観測かつ不揮発な要素について，低オーバヘッドでの実行状態復元方法を確立する．そして，次に不可観測要素かつ揮発な要素についてその実行状態復元方法の拡張を行い，システム全体の実行状態の復元を実現する．

2.2.4　本研究における課題

前述のように，不揮発メモリを主記憶として用いることによって明示的な保存作業は排除可能となる．一方，CPU の状態は電源切断とともに失われるため，CPU の状態は明示的に保存し復元されなければならない．このことから，復元される CPU 状態との一貫性を保つため，不揮発主記憶システムにおいても主記憶領域を明示的に保存する必要が生じる．

従来研究では主記憶は揮発であったためチェックポイント後のプログラムの実行によって変化した主記憶状態は，単に故障とともに失われることとなった．また，永続記憶装置への保存は明示的なコマンド発行によって行われるため，チェックポインティング後のプログラムの実行によってチェックポインティングで保存された情報に影響を及ぼすことはなかった．一方，不揮発主記憶システムでは，チェックポイント後の通常実行によって変更される状態も主記憶に残されることとなる．このため，チェックポイント後の主記憶に対する変更を無効化することができるようにしなければならない．

さらに，不揮発主記憶システムでは，チェックポインティングやリカバリ作業での主記憶の更新も電源切断後に残されてしまうことになる．このとき，主記憶に対してなされるストア命令一つ一つが保存されることなるため，CPU のインストラクション単位で，主記憶に残される状態について考慮しなければならないことになる．特に，チェックポインティングやリカバリ処理によって変更される主記憶の状態について，後のリカバリ処理において適切に実行状態を復元可能となるように考慮する必要がある．

また，本研究では，周辺デバイスの状態も含めた実行状態の復元を行う．このとき，復元されるデバイスの状態は，CPU や主記憶の状態と一貫性が維持されていなければならない．しかしながら，デバイスは不可観測な要素であり，特に動作中の周辺デバイスについてその任意の時点の状態を取得することは困難である場合が多い．このため，チェックポインティング手法によってデバイスの実行状態の復元可能とすることは困難である．このことから，周辺デバイスについてはその復元可能な状態を基準にし，CPU や主記憶について復元される状態を調整を行するようにしなければならない．また，前述の議論と同様に不揮発主記憶システムでは主記憶には一つ一つのアクセスの結果が残されるため，CPU によって発行するデバイスアクセスや，デバイスからの割り込みや主記憶へのアクセスを単位として，その一貫性を考慮する必要がある．

2.3　想定するシステムアーキテクチャ

本研究では，実行状態を構成する要素として以下の 3 つを取り上げることについて述べた．

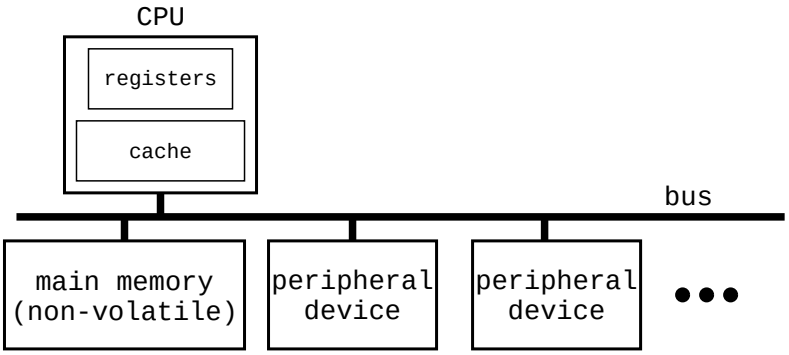

図 2.4: 想定するシステムアーキテクチャ

- CPU
- 主記憶 (main memory)
- 周辺デバイス (peripheral devices)

本研究では対象となるシステムの具体的なイメージとして PDA(Personal Digital Assistant) を想定する．PDA は，現在携帯型の計算機システムとして多くの利用がなされているものである．このようなデバイスは，太陽電池などの不安定な電力源でその電力供給がなされることによって，現在の卓上計算機に見られるように「いつでも，どこでも利用可能となる」ということが電池の充電状態に無関係に実現され，その利用価値を向上させることが予想されるためである．

そして，問題を単純化するため，本研究では図 2.4 に示すような単純な構成のシステムアーキテクチャを想定する．以下，それぞれの要素についての本研究での想定するアーキテクチャについて述べる．

2.3.1　CPU

本研究では，CPU について組み込み機器などに利用される一般的な CPU を想定する．RISC や CISC といったアーキテクチャの種別は問わない．また，キャッシュはライト・スルー (write-through) 方式，ライト・バック (write-back) 方式を問わないが，ライト・バック方式の場合には，CPU の命令によって意図的に主記憶に対するフラッシュが可能なものとする．

CPU のレジスタやキャッシュの内容は，電源切断とともに失われるものとする．また，単一プロセッサシステムとし，SMP(Symmetric Multiprocessor) システムなどのマルチプロセッサシステムは想定しない．

2.3.2　主記憶

主記憶については，不揮発メモリで構成されることを前提とする．現在，MRAM や FeRAM などの不揮発メモリの実用化がなされてきている．これらの不揮発メモリは，通常の主記憶として利用されるメモリと同様のアクセス方法でアクセス可能であり，かつ予備電源を用いることなく不揮発性を保持する．

以前から SRAM に予備電源を付加し，不揮発性を実現したメモリが存在する．しかし，予備電源を必要とすることや，容量や価格の面から主記憶として用いられることはあまりなかった．また，EEPROM や FlashROM も同様に不揮発性を持つ電気的に書き換え可能なメモリであるが，書き換えの際には，一度明示的な内容の消去が必要があり，また，書き換え可能回数にも制限があった．FlashROM を主記憶に用いるための研究も行われているが，MMU(メモリ管理ユニット) の特別な制御を必要とした [甲斐 他 01]．

これらに対して，FeRAM や MRAM などの不揮発メモリは，SRAM と同様の方法でのアクセスが可能であり，容量や速度，書き換え可能回数などの面からも主記憶として十分に利用可能なものである．表 2.1 に主な半導体メモリの性能比較を示す [松本 99, 大石 他 01, 田原 他 04]．表 2.1 からもわかるように特に MRAM はその速度 (読み出し，書き換え時間) に関して，現行の計算機システムで主記憶として用いられている DRAM の性能と比べて遜色がない．また，書き換え可能回数や消費電力，さらには集積率も遜色のないものであると言える．これらのことから，将来のメモリは全て MRAM や FeRAM はなどの不揮発メモリによって置き換えられる，との予測も存在する．よって，将来計算機システムの主記憶が，これらの不揮発メモリで構成されることは十分に予測されることである．

2.3.3　周辺デバイス

周辺デバイスについては，本研究では CPU からその状態を直接制御可能であるものを想定する．

接続方式としては，図 2.4 に示すように，CPU と直接接続されたデバイスを対象とする．問題の単純化のため，PCI や USB バスなどのバスコントローラを介して接続されるデバイスは本研究では対象外とする．このようなバスを用いた場合，デバイスの状態を復帰するためには，まずバスコントローラの状態を復元し，さらにデバイスの状態を復元する必要がある．また，バスコントローラ上のトランザクションにまで留意する必要が生じる．本研究ではまずデバイスの状態復元に関する本質的な問題を明らかにし，その基本的な方法を確立するため，CPU から周辺デバイスの全ての状態を制御可能である，という前提で研究を進める．

また，本研究で対象とする CPU は，主に組み込みシステムなどで用いられる CPU であることについて述べた．これらの CPU では，通常 CPU の IC 上に予め UART

表 2.1: メモリデバイスの性能比較 (「情報処理」Vol.45 No.1 pp.43 より)

	MRAM		FeRAM	NAND Flash	DRAM
	Type A*	TypeB†			
不揮発性	○		○	○	×
書き込み速度 (ns)	10		20〜30	200,000 (par page)	50
読み出し速度 (ns)	500	50	20〜30	Random Access 4,000	50
セルサイズ (DRAM との相対値)	0.6	1.3	2	0.6	1
書き換え耐性	No-limitation		10^{12}	10^{6}	No-limitation
消費電力 (mW)	100〜400		〜10	100	400
動作電圧 (V)	1 以下		2.5/1.2	12	2.5

*クロスポイント型
†選択トランジスタ型

や Ethernet，LCD コントローラといったある程度の周辺デバイスが存在する場合が多い．これらは，厳密には主記憶バス上に接続されるものではないが，CPU からその状態を直接制御可能である．本研究では，このような周辺デバイスも対象とする．

なお，周辺デバイスはシステムの電源切断とともにその活動は停止し，状態は失われるものとする．そして，電源切断とともにデバイスに故障が発生 (破壊される) 場合は考慮しない．また，周辺デバイスは CPU により発行される動作要求に対して 1 対 1 の関係で動作を開始するものとし，デバイス側でバッファリングを行って独自にその動作を続けることはないものする．同時に，デバイスは互いに独立に動作し，デバイス同士でその状態に影響を与えることはないものとする．そして，デバイスが動作していない期間 (idle 状態) での状態は適切な再初期化処理によって復元可能であるとする．

まずはこのような最も単純なデバイスを想定し，研究を進める．そして，6 章にて本稿で提案する手法の有効性について，適用可能なデバイスについて議論する．

2.4　本章のまとめ

本章では，実行状態の復元の基本的な手法について述べ，本研究での立場について述べた．また，本研究で対象とするシステムの概要について述べた．

本研究では，主記憶が不揮発であることを前提とし，周辺デバイスの状態を含めたシステムの実行状態の復元を行う．まず，CPU と主記憶の状態について，高速

なチェックポインティングの手法を確立する．このとき，チェックポインティング中や，リカバリ中の電源切断についても考慮する．次に，デバイスの状態について対象を広げ，この状態を復元可能とする．このとき，CPU や主記憶との一貫性の維持について考慮する．

次章にてこれらに関連する研究について述べ，4 章より具体的な手法の提案を行う．

第3章

関連研究

本章では，本研究の関連研究について述べる．本研究の目標は不揮発主記憶システムにおいてオペレーティングシステムレベルからシステムの実行状態を復元することである．また，具体的な目的は，低オーバヘッドでのチェックポイティング手法の確立と，周辺デバイスを含めた実行状態の復元である．まず，チェックポインティングの高速化手法についての取り組みについて述べ，次に既存のチェックポインティング手法におけるデバイス状態の取り扱いについて述べる．また，システムの消費電力を目的とし，システムの電源切断前後でデバイス状態の復帰を行なっているシステムの電源管理手法について述べる．

3.1 チェックポインティングオーバヘッド

3.1.1 オーバヘッドの改善

2.1 節で従来研究において，チェックポインティングオーバヘッドの低減に対する試みがなされてきたことについて述べた．また，2.2.3 節で，本研究の課題の一つは，高速なチェックポインティングを実現することであることを述べた．

その具体的な手法の一つは，保存対象となる主記憶の量を減らしチェックポインティングそのものに要する時間を低減させることである．そしてもう一つは，見掛け上のチェックポインティングによるプログラムの実行停止時間を低減させることである．それぞれ，チェックポインティングサイズの低減技術とチェックポインティング遅延の隠蔽技術として知られる [Plank *et al.* 99]．以下，それぞれの手法について，概要を述べる．

チェックポインティングサイズの低減

2.1 節で述べたように，チェックポインティングの最も単純な手法は，ある瞬間のプログラムに属する主記憶領域の状態を全て一度に永続記憶上に保存する方法で

ある．しかし，現行のシステムの多くはコード領域の変更が許されないシステムが多い．このため，コード領域はプログラムの実行ファイルから再構築し，チェックポインティング時に主記憶について保存される領域を，スタックを含めたデータ領域のみとすることができる．

そして，より主記憶状態の保存量を削減するため,「インクリメンタル・チェックポインティング」と呼ばれる手法が用いられる．直前のチェックポイントから変更されていれない情報は，過去のチェックポイントにおいて保存されたものと同一であるため，過去のチェックポイントでの情報を用いてその状態を復元することが可能である．インクリメンタル・チェックポインティングでは，直前のチェックポイントから更新のあった領域についてのみ保存を行なう手法である．このようにして，チェックポインティングにおける主記憶の保存量を減らし，チェックポインティングにかかる時間を低減する．

この手法は，主に MMU(Memory Management Unit) を用いてページアクセス情報を制御することによってプログラムに対して透過に行われる．チェックポイント毎にプログラムに関連するページについて，ページテーブルのアクセス情報を書き込み不可に設定しておく．プログラムにより書き込みがあった場合には，アクセスバイオレーショントラップ内でそのページに書き込みが行われたことを記録する．そして，そのページを書き込み可の状態にしてプログラムに実行を戻す．次のチェックポイントでは，これら記録されたページに関してのみその状態を保存する．このようにして，MMU のページ単位で更新された領域を特定する．また，ユーザレベルで行われる場合には，`mrprotect` システムコールを用いることで，同様にプログラムに対して透過に更新された領域の特定が実現できる．

この手法の問題点は，少なからず復元処理が複雑化することと，システム上に複数のチェックポイントでの状態を保持しておかなければならないことである．あるチェックポイントにおいて保存されなかった領域の状態は，同一領域について以前のチェックポイントで保存された情報から構築する必要がある．このため，システム上には過去のチェックポイントで保存された情報も残しておかなければならない．また，これら複数存在する情報から復元すべき情報を探索する必要も生じる．

チェックポインティング遅延の隠蔽

また，見掛け上のチェックポインティングにかかる時間を低減させる試みも行われている．これは，チェックポインティングの作業とプログラムの実行とを並行して行うことでなされる．保存される情報は，その完了まで有効な保存状態とはならないが，プログラムの本来の処理の停止時間を短縮し，プログラムの実行性能の低下を防ぐ．これらはチェックポインティング遅延 (レイテンシ) の隠蔽技術として知られる．

ハードディスクなどの永続記憶装置への書き込み操作は CPU の実行とは並列に

行なうことが可能である場合が多い．このため，永続記憶装置への書き込み処理中に，CPU の処理をチェックポインティングからプログラムに戻して実行を継続させ，プログラムの停止時間を短縮する．

しかし，保存対象となる領域がディスクへ保存される前にチェックポインティング開始時の状態から変更されてしまった場合，この領域について保存される状態は他の保存された状態との整合性を保てなくなる．このため，システムレベルでは，主に Copy-on-Write 技術を用いた手法が使われる．まず，MMU を用いて，ページに対するアクセスを読み込みのみに制限する．あるページに対する書き込みが生じたときには，アクセスバイオレーショントラップの処理内で，そのページのコピーを作成する．そして，物理アドレスと仮想アドレスのマッピングを変更し，同じ仮想アドレスに対するアクセスをコピーされた物理メモリ領域へのものとなるよう変更する．この後，処理を戻してプログラムの実行を継続させ，チェックポインティングでは元の物理ページの保存作業を行う．このようにして，チェックポインテングで保存対象となる主記憶領域の変更を抑制する [Elnozahy *et al.* 92]．

ユーザレベルで行われる方法としては，`fork` システムコールを用いたチェックポインティングが良く知られている．`fork` システムコールはあるプロセスのコピーを作成するシステムコールである．プロセスのチェックポインティング時，`fork` システムコールを用いてプロセスのコピーを作成する．そして，コピーされたプロセス (子プロセス) の中で状態の保存を行う．各プロセスは並行に実行されるため，プログラムは本来の実行を継続することができる．見掛けの上のチェックポインティングに要する時間は，`fork` システムコールに要する時間のみとなり，プロセスの実行はチェックポインティングにほとんど阻害されることなく，その状態の保存が行われる．

3.1.2　チェックポインティングの数学的な解析

前述のようなオーバヘッドの低減技術を用いたとしても，チェックポインティングはプログラムの本来の実行に対して多少なりともその実行性能を低下させる．チェックポイントは，システムの状態を復元するポイントとなるため，短い時間間隔であった方が故障発生時に失われる処理量が少なくてすむ．これは同時に，故障が生じた場合のプログラムの実行時間を短くすることに繋がる．しかし，あまりに短い時間間隔でチェックポイントを挿入した場合，オーバヘッドを増大させシステムの実行性能を低下させる．つまり，故障によって失われる処理の量とチェックポインティングのオーバヘッドはトレードオフの関係にある．この問題を数学的に解析し，チェックポインティングによるオーバヘッドの見積もりや適切なチェックポイント挿入間隔を決定するための研究が行われている [Vaidya 97, Plank *et al.* 98, Ling *et al.* 01]．

Vaidya はパラメータとして主に，

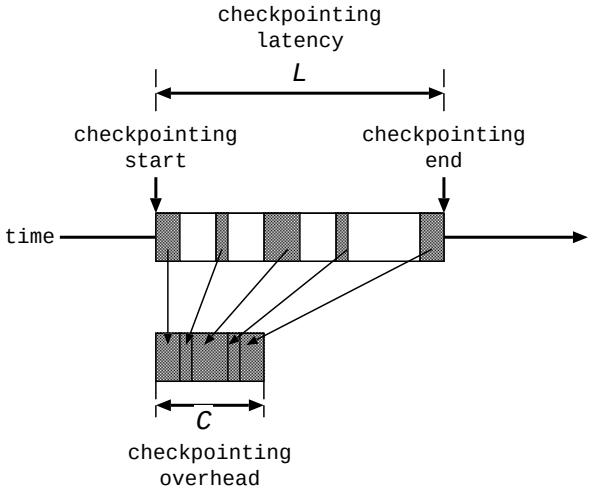

図 3.1: チェックポインティングのモデル

- チェックポインティング・オーバヘッド
- チェックポインティング・レイテンシ
- チェックポイント間隔
- リカバリにかかる時間

を取り上げ解析を行い，チェックポインティングにおけるオーバヘッドはチェックポインティングレイテンシにほぼ無関係であることを示した．

図 3.1 に，Vaidya が解析に用いたチェックポインティングのモデルを示す．図 3.1 は，左から右に時間の流れを表わしている．また，チェックポインティングの作業は四角で表わされており，このうち網がけ部分がチェックポインティングのために費やされる処理を表わしている．また，このモデルは，遅延の隠蔽技術が用いられることが考慮されており，チェックポインティング中にも本来のプログラムが走行することが考慮されている．図 3.1 中白い四角部分がチェックポインティング中，本来のプログラムが走行することを表わしている．そして，チェックポインティングが開始されてから状態の保存が終了 (チェックポインティングが完了) するまでの時間はチェックポインティング・レイテンシ L としてパラメータ化される．また，このチェックポインティング中，チェックポインティングのために費やされる計算時間の総和がチェックポインティング・オーバヘッド C としてパラメータ化される．チェックポインティング中，通常のプログラムの処理がなされる時間はチェックポインティング・レイテンシとチェックポインティング・オーバヘッドの差 $L - C$ で表わされる．

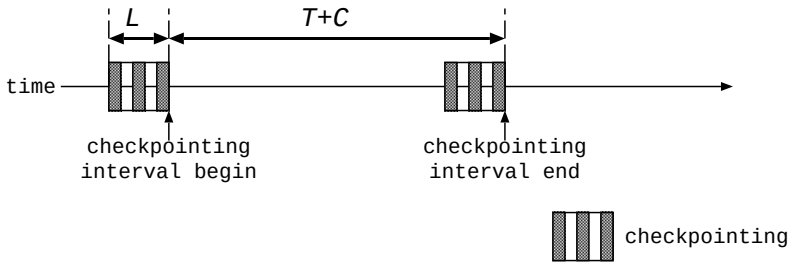

図 3.2: チェックポインティングを伴うプログラム実行のモデル

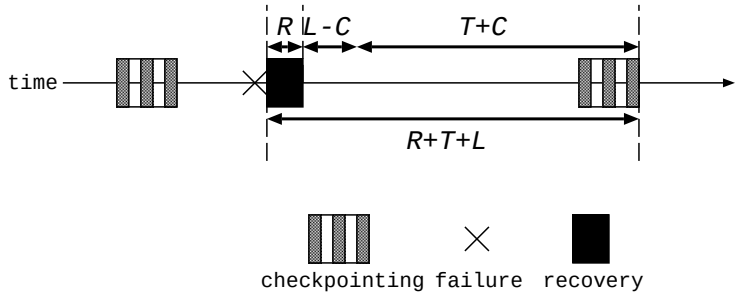

図 3.3: 故障発生時のモデル

図 3.2 にチェックポインティングを伴ったプログラムの実行のモデルを示す．また，図 3.3 には，故障およびその故障に対するリカバリ処理を含む実行モデルを示す．チェックポイント間隔は 2 つのチェックポインティングが完了する間隔として定義される．また，図 3.2 中 T はチェックポイント間隔内でプログラムがその本来の処理を行うことができる時間を示しており，チェックポイント間隔は $T+C$ として定義される．

図 3.3 に示すように，プログラムの実行中に故障が生じた場合，リカバリ処理を行った後，直前のチェックポインティングから実行が再開される．このときリカバリにかかる時間は R としてパラメータ化される．また，リカバリ処理後，実行が再開されるのは直前のチェックポイントでの最初の状態である．このため，リカバリから次のチェックポイント間隔までの時間はリカバリ時間 R およびチェックポイント間隔 $T+C$ に，チェックポイント中行われた処理時間 $L-C$ を加えた $R+T+L$ として表される．

これらのモデルをもとに，チェックポインティングおよび故障発生時に失われる実行分の時間やリカバリ，再実行処理によって増加する実行時間の増加分を「オーバヘッド率」として定義している．また，プログラムの実行を故障が発生しない場合と故障が発生した場合を状態とするマルコフ連鎖としてモデル化し，故障の発生確率をポアソン過程としてオーバヘッド率を求める式を導出している．そして，

オーバヘッド率を最小化するための条件として次式 (3.1) を導き出している．

$$\frac{\partial}{\partial T}\left(\frac{e^{\lambda(T-C)}-1}{T}\right) = 0 \tag{3.1}$$

式 (3.1) はチェックポインティング・レイテンシ L に無関係である．この結果から，チェックポインティングのオーバヘッドを低減するにはチェックポインティングの処理に費やす時間を低減させることが重要であることが示されている．

3.2　チェックポインティング高速化の具体的な手法

本研究の目的の一つは，不揮発である主記憶を利用し，高速なチェックポインティングを実現することである．以下，本研究で有効と思われる，チェックポインティングのオーバヘッドの低減方法について述べる．

3.2.1　キャッシュベースのチェックポインティング

2.2.4 節において，不揮発主記憶システムでは，CPU の状態保存後プログラムの通常実行により変更される主記憶領域について考慮しなければならないことについて述べた．これに対し CPU のキャッシュを用いて，チェックポイント間の変更を主記憶に対して反映させないことが考えられる．CPU のキャッシュを用いたチェックポインティングの研究もなされており [Bowen *et al.* 93]，これらは一般に CARER(Cache-Aided Rollback Error Recovery) と呼ばれる．

これらの手法では，CPU のキャッシュをライトバック方式で利用し，チェックポイントでの状態を主記憶上に保持させる．通常の実行はキャッシュ内のみで行われるようにし，チェックポインティングは，CPU の状態の保存と，dirty なキャッシュラインをフラッシュすることによって行われる．一方，リカバリ作業は，CPU のキャッシュを全て invalid とし，保存された CPU の状態を復元することによって行われる．その後の実行によって，CPU のキャッシュには自動的にチェックポイントでの主記憶状態が読み込まれ，チェックポイントでの主記憶状態を用いて実行が継続されることになる．

キャッシュベースのチェックポインティングでは，プログラムによって明示的に行われるチェックポインティングのほか，キャッシュの入れ替えによって 1 ラインでもキャッシュラインがフラッシュされるときもチェックポイントとしなければならない．このとき，前述のように dirty なキャッシュラインは全てフラッシュする必要があり，また，キャッシュのフラッシュ中にキャッシュの状態が変更されないように，CPU の動作を停止させる必要がある．さらに，同時に CPU の状態も保存しなければならないため，これらの理由からオーバヘッドは予想以上に増大する．

また，キャッシュのフラッシュにはある程度の時間を要するため，厳密にはキャッシュのフラッシュ中の故障も考慮しなければならない．

Hunt が提案する手法では，オーバヘッドを低減するためチェックポイントにおいて CPU の状態を CPU 内部に存在するバックアップ領域に保存するようにしている [Hunt *et al.* 87]．また，キャッシュの状態として”unchangeable” な状態を新たに設け，チェックポインティングにおいて dirty なキャッシュライン全てをフラッシュする必要性を排除している．この手法では，キャッシュ内のあるラインのフラッシュが行われるとき，他の dirty なキャッシュラインを unchangeable にする．unchangeable なキャッシュラインが更新されようとするとき，一度そのラインの内容をフラッシュして clean な状態 (主記憶と同一の状態) にし，そして実行を継続するようにする．このようにすれば，リカバリ時には，unchangeable なラインと主記憶の内容を用いて実行を再開すれば良いことになる．これによって，ある 1 つのキャッシュラインによって生じる全ての dirty なキャッシュラインのフラッシュを防ぎ，CPU の停止時間を短縮している．

この手法はチェックポインティングを高速に実現するが，プロセッサ自体の故障や電源切断への対応を考慮したものではない．CPU 内にそのチェックポイント時の状態が保持されるため，CPU 内部の状態も不揮発である必要が生じる．また，キャッシュラインのフラッシュ中に故障が生じた場合については考慮されていない．

Sequoia システムでは，CPU の状態の保存およびキャッシュラインのフラッシュを 2 回行うことによって，キャッシュのフラッシュ中の故障に対応する [Bernstein 88]．Sequoia システムでは，書き込み可能な主記憶領域はプライマリ領域とバックアップ領域の 2 つに 2 重化される．まず，キャッシュや CPU の状態はバックアップ領域にフラッシュされる．次に，同じ内容がプライマリ領域にフラッシュされる．バックアップ領域にフラッシュしている最中に故障が生じた場合には，プライマリに存在する以前のチェックポイントでの情報がリカバリに用いられる．逆に，プライマリ領域にフラッシュしている最中に故障が生じた場合には，バックアップ領域に存在する情報がリカバリに用いられる．

どちらの状態をリカバリに用いることができるか，ということを判別可能とするため，Sequoia システムでは二つのロック変数が用いられる．バックアップ領域にフラッシュを開始するとき，1 番目のロックを獲得する．次にプライマリ領域にフラッシュを行うとき，2 番目のロックを獲得する．このようにすれば，リカバリ時に 1 番目のロックが獲得されており，2 番目のロックが獲得されていないときにはバックアップ領域のフラッシュ中に故障が生じたこととなる．このため，リカバリ時にはプライマリ領域を用いて実行を再開すれば良い．また，1 番目のロックおよび 2 番目のロック両方とも獲得されている場合には，プライマリ領域のフラッシュ中に故障が生じたこととなり，バックアップ領域に存在する情報を用いて実行を再開すれば良い．なお，実行を再開するときには後に続く故障に対応するため，再実行のために用いられる領域をもう一方の領域にコピーする．

この手法は，キャッシュのフラッシュを 2 度行うためシステム性能を低下させるという問題があるものの，チェックポイントでの完全な状態が常に主記憶に存在することが保証できる．同様の手法を用いて，不揮発メモリを前提とするトランザクションを対象としたシステムも存在する [Banatre *et al.* 90, Banatre *et al.* 91]．しかしながら，この手法ではキャッシュのフラッシュを 2 度行うための機構，フラッシュの対象となる主記憶領域を切り替える機構，そして，それぞれのキャッシュのフラッシュ直前にロックの獲得を行うなど何らかの処理を行うための機構が必要となる．これを実現するためには，キャッシュを制御するハードウェアのサポートが必要となる．

不揮発主記憶システムでは，キャッシュベースのチェックポインティングを用いて低オーバヘッドでチェックポインティングが実現される可能性をもつ．しかし，これらの手法に見られるように特別な CPU を用いなければならない．現在の CPU に用いられているキャッシュは，そのほとんどが CPU に対して透過にフラッシュされる．このため，ソフトウェアのみで上記のような対応を行うことは困難である．

3.2.2　保存する主記憶量の低減

インクリメンタル・チェックポインティング

3.1.1 節では，チェックポインティング時に保存する主記憶量を低減することによってチェックポインティングのオーバヘッドが低減されてきたことについて述べた．また，主な手法はインクリメンタル・チェックポインティングであることについて述べた．

インクリメンタル・チェックポインティングは，その保存対象の特定の粒度から，さらに下記の 3 つに分類することができる．

- MMU のページ単位で行う
- CPU のワード単位で行う
- ブロック単位で行う

3.1.1 節で述べたように，MMU を用いてページ単位で更新された領域を特定可能とする手法はプログラムに対して透過に実現可能であるため，実システムでは多く用いられている．しかし，この方法ではページ内でアクセスされる領域が数バイト程度であったとしても，ページ単位での認識しかできない．このため，数バイト程度の更新がページに対して分散する場合，多くの不必要な領域が保存対象となる．つまり，変更された領域に対して不適当な保存量や時間が費やされる可能性がある．

これに対し，Plank らはワード単位で変更を特定する手法を提案している [Plank *et al.* 95c]．Plank らの手法では更新前後の主記憶状態の比較を行う．ページが更新されるときにそのページの内容をコピーし，更新前の状態を保存しておく．チェックポインティングでは，これらの比較を行って保存対象となる領域を特定する．この手法を用いた実験からは，多くの場合保存が必用となる量が低減し，オーバヘッドが低減することが示されている．

また，Nam らは数バイト程度のオブジェクト毎にチェックポイント間で変更された領域を特定する手法を提案している [Nam *et al.* 97]．この手法では，メモリ空間を数十から数百バイト毎のブロックに分け，各ブロックに対してキーとなる値を計算する．このキーの値をチェックポインティング毎に計算し，キーの値が変化したものは更新されたオブジェクトと認識し，保存の対象とする．この手法を様々なプログラムに対して適用した実験の結果からは，4Kbyte のページサイズをベースとした手法と比較して最大 11.7%のパフォーマンスの改善が見られている．

しかし，Plank らの手法におけるページのコピーとその比較や，Nam らの手法におけるキーの計算は，保存対象がハードディスクなどの入出力性能の低い永続記憶装置を対象として始めて有効となる手法である．これらの処理に費やす時間が保存に費やす時間の減少量よりも少なくなければ，逆にチェックポインティングのオーバヘッドを増す結果となる．

また，これらの処理は保存対象となるオブジェクトをプログラムに対して透過に特定するために用いられており，直接保存対象となるオブジェクトが特定できれば必要のない処理である．

明示的な保存対象の指定

また，Plank らは，より主記憶の保存量を低減させるための手法として，プログラマが明示的に保存対象となる領域の指定とチェックポイントを指定することを提案している [Plank *et al.* 99]．例えば，ある期間内でのみ一時的に利用されるメモリ領域を保存対象外とし，この領域外にチェックポイントを挿入する．保存対象とする領域の指定やチェックポイントの挿入は，プログラマが十分にそのプログラムの性質を理解した上で行われるため非常に効率の良いものとなる．

Plank らは，これを実現するために次の 3 つの API を定義している．

- `checkpoint_here()`
- `exclude_bytes(char *addr, int size, int usage)`
- `include_bytes(char *addr, int size)`

`checkpoint_here` では，強制的にチェックポインティングを行う．`exclude_byte` は保存が不必要となるメモリ領域を指定する．引数 `addr` と `size` には，対象とな

るメモリ領域のアドレスとサイズが渡される．このとき，引数となる usage には READONLY か DEAD が指定される．READONLY は，プログラムから参照のみ行われる領域に用いられる．次のチェックポインティングにおいて保存はなされるが，それ以降のチェックポインティングでは保存の対象外となる．DEAD は，以降更新される前に参照されることのない領域に対して用いられ，以降のチェックポイントでは保存されなくなる．include_byte は以降のチェックポインティングにおいて保存対象となるメモリ領域を指定する．

プログラマはプログラムに対し明示的にこれらのコードを埋め込む必要があるが，Plank は様々なプログラムに対して適用した結果，さほどプログラマの労力を要することなくオーバヘッドの改善がなされたと結論づけている．また，Silva らは，プログラムに透過に行う場合と非透過に行う場合について，プログラマの労力，チェックポイント時に保存されるメモリ量，チェックポイント挿入に関する柔軟性などの観点から，ユーザに透過に行うべきか，非透過に行うべきか，について議論を行っている [Silva *et al.* 98]．そして，プログラマの労力は存在するものの，チェックポインティングはユーザに非透過に行うべきである，と結論づけている．

一方，この手法はバグが混入しやすいという問題もある．プログラマが手動で行うため，保存が必要となる領域を指定し忘れる可能性が生じる．これに対して，コンパイラによって保存が必要であるか不必要であるかを判別する試みもなされている [Plank *et al.* 95a]．

Plank らの手法は，主記憶の状態が故障とともに失われるシステムを対象とするため，基本的にはチェックポイントにおいて全ての主記憶領域が永続記憶上に保存されることが前提とされている．このため，保存対象となる領域を減らすことに焦点があてられて設計されている．include_bytes は exclude_bytes によって保存対象から除外された領域を再び保存対象とすることに用いられる．一方，不揮発主記憶システムでは，チェックポインティングでの状態は自動的に保存されていることになる．このため，むしろチェックポイント後に変更される領域について着目し，保存が必要となる領域の増加分に対して着目することが必要であると考えられる．

3.3　周辺デバイスの状態

本研究のもう一つの目的はオペレーティングシステムからのシステム状態の復元であり，デバイスの状態を含めた実行状態の復元である．以下，チェックポインティングティングを行うシステムにおける，デバイス状態の取り扱いについて述べる．

3.3.1　永続オペレーティングシステム

アプリケーションのチェックポインティングをオペレーティングシステムレベルで行うシステムの開発が行われている．特に，主記憶の状態を永続記憶装置にマッピングし，長期的に利用されるデータ，一時的なデータを分け隔てなく永続化 (保存) するシステムは永続システムと呼ばれ，これをサポートするオペレーティングシステムは「永続オペレーティングシステム (Persistent Operating System)」と呼ばれる．永続オペレーティングシステムは，アプリケーションプログラムのファイル操作を簡略化し，また，アプリケーションプログラムの実行状態回復をサポートする．しかし，永続オペレーティングであっても，周辺デバイスに対する考慮はほとんどなされてこなかった．

EROS[Shapiro *et al.* 99] およびその前身である KeyKOS[Landau 92] では，定期的にその主記憶状態をハードディスクやテープなどの永続記憶装置上に保存する [Shapiro *et al.* 99, Landau 92]．これらのシステムでは，チェックポインティング時，全プロセスを一度停止した後，システム上の全メモリ空間に対して 3.1.1 節で述べた Copy-on-Write 技術が用いられる．カーネルのメモリ空間を含め主記憶全体の状態の保存が行われるものの，周辺デバイスが保持する状態の対応については述べられていない．

L3[Liedtke 93]，また L4[Ceelen 02] は，EROS システムと同様に Copy-on-Write 技術を用いてシステム上のメモリ領域をハードディスクに保存する．これらのカーネルは，マイクロカーネルアーキテクチャをベースとし，デバイスドライバはユーザレベルに実装される．しかしながら，デバイスドライバ自体の状態は保存されず，また，周辺デバイスが持つ状態については考慮されていない．

Grasshopper は，ユーザに提供されるプログラムの実行のアブストラクションとして，コンテナ (container) とルーカス (locus) を定義している [Lindstrom *et al.* 95, Dearle *et al.* 94, Rosenberg *et al.* 96]．コンテナはプログラムそのものやデータなどから構成される記憶領域のアブストラクションである．ルーカスは実行の流れそのもののアブストラクションである．ルーカスがコンテナの中やコンテナ間を移動することによってプログラムの実行がなされる．Grasshopper では，状態の永続化はコンテナ毎に行われる．しかし，いつ行うかなどのポリシに対しては，コンテナを管理するマネージャが決定するものとされ，詳細は述べられていない．マネージャはユーザレベルに実装され，カーネルはそのメカニズムを提供するのみとされている．プログラムの実行状態を復元可能とするため，ルーカスに関連する情報 (CPU のコンテキストなど) やカーネルに保持されるデータも同様に保存されるが，やはり周辺デバイスが持つ状態については考慮されていない．

また，Grasshopper の後継である Charm[Dearle *et al.* 00] では，ユーザレベルに対してカーネル内の情報のほとんどを開示することで，実行状態保存の枠組みを提供する．Charm では通常カーネル内に実装される機能のほとんどがユーザレベル

に一任される．デバイスドライバもユーザレベルで実装され，デバイスからの割り込みはカーネルからのアップコールにより，対応するプログラムが呼び出されるようになっている．インタラプトベクタはユーザレベルから参照可能な情報とされ，ユーザレベルから保存が可能な状態となるが，周辺デバイスが保持する情報についてその保存方法や復元方法はデバイスドライバの実装にまかせられている．

3.3.2　既存のチェックポインティングシステムにおけるデバイスの取り扱い

この他，ユーザレベルでのチェックポインティングをサポートするライブラリやオペレーティングシステムでも，周辺デバイスがもつ状態については，ほとんど扱われてこなかった．例えば libckpt[Plank *et al.* 95b] では，オープンしているファイルのファイルディスクリプタおよびファイル上の位置を示すポインタの情報を保存し，復元するのみである．

libckpt は，UNIX を対象とするユーザレベルでチェックポインティングのライブラリである．UNIX ではデバイスはファイルとして抽象化される．ファイルはファイルディスクリプタを通して操作される．ファイルディスクリプタは `open` システムコールにより，カーネルからユーザプログラムに返され，プログラムが利用するファイルの識別子として用いられる．カーネルでは，ファイルディスクリプタと実際のファイルとを結びつける．ファイルディスクリプタはプロセスに独立であり，同一のファイルだったとしても，生成されるプロセス毎に異なる可能性がある．

プログラムが停止する前のファイルディスクリプタが指すファイルと，プログラムが再起動したときにファイルディスクリプタが指すファイルとが同一でなければ，その実行に矛盾が生じる．libckpt ではユーザプログラムに対して透過にこの矛盾を回避するため，仮想的なファイルディスクリプタをユーザプログラムに返すようにしている．ファイルのオープン時には，`open` システムコールをフックし，ユーザにはライブラリ内で適当に決定されるファイルディスクリプタが返される．ライブラリ内部ではオープンしたファイルのパスや，カーネルから返された実際のファイルディスクリプタとユーザに返したファイルディスクリプタの対応関係を保持しておく．`read` や `write` などのファイルに対する操作についてもやはりフックし，この中でユーザプログラムから渡されるファイルディスクリプタの値と実際のシステムコールとして渡されるファイルディスクリプタの変換を行う．リカバリ時には，アプリケーションの再開を行う前に保存されたファイルを再びオープンし，この時のシステムコールにより得られたファイルディスクリプタで，その対応関係を更新する．

また，オープンされているファイルについては，その現在のアクセス位置を示すためのポインタが存在する．ファイルディスクリプタの取り扱いと同様に，`read` や

write システムコール時にフックする処理では，何バイトの書き込みや読み込みがあったか，ということを記録しておく．リカバリ時にはこの情報をもとに lseek システムコールを用いて，そのファイルのポインタを移動させる．このようにして，ユーザプログラムへの透過性，およびリカバリ前後において矛盾が起こらないようにする．

また，Kckpt はカーネルレベルでユーザプロセスのチェックポインティングをサポートする [Hong *et al.* 00]．Kckpt では，ファイルに関して，カーネル内のファイル管理構造体を保存する．ファイル管理構造体は実際のファイルのパスなどが記録され，ユーザから渡されたファイルディスクリプタと実際のファイルとの対応関係を保持する．リカバリ時には，ファイルシステム内でユニークなファイル ID を返す関数を用いて，プロセスがオープンしていたファイルを再びオープンする．この情報をファイル構造体に反映させ，ユーザレベルのファイル操作の一貫性を保つ．

また，WindowsNT において，DLL(Dynamic-Link Library) を用いてチェックポインティング機能を提供する手法も提案されているが，周辺デバイスの取り扱いについては述べられていない [Srouji *et al.* 98]．

この他，既存のチェックポインティング機能を提供するライブラリやシステムでは，プロセス ID やシグナルの状態などが問題視されてはきたが，デバイス自体が持つ状態については，ほとんど考慮されてこなかった．デバイスの状態は OS が再起動したときに自動的に初期化されるものとされ，復帰されるユーザプロセスとの一貫性については，ファイルディスクリプタとファイルのポインタの情報が取り扱われるのみであった．

3.3.3　分散システムにおけるデバイスの取り扱い

メッセージパッシングシステムでは，デバイスの状態はシステムの外部の状態であると考えられ，一つの特殊なプロセスとしてモデル化される．このプロセスは，OWP(Outside World Process) と呼ばれる [Elnozahy *et al.* 99]．

OWP から受け取るメッセージ (デバイスからシステムへの入力) は，アプリケーションに渡される前にその内容が保存される．これは，リカバリ後，OWP が再びそのメッセージを再生しないためである．リカバリ時には，メッセージとしてこの情報が対応するプロセスに渡される．

一方，OWP へのメッセージ (システムからデバイスへの出力) の対応は困難とされ，コミット問題 (commit problem) と呼ばれる．プリンタなどは一度出力したものは元に戻せない．このことから，Strom らはシステムの状態が”commitable” になるまで OWP へのメッセージの出力を遅延させている [Strom *et al.* 85]．“commitable” な状態とは，システムの状態が復元可能となる状態である．デバイスへの出力を行う瞬間の状態を復元可能とし，システムの状態が復元されるときには再びデバイスに対するメッセージが送信されないようにすることで，デバイスに対する出力操作

が繰り返されないようにしている．

また，同様のデバイスの取り扱いは，Plurix システムでも行われている [Bindhammer *et al.* 02]．Plurix システムはトランザクション型メモリ一貫性を提案する分散システムである，Plurix システムでは”smart buffer” を提案し，デバイス操作中にトランザクションが abort したとき，その操作が出力されないよう，commit までデバイスに対する実際のコマンド発行を遅延させる．

OWP からの入力についてはその処理を行う前に一度保存が可能であることが前提とされている．これはアプリケーションレベルのプロセスが対象とされるためであり，本研究のようにデバイスの状態を直接扱う場合にはプログラム (オペレーティングシステム) に対しその情報が渡される前に保存作業は行うことはできない．

また，OWP に対する出力に対しては，その出力操作の原子性を保証するための対応であり，OWP(デバイス) 自体の状態の復元は考慮されていない．このため，電源切断の対応を可能にするものではない．OWP としてモデル化されるデバイスは，あるプロセスに故障が発生したとしても動作し続けるものとして考えられている．つまり，メッセージを送信したプロセスに故障が発生したとしても，メッセージを受け取ったデバイスはその動作を続けるものとされている．

また，デバイスの操作としてメッセージやバッファなどが単位として考えられており，デバイスはそれを受け取ることによってある意味のある動作を行うものとしてモデル化されている．しかし，デバイスは 1 つ以上のレジスタが CPU によって設定されることにより，始めて意味のある動作がなされる場合が多い．同時に，2.2.4 節で述べたように本研究では不揮発主記憶システムを対象とするため，デバイスに対するアクセス (レジスタの操作) を単位とし，CPU や主記憶の状態との一貫性を維持した状態の復元を考慮する必要がある．

3.3.4　システム電源管理手法

APM[APM 96] や ACPI[ACPI 02] などのシステム電源管理仕様は，サスペンド／レジュームやハイバネーション機能を提供する．これらの機能は，本研究で目標とする「システム全体の実行状態の復帰」を実現する．しかし，これらはシステムの電源切断が予め通知され，かつシステムに供給される電力量が十分に存在する時に動作することが前提とされている．このため，突発的な電源切断に対応可能なものではない．

電源切断の通知は，例えばシステムの電源ボタンが押された時や，長時間キーボードの操作が無かったとき，また，“smart battery” と呼ばれ，残留電力を常に監視するバッテリによって残留電力量の低下が認識された時などに行われる．これらの実装では，システムの電源切断が要求された時に各周辺デバイスの状態を調べての状態の保存を行う．電源切断の要求時，デバイスが稼動中など状態が適切に保存可能でない場合にはシステムの電源切断自体を遅延させる．そして，後に再びデ

バイスの状態を調べ，これを繰り返す．このため，突然電源が失われるような状況には対応できない．

なお，ACPI では”Emergency Shutdown” として状態保存に対して必要な電力が残されていない場合の対応についても触れられている．しかし，システムの状態損失や破損を最小限に抑えるため，電源切断時に各デバイスの状態を「なるべく保存する」と述べられているのみであり，実行状態の復元を保証するものではない．

予備電池などの補助的な電源を用いて実際のシステム電源切断を遅延させ，これらシステム電源管理手法を適用することも可能である．しかし，システムに接続されるデバイスの種類や数，その処理にかかる時間は不定であり，結果として補助電源の大型化を招かざるを得ない．これはできる限りの小型化や軽量化が必要となるシステムでは問題となる．このため，本研究で想定するようなシステムには不適切である．電力が突然失われるような場合については，これらの手法のように電源切断要求が通知されてから対処するのではなく，システムの通常実行中に突然の電源切断に対して予め何らかの対応策を施しておく必要がある．

3.4　本章のまとめ

本章では，本研究の目的である「低オーバヘッドでのチェックポイティング手法の実現」と,「周辺デバイスを含めた実行状態の復元」という立場から，関連研究について述べた．

チェックポインティングの高速化手法についてキャッシュをベースにした手法が考えられるが，ハードウェアの変更を必要とする．また，保存対象となる主記憶量の低減し高速化する手法として，MMU のページ単位よりも小さい粒度で保存対象を特定する研究もあるが，ディスクなどの低速な記憶装置を意識したものであった．

また，周辺デバイスを含めた実行状態の復元は従来研究についてほとんど扱われてこなかった問題である．ライブラリなどのサポートではファイルディスクリプタやポインタの状態は復元されたが，デバイス自身が持つ状態は考慮されてこなかった．分散システムでは，特殊なプロセスとしてモデル化されることもあるが，そのプロセス (デバイス) の状態を復元することは考えられてこなかった．APM や ACPI などのシステムの電源管理仕様では，デバイスの状態を明示的に保存および復元する方法を規定するが，本研究で対象とする「突発的な電源切断」に適用可能なものではなかった．

第4章

CPUと主記憶状態の復元

本章では，可観測かつ揮発な要素であるCPU，および可観測かつ不揮発な要素である主記憶について，その実行状態を復元する手法について述べる．図4.1に本章で扱う対象を示す．このとき，電源切断時に不揮発である主記憶に残される状態をできる限り利用し，高速な状態保存作業を可能にする．また，CPUの主記憶に対するストア命令一つ一つが電源切断後に残されることを考慮し，チェックポインティング中やリカバリ処理中においても適切に実行状態を復元することを可能にする．

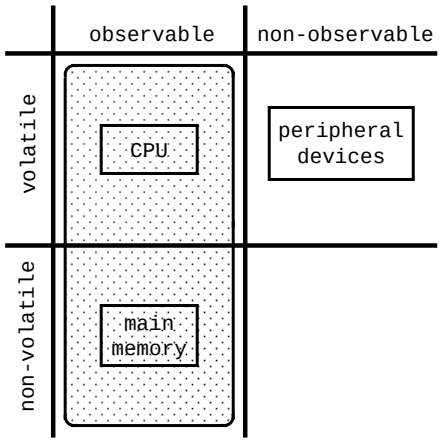

図 4.1: 4章で対象とする要素

4.1 基本方針

主記憶は不揮発であり電源切断その時の状態が残される．しかし，CPUは揮発な要素であり，電源切断とともにその状態は失われるため，明示的に保存を行わなければならない．このとき，復元されるCPUの状態との一貫性を保つため，主記

憶の状態も CPU の状態が保存される時点と同一のものを復元可能とする必要が生じる．

CPU と主記憶についてその実行状態を復元可能とするため，従来研究と同様に，チェックポイントでの主記憶の状態および CPU の状態を保存しておく．そしてリカバリ時にはこれらを復元し，システムの状態を最後のチェックポイント時に戻す．本章で提案する手法ではプログラマがマクロ化可能な程度のコードをプログラム (カーネル) に対して埋め込み，また，明示的にチェックポインティングを行う手法を採用する．この理由は以下の 2 つである．

一つは，3.2.2 節で述べたように，プログラマが明示的にその保存対象とチェックポインティングのタイミングを特定することによって，オーバヘッドの大幅な高速化が得られるという結果が得られていることである．プログラマに負担は強いるもののその労力はあまり大きくはない，と結論づけられており，コンパイラによるサポート方法も提案されている．

そして，もう一つは，デバイスが動作していない時点において明示的にチェックポインティングを行うことで，本手法のみでシステムの実行状態回復を実現するためである．本章ではデバイスに対する考慮は行わないが，デバイスが動作していない期間における状態を復元可能とすれば，リカバリ処理においてデバイスの再初期化を行うことによって，本章で提案する手法のみでも適切に実行状態の復元が可能となることが考えられる．

3.2.2 節では，従来研究において，保存対象となる主記憶領域の特定に対しワード単位，ブロック単位で行うことによって MMU のページ単位で行ったときよりもチェックポイティングの性能が改善されたことについて述べた．本章で提案する手法では，malloc などのオブジェクト獲得ルーチンにより確保されるオブジェクトを更新の検出単位として用いる．厳密にワード単位で変更領域を特定することは従来研究の例から，必要となる処理量が多くなり過ぎると考えられるためである．そして，オブジェクトの管理構造を工夫することによって，実行中に埋め込まれるコードの高速化および，チェックポインティング時に保存が必要となるオブジェクトの特定を高速化する．

また，3.2.2 節では，基本的には全ての主記憶領域が保存され，主記憶の保存量を減らす目的でオブジェクトの明示的な指定が行われてきたことについて述べた．このため，3.2.2 節で述べた手法ではチェックポイント以前に保存対象から除外される領域が指定された．しかし，本研究で対象となる主記憶は不揮発であるため，チェックポイント後変更される領域のみを特定すれば良い．このため，オブジェクトが変更される時にその特定が行われれば良い．

本手法では，実際のプログラムでは次のような記述を行うこととなる．

```
      :
p = malloc(sizeof(struct object));
      :
checkpoint();     // チェックポイント
      :
/* オブジェクトの更新 */
WRITE_OBJECT(p); // 挿入されるマクロ
p->member1 = 1;
p->member2 = 2;
      :
checkpoint();     // チェックポイント
      :
```

各オブジェクトの管理構造体には，更新されたことを示すフラグを付加する．WRITE_OBJECT 内ではオブジェクトの管理構造体を特定し，フラグをセットする．またチェックポイントやリカバリ時には各オブジェクトについてこのフラグを参照し，状態の保存や復元処理を行う．WRITE_OBJECT 内でのオブジェクトの特定やチェックポイントでのオブジェクトの特定が高速に動作するように，オブジェクトの管理構造には構造化された管理方式を用いる．詳細は 4.2.2 節で述べる．

主記憶の状態としてはオブジェクトのほか，スタックや広域変数の状態が存在する．スタックの扱いについては 4.2.5 節で述べる．広域変数の扱いについては 4.2.6 節で述べる．

CPU の状態は実行とともに保存が行われないため，チェックポイントにおいて明示的に保存を行う必要があることについて述べた．CPU の状態保存中に電源切断が生じた場合に対応するため，CPU の保存を行う領域は 2 つ設け，これらをチェックポイント毎に切り替えて使用する．

実行時の環境とチェックポイント時のシステム (データ) の状態を 2 重化する，という考え方はシャドウページングなどの「サイドファイル」手法 [Gray *et al.* 93, Lorie 77, Challis 78] に類似する．これらの手法では，時間のかかるディスク間のコピー作業を削減するために，プログラムが使用する物理的な領域 (ファイルやファイルに対応するスロット) の切り替えを行う．これは，ファイルの内容が一度主記憶上のバッファに読み込まれることを利用し，バッファの書き込み先を変更する方法で行われる．そして，チェックポインティングではこれら物理的な領域に対して有効となるマッピングの情報 (参照関係情報) を切り替える．しかし主記憶を対象とした場合，その状態を保存するためには主記憶間でのコピーが必須となる．またメモリ間のコピーは高速に行えるため，参照の切り替えを行うことによるオーバヘッドの方が大きくなる．そこでプログラムからはオブジェクトとして常に同じ領域が

参照されるようにし，参照の切り替えは行わない．オブジェクトの内容は常に保存のために用意された領域に保存を行う．

またサイドファイル手法では，チェックポインティング中に故障が発生した場合に対応するため，チェックポインティングでの作業 (ファイルのコピー) の原子性を確保する必要がある．シャドウページングでは，この目的のためにも参照の切り替えが用いられている．本手法では，CPU の保存領域を指す 2 つのポインタを用いることでこれを実現する．詳しくは 4.2.8 節で述べる．

4.2　設計および実装

本節では，まずプログラムの処理の流れに対して状態保存時および復旧時に行うべき処理について検討する．次にその具体的な実現方法について述べる．

4.2.1　処理の流れに対する要求

前述のように本手法はチェックポイント後に更新されるオブジェクトの状態を保存することで，チェックポイント時のオブジェクトの状態を復元可能とする．このときオブジェクトの保存のタイミングとして以下の 2 つを考えることができる．

(1) write アクセスが生じるときにそのオブジェクトの更新前の状態を保存する

(2) チェックポイントにおいて，以前のチェックポイントから更新されたオブジェクトの状態を一括して保存する

(1) の手法では，オブジェクトの保存のための領域を write アクセスが生じたときに確保する．また，確保された保存領域はチェックポイント時に破棄する．このため (1) の手法は「動的割り当て手法」と呼ぶことにする．(2) の手法では，プログラム中の malloc ルーチンなどを用いてオブジェクトを確保するときに，そのルーチン内で保存のための領域を同時に確保する．またオブジェクト自体がプログラムによって破棄される (free) まで保存領域は破棄されない．このため (2) の手法は「静的割り当て手法」と呼ぶことにする．

以下，それぞれについて詳細を述べる．

動的割り当て手法

図 4.2 にチェックポイント間でオブジェクト A と B が write アクセスされる場合の処理を示す (2 段目)．図 4.2 は最上段にオブジェクトに対するフラグの操作，3 段目に電源切断が生じた場合に復元すべき内容，最下段にシステムの状態が復元される時点を同時に表わしている．

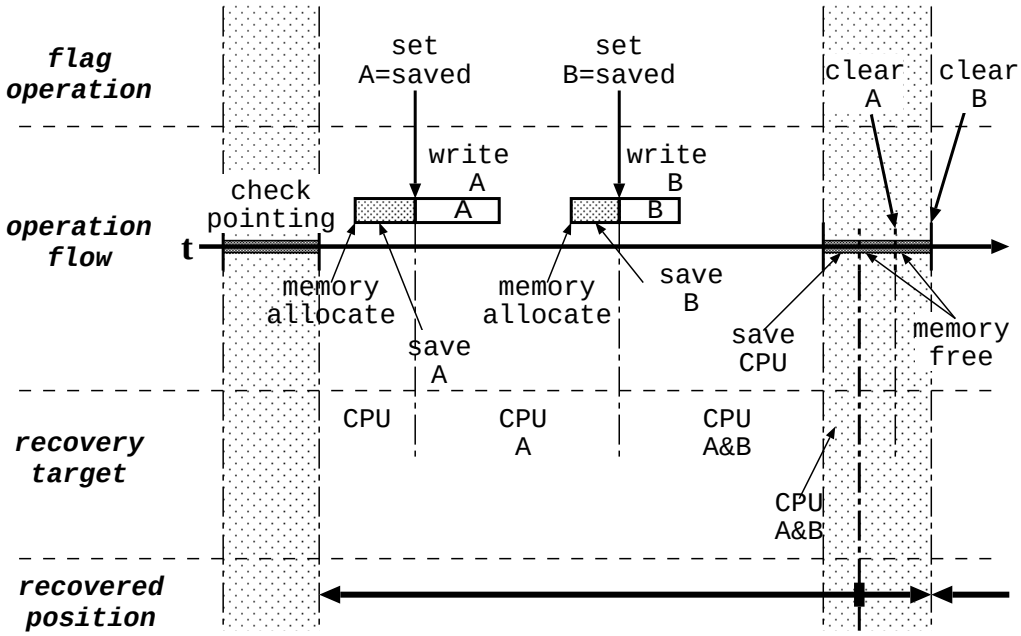

図 4.2: 動的に保存領域を確保した場合

この手法ではそれぞれオブジェクトが write アクセスされる直前に保存領域を動的に確保し，その時点でのオブジェクトの状態をコピーする (図 4.2 中 2 段目の濃い網掛け部)．このとき保存される内容は，直前のチェックポイント時の内容と同一のものとなる．

リカバリ作業では CPU の状態と保存が完了したオブジェクトの状態を復元すれば，チェックポイントでのシステムの状態を復元することができる．しかしオブジェクトの保存中に電源切断が生じた場合には，保存領域内の内容は不定となるため保存領域の内容を復元してはならない．またチェックポイント間で同一のオブジェクトが 2 度以上 write アクセスされる場合，2 度目以降ではオブジェクトの保存は行ってはならない．保存を行った場合，オブジェクトのチェックポイントでの状態が書き潰されてしまうためである．このためオブジェクトの保存が終了した時点でオブジェクトの保存が完了したことを示すフラグをセットする (図 4.2 中最上段)．このフラグを参照し，すでにセットされている場合には保存作業は行わない．リカバリ時には，このフラグがセットされているオブジェクトについてのみ状態を復元し保存領域の破棄を行う．保存領域が確保されたとしてもフラグがセットされていないものは，保存用の領域の破棄作業のみ行う (図 4.2 中 3 段目)．

チェックポイントでは CPU の状態を保存した後，各オブジェクトについて保存のために確保された領域を解放しフラグをクリアする．CPU の状態保存中に電源切断が生じたときには，保存領域にあるオブジェクトの状態と保存が完了している CPU の状態を用いて 1 つ前のチェックポイントの状態に戻す．一方 CPU の状態保

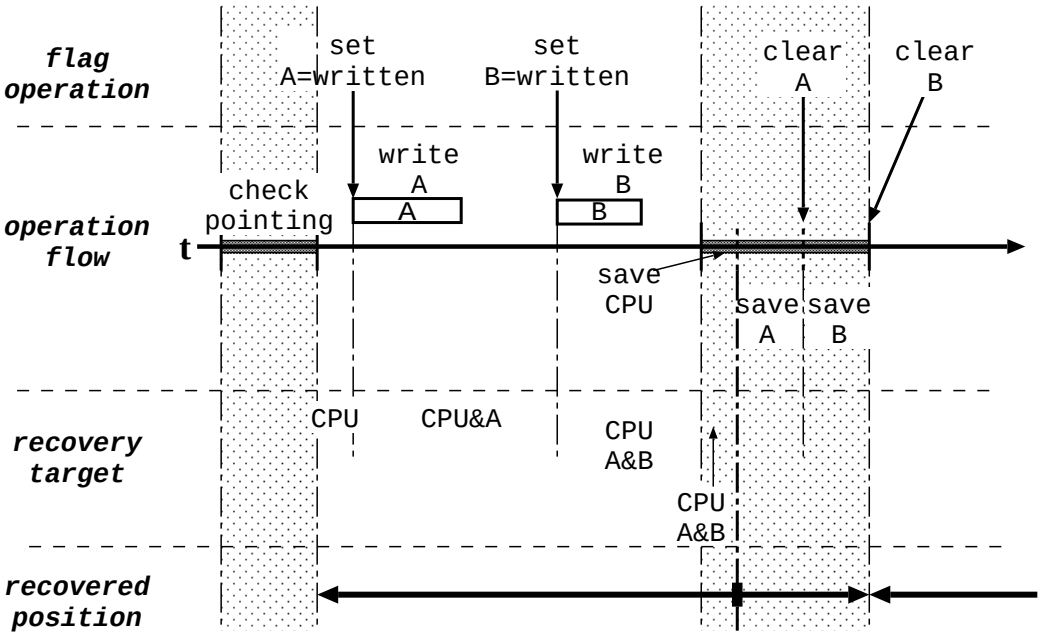

図 4.3: 静的に保存領域を確保した場合

存後であれば，主記憶は不揮発であるため各オブジェクトはそのチェックポイント時の内容を保持したままリカバリに臨むことが可能である．このため各オブジェクトのオリジナルの領域を用いて，電源切断が生じたチェックポイントでの状態を復元することができる．つまり CPU の状態保存が完了したかどうかで復元されるシステム状態が切り替わる (図 4.2 中最下段)．なお，CPU の状態保存後にシステムの電源切断が生じた場合，各オブジェクトに対してチェックポイントでの作業 (保存領域の解放およびフラグのクリア) を行った後，システムの動作を再開させる必要がある．これはシステムの復元後，チェックポイントに到達する前に再び電源切断が生じた場合，リカバリ作業で実際には保存されていないオブジェクトが保存されたと誤認識されることを防ぐためである．

静的割り当て手法

図 4.2 と同様に，この手法について処理の流れと要求される動作を図 4.3 に示す．

この手法の場合オブジェクトの保存領域は，malloc ルーチンなどでオブジェクト自体が確保されるときに同時に確保する．write アクセスが生じたときにはオブジェクトのフラグのセットのみ行う．そしてチェックポイントでフラグがセットされているオブジェクトの保存を一括して行う．また，各オブジェクトのフラグは保存が完了した時点で順次クリアする．通常処理中に電源切断が生じた場合，フラグがセットされているオブジェクトのみ状態の復元を行う．

この手法では，各オブジェクトの直前のチェックポイントでの状態は常に保存領域に存在することとなる．オブジェクトが直前のチェックポイントで保存されなかった場合には，write アクセスが生じなかったことを意味し，以前のチェックポイントで保存されたオブジェクトの状態は直前のチェックポイントと等しい．このため，保存領域に存在する状態を用いて直前のチェックポイントの状態を復元することができる．チェックポインティング中に電源切断が生じた場合には動的割り当て手法と同様に，CPU の状態の保存が完了したかどうかによって復元される時点が切り替わる．また CPU の状態保存後電源切断が生じた場合も同様に，チェックポイントでの作業 (オブジェクトの保存およびフラグのクリア) を行った後にシステムの復元をする必要がある．

動的割り当て手法ではチェックポイント後にアクセスされるオブジェクト分のみの保存領域を確保するため，メモリ使用量を節約できる．また状態を保存するための処理は分散し，チェックポイントにおいてプログラムの処理を長時間停止させる必要がないという利点がある．しかし，通常の処理中に保存作業が入り，状態保存のオーバヘッドがいつ生じるかという予測が困難となる．またメモリを確保／解放するためのオーバヘッドが余分にかかるなどの欠点がある．一方，静的割り当て手法では，常にオブジェクトとして利用しようとするメモリ領域の倍のメモリ量を必要とする．またチェックポイント時に CPU はその処理に専念しなければならない．しかし，状態保存のオーバヘッドが集約されるため CPU のアイドル時間を利用しオーバヘッドを隠蔽するなどの制御が可能となる．このような一長一短がそれぞれの手法に存在するため，どちらを選択するかはシステムの開発ポリシに依存することとなる．

以下，具体的なオブジェクト管理手法およびチェックポイント時とリカバリ時の処理について述べる．

4.2.2　slab の拡張

オブジェクトの管理について，本研究では slab メモリアロケータ (以下，slab) [Bonwich 94] に着目した．本節ではまず slab について述べ，次に slab をもとにしたオブジェクト管理の拡張について述べる．

slab では MMU のページを単位として図 4.4 に示すような管理構造を作成し，オブジェクトの利用状況を管理する．`kmem_cache` はオブジェクトの種類やサイズ毎に作成される．`kmem_slab` はページ毎に作成され，`kmem_cache` で管理されるオブジェクトの種類について，ページ単位での利用状況を管理する．`kmem_bufctl` は `kmem_slab` の元で実際のオブジェクトの利用状況を管理する．実際のオブジェクトは 1 つのページがあるオブジェクトのサイズで区切られ，それぞれのオブジェクトがプログラムからの要求に応じて渡される．

`kmem_cache` 構造体はオブジェクトの種類 (サイズなど) 毎に一つずつ作成され，オ

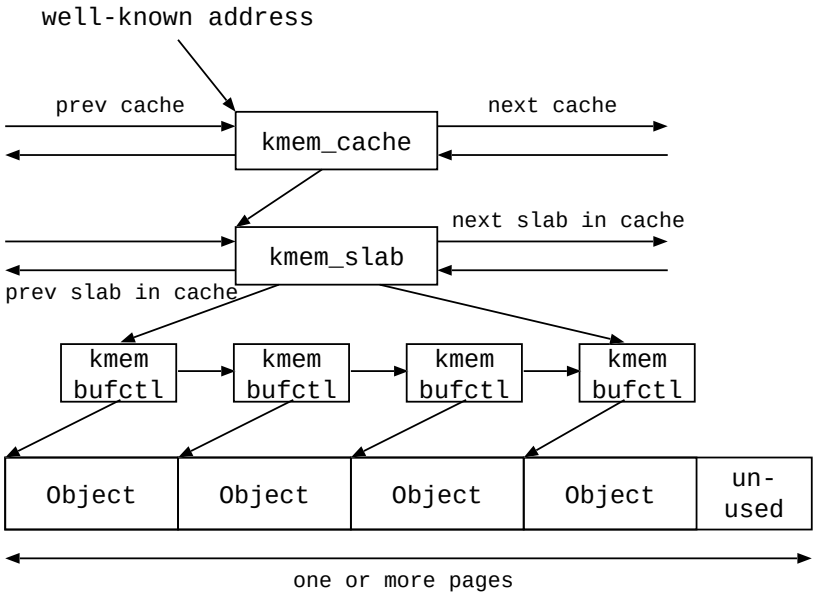

図 4.4: slab メモリアロケータの管理構造

ブジェクトの名前やサイズなどの情報や kmem_slab へのポインタを持つ．slab では新たなオブジェクトを作成する際に物理メモリページを取得し，これをオブジェクトのサイズで区切る．そして，ページに対し，kmem_slab を割り当てる．kmem_slab は対応するページへのポインタや kmem_bufctl へのポインタなどを持つ．区切られた各オブジェクト毎に kmem_bufctl 構造体を作成し，対応するオブジェクトが使用されているかどうかを管理する．それぞれの構造体は他の同構造体へのポインタを持ち，リスト構造によって管理される．

slab はシステムの初期化時にグローバルなポインタをルートとして，このような木構造に似た構造を作成する．この構造によって，オブジェクト毎の情報の追加を容易に行うことができ，また保存を必要とするオブジェクトの探索を高速に行うことができる．さらに slab 自体は同種のオブジェクトの確保／破棄が繰り返し行われる場合の高速な処理を特徴とする．これは動的割り当て手法について大きな利点となる．以上のような理由から本研究では slab に着目した．

4.2.1 節にて述べたオブジェクトの状態を示すフラグおよびオブジェクトの保存領域を指すポインタをこの kmem_bufctl に追加する．またチェックポインティングですべての kmem_bufctl を走査する必要を無くすため，kmem_slab に対してもフラグを付加する．このフラグは，その kmem_slab 管理下のオブジェクトが write アクセスされた際にセットする．チェックポインティング時にこのフラグを参照し，セットされていなければその kmem_slab 以下の kmem_bufctl の走査は行わない．これによりチェックポインティングの高速化をはかる．

4.2.3　挿入されるコード

プログラム中に挿入するコードは，それぞれ以下のような処理を行う．動的割り当て手法の場合，以下のようになる．

```
オブジェクトに対応する kmem_slab の特定;
オブジェクトに対応する kmem_bufctl の特定;
if(!kmem_bufctl 内の保存領域へのフラグ){
    保存用領域の取得;
    kmem_slab のフラグのセット;
    オブジェクトの保存;
    kmem_bufctl のフラグのセット;
}
```

また静的割り当て手法の場合には，以下のようになる．

```
オブジェクトに対応する kmem_slab の特定;
オブジェクトに対応する kmem_bufctl の特定;
kmem_slab のフラグのセット;
kmem_bufctl のフラグのセット;
```

オブジェクトに対応する kmem_slab および kmem_bufctl の特定は図 4.5 のように動作する．図 4.5 中左側は実際のメモリ空間を表わし，この中にプログラムによって利用されるオブジェクトが存在する．そして，図 4.5 中中央はページ管理テーブルであり，それぞれのエントリはメモリ空間のページに対応する．そして，図 4.5 中右側は，ページに対応する slab の管理構造体を示している．

オブジェクトの存在するページを特定することにより，ページに対応するページ管理構造体が特定できる (図 4.5 中 1.)．kmem_slab はページ毎に作成されるため，ページ管理構造体に対応する kmem_slab へのポインタを保持することにより，オブジェクトに対応する kmem_slab を特定することができる (図 4.5 中 2.)．kmem_bufctl は作成時にオブジェクトの後方に作成される場合と kmem_bufctl 専用の領域に作成される場合がある [Bonwich 94] が，どちらの場合でもオブジェクトのページ内のオフセットから特定することができ，リスト構造を辿る必要は無い (図 4.5 中 3.)．このように slab ではポインタを 3 段辿るだけでオブジェクトに対応した kmem_slab と kmem_bufctl の特定が可能である．

チェックポインティングではフラグがセットされた kmem_bufctl を探しだし，動的割り当て手法の場合には，保存領域の解放および kmem_bufctl 内のポインタとフラグのクリアを行う．静的割り当て手法の場合にはオブジェクトの保存とフラグのクリアを行う．また一つの kmem_slab 下のオブジェクトに対してこの作業が終了

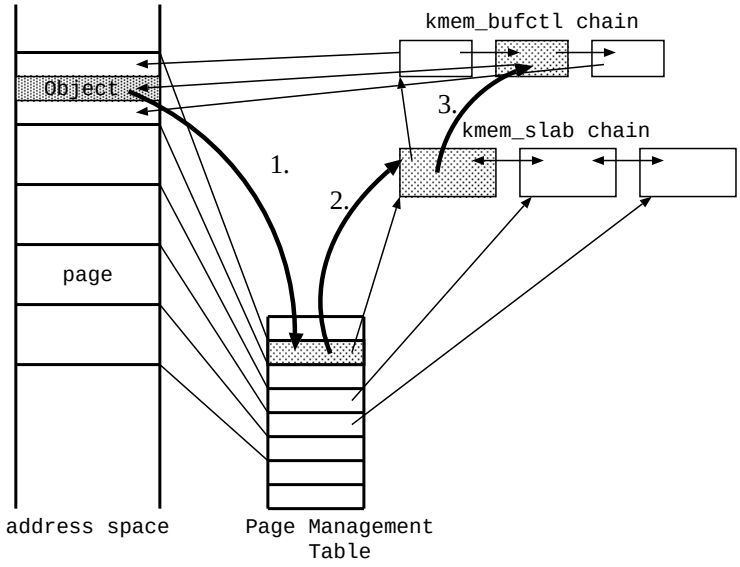

図 4.5: 管理構造体の特定

した後，kmem_slab のフラグをクリアする．

4.2.4　ページ管理

4.2.2 節で述べたように，slab はオブジェクトの取得要求が到着した際に管理構造内に割り当て可能なオブジェクトが無ければ，ページ管理機構からページを取得する．そしてこのページの中に新しくオブジェクトを作成する．このためページの利用状況も復元可能としなければ，復帰後の処理においてメモリリークが発生することになる．

ページに対する操作 (割り当てと解放) は，そのログを取ることでリカバリを可能とする．チェックポイントではこのログは破棄する．リカバリ時にはこのログを用いて undo 作業を行う．ログ領域は予めテーブルとして用意しておき，ログ量の増加とともにテーブルのエンティティを指すインデックス値を増加させる．ログの破棄は，そのインデックス値を 0 とすることで行う．

4.2.5　スタック領域

スタック領域は処理の実行に伴い非明示的に write アクセスが生じる．またそのアクセスされる範囲 (アドレス) もチェックポイント間で大きく増減する可能性があり，予測を行うことは困難である．このためスレッドに対して割り当てられたスタック領域の最上位から，現スタックポインタが示す範囲までをチェックポイント

時に保存する．

スレッド作成時にスレッドが使用するスタック領域と同時にスタック保存用の領域を割り当てる．スケジューラがスレッドを選択した際にスレッド管理構造体に対してスレッドが走行したことを示すフラグをセットし，チェックポイントではこのフラグがセットされているスレッドのスタック領域を保存する．これは静的割り当て手法におけるオブジェクトの扱いと同一の方法となる．

なお，スタック領域が増大し，ページサイズ以上となったときには，MMU を用いて更新のない領域を保存対象から除外することができる．しかし，このとき用いられる手法は通常のインクリメンタル・チェックポインティングにおける手法と同一であるため，本稿では述べない．

4.2.6　広域変数

広域変数については，その対応は 2 種類の方法が考えられる．

一つは，オブジェクトと同様に扱う方法である．グローバル (BSS) 領域には，オブジェクトへのポインタを保持させ，プログラムの初期化時にオブジェクトとして確保する．値の初期化が必要な場合には同様に初期化時に，オブジェクトの内容を定義する．プログラムからはグローバル領域に存在するポインタを通してこのオブジェクトを利用する．プログラムの改変が必要になるものの，保存の方法を他のオブジェクトと同様の扱いとすることが可能となる．

もう一つの方法は，MMU を用いてページ単位でその更新を特定する方法である．この場合には BSS 領域を通常の MMU を用いた手法で管理することとなり，ページ単位での保存がなされることになるが，プログラムの改変は必要ない．

それぞれ，プログラマの実装方式に委ねられる．

4.2.7　CPU 状態の保存

CPU の状態を保存中に電源切断が生じた場合，復元すべき CPU の状態が書き潰されてしまうため，CPU の状態を保存するための領域は 2 つ設ける．また 4.2.1 節で述べたように，CPU の状態保存が終了したかどうかによってリカバリされる時点が切り替わる．このとき「主記憶についてリカバリされる時点」と「どちらの CPU 状態を用いてリカバリを行うか」という 2 つの独立な事象を原子的に切替えることができなければならない．このため，CPU 保存領域を示す 2 つのポインタ，`current` と `next` を設ける．通常実行時には `next` と `current` はそれぞれ別々の CPU 保存領域を指す．これらのポインタの動作は次節で述べる．

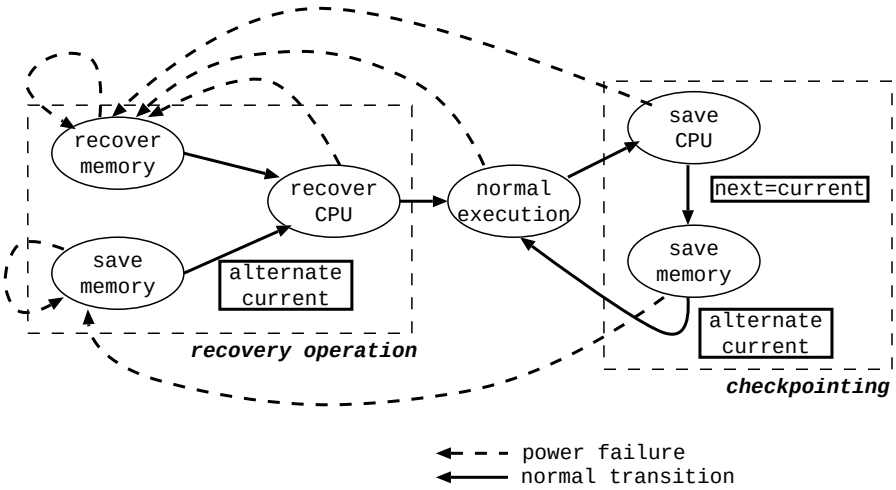

図 4.6: 状態遷移図

4.2.8　チェックポインティングとリカバリ

チェックポインティングは以下のように行う．

next が示す領域へ CPU の状態を保存;
next = current;
ページ操作ログの破棄;
スタックの保存;
オブジェクトの処理;
current = 以前の next;

「オブジェクトの処理」は，動的割り当て手法では保存領域の破棄とフラグのクリアであり，静的割り当て手法ではオブジェクトの保存とフラグのクリアである．

current と next のポインタの切り替えによる，システムの状態遷移を図 4.6 に示す．破線矢印は，電源切断が生じたときの状態遷移である．実線四角内に current と next への操作を記した．

通常実行時および CPU の状態保存が完了する前までは，4.2.1 節での議論より一つ前のチェックポイントでの状態を復元しなければならない．このとき current と next の値は異なる．一方チェックポインティング中 CPU の状態保存が完了した後では，そのチェックポイント時の状態を復元する．このとき current と next の値は同一となる．

リカバリ処理のフローチャートを図 4.7 に示す．リカバリ時には，まず current と next の値を比較する．current と next の値が異なる場合には，ログによるページ

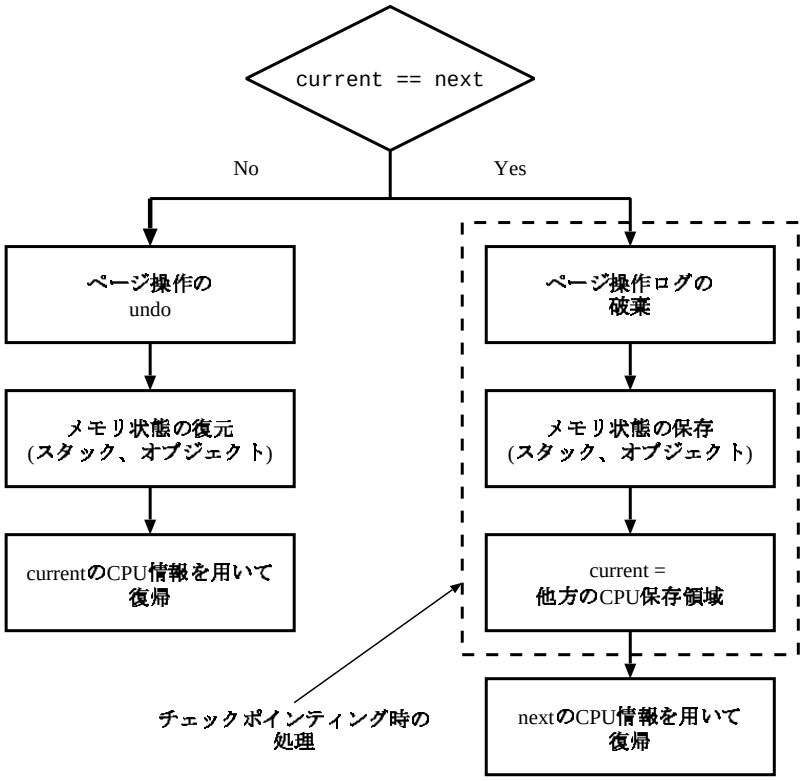

図 4.7: リカバリ処理のフローチャート

操作の undo を行い，オブジェクトとスタックの状態を復元する．そして，current が指す領域には一つ前のチェックポイントでの CPU 状態が存在するため，これを用いてシステムを復元すれば良い．

current と next の値が同一の場合には，チェックポインティングでの処理 (CPU 状態の保存以外) を再び行う．そして，current を next とは逆の領域にセットした後，next が指す CPU の状態を用いてシステムを復元すれば，電源切断が生じたチェックポイントの状態を復元することができる．

ポインタの操作は CPU のストア命令一つで実行が可能なため，このようにすることで前述の 2 つの独立した事象を原子的に切替えることが可能となる．

リカバリで行われる作業は，ログを用いたページ操作の undo およびオブジェクトとスタックの状態の保存／復元であり，巾等に行うことが可能な処理である．リカバリ作業中電源切断が生じた場合，再び current と next の値が比較され同一の処理が行われることとなるが，処理の巾等性より保存状態を損なうことなく適切にシステムの状態の復元が可能である．current と next が同一であり current ポインタが変更された後にシステムの電源切断が生じた場合，復元時には current と next が異なっていると判断される．しかしフラグのクリアやログの破棄はすでに

表 4.1: 実装環境

CPU	モトローラ社 MPC860 50MHz I-cache 4Kbyte D-cache 4Kbyte(Write Through)
ボード	モトローラ社 MPC860FADS
プログラム領域	MPC860FADS 付属 FlashROM サイクルタイム 140nsec (実行時 70ns EDORAM 領域にコピー)
データ領域	ラムトロン社 FeRAM FM1808-70 サイクルタイム 140nsec

なされているため，行われる処理は `current` が示す CPU 状態を用いて復元するのみとなる．このため適切にシステムを復元させることが可能である．

4.3　実験

不揮発メモリとして FeRAM を用いたメモリボードを作成し，プロトタイプ OS を作成してこれまでに述べた手法の実装を行った．実装環境を表 4.1 に示す．

4.3.1　電源切断の実験

作成したシステムを用いて，通常実行時，挿入コードの実行時，チェックポイント時 (CPU 保存前，CPU 保存後) の各場合において，何も処理を行わない for 文を挿入して電源切断のタイミングを作成し，故意にシステムの電源を落した．そして再び電源を投入し復帰を行った．動的割り当て手法および静的割り当て手法それぞれについて行い，いずれもシステムが正しく復帰することを確認した．同様にリカバリ処理中に再び電源を切断したときにも，誤動作することなくシステムの状態が復元されることを確認した．

4.3.2　オーバヘッドとなる時間の測定

通常実行時にオーバヘッドとなる時間を測定するため，挿入コードにより消費される時間とチェックポインティングに要する時間の測定を行った．10 個のオブジェクトのサイズを 32〜4096byte まで変化させたときの動的割り当て手法による結果を図 4.8 と図 4.10 に示す．また静的割り当て手法における結果を図 4.9 および

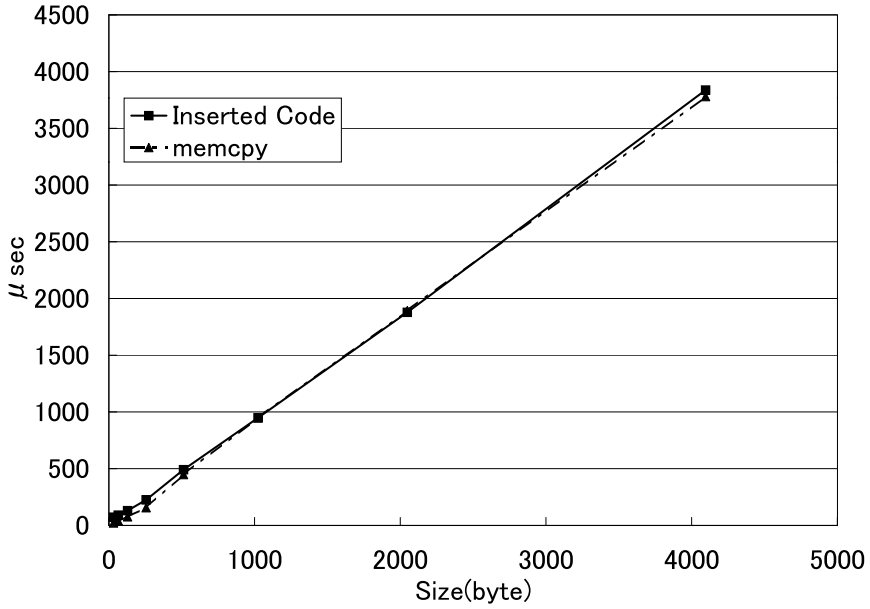

図 4.8: 挿入コードによるオーバヘッド (動的割り当て手法)

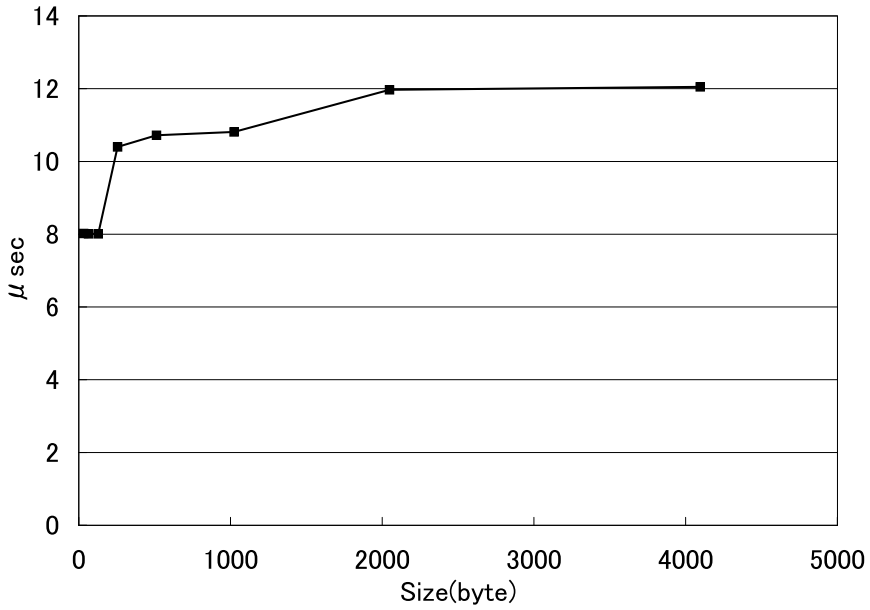

図 4.9: 挿入コードによるオーバヘッド (静的割り当て手法)

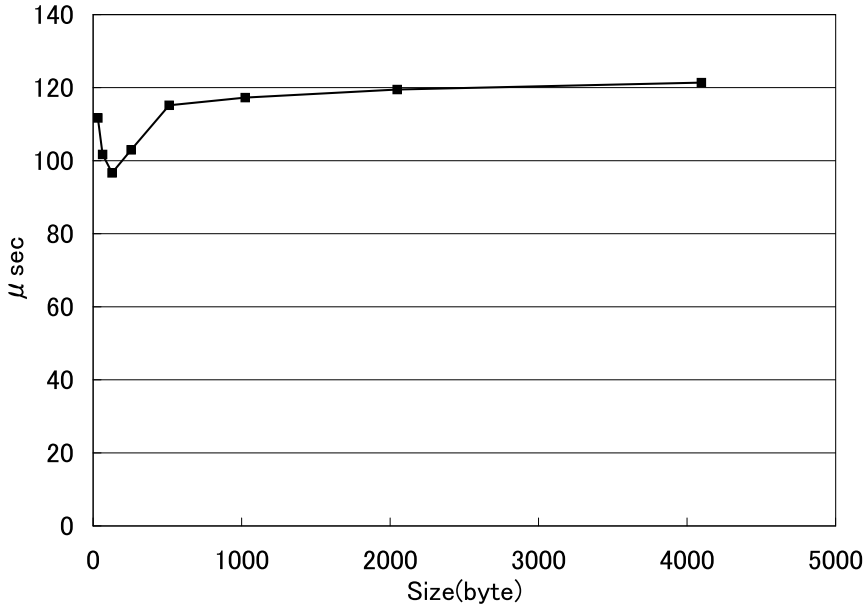

図 4.10: チェックポインティングに要する時間 (動的割り当て手法)

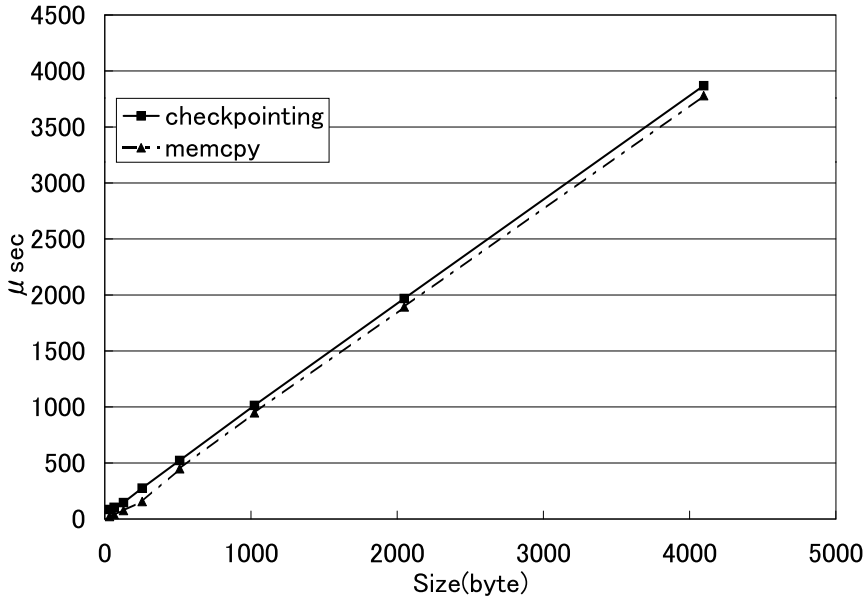

図 4.11: チェックポインティングに要する時間 (静的割り当て手法)

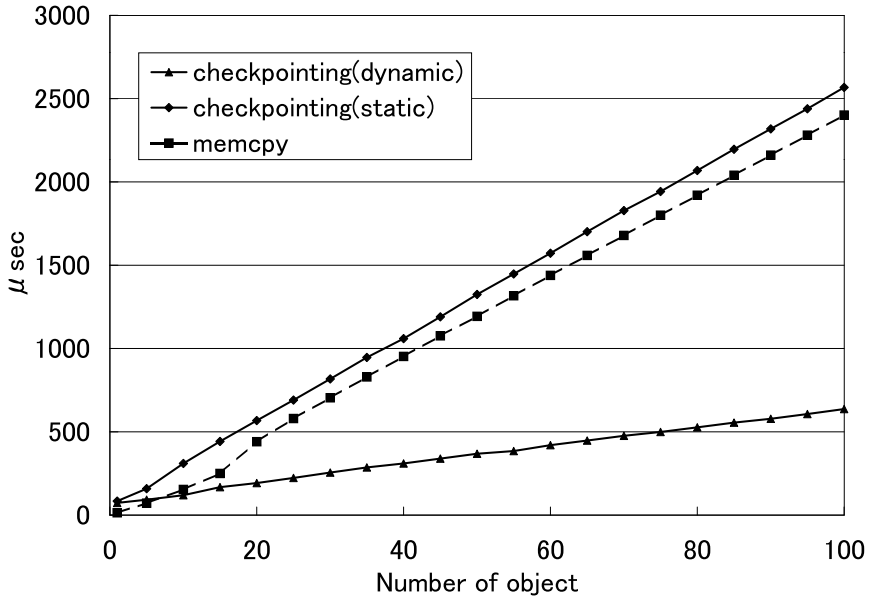

図 4.12: オブジェクトの数とチェックポインティングに要する時間

図 4.11 に示す．それぞれ，挿入コードにより消費される時間とチェックポインティングにより消費される時間を表わしている．なおスタックとして使用されている領域は 250byte 程であった．

挿入コードにより消費される時間はオブジェクトへの write アクセス毎に発生する時間であり，それ自体はオブジェクトの数には依存しない．一方チェックポインティングに要する時間は，チェックポイント間でアクセスされたオブジェクトの数に依存すると考えられる．そこでこの関係を調べるため，オブジェクトのサイズを 256byte に固定しオブジェクトの数を 1〜100 個まで変化させたときのチェックポインティングに要する時間の測定を行った．結果を図 4.12 に示す．以下，これらの結果を元にオーバヘッドとなる時間を概算するための式の導出を行う．

挿入コードによるオーバヘッド

動的割り当て手法では挿入コードによってオブジェクトの保存作業が行われる．このため，図 4.8 にはオブジェクトの保存のみに要する時間 (`memcpy`) も記載した．図 4.8 よりオーバヘッドとなるのはそのほとんどが `memcpy` に要する時間であり，提案したオブジェクトの管理手法が高速に動作していることがわかる．またオブジェクトのサイズに対して線形増加しており，この傾きは約 0.93 であった．測定時のオブジェクトの数は 10 個なので，オブジェクト 1 個あたりの時間 $E_{dyn}(\mu\mathrm{sec})$ はオブジェクトのサイズを S としたとき次式で見積もることができる．

$$E_{dyn} = \alpha \times S \qquad (\alpha = 0.1) \tag{4.1}$$

また図 4.9 より，静的割り当て手法の処理の場合にはオブジェクトのサイズにほとんど依存しないことがわかる．約 12μsec 以下で一定と見なせるため，静的割り当て手法でのオブジェクト 1 個あたりの時間 $E_{sta}(\mu\text{sec})$ は次式で見積もることができる．

$$E_{sta} = \beta \qquad (\beta = 1.2) \tag{4.2}$$

チェックポインティングによるオーバヘッド

動的割り当て手法におけるチェックポインティングにかかる時間は，図 4.10 よりオブジェクトのサイズに対してほぼ一定であり，また図 4.12 からオブジェクトの数に対してほぼ線形増加となることがわかる．図 4.12 での傾きは約 5.64 であった．この処理ではオブジェクトの保存領域の破棄およびフラグの操作，スタックの保存，ページ操作ログの破棄，CPU の状態保存が行われる．このうちページ操作ログの破棄と CPU の状態保存にかかる処理は，その処理内容からほぼ一定と考えることができる．またスタックの保存作業はほぼ `memcpy` の作業であるため，スタックの量に係数 α で比例すると考えることができる．図 4.10 の値についてスタックサイズおよびオブジェクトの個数を無視すれば，CPU の状態保存およびページ操作ログの破棄にかかる時間は約 120μsec と多めに見積もることができる．よって保存が必要となるオブジェクトの数を n，スタックのサイズを P とすれば，動的割り当て手法におけるチェックポインティングの時間 $C_{dyn}(\mu\text{sec})$ は次式で見積もることができる．

$$C_{dyn} = a + \sum^{n} b + \alpha \times P \tag{4.3}$$
$$(\alpha = 0.1,\ a = 120,\ b = 5.7)$$

図 4.11 および図 4.12 より，静的割り当て手法によるチェックポインティングはオブジェクトの数およびオブジェクトのサイズに依存することがわかる．またそれぞれ傾きは式 (4.1) と同様に `memcpy` とほぼ同等と見なすことができる．動的割り当て手法との処理違いは，オブジェクト保存領域の破棄が実際の保存となるので，その他の作業時間は同一と考えることができる．よって，静的手法時のチェックポインティングにかかる時間 $C_{sta}(\mu\text{sec})$ は次のようになる．

$$C_{sta} = a + \sum^{n}(\alpha \times S_i) + \alpha \times P \tag{4.4}$$
$$(\alpha = 0.1,\ a = 120)$$

全体のオーバヘッド

以上，式 (4.1) および式 (4.3) より，動的割り当て手法によるオーバヘッド $O_{dyn}(\mu\text{sec})$ は，式 (4.1) がチェックポイント間でアクセスされるオブジェクト数に比例することから次式で見積もられる．

$$\begin{aligned} O_{dyn} &= \sum^{n} E_{dyn} + C_{dyn} \\ &= \alpha \sum^{n} S_i + a + bn + \alpha P \qquad (4.5) \\ &\quad (\alpha = 0.1,\ a = 120,\ b = 5.7) \end{aligned}$$

同様に式 (4.2) および式 (4.4) より，静的割り当て手法によるオーバヘッド $O_{sta}(\mu\text{sec})$ は次式のようになる．

$$\begin{aligned} O_{sta} &= \sum^{n} E_{sta} + C_{sta} \\ &= \beta n + a + \alpha \sum^{n} S_i + \alpha P \qquad (4.6) \\ &\quad (\alpha = 0.1,\ \beta = 1.2,\ a = 120) \end{aligned}$$

なお，両手法ともチェックポイント間で同一オブジェクトに対して複数回 write アクセスがあったとしても，保存作業は 1 度しか行われない．しかしここでは簡単化のため，チェックポイント間ではすべて別々のオブジェクトがアクセスされるものとして O_{dyn} と O_{sta} の導出を行っている．

4.3.3　MMU による手法との比較

提案手法との比較のため，オブジェクトへの write アクセス検出に MMU を用いた手法を実装し実験を行った．この手法の動作の概略は以下のようになる．

チェックポイント時に MMU ページテーブルの各ページの書き込みを禁止する．プログラムの実行中主記憶に対して書き込みが起こるとアクセスバイオレーションが生じる．アクセスバイオレーショントラップ内では，書き込みが行われた仮想ページに対して新たな物理ページをマップし，現在マップされている物理ページの内容をコピーする．また仮想ページアドレス，元マップされていた物理ページアドレス，新しくマップした物理ページアドレスをログとして保存する．電源切断が生じた場合には，このログをもとに元の物理ページをマップし直し，新しいページを解放して主記憶の状態を復元する．チェックポインティングではログを元に，元の物理ページを解放し再びページを書き込み禁止にする．MMU の操作に関して，これとほぼ同様の手法は [Lindstrom *et al.* 95] や [Shapiro *et al.* 99] などで用いられている．

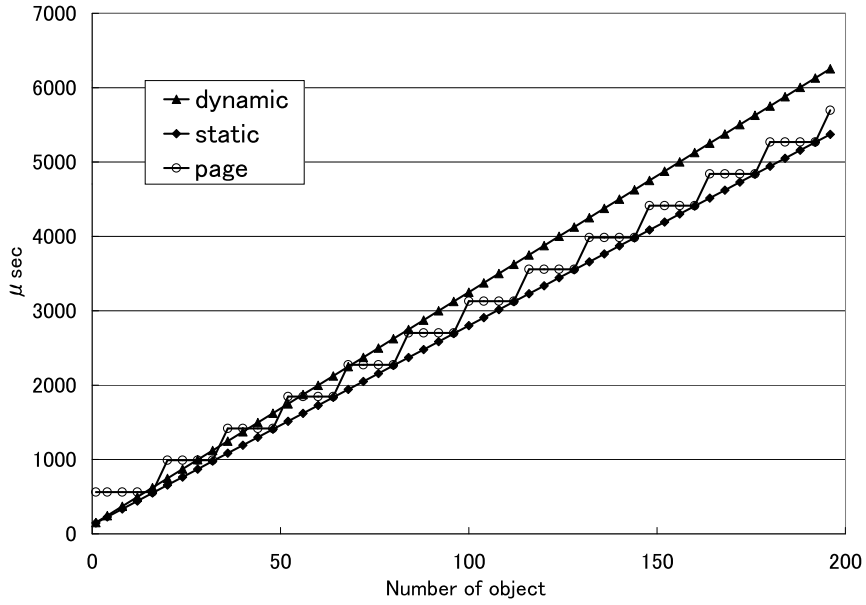

図 4.13: 見積もられるオーバヘッド

この手法でオーバヘッドとなるのはアクセスバイオレーショントラップ時の作業とチェックポインティングでの作業である．ページサイズを 4K バイトとして実験を行った結果，それぞれにかかる時間 E_{page}，C_{page} および全体のオーバヘッド O_{page}(それぞれ単位は μsec) は，アクセスバイオレーションが発生する (更新される) ページ数を q としたとき，以下のように得られた．

$$E_{page} = \gamma \tag{4.7}$$

$$C_{page} = c + d \times q + \alpha \times P \tag{4.8}$$

$$\begin{aligned} O_{page} &= \sum^{q} E_{page} + C_{page} \\ &= (\gamma + d)q + c + \alpha P \end{aligned} \tag{4.9}$$

$$(\alpha = 0.1,\ \gamma = 426.5,\ c = 134,\ d = 1.4)$$

この結果から提案手法と MMU を用いた手法との比較を行う．

メモリ上に 256byte のオブジェクトが連続して並んでいる状況を想定する．これらすべてのオブジェクトに対して連続的にアクセスが生じたと仮定したときの各オーバヘッドの見積りを図 4.13 に示す．q は $4096 \div 256$ よりオブジェクト 16 個毎に 1 増加させ，P(スタックサイズ) は 0 とした．図 4.13 より静的割り当て手法は，ほとんどの場合において MMU を用いた手法よりも低く見積もられることがわかる．動的割り当て手法については，オブジェクトの数が 50 を超えた程度から常に MMU を用いた手法よりもオーバヘッドが大きく見積もられている．しかし MMU を用い

表 4.2: ライフゲームの実行時間 (1000 世代)

サイズ	通常実行	動的割り当て手法				静的割り当て手法			
		実行時間	差	割合	O_{dyn}	実行時間	差	割合	O_{sta}
10×10	629	710	81	12.88%	151	684	55	8.47%	142
25×25	4,018	4,148	130	3.24%	256	4,140	121	3.01%	247
50×50	16,220	16,731	511	3.15%	631	16,697	477	2.94%	622
75×75	37,773	38,927	1,154	3.06%	1,256	38,942	1,169	3.09%	1,247
100×100	70,118	72,058	1,940	2.77%	2,131	72,044	1,926	2.75%	2,122

単位：msec

た方法では，たとえ数 byte のオブジェクトだったとしても，以前にアクセスが行われたページでなれば E_{page} のオーバヘッドが発生することとなる．例えば 256byte のオブジェクト 10 個が別々のページに存在していたとすれば，O_{dyn} は 433(μsec) であるのに対し，O_{page} は 4,413(μsec) となり，約 10 倍の差を持って見積もられる ($P = 0$ のとき)．

また，チェックポイント間で同一オブジェクトに対して繰り返しアクセスが行われたとする．静的割り当て手法の場合 $(O_{page} - C_{sta}) \div E_{sta}$ より 346 回以上のアクセス (挿入コードの繰り返し) がなければ $O_{page} > O_{sta}$ とはならない ($S = 256, n = 1$ のとき)．動的割り当て手法の場合にも 2 回目以降での処理は E_{sta} とほぼ同様の内容となることから $E_{dyn}(2\text{回目以降}) = 1.2$ とすれば，試算の結果は 343 回となる．

オブジェクトの配置やオブジェクトへのアクセスパターンはプログラムの内容やチェックポイントの挿入間隔に依存する．このため一概にいうことはできないが，一区切りの処理中でページ全域を利用することは稀であると考えられる．特にシステムソフトウェアではリスト構造などを用いてオブジェクトを管理することが一般的であり，対象となるオブジェクトがページに対して分散する確率が高いと考えられる．また細粒度でチェックポイントを挿入した場合，同一オブジェクトへのアクセスの重複は少なくなる．このため O_{page} よりも O_{dyn} や O_{sta} の方が小さくなる可能性が高いといえる．以上の結果から，本手法は MMU を用いた手法よりも高速に動作し，特に細粒度でシステムの状態保存を行う場合には有効であると考えられる．

4.3.4　オーバヘッドの割合の測定

実プログラムの実行に対するオーバヘッドの割合を観察するため，サンプルとしてカーネル内でライフゲームの実行を行った．マトリックスサイズは 10×10 から 100×100 まで変化させ，各世代の始めにチェックポイントを挿入した．表示などの処理は行わず，純粋に CPU の利用時間のみについて実行時間の測定を行った．

ライフゲームのような処理は実プログラムとしては稀であるが，サンプルとして選んだ理由は次のとおりである．世代毎に同様の処理が反復さるため，処理の量に対してチェックポイントを一定間隔で挿入しやすいこと．処理が簡単なため処理量を概算しやすいこと．コード量が少ないため一連の作業すべてが I-cache 内に収まり実際のプログラムよりもオーバヘッドの割合が大きく現れると考えられることなどである．

測定結果を表 4.2 に示す．マトリックス全域を一つのオブジェクトとして確保しており，世代毎に前世代の状態と現世代の状態を持つため，オブジェクトの数は 2 つである．スタックとしては 140byte 程用いられていたが，表中の O_{dyn} および O_{sta} は $P = 0$ として計算している．

ライフゲームの動作は，各世代毎に前世代の状態の保存 (マトリックスサイズのコピー) と，各要素毎に 10 回の単純な比較 (周囲の状況判断と生存判定) および現世代状態の作成となる．処理の量が少ないことから，オーバヘッドの割合が一番大きいのは 10×10 のときとなる．10×10 の場合の大体の処理量は，世代毎にのべ 300byte のメモリ操作 (read と write×2) と 1000 回の単純な比較作業と見積もることができる．また表中の実行時間を世代数で割った結果から，約 0.7msec 毎にチェックポイントが挿入されていると考えることができる．このときのオーバヘッドの割合は動的割り当て手法では約 12.3%，静的割り当て手法では約 8.5%であった．許容される範囲には議論の余地があるが，処理量が非常に少なく，0.7msec に 1 回という粒度でチェックポイントを挿入したときのオーバヘッドが 10%程度であれば，十分に低オーバヘッドでの状態保存の実現がされていると考えられる．

また，25×25 以降では両手法ともほぼ 3%で一定している．オブジェクト数が少ないためオブジェクト数に依存するオーバヘッドが表れ難くなっているが，オブジェクト毎のオーバヘッドの増加は式 (4.5) および式 (4.6) より，それぞれの手法において 5.7μsec および 1.2μsec と見積もられる．例えば動的割り当て手法を用いたときに，総オブジェクトサイズが 50×50 のときと同一 (のべ 5000byte) であるとすれば，オブジェクト 1 個あたりのオーバヘッドの増加率は約 0.04%となる．オブジェクトの数が 100 個となったとしてもオブジェクトのサイズがのべ 5000byte 程であれば，オーバヘッドの割合は 7%程と見積もられる．また 50×50 においては約 16msec 毎にチェックポイントが挿入されていると考えることができる．例えば linux での内部基準時間は 10msec であり，50×50 の結果は，ほぼコンテキストスイッチ毎にチェックポイントを挿入した場合と同等と考えることができる．このときの結果は約 3%程であり，本手法は短い時間間隔でチェックポイントを挿入したとしても，十分に小さなオーバヘッドでシステムの永続化が可能であることが示唆されている．

また 50×50 程度から見積り値と実際のオーバヘッドの値はよく一致している．さらに全ての場合において見積り値よりもオーバヘッドの値は小さくなっており，式によって見積もられる値およびそれらの値を元にした評価内容が信頼できるもので

あることが確認できる．

4.4　本章のまとめ

本章では，可観測かつ揮発な要素である CPU と可観測かつ不揮発な要素である主記憶について低オーバヘッドでの状態保存を可能とするための手法について述べた．揮発である CPU について復元される状態に合わせるため，不揮発である主記憶の状態についても保存を行い，これを復元した．

主記憶の保存に対し，無駄なコピー作業をなるべく排除するため，malloc などにより確保されるオブジェクトを単位としてその状態を保存した．また，構造化された主記憶管理手法を拡張することによって，実行中のチェックポインティング作業の低オーバヘッド化をはかった．CPU の保存領域を指す 2 つのポインタの切り替えにより，チェックポインティング中およびリカバリ作業中に電源切断が生じた場合にも，正しくシステムが復帰することを可能とした．

実際に各処理中に電源切断を生じさせた実験の結果，通常実行時，チェックポインティング時，リカバリ時に電源切断が発生したとしても，適切にシステムの状態が復元することを確認した．また測定および考察により，提案手法が低オーバヘッドでシステムの状態保存を可能としていることを示した．

第5章

周辺デバイスの状態復元

前章では可観測かつ揮発な要素であるCPUと可観測かつ不揮発な要素である主記憶について，低オーバヘッドで状態復元を可能とする手法を提案した．図5.1に示すように，本章ではこれを拡張し，不可観測かつ揮発な要素である，周辺デバイスの状態の復元手法について述べる．デバイスとデバイスドライバの関係に着目し，システム全体の実行状態の一貫性を維持した状態復元を行う．

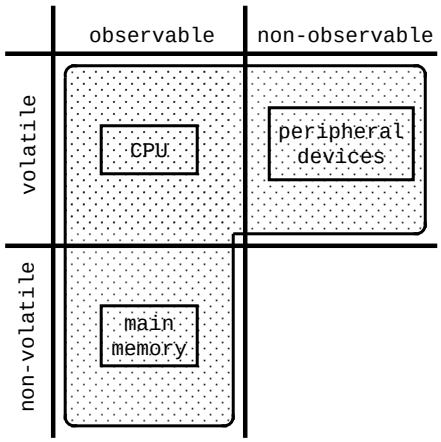

図 5.1: 5章で対象とする要素

5.1 基本方針

本章では，4章で行ったCPUと主記憶状態の復元を基本とし，周辺デバイスの状態復元を行う．このとき，2.3.3章で述べたように，周辺デバイスはそれぞれ独立に動作するものとし，デバイス間で相互作用によってその状態に依存関係が発生しないものとする．また，一つのデバイスは対応する一つのデバイスドライバによってその状態が制御されるものとする．

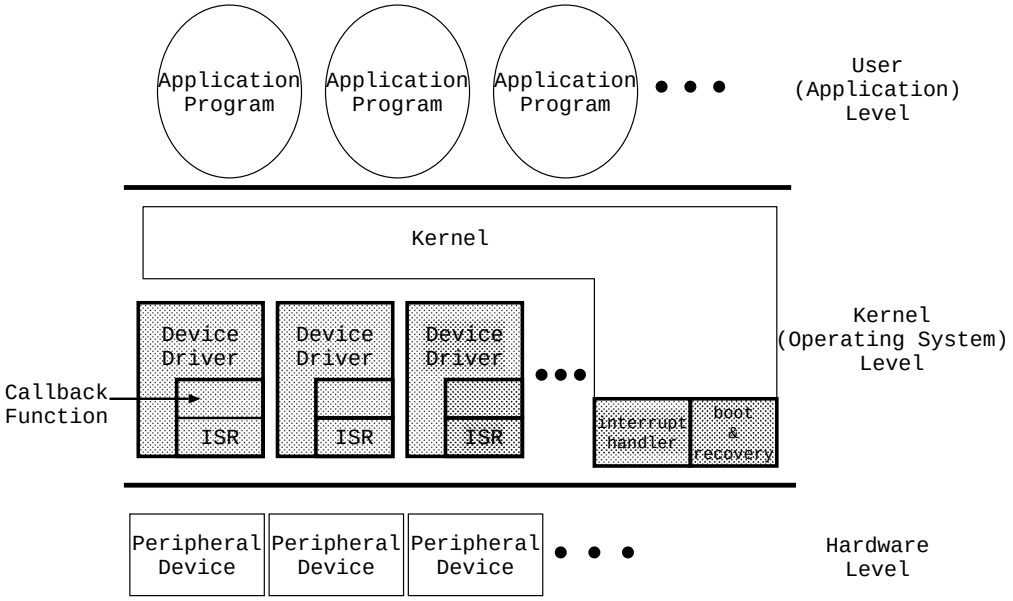

図 5.2: 実行状態復元に関するソフトウェア

複数の要素で構成されるシステムにおいて，各要素間に依存関係が発生しなければ，各要素が独立に状態を復元しシステム全体の一貫性を維持することが可能である．よって，まずデバイスとの依存関係が発生する要素であるデバイスドライバに注目する．そして，個々のデバイスとデバイスドライバの間で一貫性を維持するリカバリ手法を確立する．その手法をシステムに接続される各デバイスドライバにそれぞれ適用することによって，システム全体の状態を復元することができる．

図 5.2 に本手法における実行状態復元に関するソフトウェアコンポーネントを示す．図中，網がけとなっている部分が本章の手法で着目するソフトウェアコンポーネントである．

本章で提案する手法では，それぞれのデバイスの状態復元に対して，デバイスドライバにコードを追加し，デバイスドライバやデバイスに発行されるコマンドの情報の保存や復元を行う．デバイスドライバの通常処理に対して追加されるコードでは，リカバリ時のデバイスドライバとデバイスの間で一貫性を維持可能なように，必要となる情報を保存する．また，デバイスドライバにはコールバックルーチンを設ける．リカバリ時には，カーネルのブート／リカバリを扱うコードが各デバイスドライバのコールバック関数 (図 5.2 中 callback function) を呼出す．各コールバック関数では，デバイスドライバに追加されたコードによって通常実行中に保存された情報をもとに，各デバイスの状態を復元する．

本研究では，CPU および主記憶と，デバイスの間での一貫性を維持した状態を復元するにあたり，デバイスドライバとデバイスの間にメッセージパッシングシステムで用いられてきた一貫性の議論を適用する．

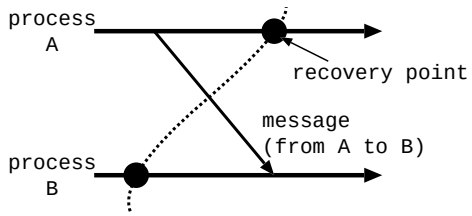

図 5.3: 一貫性が保たれる復元状態

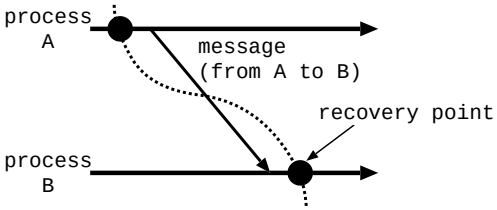

図 5.4: 矛盾を生じる復元状態

5.2　システムモデル

デバイスの状態の復元について，CPU および主記憶の状態とデバイスは一貫性を保ち，復元されなければならない．本研究ではデバイスドライバとデバイスをシステムを構成するプロセスとしてみなし，メッセージパッシングシステムでの一貫性の議論およびその方法論を適用する．そして，デバイスとデバイスドライバについて，一貫性が維持された実行状態の復元を行う．以下，メッセージパッシングシステムモデルでの一貫性維持について述べ，デバイスドライバとデバイスの動作について述べる．

5.2.1　メッセージパッシングシステムのモデル

最も単純には，復元される状態が全て同一時刻のものであれば，その状態は一貫性を保つことができる．このため，単一プロセッサシステムでは，4 章で述べた手法や，2.1 章述べたようにプログラムの実行をチェックポイントにおいて停止し，そのときの CPU や主記憶の状態を保存することが基本となる．しかし，分散システムにおいて各ノードの時刻を完全に同期させることは困難であり，同一時刻においてアプリケーションを構成する要素である全てのプロセスの状態を一斉に保存することはほぼ不可能であるといって良い．このため，特にメッセージパッシングシステムの実行状態の復元では，プロセス間で取り交わされるメッセージに着目し，その一貫性の維持がはかられる．

各プロセスにおいて，復元される状態がメッセージを横切る場合を図 5.3 および図 5.4 に示す．横の矢印は，各プロセスの実行を表わしている．斜めの矢印は，プロセス間でやり取りされるメッセージを表わしており，両図ともプロセス B がプロセス A からメッセージを受け取る状況を示している．図中の黒丸はそれぞれのプロセスの状態が復元される時点を示している．図 5.3 はプロセス A についてはメッセージを送信した後の状態，プロセス B についてはメッセージを受信する前の状態が復元される場合を表している．図 5.4 はプロセス A についてはメッセージを送信する前の状態，プロセス B がメッセージを受信した後の状態が復元される場合を示している．

これらの図において，図 5.3 は一貫性を保った復元状態となり，図 5.4 は矛盾が生じる (一貫性が保てない) 復元状態となる [Elnozahy *et al.* 99]．図 5.4 では，プロセス B についての復元状態がプロセス A が送ったメッセージを反映しているにもかかわらず，プロセス A の復元状態はそのメッセージを送る前の状態となり，これは，現実には起り得ない状態となるためである．それぞれの図で見られるように，各プロセスの復元される状態を結んだ線 (図中点線) に着目すれば，この線がメッセージと「X」の形で交差する場合には一貫性が保たれた状態であり，メッセージを追い越すような形で交差する線は一貫性が保てない状態となる．

なお，図 5.3 において，復元状態を示す線がメッセージを横切るということは，「メッセージは送られたが，まだ届いていない状態」が復元されるということを意味する．このようなメッセージはイン・トランジット・メッセージ (in-transit message) と呼ばれる．イン・トランジット・メッセージは各プロセスの実行状態の復元後，受信したプロセスに再度渡される必要が生じる．

これらの一貫性の議論をもとに，メッセージパッシングシステムおけるシステム全体の実行状態復元の実際の方法は，大きく以下の 2 種類に分類される．

- チェックポインティングベースのロールバックリカバリ
- ログベースのロールバックリカバリ

チェックポインティングベースのリカバリは，各プロセスで行われるチェックポインティングのみを用いる方法である．図 5.5 は，チェックポインティングベースのリカバリ手法における実行状態復元の例を示している．前述の例と同様に，各プロセスの実行は横の矢印で表し，メッセージは斜めの矢印で表している．各プロセスのチェックポイントは，図中黒丸で表している．通常実行中，各プロセスはそれぞれ自身の状態を保存 (チェックポインティング) する．あるプロセスに故障が生じリカバリが必要になったとき，リカバリ作業では一貫性が保たれる各プロセスのチェックポイントを選びだし，その状態まで各プロセスの実行状態を戻す (ロールバック)．図 5.5 の例では，前述の議論に基づき $(c^3_{\mathrm{A}}, c^2_{\mathrm{B}}, c^3_{\mathrm{C}})$ が一貫性が維持された状態となり，各プロセスについてこれらのチェックポイントの状態が復元されるこ

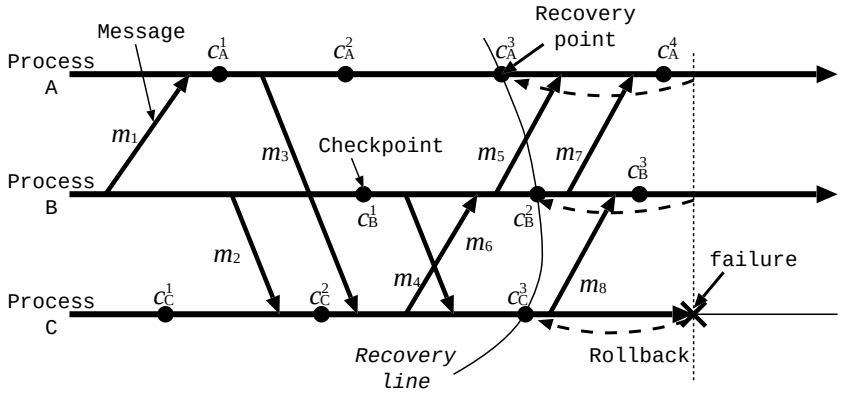

図 5.5: チェックポインティングベースのロールバックリカバリ

とになる．なお，チェックポインティングベースの手法では，リカバリ作業によって各プロセスが復元されるポイントは「リカバリポイント (recovery point)」と呼ばれる．また，各リカバリポイントを結んだ線は「リカバリライン (recovery line)」と呼ばれる．

ログベースのロールバックリカバリでは，各プロセスで行われるチェックポインティングに加えプロセス間で取り交わされるメッセージの情報をログとして保存する．リカバリ時には，このログを用いてメッセージを再生することにより，チェックポイント後の各プロセスの状態を再構築する．このときやはり再構築される状態は，一貫性が維持されたシステム状態である必要がある．

5.2.2　デバイスドライバとデバイスの動作

本研究では，デバイスの取り扱いに対し，以下の 3 つをプロセスとしてみなして前節で述べたメッセージパッシングシステムでの一貫性の議論を適用する．

- デバイス
- デバイスドライバ (通常実行)
- 割り込み処理

CPU とデバイスの関係を図 5.6 に示す．横の矢印は，それぞれ CPU とデバイスが動作していることを表している．デバイスの横の矢印のうち，細い部分はデバイスが停止している (idle 状態) ことを示す．また，縦の矢印は CPU からデバイスのレジスタへのアクセスを示し，それぞれ CPU からデバイスへ向うものは write，デバイスから CPU へ向うものは read アクセスを示している．割り込みも同様に，デバイスから CPU へ向う縦の矢印で示している．

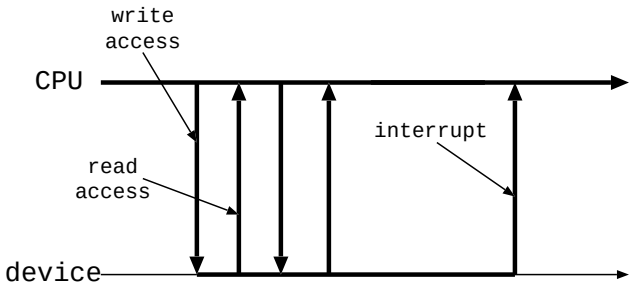

図 5.6: CPU とデバイスの動作

通常デバイスは CPU からのアクセスによってレジスタが設定された後，その動作を開始する．多くのデバイスは，CPU とは並列に動作し，その動作の完了は割り込みによって通知される．つまり，デバイスは CPU からのアクセスによってその状態が変化し，また，CPU と並列に動作している最中にも自身でその状態を変化させる．

図 5.6 は分散システムの図 5.5 に非常に良く似ている．CPU とデバイスが並列かつ独立に動作することは，分散システムにおける各プロセスの扱いと同様である．また，デバイスへのアクセスおよび割り込みによって，それぞれ CPU およびデバイスの状態が変化することも，分散システムにおけるメッセージの取り扱いと同様である．

これらのことから，デバイスとデバイスドライバの関係において，分散システムにおける一貫性の議論を適用する．つまり，デバイスドライバおよびデバイスについて，デバイスアクセス (メッセージ) を横切らないリカバリラインを構築することができれば，それは，一貫性の保たれた復元状態となる，ということである．なお，このことに関する妥当性については 6.1 節で述べる．

デバイスはデバイス自身でその状態を保存することができない，という点において，メッセージパッシングシステムと異なる．このため，2.3.3 節で述べたように，デバイスが停止している状態は CPU からの適切な再初期化処理によって復元可能とし，リカバリ処理によってその状態が復元されるものとする．また，メッセージパッシングシステムでは，図 5.3 のようにメッセージを横切った場合，リカバリ後に再生する必要があった．これは，メッセージパッシングシステムでは通信チャネルによってアプリケーションには透過に行われることが多かった．これは，デバイスと CPU の関係においては，リカバリ時に CPU が明示的にアクセスを再生することによって行うことができる．また，デバイスへのアクセスは即座にデバイスの状態に反映されるものとし，イン・トランジット・メッセージは考慮しない．

このようにモデル化したとき，デバイスドライバとデバイスの一貫性を保つためには，デバイスについて復元可能な状態と，CPU および主記憶 (デバイスドライバの実行) について復元可能な状態とを基準にして各アクセスを横切らないリカバリ

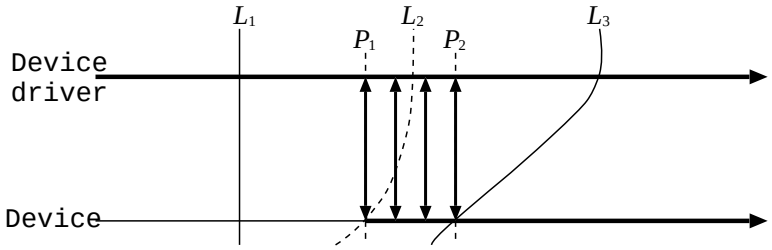

図 5.7: 基本的なデバイスアクセス

ラインを発見し，このリカバリラインを構築するために必要なデバイスドライバでの処理について考えれば良いことになる．以下，このモデルを用いて，デバイスおよびデバイスドライバ間での一貫性の維持のための処理について，検討を行う．

なお，デバイスドライバの実行のうち，割り込み処理についても一つのプロセスとみなしてモデル化する．なぜなら，割り込み処理は CPU の通常実行に対して非同期に起動され，また，そのコンテキストはデバイスドライバとは独立のものとなるためである．

5.3　解析

本節では，デバイスおよびデバイスドライバの実行をメッセージパッシングシステムとしてモデル化し，適切なリカバリラインを構築するために必要な処理について解析する．まず，純粋にデバイスドライバとデバイス間での一貫性の維持のための処理に対する要求事項を抽出するため，どのような時点でもデバイスドライバの状態の復元が可能であると仮定し，適切なリカバリラインを発見する．そして，そのリカバリラインを構築するために必要なデバイスドライバもしくはデバイスに対するリカバリ時の処理について検討する．実際のデバイスドライバの状態復元処理については 5.4 節で述べる．

5.3.1　デバイスへのコマンド発行

図 5.7 にモデル化したデバイスハンドリング時の様子を示す．図 5.6 と同様に，横の矢印は時間の経過と各要素の実行状態を表し，細い部分はその要素が実行されていないことを表す．これは，デバイスについては idle 状態を意味する．縦の矢印はデバイスへのアクセスを表す．read および write アクセスは混在する場合が多く，また，read によってデバイスがその状態を変更することも存在するため，上下両方を指す矢印として示してある．

P_1 において，デバイスドライバはデバイスへのコマンドを発行し始める．デバ

イスが持つレジスタの設定や読み込みを複数回行った後，P_2 においてそのコマンドの発行が終了する．そして，デバイスはコマンドに対応する実行を開始する．

P_1 以前のデバイスドライバの状態が復元される場合，このときのデバイスは idle 状態である．よって，リカバリラインは L_1 のようになりデバイスについてはこのときの idle 状態を復元する再初期化処理のみを行えば良い．

P_1 と P_2 の間のデバイスドライバの状態が復元される場合，デバイスについての復元可能な状態は P_1 での状態である．そのままデバイスドライバの実行を再開するとすれば，このときのリカバリラインは L_2 で示される点線となる．この場合，P_1 から状態復元時点までのアクセスによって変更されたデバイスの状態は失われ，デバイスはコマンドを正確に実行できなくなる．これは，リカバリラインがデバイスアクセスを横切るとき，リカバリ後のシステム状態の一貫性が維持できない，ということに対応する．

この解決には 2 通りの方法が考えられる．一つは，P_1 でのデバイスドライバの状態 (CPU，主記憶) およびデバイスの状態を復元し，実行を再び P_1 から開始する方法である．これは，リカバリラインを P_1 での垂直線にすることである．もう一つは，実行中に各アクセスのログを保存し，P_1 でのデバイス状態を復元した後に，ログを用いてアクセスをもう一度行う方法である．それぞれ 5.2.1 節で述べたチェックポインティングベースのリカバリ手法，ログベースのリカバリ手法に相当し，デバイスドライバの製作者が P_1 から P_2 に行われる処理内容に応じて選択すれば良い．ただし，チェックポインティングベースの手法を用いる場合には，P_1 から P_2 においてデバイスドライバ外へ影響を及ぼす処理が行われないことが条件となる．この理由は後に述べる．

なお，以降それぞれを「チェックポインティング手法」および「ロギング手法」と呼ぶ．4 章で述べたチェックポインティング手法はシステム全体を対象としたが，本章で述べるチェックポインティング手法はデバイスドライバ内においてデバイスアクセス中に更新される状態についてのみである．本章では，特に断らない限り後者についてチェックポインティング手法という言葉を用いる．

P_2 以降，デバイスが動作を開始した後のデバイスドライバの状態が復元される場合には，リカバリラインは L_3 に示す形が考えられる．P_1 から P_2 に行われるデバイスへのアクセスを保存しておき，P_1 におけるデバイスの状態を復元した後それらを再度実行すれば，P_2 でのデバイスの状態を復元できる．P_2 以降，デバイスドライバはデバイスとは無関係に動作するので，デバイスドライバの状態をそのまま使用することができる．

この方法はデバイスへのコマンドをログとしてみなしたロギング手法となる．デバイスドライバのみに着目すれば，チェックポインティング手法を用いて P_1 でのデバイスドライバの状態を復元し，リカバリラインを P_1 での垂直線とすることも考えられる．しかし，以下の理由のため，この場合にはあまり適さない．デバイスが動作している間，デバイスドライバはデバイスドライバ以外の要素と依存関係を

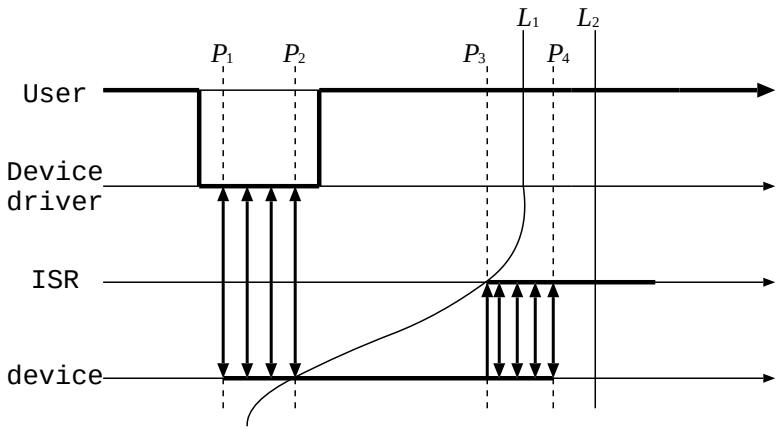

図 5.8: コマンドに対応する割り込み処理

生じる処理が行われる可能性が高い．このため，デバイスドライバの状態を P_1 にロールバックした場合，システム全体の一貫性の維持のためにはデバイスドライバと依存関係を生じた各要素の状態も P_1 時点に復元しなければならない．この問題は,「ロールバックプロパゲーション (Rollback propagation)」もしくは「ドミノ効果 (Domino effect)」と呼ばれる．最終的に，システム全体の一貫性を保証するためには，P_1 におけるシステム全体の状態を保存し，これを復元して実行を再開させなければならなくなる．よって，P_2 でのデバイスの状態を復元するには，P_1 から P_2 間のアクセスを保存し，リカバリ時に再実行する方法が適切である．以降，P_1 と P_2 間のアクセスを一塊として,「デバイスへのコマンド」と呼ぶ．

同様の議論は P_1 から P_2 の間についても適用される．前述したように，チェックポインティング手法を用いる場合には，P_1 と P_2 間で外部との依存関係が発生しないことが必要である．

5.3.2　割り込み処理

非同期に動作するデバイスの動作の完了は通常割り込みを用いて行われる．また，マウスやキーボードなど，デバイス側から発生するイベントも同様に割り込みを用いて CPU に通知される．それぞれの場合をモデル化した図を図 5.8 および図 5.9 に示す．

コマンド終了通知としての割り込み

図 5.8 は P_1 から P_2 にかけて発行されたコマンドの完了通知 (割り込み) が P_3 において発生した場合である．割り込みに応じて，対応するデバイスドライバ内の割

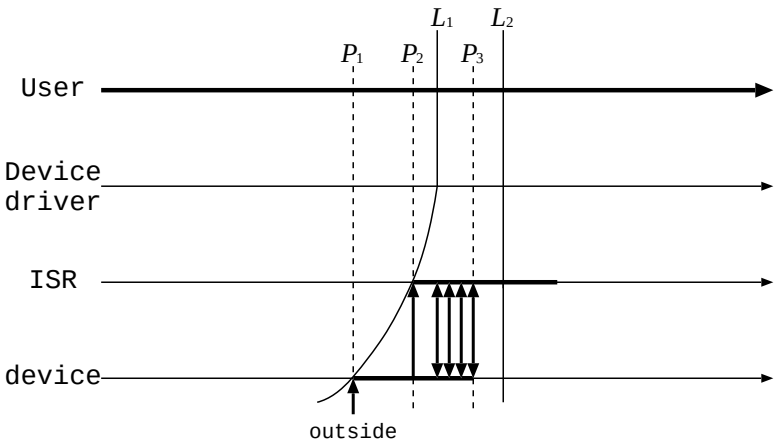

図 5.9: 外部イベントに対応する割り込み処理

り込みサービス処理 (ISR：Interrupt Service Routine) がカーネルより呼び出される．ISR の処理では，デバイスの制御やデバイスからのデータの取得などのためのアクセスが発生する (以降，このような処理を「コマンドの後処理」と呼ぶ).

P_3 から P_4 において，デバイスからデータを取得するための処理が行われるとすると，P_4 にてこの処理が完了する以前のデバイスドライバの状態が復元される場合，デバイスから取得されるデータは不完全なものとなる．このため，システムの再起動時 P_1 から P_2 で発行されるコマンドを再度実行し，デバイスが同じデータを保持するようにする必要がある．これは，前節で述べたように，保存された「デバイスへのコマンド」により行うことができる．ここから L_1 に示すリカバリラインが導き出される．そして，ISR では復元される時点までに行われた処理を無効化し，状態を割り込み発生以前 (P_3 以前) のものにする．つまり，ISR では P_3 での状態を保存しておき，リカバリ時に復元するようにする必要がある．

P_4 以降 ISR 内においてデバイスアクセス (コマンドの後処理) が終了した後には，各処理は CPU と主記憶の間で完結するため，デバイスドライバについて復元可能な時点からそのまま再実行させることが可能となる．このためリカバリラインは L_2 のようになる．このときもはやデバイスの状態を P_2 に戻す必要はなくなるので，保存されていたデバイスへのコマンドは P_4 以降，最初にデバイスドライバが復元可能となる時点で破棄することができる．

イベント発生通知としての割り込み

図 5.9 は P_1 においてキーボードの入力やパケットの到着などのイベントが発生し，P_2 において CPU に対して割り込みが発生した場合である．P_1 と P_2 は等しくても構わない.

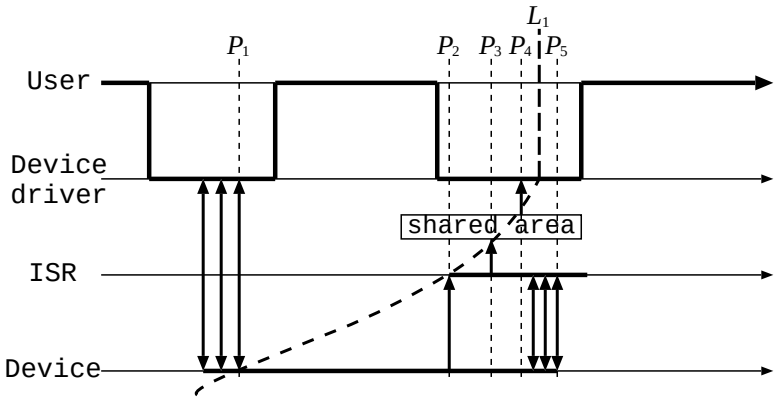

図 5.10: 問題を生じる場合

P_2 と P_3 のデバイスドライバの状態が復元される場合，デバイスが保持するデータはまだ取得されていない．また，デバイスの状態は失われ P_1 時点のものしか復元できない．このときのリカバリラインは L_1 となり，前述の議論と同様に ISR の状態は P_2 以前に戻す必要がある．

P_1 以降のデバイスの状態を復元するための情報は，CPU 側からは取得および保存する術がないため，このとき生じた割り込みはシステムのリカバリ後完全に失われることとなる．しかし，このようなイベントは，もともと生じるかどうか不確定であるため，無視できる場合が多い．キーボードやマウスの入力がこれにあたる．ネットワークパケットの Ack のように，予め期待されるものも存在するが，これらについてはより上位層の問題となる．このことについての議論は 6.1.5 節で述べる．

なお，この場合にも，デバイスアクセスが終了する P_3 以降では，リカバリラインは L_2 のようになり，デバイスドライバの復元可能の状態からそのまま実行を再開することが可能である．

5.3.3　共有領域のアクセス

前節にて，ISR の状態はロールバックされる必要がある場合があることについて述べた．5.3.1 節でも述べたように，ロールバックされる要素が存在する場合，ロールバックプロパゲーションの可能性について考慮しなければならない．

図 5.10 は ISR がロールバックされるときにおいて，この状況が発生する場合である．ISR 内でデバイスへのアクセスが終了する (P_5) 以前に，P_3 において ISR が共有データに対して書き込みを行い，かつ P_4 においてデバイスドライバがそのデータを読み込み作業を行ったとする．このとき，ISR とデバイスドライバの状態に依存関係が発生する．そして，P_4 と P_5 の間のデバイスドライバの状態が復元されるとすれば，ISR の状態は P_2 にロールバックされるため，デバイスドライバの状態

も P_4 以前に戻さなければならない．

問題の解決には，ロールバックされる範囲内で ISR とデバイスドライバの間に依存関係が発生しないようにすれば良い．このことから，一つの解決策は，ISR から共有領域への write アクセスは，デバイスへのアクセスが終了した後に行うように ISR をプログラムすることである．それが困難な場合，ISR が共有領域に関連する同期プリミティブのロックをデバイスのアクセスが終了するまで保持し続けるようにする．これによって，デバイスドライバからの共有領域への Read アクセスを P_5 以降まで行われないようにすれば良い．

5.3.4　解析のまとめ

これまでに述べた解析をまとめると以下のようになる．

1. コマンド発行中のデバイスドライバの状態が復元される場合には，デバイスへのアクセスは各アクセス毎にログをとるか，デバイスアクセス前にチェックポインティングを行う
2. デバイスへ発行されるコマンドを保存する
3. 保存されたデバイスへのコマンドは，そのコマンドの後処理が終了し，最初にデバイスドライバの状態が復元可能となる時点で破棄することができる
4. ISR 内でデバイスへのアクセスが完了する前のデバイスドライバの状態が復元される場合，ISR の状態を割り込み発生以前にロールバックさせる
5. ISR 内で他の要素との共有領域へ write アクセスを行う場合，デバイスアクセス終了後に行うようにするか，ISR がデバイスアクセス終了までロックを保持し続けるようにする

5.4　デバイスドライバの状態復元

前節で行った解析は，デバイスドライバの状態が復元される時点について，特別な考慮は行わなかった．以下，実際のデバイスドライバの状態の復元時点について述べる．

4 章の手法の適用

4 章では，システム全体に対して CPU の状態および主記憶の状態を保存し，これらの状態の復元を行った．このとき，状態保存は明示的にプログラマによって指

定されるため，自由にその瞬間を決定することができる．前節で行った解析の結果，デバイスドライバに対してその制御が必要となるのはコマンドの発行中および ISR の処理中 (コマンドの後処理が終了するまで) である．よって，これらの期間以外でシステム全体のチェックポインティングを行うようにすれば，デバイスドライバに対する特別な制御をすることなく実行状態の復元が可能となる．ただし，デバイスの動作中に，他の要素の実行によってシステム全体のチェックポインティングが行われる場合に対応するため，デバイスへのコマンドの再発行は行うことができるようにしておく必要がある．このため，5.3.4 節でまとめた事項のうち，2. は必須な処理となる．コマンドのログの破棄は，コマンドの後処理後，最初のシステム全体のチェックポインティングによって行うようにする．

なお，コンテキストスイッチなどによって，デバイスへのコマンド発行中や ISR の処理中にシステム全体のチェックポインティングが行われる場合には，これまでに述べた解析の結果を組み合わせて行うことで対応することができる．このときには，システム全体のチェックポインティングとは別に，それぞれ個別のデバイスドライバに対して前節での解析結果を適用する．リカバリ処理ではシステムの主記憶状態を復元し，その後，デバイスドライバについてデバイスアクセス前へのロールバックやデバイスアクセスログの再実行，ISR のロールバックを行うようにすれば良い．

電源切断時の CPU 状態の保存

また，もし電源切断時に CPU の状態を保存することが可能であれば，電源切断時の CPU 状態，主記憶状態は一貫性を維持したまま復元することが可能となる．

例えば，図 5.11 に示すように，電源を監視する IC を用いてシステムに供給される電源が一定値以下になった場合に CPU に対して割り込みをかける．電源切断時の電力の減衰はある程度の期間を持つため，この割り込み処理において，特定の領域に CPU の状態を保存することができればシステムの電源切断時の CPU 状態および主記憶状態が復元可能となる．

このようにして，電源切断時の CPU と主記憶の状態を回復可能であることが可能であるとすれば，デバイスドライバについて復元可能な状態は，まさに電源切断時のものとすることができる．そして，5.3.4 節で述べた解析の結果をそのまま適用することができる．

多少のハードウェアの変更が必要とはなるものの，主記憶の保存が必要なくなることによる低オーバヘッド化も見込むことができるため，以下，電源切断時に CPU の状態が保存可能であることを前提として議論を進める．

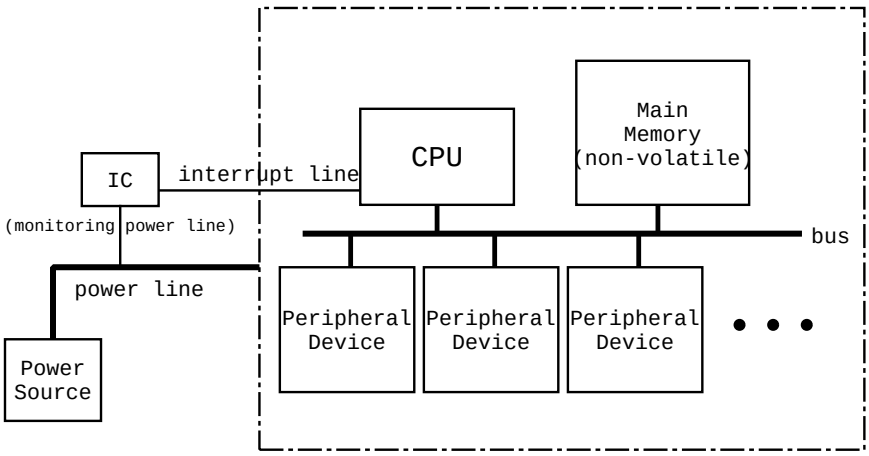

図 5.11: 電源切断時に CPU 状態の保存を行う場合のハードウェア構成

5.5　デバイスドライバへの適用

本節では，図 5.12 に示す例を用い，5.3.4 節で総括した事項のデバイスドライバへの適用方法について述べる．図 5.12 中，`write`，`read`，`ioctl` の各メソッドの役割は，POSIX で規定される同名の API の機能と同一である．また，`interrupt` メソッドは割り込み発生時に実行される ISR である．

なお，本節での議論は，電源切断時に CPU の状態が保存可能であるものとして行う．

5.5.1　idle 状態の復元

まず，idle 時のデバイスの状態を復元できるようにしなければならない．これは基本的にはデバイスの初期化作業を行えば良い．しかし，`ioctl` システムコールのように，明示的なデバイスの動作コマンドとは別にデバイスの状態を変化させる場合がある．例えば UART デバイスのボーレート設定などが該当する．

これらの設定を復元するため，デバイスの状態を変更する際に，変更後のデバイスの状態をリカバリ時に参照可能な領域へ保存しておく必要がある (図 5.12 中 26 行目)．リカバリ時にはこの情報を参照し，デバイスの状態の初期化と再設定を行うようにする．なお，コマンドの実行などがデバイスの idle 状態に影響する場合も，同様にリカバリ時に参照可能な領域へその情報を保存し対応する．

```
 1  void write(void* request){
 2    if(idle == get_device_state()){
 3      DEV_ACCESS_START();
 4      tell_device_request(request);
 5      DEV_REQUEST_SAVE(request_log,request);
 6      DEV_ACCESS_END();
 7    }else{
 8      lock(request_queue);
 9      enqueue(request_queue,request);
10      request_count++;
11      unlock(request_queue);
12    }
13  }
14  void read(void* return_buf){
15    lock(event_buf);
16    while(!event_buf_count){
17      unlock(event_buf);
18      sleep(read_wait);
19      lock(event_buf);
20    }
21    memcpy(return_buf,event_buf);
22    event_buf_count--;
23    unlock(event_buf);
24  }
25  void ioctl(void* new_state){
26    dev_state = *new_state;
27    device_state_change(new_state);
28  }
29  void interrupt(){
30    do{
31      if(request_done & interrupt_reason()){
32        ISR_STATE_SAVE();
33        post_processing_for_request();
34        ISR_STATE_DISCARD();
35        DEV_REQUEST_DISCARD(request_log);
36        lock(request_queue);
37        if(request_count){
38          request = dequeue(request_queue);
39          DEV_ACCESS_START();
40          tell_device_request(request);
41          request_count--;
42          DEV_REQUEST_SAVE(request_log,request);
43          DEV_ACCESS_END();
44        }
45        unlock(request_queue);
46      }
47      if(event_arose & interrupt_reason()){
48        ISR_STATE_SAVE();
49        lock(event_buf);
50        data = get(event_buf);
51        post_processing_for_event(&data);
52        event_buf_count++;
53        ISR_STATE_DISCARD();
54        unlock(event_buf);
55        if(someone_is(read_wait))
56          wakeup(read_wait);
57      }
58    }while(interrupt_reason());
59  }
```

図 5.12: コード例

5.5.2　デバイスアクセス

デバイスへのコマンドを発行するためのアクセスは，チェックポインティング手法もしくはロギング手法によって対応することを述べた．

チェックポインティング手法の場合には，デバイスへのアクセス開始時にデバイスドライバの主記憶状態，CPU 状態の保存を行う．そして，アクセスが終了した時点でそれらの情報を破棄する．リカバリ時には，これらの情報が存在するかどうかを調べる．存在する場合には，保存された主記憶状態の復元を行った後，このときの CPU 状態を用いてシステムの実行を再開すれば良い．

このとき，主記憶状態の保存については，デバイスドライバ内全ての領域を保存する必要はない．デバイスアクセス開始時の状態が復元できれば良いため，デバイスアクセスが終了するまでに変更され，かつ復元が必要となる主記憶の状態を保存すれば良い．

ロギング手法を用いる場合には，各デバイスアクセス毎に，どのようなアクセスが行われるかを示すログをとる．そしてデバイスアクセスが終了した時点でこれらのログを破棄する．リカバリ時には，ログの存在を確認し，存在する場合には，それらのログを実行する．そして，電源切断時の CPU 状態を用いてシステムの実行を再開すれば良い．

図 5.12 中 DEV_ACCESS_START および DEV_ACCESS_END (3，6，39，43 行目) は，それぞれ，この議論においてデバイスへのアクセスの開始とデバイスへのアクセスの終了に対応する．ロギング手法では DEV_ACCESS_START において何も行う必要はなく，tell_device_request 関数内でログをとるための変更を加える．

5.5.3　デバイスへのコマンドの保存

デバイスへのコマンドは，デバイスへのアクセスが終了した後，かつこれらアクセスを復元するための情報の破棄 (DEV_ACCESS_END) が行われる前に保存する必要がある．アクセスの情報を破棄した後，コマンドを保存したとする．もしこれらの処理の間で電源切断が生じたとすると，アクセスを再実行するための情報は失われ，かつデバイスへのコマンドもまだ保存されていないこととなる．このため，デバイスの実行状態を復元できなくなってしまう．

また，5.3.2 節で述べたように，この情報が破棄可能となるのは ISR 内となる．ISR のロールバックと関連するため，破棄作業の注意点については次節に述べる．

5.3.1 節では，デバイスドライバのコンテキスト内でコマンドが発行される場合を述べたが，デバイスへのコマンドは ISR の中でも発行される場合がある．デバイスが busy 状態のときに新たなデバイスへのコマンドが発行された場合，デバイスドライバのコンテキストではそのコマンドをキューイングすることがある．キューイングされたコマンドは，実行中のコマンドが終了したとき，ISR の中でデバイス

に発行される．しかし，ISR 内で実行されたとしても，ロールバックされる範囲でなければ，5.3.1 節で述べた議論と同一となる．つまり，通常実行時と同じ対応を ISR 内で行えば良い．このことは図 5.12 中，8〜11 行および 38〜43 行に対応する．

図 5.12 中，コマンドのログをとる処理に対応するのが DEV_REQUEST_SAVE(5，42 行目) である．また，ログの破棄に対応するのは DEV_REQUEST_DISCARD(35 行目) である．

5.5.4　ISR のロールバック

ISR の中でデバイスアクセスが終了する以前に電源切断が生じた場合，ISR の状態をロールバックしなければならない．このため，ISR が実行を開始する前に ISR の主記憶状態を保存する必要がある．これは，デバイスへのアクセスが終了した時点で破棄可能となる．

図 5.12 中，ISR の状態保存を行う処理に対応するのが ISR_SATE_SAVE(32，48 行目)，保存された情報を破棄する処理が ISR_STATE_DISCARD(34，53 行目) である．

これらの処理は，デバイスアクセスをチェックポインティング手法で復元させる場合の対応と類似するが，復帰後 ISR を再実行する必要はないため，CPU の状態を保存する必要はない．リカバリ時には，カーネルが割り込み受け付けたときに保存した CPU 状態 (割り込み時に実行されていたコンテキスト) を用いてシステムを再開させれば良い．

再実行が必要なデバイスへのコマンドについて，ISR がロールバックされるときに失われないようにする必要がある．このため，この情報の破棄は，ISR の情報が破棄された後に行うようにする (35 行目)．もしくは，ISR の状態がロールバックされたときに，破棄されたコマンドのログも復元するようにする必要がある．

ISR は複数の要因で起動されることが多い．例えば，ネットワークに対する送信終了割り込みと受信割り込みは一般に同一の ISR で処理される．このため，ISR では，割り込み要因の特定を行った後，対応する割り込みを処理する．また，ISR 実行中に割り込みが発生した場合に備えて，ISR の処理は割り込み要因が無くなるまでループする場合がある．

ロールバックされる範囲はそれぞれの割り込み要因の特定がなされてからのものとする．例えば，コマンドの終了とイベントによる割り込みが同時に発生したとする．このとき ISR が起動され，コマンドの後処理終了に続いて，イベントの処理が行われることになる．コマンドの後処理が終り，50 行目の処理中に電源切断が生じたとすれば，この場合ロールバックされなければならない内容はイベント処理に関するもののみであり，コマンドの後処理までロールバックする必要はない．よって，ISR_STATE_SAVE と ISR_STATE_DISCARD は図 5.12 のように挿入される．(32 と 34 行目および 48 と 53 行目)

このとき，ISR 内で割り込み要因を特定するときには,「現在の状態」を確認する

```
1  int recovery_call_back(CPU_t **cpu_state){
2    initialize_device(dev_state);
3    recover_memory_area();
4    replay_request();
5    return flag;
6  }
```

図 5.13: コールバック関数の例

必要がある．同上の例において，38 行目で電源切断が生じたとする．リカバリ後 38 行目以降の処理が継続されるが，デバイスの状態は失われているため電源切断前のイベントの処理を続けて行うことはできない．このため，復帰後には 31，47 行目や 58 行目でその時点の割り込み状態の確認がなされるようにする必要がある．

49 行目と 54 行目のロックの操作は 5.3.3 節で述べた議論に対応するものである．50 行目で read メソッドとの共有領域である event_buf をアクセスする．このため 49 行目で取得したロックは 54 行目まで保持される．ただし，ISR とデバイスドライバのコンテキストが同時に実行しないことが分かっている場合には，これらの操作は必要ない．

5.5.5　リカバリ

コールバック関数の例を図 5.13 に示す．この関数は対応するデバイスの状態を復元するためにリカバリ時にカーネルから呼び出される．

前節までの議論から，この関数では主に，デバイスの再初期化，主記憶状態の復元，デバイスへのコマンドの再実行を行うことになる．まず，5.5.1 節で述べたように，デバイスの再初期化は通常実行中保存された情報を元に，idle 時のデバイス状態を復元する (2 行目)．次に，DEV_ACCESS_START(チェックポインティング手法の場合) や，ISR_STATE_SAVE で保存された主記憶状態が存在する場合にはそれぞれを復元する (3 行目)．そして，デバイスアクセスのログ (ロギング手法の場合) や再実行すべきデバイスへのコマンドが存在する場合にはこれらを実行する (4 行目)．

また，リカバリ時には ISR_STATE_SAVE で保存された情報の破棄を行なう必要がある．これは，ISR の状態を復元した後に行えば良い．DEV_ACCESS_START での主記憶状態やログについては，リカバリ後の通常動作によって破棄されるため，この関数内では何も行う必要はない．

5.5.2 節や 5.5.4 節で述べたように，ロールバックが必要となる範囲で電源切断が生じた場合，システムを再開させるための CPU 状態をカーネルが把握する必要がある．このため，このメソッドは図 5.13 に示すように，引数として CPU 状態の構造体の参照受け渡しや，フラグを返戻値として返すようにする．

例えば，フラグは ISR 内で電源切断が生じた場合にセットする．この場合，カーネルは割り込み発生時に保存された CPU 状態を用いて復元する．また，CPU 状態

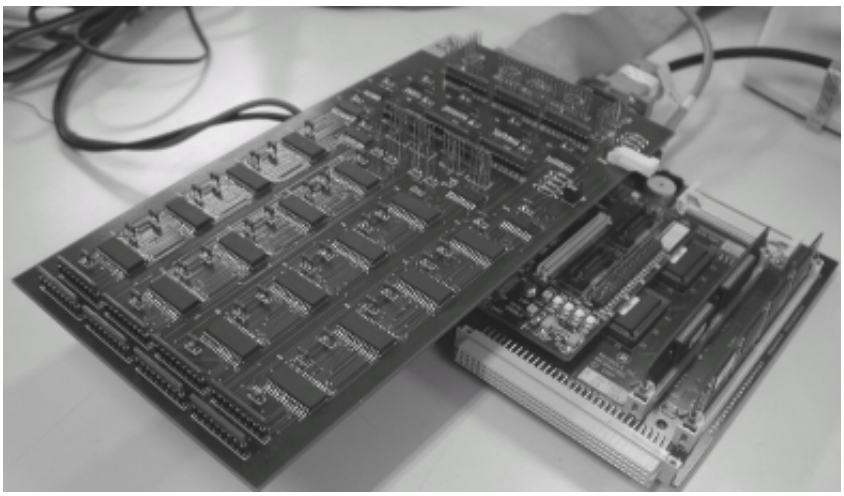

図 5.14: FeRAM ボードと MPC860FADS

には，DEV_ACCESS_START で保存された CPU 状態が存在する場合にはこの情報を返すようにする．カーネルはこれを参照し，システムを復元する．

なお，これまでの議論は基本的な場合について述べたものである．例えば，コマンドの再実行が必要ないようなデバイスでは，リカバリ時にコマンドを再実行しなくとも良い．また，デバイスへのコマンド自体を保存しなくても構わない．これまでの議論をもとに，対象となるデバイスの特徴に応じてこれまで述べた方法を応用し，実際のデバイスドライバに適用すれば良い．

5.6　実験

本章で提案する手法を linux 2.4.4 カーネルに対し適用した．CPU はモトローラ社の MPC860(40MHz) を用いた．また FeRAM を用いた不揮発 RAM ボードを作成し，これを主記憶として用いた (図 5.14)．デバイスとしては，オンチップ上に存在する UART(9600bps) および Ethernet(10Mbps) コントローラを用いた．また，図 5.11 に示したように，電源ラインを監視する IC をとりつけ，一定値以下の電圧になったときに割り込みを発生させるようにした．この割り込み処理の中では CPU の状態を保存し，リカバリ時にはこの CPU の状態を参照することができるようにした．

まず，提案手法に基づいてシステムの実行状態の復元が可能であるかどうかを調べる実験を行った．システム上でデバイスを利用するプロセスを実行している最中に，故意にシステムの電源の切断／復帰を行いシステムの挙動を観察した．

次に，提案手法がシステムのパフォーマンスに与える影響について調べた．デバイスを高頻度で扱うプロセスを走らせ，提案手法を適用しない場合，および適用し

た場合について，プロセスの実行時間を計測した．

以下，それぞれの結果について述べる．

5.6.1　実行状態復元の確認

提案手法を実装したシステムにおいて，システム上で UART デバイスに対して出力を続けるプロセスを実行し，故意に電源を切断した．そして電源を再度入れシステムを再起動した．

このとき，走行していたプロセスの実行が復元され，再び UART デバイスに出力を開始することを確認した．これは，システムの実行状態が適切に復元されたことによるものである．また，この実験中 UART デバイスから電源切断直前の文字が出力される場合があることを確認した．これは，UART デバイスに対して，コマンドの再実行が適切に行われた結果といえる．

また，実験ではルートファイルシステムに NFS(バージョン 2) を用いたが，システム再起動後，電源切断以前と同様にファイルに対してアクセス可能であることを確認した．さらに，NFS 上のファイルのコピー作業中に電源を切断した場合にも，リカバリ後ファイルのコピーが継続することを確認した．

さらにより詳細にデバイスアクセスの復元や ISR の状態復元が適切に動作していることを調べるための実験を行った．システム上の最も高いレベルの割り込みにプッシュボタンを取りつけ，この割り込みを仮想的な電源切断として，このときの CPU 状態を保存するようにした．そして，システムの実行中にこの割り込みを発生させた．割り込み発生時のアドレスを確認し，一度システムの電源を切断した後再度システムの電源を投入してシステムを再起動させた．

UART デバイス，Ethernet デバイス両方について，デバイスアクセス中および ISR 処理中に割り込みが発生した場合でも，システムが適切に処理を継続することを確認した．このことから，デバイスへのアクセスの復帰や，ISR の状態復元処理が適切に行われているといえる．

5.6.2　オーバヘッドの測定

実装したシステム上での提案手法によるオーバヘッドの測定を行うため，以下の測定を行った．

測定 1 `ioctl` メソッドの中にチェックポインティングを行う動作，およびデバイスアクセスのログを保存する動作を行う処理を設け，ユーザレベルから `ioctl` システムコールを用いてその完了にかかる時間を測定した．1 バイトから 4096 バイトのメモリ領域に対して書き込みを行った場合の `ioctl` システムコールにかかる時間を基準にし，チェックポインティング手法の場合には，この動作

に加えて CPU の保存と同サイズのメモリ領域のコピー作業にかかった時間を測定した．ロギング手法の場合には，この動作に加えて，同サイズ分 (16 バイトであれば 16 回) ログを保存する処理を行った場合の時間を測定した．基準からのそれぞれの時間の増加分をオーバヘッドの割合として求めた．

測定 2 測定 1 に加え，各 `ioctl` メソッドでの処理に対し，`udelay` を挿入した．`udelay` は正確ではないもののマイクロ秒単位での遅延を生じる関数である．この遅延をデバイスの動作と見立て，チェックポインティング手法について 256 バイト，1024 バイト，4096 バイト，ロギング手法において 4 アクセス，16 アクセスが保存される場合に対し，udelay による遅延を 0 から 1000(μsec) まで変化させた．そして，測定 1 と同様に，チェックポインティング手法もロギング手法も用いないときの時間を基準とし，実行の増加分をオーバヘッドの割合として求めた．

測定 3 実際のアプリケーションによるデバイスのアクセスに対するオーバヘッドを測定するため，提案手法を用いない場合 (通常実行)，提案手法においてデバイスのアクセスに対しチェックポインティング手法を用いた場合，同じくロギング手法を用いた場合それぞれについて 2 つのタスクを実行した．タスク A は，UART デバイスに対し 1 文字を出力する動作を 10000 回繰り返すプロセスを実行した．タスク B は NFS 上の 2Mbyte のファイルを cp コマンドによりコピーした．コピー先も同様に NFS 上とした．これらのタスクの実行時間を測定した．

それぞれの結果を図 5.15，図 5.16，図 5.17 に示す．なお，測定 1 および測定 2 では `ioctl` システムコールの発行は 100,000 回繰り返して `gettimeofday` によって開始時刻と終了時刻の差を求めた．測定 3 は time コマンドによって実行時間を測定した．

図 5.15 の結果から，デバイスへのコマンド発行においてその処理が 16 回のアクセスであればロギング手法を用いた方がオーバヘッドが少なくてすみ，それ以上の場合にはチェックポインティング手法を用いた方が良いことがわかる．チェックポインティング手法は，デバイスアクセス間で更新される主記憶の情報を保存するものであり，一方，ロギング手法は，デバイスアクセス自体を保存する手法である．このため，一概に比較はできないが，チェックポインティング手法の場合，約 20%から 30%程のオーバヘッドで推移しており，また，ロギング手法の場合には，2 次関数的に増加していることがみてとれる．

図 5.16 では，1000μsec の遅延を挿入したときにはチェックポインティング手法で 4096byte を保存したとき以外は 10%以下におさまってはいるものの，100μsec の遅延を挿入したときには 10%〜30%程度とそのオーバヘッドはかなり大きいものとなってしまっている．ここで，1000μsec の遅延は，9600bps の UART デバイスに

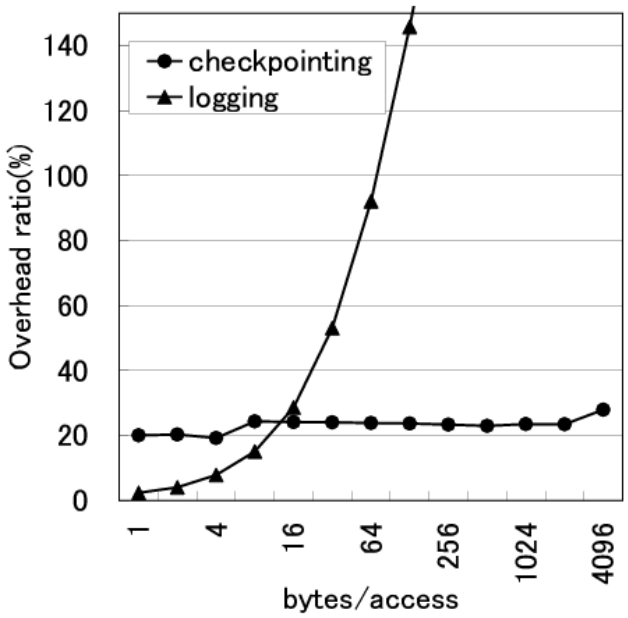

図 5.15: オーバヘッドの割合

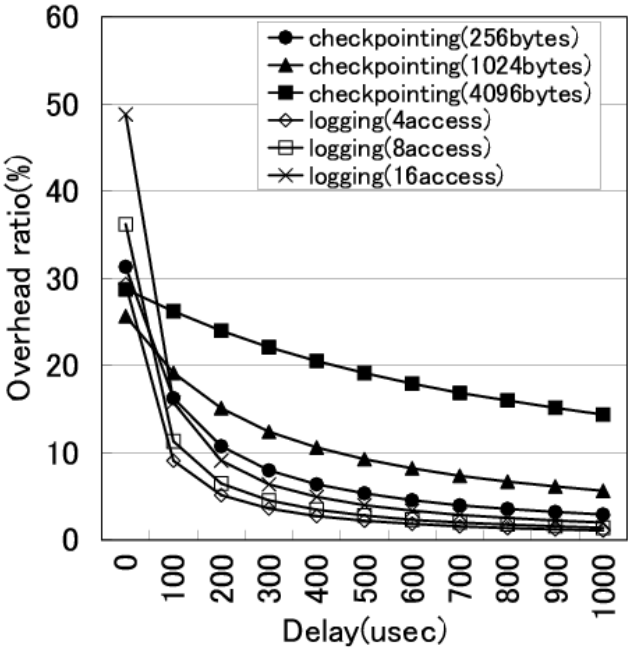

図 5.16: 遅延時間とオーバヘッドの割合の関係

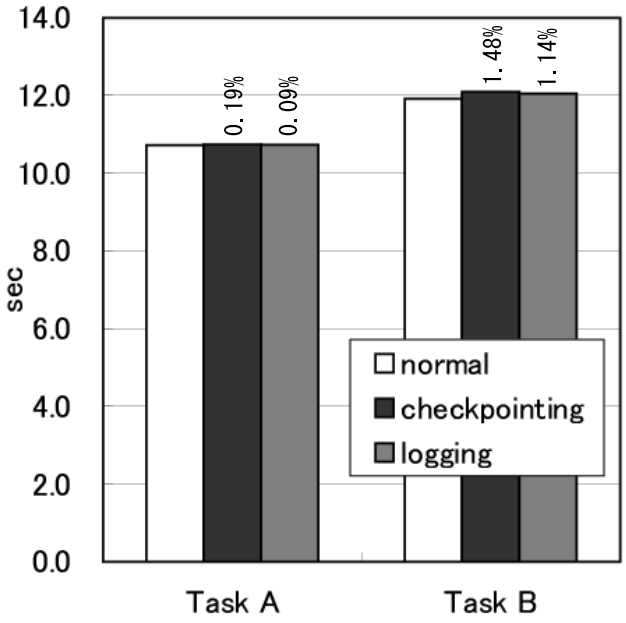

図 5.17: 各タスクの実行時間

対し一文字送信する速度や，10Mbps の Ethernet で 1000byte 程度のパケットを送信する時間と考えることができる．

図 5.17 は，9600bps の UART デバイス，10Mbps の Ethernet デバイスを用いたときの実際のアプリケーションの実行を含めた実行時間を示している．図 5.17 からは，通常実行に対し，チェックポインティング手法，ロギング手法いずれを用いた場合にも実行時間の増加はほとんど見ることができない．具体的には，そのオーバヘッド率は Task A についてはチェックポインティング手法が 0.19%，ロギング手法が 0.09%，Task B はそれぞれ 1.48%，1.14%となった．

実際のデバイスドライバでは，ほぼチェックポインティング手法では 256byte，ロギング手法で 8 アクセス程度の保存作業となる．それぞれ，図 5.16 の結果からは，約 3%と約 2%程と得られており，実際のオーバヘッドは小さくなっている．このため，図 5.16 から得られたオーバヘッドは，実際にはより小さくなると考えられる．この原因としては，カーネル内での計算処理や，アプリケーション自体が行う計算処理などが考えられる．また，図 5.16 の測定では `ioctl` のみを繰り返す方法で測定がなされた．このため実際のアプリケーションでのデバイスアクセス頻度はより低下するものと考えられる．さらに，特にチェックポインティング手法では，デバイスアクセスを一つの関数としてまとめローカル変数の保存を行う必要を無くすなど，デバイスドライバの実装を工夫することによって保存量の低減が可能であり，4096byte ものデータを保存する必要がある場合は稀であると考えられる．

例えば 100Mbps の Ethernet では，100μsec 程度の遅延を挿入したときと同程度と考えることができる．前述のように，このときのオーバヘッドはチェックポインティング手法で 4096byte を保存する以外でも 10〜20%と得られているが，以上述べた理由により実アプリケーション動作時のオーバヘッドはより低下するものと考えることができ，図 5.16 の測定において 20%程であれば，図 5.17 と同程度とまでは行かないまでも，実用上十分に低オーバヘッドで実行状態の復元が可能になったものと考える．

5.7　本章のまとめ

本章では対象を不可観測かつ揮発な要素である周辺デバイスに拡張し，可観測な要素である CPU や主記憶との一貫性を維持した実行状態の復元を行う手法について述べた．デバイスドライバの状態復元や，デバイスアクセスおよびデバイスのコマンドのログ再実行を行うことによって CPU や主記憶の状態との一貫性を維持した．

このようにして，まずデバイスドライバとデバイス間の一貫性を保証し，これをシステム上の各デバイスドライバに適用することによってシステム全体の復元状態の一貫性を保証した．デバイスドライバとデバイスの関係において，一貫性維持のための処理は，メッセージパッシングシステムでの一貫性の議論を適用し，このモデルを用いた解析によって明らかにした．また，例を用いてこれらの処理の実際のデバイスドライバへの適用方法を示した．そして，linux2.4.4 カーネルおよび UART デバイス，Ethernet デバイスに提案手法を適用したシステムを用いた実験によって，提案手法によるシステムの実行状態の回復が適切に動作することを確認し，提案手法が十分に低オーバヘッドで実現可能であることを示した．

第6章

議論

4章ではCPUの状態と主記憶の状態を対象とし，高速なチェックポインティングを実現する手法について述べた．そして，5章で対象を周辺デバイスの状態まで広げ，周辺デバイスが動作中している最中においてもシステムの実行状態を復元することを可能とする手法について述べた．以下，4章で述べた手法をチェックポインティング手法，5章で述べた手法をデバイス状態復元手法と呼ぶ．

本章では，まず，デバイス状態復元手法の適用範囲について議論を行う．そして，2つの手法の統一的な扱いについて述べる．さらに，CPU，主記憶，周辺デバイスについて，可観測・不可観測要素，揮発・不揮発要素としてこれらを一般化した場合におけて，それぞれの手法から得られた知見について述べる．

6.1 デバイス状態復元手法の適用範囲

デバイス状態復元手法では，各周辺デバイス自身が持つ状態と，CPUおよび主記憶の状態について一貫性を保証し復元する手法を実現した．この手法は，メッセージパッシングシステムのモデルを用いてその一貫性維持に関する議論を適用した．以下，メッセージパッシングシステムのモデルを用いる妥当性について述べ，また，本手法の適用可能なデバイスモデルについて述べる．

6.1.1 メッセージパッシングシステムにおける一貫性の定義

まず，メッセージパッシングシステムにおける一貫性の厳密な定義について述べる．

メッセージパッシングシステムをはじめとする分散システムでは，一つのアプリケーションを構成する要素となる各プロセスは，一つ以上の異なる計算機に分散して存在しそれぞれ独立に動作する．しかし，分散システムでは完全な時刻の同期は困難であるため，同一時刻において全てのプロセスの実行状態を保存することはほぼ不可能と言って良い．このため，事象生起関係 (happend-before relation)[Lamport 78]

をもとに，各プロセスに発生するイベントおよびそのイベントに伴う状態遷移の順序関係が決定され，一貫性についての議論がなされる．

事象生起関係とは，ある 2 つの事象 a と b について，a と b の順序関係を示すものである．「$a \rightarrow b$」と表わされ，a は b の前に生じたことを表わす．そして，以下のような場合に「$a \rightarrow b$」と表される．

- a と b が同一プロセス内のイベントであり，b よりも前に a が生じる
- あるプロセスがメッセージを送信することが a であり，別プロセスがメッセージを受信することが b である場合 (送信されていないメッセージは受信されない)

また，この関係は推移的であり，$a \rightarrow b$ かつ $b \rightarrow c$ であれば，$a \rightarrow c$ である．これらの関係をもとに，各プロセスにおけるイベントの順序は，システム全体でその順序が決定される．なお，$a \rightarrow b$ ではなく，$b \rightarrow a$ でもない場合は a と b は「並行 (concurrent) である」と呼ばれ，その論理的な発生時刻は同一であるとみなされる．

一貫性のある状態とは,「生じうる状態」もしくは「矛盾がない状態」である．並行もしくは並列に動作する全てのプロセスについて，その状態を引き起こす原因となった事象が特定可能であれば，それは「生じうる状態」であり一貫性のある状態である．逆に，あるプロセスの実行が他のプロセスなどから影響をうけているにもかかわらずこれらの因果関係が特定できない場合，それは矛盾が生じる状態であり一貫性のある状態とはならない．つまり，システムを構成する各々のプロセスの状態に着目したときに，システム全体としてこの因果関係 (事象生起関係) が満たされれば，それは一貫性が維持された状態となる．このことは，厳密に以下のように定義される [Coulouris *et al.* 01]．

N 個のプロセスで構成されるシステム Σ を考えるとする．図 6.1 はこのシステムをモデル化した図である．横の矢印はそれぞれプロセスの実行を表しており，左から右に向って時間が増加する．斜めの矢印は各プロセス間でやり取りされるメッセージを表している．また，図 6.1 中の黒丸は各プロセスで発生するイベントを表している．イベントにはメッセージの送信，メッセージの受信のほか，プロセス内で完結するイベント (変数の更新などにより自身の状態を変更するもの) を含む．つまり，図 6.1 は各プロセスがメッセージのやり取りをしながらそれぞれ独自にも状態が遷移し，システム全体の実行が進められている状況を示している．

各プロセス $p_i (i = 1, 2, ..., N)$ の状態を s_i とし，その初期状態を s_i^0 とする．また，プロセス p_i で発生するイベントを e_i とする．そして，イベントの履歴を h_i とおいて，次式のように定義する．

$$h_i =< e_i^0, e_i^1, e_i^2, ... > \tag{6.1}$$

そして，プロセス p_i について k 番目のイベント e_i^k までの有限の履歴を次式で表わ

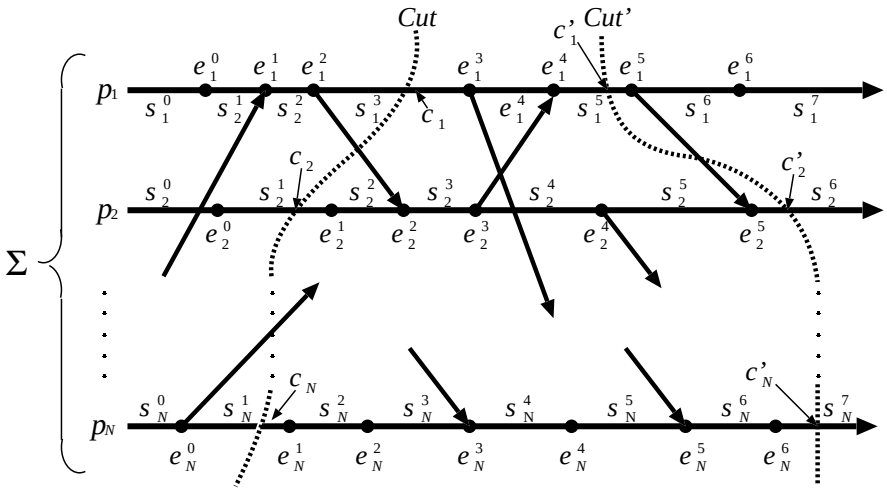

図 6.1: 分散システムにおける一貫性

すこととする．

$$h_i^k =< e_i^0, e_i^1, e_i^2, ..., e_i^k > \tag{6.2}$$

プロセスのある時点の状態 s_i^k はその直前までのイベントの履歴 h_i^{k-1} を反映した結果である．このため，e_i^k が発生する直前のプロセス p_i の状態を s_i^k とし，δ_i をプロセス p_i における状態遷移の遷移関数とすれば，プロセス p_i のある時点での状態 s_i^k は次式で表わされる．

$$s_i^k = \begin{cases} s_i^0 & (k = 0) \\ \delta_i(s_i^0, h_i^{k-1}) & (k > 0) \end{cases} \tag{6.3}$$

一方，システム全体の状態は次式で表わされる．

$$S = (s_1, s_2, ..., s_N) \tag{6.4}$$

このため，システム全体の実行はシステム全体のイベントの履歴として考えることができる．システム全体のイベントの履歴を H とすれば H は次式のようになる．

$$H = h_1 \cup h_1 \cup ... \cup h_N \tag{6.5}$$

各プロセスについて任意の時点 $(c_1, c_2, ..., c_N)$ の状態を結んだ線をカット (cut，図 6.1 中の波線) と呼べば，このときの状態 S^{cut} は，次式で表わされる．

$$S^{cut} = (s_1^{c_1}, s_2^{c_2}, ..., s_N^{c_N}) \tag{6.6}$$

そして，このカットまでに各々のプロセスに発生したイベントの履歴全体を C とすれは，C は次式で表わされる．

$$C = h_1^{c_1 - 1} \cup h_2^{c_2 - 1} \cup ... \cup h_N^{c_N - 1} \tag{6.7}$$

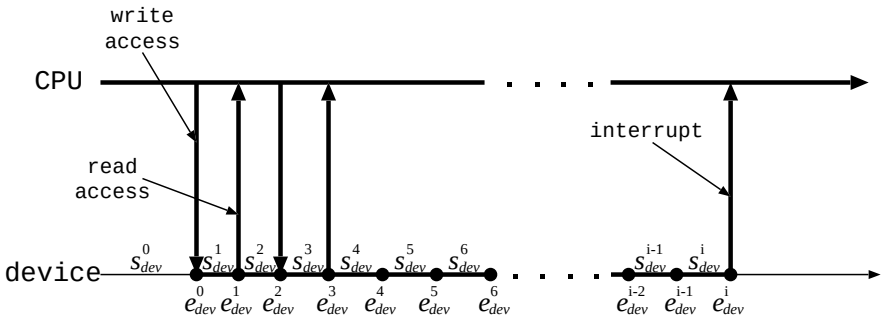

図 6.2: デバイスの状態遷移

カットにおけるシステムの状態はこの C に含まれるイベントが反映した結果である．このため，C に含まれる全てのイベント e に対して，$e' \rightarrow e$ となる e' が C の中に含まれれば，その C には各イベントの因果関係が含まれており,「起り得る状態」として一貫性が保たれた状態となる．つまり，次式の条件を満たすカットが一貫性が保たれた状態として定義される．

$$\forall e, e' : (e \in C) \wedge (e' \rightarrow e) \Rightarrow e' \in C \tag{6.8}$$

図 6.1 中，*Cut* は一貫性の保たれた状態である．一方，*Cut'* は一貫性が保たれない状態である．なぜなら，e_2^5 に対し，$e_1^5 \rightarrow e_2^5$ となる e_1^5 がカットでの履歴の中に含まれないためである．なお，一貫性の保たれるカットは特に「コンシステント・カット (consistent cut)」と呼ばれる．

5.2.1 節で述べたメッセージパッシングシステムにおける一貫性は，この定義をもとにしたものである．換言すれば，メッセージパッシングシステムではコンシステント・カットとなるリカバリラインを構築し，実行状態の復元がなされる．

6.1.2　デバイスの動作とメッセージパッシングシステムモデル

図 6.2 に，デバイスのへのアクセスとその状態の遷移を示す．この図は図 5.6 と同様の図であるが，デバイスの状態，およびデバイスの状態を変化させるイベントを記入してある．

式 (6.8) はメッセージの関係のみでなく，あるプロセス内での状態変化についても考慮されたものである．また，メッセージの内容に対してはなにも前提は設けられていない．単に送信，受信，というイベントによってそのプロセスの状態が遷移する，ということのみである．このため，オートマトンで表わすことができる要素全てについて，この一貫性の定義を当てはめることができる．

デバイスの状態遷移はオートマトンそのものである．また，CPU がデバイスのレジスタを設定 (write アクセス) することと読み込む (read アクセス) ことによっ

て，デバイスの状態およびデバイスドライバの状態が変化することも前述の一貫性の議論と同一のものである．さらに，割り込みによってデバイスドライバの状態が変化することも同様である．これらのことから，分散システムの一貫性の定義，およびメッセージパッシングシステムにおける一貫性維持の方法論をデバイスやデバイスドライバに対して適用することは妥当であると言える．

5.2.1 節で，メッセージパッシングシステムにおけるリカバリの手法は大きく分けてチェックポインティングベースのリカバリとログベースのリカバリの 2 つがあることについて述べた．また，5 章で提案した手法では，デバイスへのコマンド発行中のデバイスアクセスに対して，チェックポインティング手法，およびロギング手法それぞれを用いた．

メッセージパッシングシステムにおけるチェックポインティングベースのリカバリでは，各プロセスで保存された状態からコンシステント・カットとなる状態が選択され，これを復元する．特別な前提は必要とされず，例えリカバリ後の実行が通常実行時のものと異なったとしても，それはそれで正しい実行結果となる．このため，チェックポインティング後に変更される主記憶状態にさえ考慮すれば，デバイスへのコマンド発行中にチェックポインティング手法を適用することに問題はない．

一方，メッセージパッシングシステムにおけるログベースのリカバリでは，メッセージのログを再生することによって各プロセスは同一のプロセスの状態が復元されることが前提とされる．これは PWD 仮定 (piecewise deterministic assumption) と呼ばれる [Strom *et al.* 85, Alvisi *et al.* 95, Slye *et al.* 98]．

プロセスの実行は，実行時に動的に決定される要因によって変化することがある．メッセージの受信もその一つであり，その他，シグナル (割り込み) や，スケジューリングによるスレッドの走行順序などがある．これらは非決定性要素 (nondeterminism) と呼ばれる．ログベースのリカバリは，メッセージという非決定性要素についてそのログを取得し後に再生することで，対象となるプロセスの一貫性を維持する時点までその実行状態を復元する手法である．しかし，その他の非決定性要素が原因となり，リカバリ後に得られる状態が通常実行時のものと異なれば，結果としてプロセス間の状態に不整合が生じてしまうことになる．

例えば，通常実行中の状態がプロセス A については a であり，プロセス B については b だったとする．プロセス B はプロセス A が a という状態にあることを前提とし処理を進めている場合，リカバリによってプロセス A の状態が a' となったとすれば，システムの実行は復元後正しく継続されなくなる．

5 章で提案した手法において，ログは，コマンド発行のためのデバイスアクセスの再生に用いられる．デバイスについての非決定性要素は，外部 (物理) 世界や他のデバイスからのデバイスへの入力，また，デバイス自身がその動作の完了によってバッファに空き領域を作ることなどが考えられる．デバイスへのコマンド発行中にこれらの要因によってデバイスの動作が変化する場合，ログベースの手法では適切にその状態を復帰できなくなる可能性が生じる．このようなデバイスについて

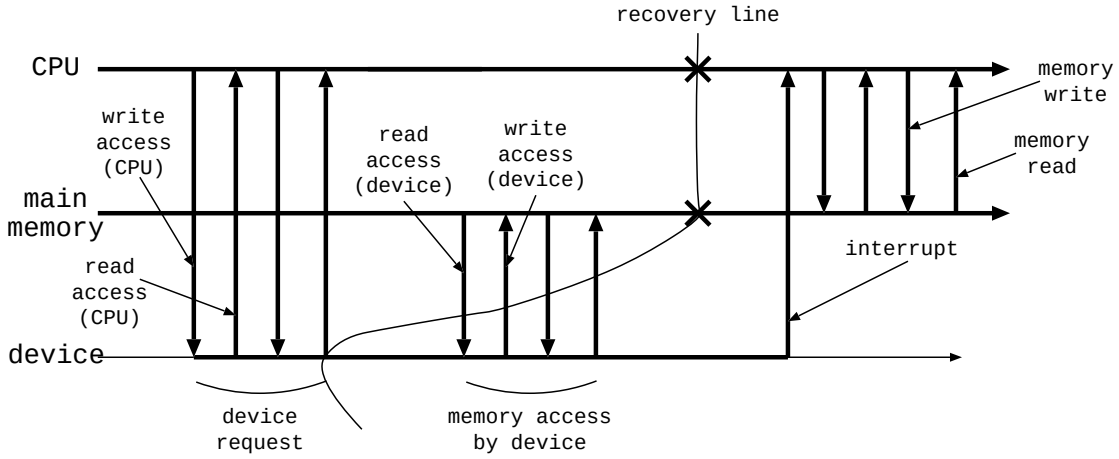

図 6.3: デバイスからの主記憶操作

は，そのデバイスの動作に合わせてリカバリ処理を行う (コールバックルーチンの中で条件分岐を挿入するなど) か，もしくは，デバイスへのコマンド発行に対してチェックポインティング手法を用いる必要が生じる場合がある．

6.1.3　適用可能なデバイスのアーキテクチャ

本節では，5 章では実際のデバイスの動作として取り扱わなかった DMA やデバイス上バッファについて取り上げ，提案した手法の適用可能性について議論する．

DMA およびバスマスタ DMA

DMA およびバスマスタ DMA は，CPU を介さずデバイスと主記憶の間でデータの転送を行い，CPU の利用率を向上させる機構である．DMA やバスマスタ DMA では事前にその対象となる主記憶空間の位置 (アドレス) とサイズを設定する．そして，デバイスに対してその主記憶の内容を転送するか，逆にデバイスが保持するレジスタ (バッファ) の内容を主記憶に対して転送する．通常の DMA の場合にはプログラムによって明示的に転送要求が出される．バスマスタ DMA の場合には，転送要求の開始はデバイスの動作とともに自動的に行われる．また，双方ともその転送の完了通知には割り込みが用いられる．以下，DMA およびバスマスタ DMA をまとめて DMA と呼ぶ．

デバイスが DMA 転送を用いた場合について，その主記憶へのアクセス状況を図 6.3 に示す．図 6.3 は説明のため，デバイスドライバを CPU と主記憶の要素に分け，それぞれ別々に記している．図 6.3 中左側のデバイスへの縦の矢印は，今までの図

と同様にそれぞれ CPU からデバイスに対して発行される read および write アクセスを表わしており，これには DMA 転送の設定のための処理も含まれる．図 6.3 中中央部の縦の矢印は，DMA による主記憶へのアクセスを示している．主記憶からデバイスに向う矢印は主記憶からデバイスに対するデータ転送を表わす．逆にデバイスから主記憶へ向う矢印は，デバイスから主記憶に対するデータ転送を表わす．また，図 6.3 中右側の矢印は DMA によってデバイスから転送された結果を CPU が処理することを示している．

図 6.3 に示すリカバリラインは，DMA によるデータ転送が行われた後に電源切断が生じた場合において，デバイス状態復元手法により復元される各要素の状態を示している．この場合，リカバリラインは DMA による主記憶のアクセスを横切ることになる．

まず，図 6.3 中，DMA の処理が全て主記憶からデバイスへ向う矢印だったと仮定する．すなわち，主記憶からデバイスへのデータ転送について考慮すれば，これは式 (6.8) による一貫性の定義から，一貫性が維持されるリカバリラインとなる．実際に，リカバリ後 DMA に対して通常実行時と同じアドレスとサイズが設定されれば，デバイスに対して転送される主記憶の内容は同一となると考えられる．ただし，DMA によって転送される主記憶領域が変更されていないことが前提となるが，一般的な DMA の動作ではデバイスに対して転送される領域はその転送の完了通知 (割り込み発生) まで放置されることが多い．このため，主記憶からデバイスへの転送について，リカバリ後問題を生じる可能性は少ない．

逆に，DMA の動作を全てデバイスから主記憶へ向う矢印，すなわちデバイスから主記憶へのデータ転送と仮定した場合，これは一貫性が維持できないリカバリラインとなる．主記憶上にはデバイスから転送されたデータが残され，デバイスはそのデータをまだ得ていない状態となる．しかし，リカバリ後デバイスが通常実行時と同じ動作を行い，同じデータを保持するようになれば，デバイスから主記憶に対して転送されるデータも通常実行時と同一と考えられる．つまり，コマンドの復帰によりデバイスが同一動作を繰り返すことによって，電源切断時の主記憶状態とリカバリ後に DMA によって更新される主記憶状態は同一のものとなり，その後のプログラムの動作に影響を与えずにシステムの実行が復元されると考えられる．

同一の情報が転送されなかったとしても，図 6.3 に示すように，通常のプログラム (デバイスドライバ) はその転送が完了通知がなされてから対応する主記憶の状態を読み込んで処理を行う．このとき，リカバリ後のプログラムは新たに DMA によって転送された情報をもとに処理を継続することになるため，通常実行時と処理結果は異なるかも知れないがプログラムの実行自体に問題は発生しないと考えられる．つまり，DMA によって変更される主記憶領域がプログラムの実行に対して影響を与える前であれば，リカバリ後 DMA によって転送される情報が異なったとしても問題は発生しない．

割り込みが発生した後電源切断が生じた場合には，CPU が主記憶の状態を読み

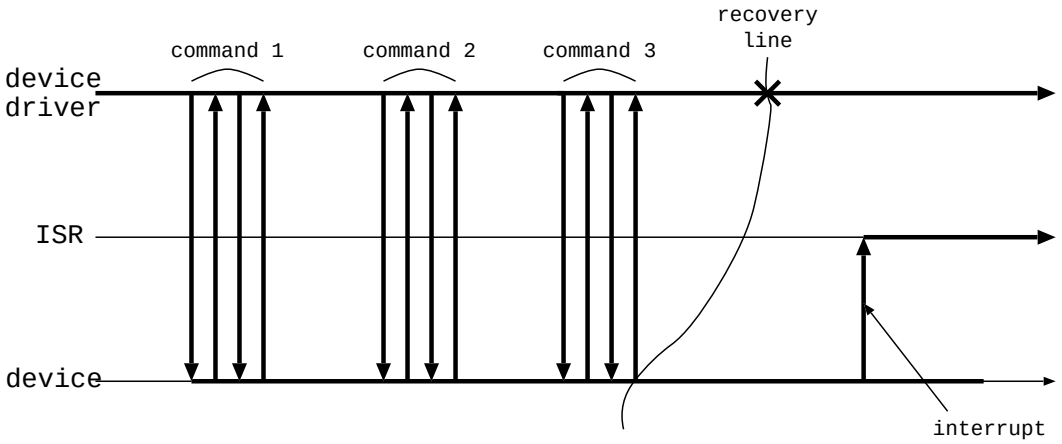

図 6.4: デバイスでのバッファリング

込み，プログラムの実行に影響が生じると考えられる．しかし，DMA による影響を最初に処理するプログラムである ISR の処理は，その必要があればロールバックされる．このため，DMA によって同一の情報が転送されなかったとしてもその影響は破棄され，新しく転送された上を用いて処理の継続が可能となる．

通常の DMA の動作は，主記憶からデバイスもしくはデバイスから主記憶のどちらか一方であるが，もしこれらが混在する場合には以下のような問題が発生する可能性が生じる．デバイスがある主記憶領域を参照して処理を行った後，同一領域の内容を上書きするような場合，デバイスの動作はその主記憶の状態に依存することになり，リカバリ後その動作が変化する可能性が生じる．しかし，これはデバイスへのコマンド発行時に対象となる主記憶領域を保存しておき，リカバリ時にこの状態を復元すれば対応可能である．主記憶からの読み込みが混在していたとしても，デバイスが最初に書き込みを行う場合や，読み込む領域と書き込む領域がそれぞれ異なる場合には，前述の読み込み，書き込み単一の場合それぞれについての議論と同様となりこのような対応は必要ない．

このように，DMA やバスマスタ DMA を用いてデバイスが独自に主記憶に対してその読み書きを行う場合においても，多くの場合において本論文で提案したデバイス状態復元手法は適用可能であると考えられる．

デバイス上のバッファ

デバイスには，独自にそのデバイス上にバッファを持つものが多く存在する．この場合，デバイスのバッファ上に保持されたコマンドおよび情報は電源切断とともに失われることになってしまう．

デバイス上にバッファが存在する場合のデバイスハンドリングの様子を図 6.4 に

示す．図 6.4 は最初のコマンドの発行によってデバイスが動作を開始し，そのデバイスの動作中にさらに 2 つのコマンドが繰り返し発行されている状況を示している．

3 つめのコマンドが発行された後に電源切断が生じた場合，保存されたコマンドの再実行によって 3 つめのコマンド実行後の状態が復元できれば，図 6.4 に示すリカバリラインを構築することができる．このため，各デバイスへのコマンド発行の際，それぞれのコマンドを保存しておき，リカバリ時にそれぞれを再実行することが考えられる．

このとき，前述のログベースのリカバリにおける PWD 仮定と同一の問題が発生する．コマンドの実行が発行のタイミングなどに依存し，これによってデバイスの動作やデバイスが持つ状態が変化する場合には一連のコマンドの再実行に対して問題を生じる可能性がある．一方，それぞれのコマンドが独立に実行され，その再実行によって通常実行時にデバイスドライバがデバイスに対して想定していた状態と，リカバリ後の実際のデバイスの状態が同一になるようなデバイスであれ問題は生じない．これは，換言すれば一連のコマンドの実行が巾等 (idempotent) に行われるデバイス，ということになる．

コマンドが巾等に実行されるデバイスの良い例として，ハードディスクがあげられる．ハードディスクへのコマンドはあるセクタに対する読み込み，もしくは書き込みとなる．読み込みの場合には巾等な処理となることは明らかである．一方，書き込みの場合には同じデータをある同じセクタに対して書き込むことは巾等な動作となる．ハードディスクのようにその処理が巾等となるようなデバイスであれば，コマンドの再実行によってバッファ中に存在した処理が再度繰り返されたとしても，デバイスがもつ状態はプログラムによって想定されたものと同一となる．

ただし，ハードディスクのようなデバイスであったとしても，前述の DMA の項で述べた議論と同様に，同一セクタに対する読み込みと書き込みが混在する場合には問題が発生する可能性がある．同一セクタに対する読み込みと後に続く書き込みの操作が復元される場合，リカバリ後読み込まれるデータは，通常実行時に変更されたものとなる可能性がある．この場合には，デバイスドライバ内でコマンドの発行時にその操作や対象セクタを確認し，書き込み操作を遅延させるなどの対応が必要となる．なお，やはり DMA と同様に同一セクタに対するものでない場合や，書き込みが先に行われる場合にはこの問題は発生しない．

この同一領域に対する読み込みの後の書き込みに関する問題は，ハードディスクが保持する情報が永続化されることに起因するものである．電源切断とともにその状態が失われるものであれば，リカバリ時に再初期化処理によって構築されるデバイスの状態は通常実行時の状態に依存しない．このため，このような問題は発生しない．

6.1.4　適用可能なデバイスアーキテクチャの定式化

以上，DMA およびバッファに対してその適用可能性について議論した．これらの議論には以下のような共通項がある．一つは，双方ともデバイスへのコマンドの再実行に対し，同一 (巾等) の動作が繰り返される場合には問題が生じないということである．また，もう一つはその対象が不揮発であるため，その問題が生じる可能性があることである．DMA は不揮発な主記憶を対象とし，ハードディスクもそのコマンドの実行結果が永続化 (不揮発) される．

よって，前述までの議論から，デバイス状態復元手法について以下のように考えることができる．

> 対象とするデバイスについて再実行されるコマンドの影響が永続化され，かつそのコマンドが巾等に実行できないデバイスはデバイス状態復元手法では問題が発生する可能性がある

逆に言えば，不揮発であってもコマンドが巾等に実行される場合や，巾等に実行できなかったとしても揮発な場合にはデバイス状態復元手法は適用可能である，ということができる．

このことから，デバイス状態復元手法が適用可能な場合を以下の論理式によって表わすことができる．

$$Idempotent(c) \vee Volatile(c) \Rightarrow Recoverable(c) \tag{6.9}$$

ここで，c はデバイス状態復元手法によって復元されるコマンドを表わす．
$Idempotent(c)$，$Volatile(c)$，$Recoverable(c)$ はそれぞれ，デバイスへのコマンド c が巾等に実行されること，コマンド c の実行結果が電源切断とともに失われること，コマンド c が本論文で提案したデバイス状態復元手法によって適切に復元されることを示す．

そして，デバイス状態復元手法が適用可能なデバイスは，デバイスドライバによってデバイスに発行されるコマンドの集合を C とすれば，次式で表わすことができる動作をするデバイスとなる．

$$\forall c \in C : Idempotent(c) \vee Volatile(c) \tag{6.10}$$

6.1.5　デバイスの動作と外部世界への影響

周辺デバイスには外部世界との相互作用を行うものが多い．外部世界とは，ここではある一つの計算機システムに内包されない物を意味し，物理世界や他の計算機システムなどを意味する．外部世界の状態は不可観測かつ不揮発な状態であり，また，特に物理世界の状態は時間に依存する場合が多いことから，いままで述べてこ

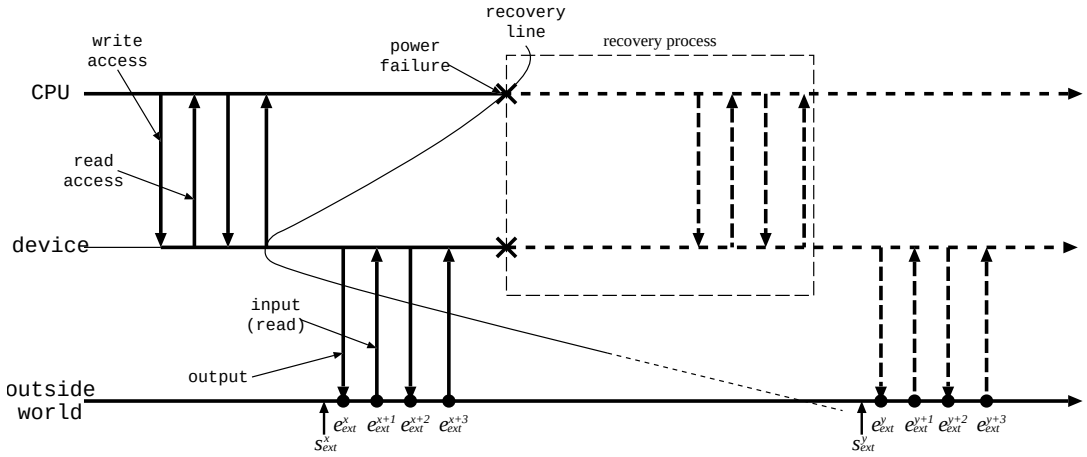

図 6.5: デバイスの外部世界への影響

なかった．しかし，ある特定の条件下やアプリケーションの内容によって外部世界を扱うデバイスに対してもその動作を適切に継続可能であると考えられる．以下，外部世界とデバイスの動作について考察する．

図 6.5 に外部世界へ影響を及ぼすデバイスについて，その動作中に電源切断が生じた時の状況を示す．図 6.5 では，3.3.3 節で述べたメッセージパッシングシステムにおける外部世界 (デバイス) の取り扱いと同様，外部世界も一つのプロセスとしてみなしモデル化している．デバイスから外部世界へ向う矢印は外部世界への出力を表わし，外部世界からデバイスへ向う矢印は外部世界からデバイスへの入力を表わしている．

デバイスが外部世界に対して入出力を繰り返した後，システムの電源切断が生じたとすれば，外部世界の状態は復元できないため，リカバリラインは図 6.5 中に示すようになる．電源切断後システムではリカバリ処理が行われ，そして実行が継続される．一方，外部世界の状態は元には戻らない．なお，CPU とデバイスについて，リカバリ処理とその後継続する実行は点線で表わされている．

このリカバリラインにおける一貫性に着目すれば，5.2.1 節で述べた議論から，外部世界からの入力は一貫性の維持されるアクセスとなる．一方，外部世界に対する出力は一貫性が維持できないアクセスとなる．

まず，デバイスに対する入力について考えれば，デバイスに対する入力は「デバイスが能動的に外部の状態を取得するもの」と「外部世界からデバイスに対して入力されるイベント」の 2 つを考えることができる．

リカバリ後のデバイスが能動的にその外部世界の状態の取得を行う場合，デバイスはこの作業でリカバリ後の外部世界の状態を取得することになる．このことは，一貫性は維持されるものの，アプリケーションの期待する動作によっては問題を発生する可能性を生じさせる．例えば，画像をキャプチャアするプリケーションでは，

リカバリ後取得された画像とリカバリ前に取得された画像とで時間的な連続性が失われることになる．一定期間の連続的な画像の録画を期待するアプリケーションでは，画像は途切れた画像となるため期待されたアプリケーションの実行結果とはならない．一方，同じ画像をキャプチャするデバイスを用いていたとしても，単に画像のモニタリングのみを行うようなアプリケーションでは問題とはならない．つまり，入力に関して，リカバリ後の処理が「意味をなすものであるかどうか」ということは，アプリケーションレベルでの処理に依存するものとなる．

外部から非同期に発生しデバイスへの入力となるイベントの場合には，このイベントは失われる (再発生しない) ことになる．しかし，このような外部世界から送られる非同期のイベントの内，偶発的に発生するものは例えばキーボードやマウスの入力などと考えられ，この扱いは 5.3.2 節で述べたように無視してもその実行に支障をきたすことは少ないと考えられる．一方，発生することが期待されるイベントの例は，例えばネットワークパケットの Ack などが考えられ，このパケットが失われた場合にはアプリケーションレベルでは接続先とのコネクションが切断されたと認識がなされる．このため，アプリケーションレベルやプロトコルスタック，デバイスによってはデバイス自体でリカバリ時にコネクションの再接続を行うなどの対応が必要になることが考えられる．

次に，外部世界への出力について考慮すれば，コマンドの再実行によってシステム (アプリケーション) が想定する外部世界の状態と実際の外部世界の状態とで不整合をおこす可能性がある．これは 6.1.3 節で述べた議論と同一のものであり，外部世界の状態が不揮発であるために生じる問題である．つまり，式 (6.10) において，外部世界の状態は $Volatile(c)$ が常に偽となる状況と考えられる．換言すれば，デバイスの操作が外部世界に対して巾等 ($Idempotent(c)$ が真) となるデバイスであれば，デバイス状態復元手法によって矛盾をおこすことなくその実行状態を復元することができると考えられる．

外部世界の影響に対し巾等となるコマンドは，例えばモータなどの指令値が絶対的な角度によって与えられるときなどが考えられる．このときにはデバイス自身がフィードバックループを持ち，その巾等性はデバイス自身によって保証される．また，デバイスドライバ外で巾等とされる処理が行われる場合も存在する．前述と同様にモータの例では，デバイス自身が基準となる値を持たない様な場合でも，アプリケーションレベルでポテンショメータなどと組み合わせ，フィードバックループを形成することによって，アプリケーションレベルでの巾等性を確保することができる．この他，TCP プロトコルを用いる通信ではデバイスのコマンドの再実行によって同一のパケットが 2 度送信される可能性があったとしても，下位レイヤのネットワークは元来信頼性のないものと考えられて設計されているため，TCP プロトコルの処理でその対応がなされる．TCP プロトコルでは，ヘッダの中のシーケンス番号によってネットワークから 2 重に到着したパケットの破棄が行われる．つまり，デバイスレベル，カーネルレベル (デバイスドライバ外)，アプリケーショ

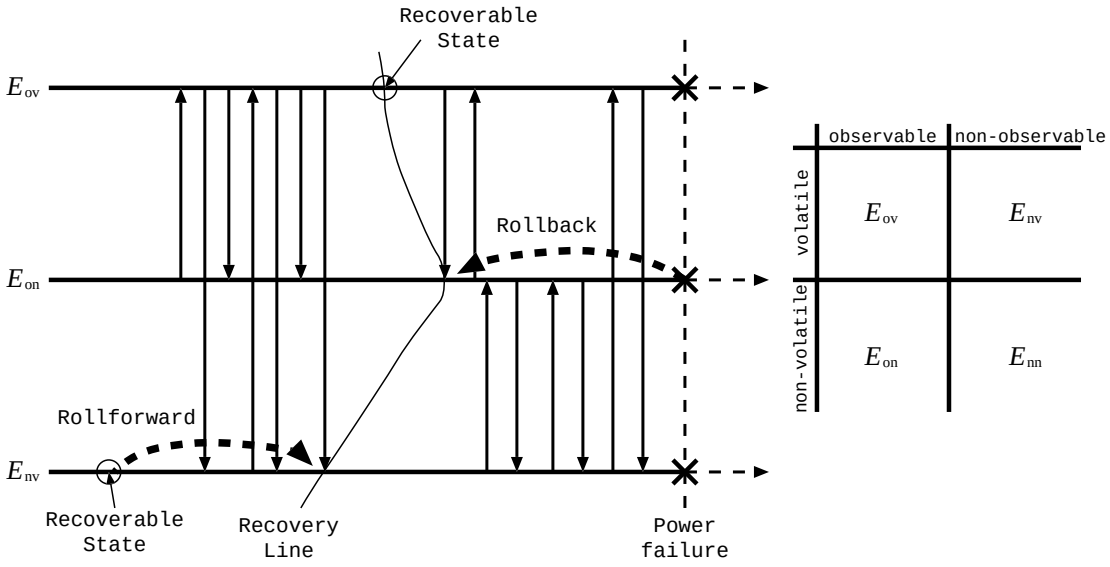

図 6.6: 各要素についての状態復元

ンレベルなどでこのような対処がなされる場合であれば，本手法によりデバイスの動作が繰り返されたとしても問題は生じないと言うことができる．

以上の考察から，デバイスへの入力について述べたようにその処理内容が電源切断中に流れる時間に依存する場合や失われた入力がプログラムの状態変化を引き起こす場合，また外部世界に影響を及ぼすデバイスについてデバイスレベルでの巾等性が保証されない場合には，アプリケーションをはじめとするデバイスドライバ外と連携をし対応を行う必要が生じる．この点において，本研究は未考慮であり残された課題として存在しているが，リカバリ時にカーネルから対応するアプリケーションやカーネルコンポーネント (プロトコルスタックなど) のリカバリ処理を呼び出す機構を設け (アップコールやシグナルの送信など)，その対応を行う枠組みを提供することが考えられる．

6.2　可観測・不可観測要素と揮発・不揮発要素

チェックポインティング手法およびデバイス状態復元手法は，それぞれ可観測かつ揮発な CPU の状態を基準に，可観測かつ不揮発な要素の実行状態，不可観測かつ揮発な要素の実行状態を一貫性を保ちつつ復元する手法と捕らえられる．

チェックポインティング手法は，CPU との一貫性を保つため，可観測かつ不揮発な要素である主記憶の状態を明示的に保存することによって実行状態の復元を行った．このとき，主記憶の状態は CPU の状態と同一の (一貫性を保つ) 時点へロールバックされた．また，デバイス状態復元手法では，不可観測かつ揮発な要素である

デバイスについてある時点での復元可能な状態をもとに，デバイスに対するコマンドを再発行することによってデバイスと CPU や主記憶の状態との一貫性の維持を行った．これは，デバイスの状態はある時点からのロールフォワードによってその状態が復元される処理と言うことができる．

これらを一般化した図を図 6.6 に示す．図 6.6 中 E_{ov} は可観測かつ揮発性要素，E_{on} は可観測かつ不揮発性要素，E_{nv} は不可観測かつ揮発性要素を表わす．また，図 6.6 中白丸は揮発性要素について復元可能となる時点を表わしている．

図に示されるように，E_{ov} について復元可能となる状態をもとに，一貫性を保つ状態について，それぞれ E_{on} の状態は明示的に状態を保存することによってロールバックされ，E_{nv} はそのアクセスを再実行することによってロールフォワードされた．そして，それぞれの手法が適切に実行状態が復元されることが確認されている．

また，図 6.6 内には示していないが，例えばディスクなどの周辺デバイスであっても不揮発である要素，つまり不可観測かつ不揮発 E_{nn} となる要素に対しては考察の結果から，式 (6.9) が得られている．また，同様に E_{nn} となる外部世界に対しては $Idempotent(c)$ とすることができればその実行状態を適切に復元できることついて述べた．このことは，逆に言えば $Idempotent(c)$ とするような処理をリカバリ時に行うことができれば不可観測かつ不揮発な要素であってもその実行状態を復元可能であると考えられる．

つまり，これらの結果をもとにより一般化して論理的に解析を行うことによって，システムの構成要素を可観測・不可観測，揮発・不揮発として分類したときに全ての要素についての基本的な実行状態の復元方法が得られることが示唆される．このことは，現在不揮発メモリが IC 上に混載され始めており，今後 CPU キャッシュやデバイスのバッファとして用いられることが目される中，システム上にそれぞれの要素が混在する状況となったときに，非常に有益な実行状態復元の方法論の確立を促すものである．

第7章

まとめ

本論文では，不揮発メモリを主記憶とするシステムを対象とし，突発的な電源切断に面したときにもその実行状態を復元可能とする手法について述べた．そして，ソフトウェアベースのアプローチによってこれを実現した．

主にフォールトトレランスを目的としプログラムの実行状態の復元に関する研究はなされてきたが，ほとんどの従来研究で対象とされてきたのはCPUと主記憶の状態のみであり，周辺デバイスが持つ状態はあまり考慮されてこなかった．これは，可観測かつ揮発な要素についてしか実行状態の復元が扱われてこなかったことを意味する．一方，本研究では，オペレーティングシステムを含めたシステム全体を対象とし，CPU，主記憶，周辺デバイスの3つを対象に実行状態の復元を行った．このとき，本研究で対象とする不揮発主記憶システムでは，それぞれの要素は，可観測かつ揮発，可観測かつ不揮発，不可観測かつ揮発な要素として分類される．

本論文では，まず可観測かつ揮発であるCPUと可観測かつ不揮発である主記憶の状態に着目し，高速にその状態を保存する方法を提案した．そして，不可観測かつ揮発なデバイスに対象を拡大し，システム全体の実行状態を復元可能とする手法を提案した．

CPUと主記憶の実行状態復元について，不揮発主記憶システムであってもCPUの状態は揮発であるため，これらの一貫性を維持するためには主記憶の状態を明示的に保存する必要が生じる．さらに，CPUのストア命令一つ一つが主記憶に残されることから，特にチェックポインティング時やリカバリ時における電源切断に留意する必要がある．

主記憶の状態保存に対し，その対象となる領域をオブジェクトを単位としてプログラムの中で明示的に指定した．このとき，構造化された主記憶の管理手法を用いることによって，通常実行時やチェックポインティング時に対象となる主記憶領域の特定の高速化をはかった．また，チェックポインティング中やリカバリ中に電源切断が生じた場合にもその実行状態を適切に復元可能とするため，2つのポインタを用いてCPUのストア命令一つでCPUの復元状態と主記憶の復元状態という2つの独立な事象を切り替える手法を用いた．

この手法をプロトタイプオペレーティングシステム上に実装し，実験を行った．その結果，システムが通常実行中だけでなく，チェックポインティング中，復帰処理中のいずれにおいて突然電源切断が生じた場合にも，適切にシステムの実行状態が復元可能であることを確認した．また従来研究で用いられてきた MMU を用いた手法よりも，高速にその実行状態の保存がなされることを確認した．

次に，不可観測かつ揮発である周辺デバイスの状態に対象を広げ，CPU や主記憶状態との一貫性を維持した実行状態の復元を行った．このとき，デバイスとデバイスドライバの関係に着目し，この関係をメッセージパッシングシステムとみなした．そして，メッセージパッシングシステムでの一貫性の議論をこれらの関係に適用し，デバイスへのアクセス中や，デバイスが動作している最中において，CPU や主記憶状態と一貫性を維持しつつデバイスの状態を復元可能とするために行うべき処理を明らかにした．

この解析により得られた結果を，Linux カーネルおよび UART，Ethernet デバイスのデバイスドライバに対して適用して実験を行った．この結果，デバイスの動作中においてもシステムの実行が適切に復元され，さらに，低オーバヘッドでの実行状態の復元が可能であることを確認した．

本研究で得られた成果は，一般家庭での利用における計算機システムの利便性や信頼性の向上のみならず，計算機システムが環境から取得される電力や無線で供給される電力で動作することが目される中，このような不安定な電力源を用いたシステムの実現に大きな貢献を果たすものである．さらに，本研究で得られた知見は，将来不揮発メモリが計算機システム内の各要素に適用され，その揮発性や可観測性に対して様々な要素がシステム内に混在する状況となったときにも，その実行状態の復元を可能とする基本的な方法論の確立を示唆するものである．

謝辞

本研究を行う機会を与えてくださり，また，研究を進めるにあたり厳しくもかつ優しく見守ってくださった 慶応義塾大学理工学部教授 安西 祐一郎先生 に心より感謝申し上げます．理工学部長，塾長兼務の激務の中，要所要所にて頂いたアドバイスは，研究のみならず一個人としての社会での処し方にまで及んだものでありました．

また，研究を進めるにあたり，途中困難に遭遇したときにも適切なご指導頂きました 慶応義塾大学理工学部 専任講師 山崎 信行先生 に厚く御礼申し上げます．論文の書き方や実装について頂いたアドバイスがあってはじめて本研究を成し遂げることができたものと思います．

本論文に対し，大変有益なご助言を下さった 慶応義塾大学理工学部教授寺岡 文男先生，慶応義塾大学理工学部 専任講師 矢向 高弘先生 には感謝の言葉もございません．先生方とのディスカッションが無ければ到底本論文の完成はあり得ませんでした．また，今後の研究についても大変に有益かつ本質的なサジェスチョンを頂き，今後に大きく繋がるご助言を頂きました．

また，慶応義塾大学理工学部専任講師 今井 倫太先生には論文の執筆の仕方，研究の進め方を始め，多くの点でご指導頂きました．私の博士課程在学中，途中からいらっしゃったためにご一緒する期間が短く，残念に思っております．もっと先生から学びたかった点が多くございます．

安西・山崎・今井研究室秘書 永坂 弘子さんには日頃の研究室生活において大変お世話になりました．また，研究室で一緒に博士課程に進学した 梅澤 猛君，川島 英之君，宮崎 崇君，また研究室の先輩として色々相談に乗って頂いた 白石 陽氏にも迷惑をかけっぱなしであったかと思います．この場を借りて，陳謝すると共に厚くお礼を申し上げる次第です．

慶応義塾大学理工学部情報工学科 安西・山崎・今井研究室のメンバーには，研究を始めとし，研究室生活そのものを助けて頂きました．皆様が居なければ，途中で挫折すると言うこともあり得たかと思います．お礼申し上げると共に，皆様のこれからの活躍を信じてやみません．

この他，学部時代よりお世話になり，本研究におきましても有益な助言を下さった ソニー CSL 河野 通宗氏 をはじめとし，様々な方々に支えられて本研究を遂行することができました．この場を借りてお礼申し上げたいと思います．

最後になりますが，幼少のころより私のことを暖かく見守り，また支えて続けてくれた父 義治，母 千賀子 そして姉 朗子には弁舌に勝る感謝の気持を持ち続けていることをここに明記するものです．特に両親のその助言が何度挫折しかかった自分を戒め，そして支えたことかわかりません．本論文の完成をもって，家族への謝辞に替えたいと思います．

2004 年 2 月
大村 廉

参考文献

[ACPI 02] Compaq Computer Corporation and Intel Corporation and Microsoft Corporation and Phoenix Technologies Ltd. and Toshiba Corporation. *Advanced Configuration and Power Interface Specification Rev. 2.0a*, Mar. 2002. http://www.acpi.info.

[Alvisi *et al.* 95] Lorenzo Alvisi, and Keith Marzullo. Message Logging: Pessimistic, Optimistic, and Causal. In *International Conference on Distributed Computing Systems*, pp. 229–236, 1995.

[APM 96] Intel Corporation and Microsoft Corporation. *Advanced Power Management BIOS Interface Specification Rev. 1.2*, Feb. 1996. http://www.microsoft.com/hwdev/archive/BUSBIOS/apm_12.asp.

[Banatre *et al.* 90] Michel Banatre, and Philippe Joubert. Cache Management in a Tightly Coupled Fault Tolerant Multiprocessor. In *Proceedings of the 20th Symposium on Fault-Tolerant Computing (FTCS-20)*, pp. 89–96. IEEE CS Press, 1990.

[Banatre *et al.* 91] M. Banatre, Ph. Joubert, Ch. Morin, G. Muller, B. Rochat, and P. Sanchez. Stable transactional memories and fault tolerant architectures. *SIGOPS Operating System Review*, Vol. 25, No. 1, pp. 68–72, 1991.

[Bernstein 88] Philip A. Bernstein. Sequoia: A Fault-Tolerant Tightly Coupled Multiprocessor for Transaction Processing. *Computer*, Vol. 21, No. 2, pp. 37–45, 1988.

[Bindhammer *et al.* 02] T. Bindhammer, R. Göckelmann, O. Marquardt, M. Schöttner, M.Wende, and P. Schulthess. Device Driver Programming in a Transactional DSM Operating System. In Feipei Lai, and John Morris, editors, *Seventh Asia-Pacific Computer Systems Architecture Conference (ACSAC'2002)*, Melbourne, Australia, 2002. ACS.

[Bonwich 94] Jeff Bonwich. The Slab Allocator:An Object-Caching Kernel Memory Allocator. In *Proc. of the summer 1994 USENIX Conference*, pp. 87–98, 1994.

[Bowen *et al.* 93] Nicholas S. Bowen, and Dhiraj K. Pradhan. Processor- and Memory- Based Checkpoint and Rollback Recovery. *IEEE Computer*, pp. 22–31, Feb. 1993.

[Ceelen 02] Christian Ceelen. *Implementation of an Orthogonally Persistent L4 μ-Kernel Based System*. PhD thesis, Universitaet Karlsruhe, 2002.

[Challis 78] Michael F. Challis. Database consistency and integrity in a multi-user environment. In *DATABASE: Improving usability and responsiveness*. Academic Press, 1978.

[Coulouris *et al.* 01] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley, Pearson Education, third edition, 2001.

[Dearle *et al.* 94] Alan Dearle, Rex di Bona, James Farrow, Frans Henskens, Anders Lindstrom, John Rosenberg, and Francis Vaughan. Grasshopper: An Orthogonally Persistent Operating System. *Computing System*, Vol. 7, No. 3, pp. 289–312, 1994.

[Dearle *et al.* 00] Alan Dearle, and David Hulse. Operating system support for persistent systems: past, present and future. *Software – Practice & Experience*, Vol. 30, No. 4, pp. 295–324, 2000.

[Elnozahy *et al.* 92] Elmootazbellah Nabil Elnozahy, David B. Johnson, and Willy Zwaenpoel. The Performance of Consistent Checkpointing. In *Proceedings of the 11th IEEE Symposium on Reliable Distributed Systems*, Houston, Texas, 1992.

[Elnozahy *et al.* 99] Mootaz Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and Dabid B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. Carnegie Mellon University, June 1999.

[Gray *et al.* 93] Jim Gray, and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[Hong *et al.* 00] Jiman Hong, Yeom Y. H., Yookun Cho, and Taesoon Park. Kckpt: an efficient checkpoint and recovery facility on UnixWare kernel. In *Proc. of the ISCA 15th International Conf. Computers and Their Applications*, pp. 303–308, Jan. 2000.

[Hunt *et al.* 87] Douglas B. Hunt, and Peter N. Marinos. A General Purpose Cache-Aided Rollback Error Recovery (CARER) Technique. In *Proceedings of the 17th Symposium on Fault-Tolerant Computing (FTCS-17)*, pp. 170–175. IEEE CS Press, 1987.

[Lamport 78] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, Vol. 21, No. 7, pp. 558–565, Jul. 1978.

[Landau 92] Charles R. Landau. The Checkpoint Mechanism in KeyKOS. In *Proceedings of the IEEE Second International Workshop on Object Orientation in Operating Systems*, 1992.

[Liedtke 93] Jochen Liedtke. A persistent system in real use: experiences of the first 13 years. In *Proceedings of the International Workshop on Object-Orientation in Operating Systems (IWOOOS)*, 1993.

[Lindstrom *et al.* 95] Anders Lindstrom, Alan Dearle, Rex di Bona, Stephen Norris, John Rosenberg, and Francis Vaugha. Persistence in Grasshopper Kernel. In *Proc. of the Eighteenth Australian Computer Science Conference*, pp. 329–338, 1995.

[Ling *et al.* 01] Yibei Ling, Jie Mi, and Xiaola Lin. A Variational Calculus Approach to Optimal Checkpoint Placement. *IEEE Transaction on Computer*, Vol. 50, No. 7, pp. 699–708, 2001.

[Lorie 77] Raymond A. Lorie. Physical Integrity in a Large Segmented Database. *ACM Transaction on Database Systems*, Vol. 2, No. 1, pp. 91–104, March 1977.

[Nam *et al.* 97] Hyo chang Nam, Jong Kim, SungJe Hong, and Sunggu Lee. Probablistic Checkpointing. In *Proceedings of the 27th Symposium on Fault-tolerant Computing (FTCS-27)*, pp. 48–57, 1997.

[Plank *et al.* 95a] J. S. Plank, M. Beck, and G. Kingsley. Compiler-Assisted Memory Exclusion for Fast Checkpointing. *IEEE Technical Committee on Operating Systems and Application Environments, Special Issue on Fault-Tolerance*, Vol. 7, No. 4, pp. 10–14, Winter 1995.

[Plank *et al.* 95b] James S. Plank, Micah Beck, and Gerry Kingsley. Libckpt: Transparent Checkpointing under Unix. In *USENIX Winter 1995 Technical Conference*, pp. 213–223, Jan. 1995.

[Plank *et al.* 95c] James S. Plank, Jian Xu, and Robert H. B. Netzer. Compressed Differences: An Algorithm for Fast Incremental Checkpointing. University of Tennessee, August 1995.

[Plank *et al.* 98] James S. Plank, and Michael G. Thomason. The Average Availability of Uniprocessor Checkpointing Systems, Revisited. Dept. of Computer Science Univ. of Tennessee, Aug. 1998.

[Plank *et al.* 99] J. S. Plank, Y. Chen, K. Li, M. Beck, and G. Kingsley. Memory Exclusion: Optimizing the Performance of Checkpointing Systems. *Software – Practice & Experience*, Vol. 29, No. 2, pp. 125–142, 1999.

[Rosenberg *et al.* 96] J. Rosenberg, A. Dearle, D. Hulse, A. Linderstrom, and S.Norris. Operating System Support for Persistent and Recoverable Computations. *Communications of the ACM*, Vol. 39, No. 9, pp. 62–69, Sept. 1996.

[Shapiro *et al.* 99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS:a fast capability system. In *17th ACM Symposium on Operating System Principles(SOSP '99)*, pp. 170–185, 1999.

[Silva *et al.* 98] Luis Moura Silva, and Joao Gabriel Silva. System-Level Versus User-Defined Checkpointing. In *Symposium on Reliable Distributed Systems*, pp. 68–74, 1998.

[Slye *et al.* 98] J. Hamilton Slye, and E. N. Elnozahy. Support for Software Interrupts in Log-Based Rollback-Recovery. *IEEE Transactions on Computers*, Vol. 47, No. 10, pp. 1113–1123, 1998.

[Srouji *et al.* 98] Johny Srouji, Paul Schuster, Maury Bach, and Yulik Kuzmin. A Transparent Checkpoint Facility On NT. In *Proceedings of the 2nd USENIX Windows NT Symposium*, pp. 77–86, 1998.

[Strom *et al.* 85] Rob Strom, and Shaula Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems (TOCS)*, Vol. 3, No. 3, pp. 204–226, 1985.

[Vaidya 97] Nitin H. Vaidya. Impact of Checkpoint Latency on Overhead Ratio of a Checkpointing Scheme. *IEEE Transaction on Computer*, Vol. 46, No. 8, pp. 942–947, 1997.

[河合 他 03] 河合基伸, 蓮田宏樹. 日経エレクトロニクス, 電源だってユビキタス, pp. 103–133. No. 849. 日経 BP 社, 6-9 2003.

[甲斐 他 01] 甲斐康司, 井上昭彦, 安浦寛人. フラッシュ・メモリを主記憶とするシステムのためのメモリ・アーキテクチャの検討. 電子情報通信学会技術研究報告,CPSY-111, pp. 51–58, 2001.

[松本 99] 松本輝恵. 日経エレクトロニクス, 解説／ DRAM 代替をねらう新メモリ「MRAM」, pp. 49–56. No. 757. 日経 BP 社, 11-15 1999.

[大石 他 01] 大石基之, 松本輝恵. 日経エレクトロニクス, そして，全てのメモリは不揮発になる．, pp. 151–177. No. 789. 日経 BP 社, 2-12 2001.

[田原 他 04] 田原修一, 與田博明. 情報処理, Vol. 45 No.1, MRAM–不揮発性 RAM の実現に向けて–, pp. 42–47. 情報処理学会, Jan. 2004.

論文目録

【 主論文に関する公刊論文 】

1. 大村 廉，山崎 信行，安西 祐一郎, “不揮発メモリを用いた永続システムの設計方法”, 電子情報通信学会論文誌 Vol.J85-D-I, No.9, pp.817-830 (2002 年 9 月).

2. 大村 廉，山崎 信行，安西 祐一郎, “主記憶に不揮発メモリを用いたシステムの実行状態復元手法”, 情報処理学会 Vol.45, No.SIG1(ACS 4), pp.88-99 (2004 年 1 月).

【 国際会議発表 】

3. Ren Ohmura, Nobuyuki Yamasaki, Yuichiro Anzai, “Device State Recovery in Non-volatile Main Memory Systems”, Proc. of The 27th Annual Intl. Computer Software and Applications Conf.(COMPSAC2003), pp.16-21, Nov. 2003.

4. Ren Ohmura, Nobuyuki Yamasaki, Yuichiro Anzai, “A Design of the Persistent Operating System with Non-volatile Memory”, Proc. of The 10th ACM SIGOPS European Workshop, pp.149-152, Sept. 2002.

【 国内学会発表 】

5. 大村 廉，山崎 信行，安西 祐一郎. “不揮発 RAM を用いた Persistent OS におけるカーネルメモリマネージメント” 情報処理学会研究会報告 (RTP2001) 2001-OS-86，Vol.2001，No.21，pp.99-106，(2001 年 3 月).

6. 大村 廉，山崎 信行，安西 祐一郎. “不揮発 RAM を用いたシステムにおける主記憶管理手法” 電子情報通信学会技術研究報告 DE2001-112，Vol.101，No.343，pp.25-32，2001.

7. 大村 廉，山崎 信行，安西 祐一郎．“不揮発 RAM を用いた Persistent OS におけるデバイスマネージメント” 情報処理学会コンピュータシステム・シンポジウム論文集，IPSJ Symposium Series Vol.2001，No.16，pp.73-80，2001.

【 同一著者による他の研究成果 】

8. 大村 廉，河野 通宗，安西 祐一郎, “移動センサノードに対応したセンサノード管理システムの拡張”, 情報処理学会 OS 研究会・電子情報通信学会 CPSY 研究会合同ワークショップ, (1999 年 5 月).