平成21年度　　卒業研究論文

耐障害性を有したマイクロカーネルOSの実装と再起動プロセスの効率化の提案

指導教官
齋藤 彰一 准教授

名古屋工業大学　情報工学科
平成１８年度入学　18115046番

加藤　雄大

目 次

第 1 章　始めに **1**

第 2 章　関連研究 **3**

2.1　MINIX 手法 4

2.1.1　マイクロカーネル 4

2.1.2　MINIX におけるドライバの隔離 5

2.1.3　MINIX におけるマイクロリブート 6

2.2　シャドードライバ 7

2.2.1　シャドードライバにおけるドライバの隔離 7

2.2.2　シャドードライバにおけるマイクロリブート 7

2.3　関連研究まとめ 8

第 3 章　提案システム **9**

3.1　アプリケーションによるカーネル内のデータの操作 10

3.2　リブートプロセスが必要とするカーネルデータ 11

3.3　アクセス許可範囲の設定 12

3.4　アクセス許可のタイミング 12

3.5　提案方式まとめ 13

第 4 章　提案方式の実装 **14**

4.1　実行環境 14

4.2　各機能毎のコードの行数 14

4.3　提案 OS の全体像 14

4.3.1 システムタスク 16
4.3.2 プロセスマネージャ 16
4.4 リブート処理の実装 17
4.4.1 ハンドラの処理 18
4.4.2 システムタスクの処理① 18
4.4.3 プロセスマネージャの処理 19
4.4.4 システムタスクの処理② 21
4.5 実装評価 23

第 5 章 まとめ **25**

参考文献 **27**

第1章

始めに

PCやインターネットの普及によって,計算機に様々な信頼性が求められるようになった．特に商用システムにおいてはサービスが安定して提供できない場合,利益を損なう可能性がある. しかしコンピュータシステムは少からず誤りを含んでいる. それらを完全に取り除くことは困難であるため,誤り許容できるような耐障害性を有したシステムが求められる.

耐障害性を考える上でオペレーティングシステム（OS）が重要な基盤であると考える．なぜなら OS の誤りは, その上で実行されているアプリケーション等の全てコンポーネントに影響を与える可能性がある．特に OS がクラッシュした場合,OS 上で動作している 全てのアプリケーションが停止する．OS のクラッシュの原因は様々だが,WindowsXP の場合はドライバが原因として起こったクラッシュが全体の８５パーセントを占めている [1]．よって他の一般的な OS でも同様にドライバが原因でクラッシュしている可能性が高いと考えられる. ドライバが OS のクラッシュの引き金になる理由は２点ある.

- ドライバは誤りを多く含んでいること
- モノリシックな OS ではドライバがカーネル内にあること

ドライバはハードウェアが複雑化するにつれて誤りや脆弱性を多く含むようになった. 同様にアプリケーションも誤りを多く含んでいるが, アプリケーションとカーネルは隔離されているため, アプリケーションのエラーが原因で OS がクラッシュする可能性は

低い. しかしドライバの場合は, カーネルから隔離をなされていないために誤りによってカーネルを巻き込み,OSがクラッシュする. そこで本研究ではマイクロカーネルを用いたドライバに対する耐障害性の研究に着目し, 作成したマイクロカーネル上に処理を効率化するための機能を加えた.

２章では提案手法の基盤となる既存研究や関連研究について述べ, ３章では提案手法の詳細について述べる. 続く４章では提案システムの実装と評価について述べる. そして５章では結論をまとめる.

第2章

関連研究

本章ではドライバに対する耐障害性を有したOSの既存研究について述べる．まずマイクロカーネルOSであるMINIX[2]における研究[3]について述べ, 次にLinux上で実装されているシャドードライバ[1]について述べる．2つの研究はカーネルをドライバから隔離することで保護していること, そしてOSがアプリケーションに対してドライバのエラーを隠蔽する事が可能であることの2点が共通している．2つの共通点のうち,ドライバをOSから隔離する方法は両者で大きくことなるが, アプリケーションに対してドライバのエラーを隠蔽する方法は次に述べるマイクロリブート[4]に基づくアプローチがとられている点が共通している．マイクロリブートとは再起動をシステム全体に対して行うのではなく,より小さな範囲に対して行うことでエラーを取り除く方法である．マイクロリブートが適用可能なシステムは次の条件を満たしている必要がある．

- コンポーネント同士が隔離されていること
- 状態を保存する仕組みがあること
- リブート中のコンポーネントへのリクエストを透過的に再試行する仕組みがあること

1つめはコンポーネントが他のコンポーネントを巻き込んでクラッシュすることを防ぐための条件である．2つめは再起動した後のコンポーネントをクラッシュ以前の状態に戻すために必要な条件である．3つめはコンポーネントがクラッシュして利用不可能

になっている事を隠蔽するために必要な条件である. 復帰後のコンポーネントに対してリクエストを再試行することでリクエストは正しく処理される. 次に MINIX の研究 [3] とシャドードライバ [1] について述べるが, マイクロリブートの実現方法を要点として述べる.

2.1 MINIX 手法

MINIX はマイクロカーネルを採用している OS である. MINIX の手法について述べるにはマイクロカーネルの概要について述べる必要があるので, まずマイクロカーネルについて述べ, 次に MINIX におけるドライバのマイクロリブートを行う方法について述べる.

2.1.1 マイクロカーネル

カーネルは OS の中核を担うコンポーネントである.OS はユーザに対し リソースを抽象化するがカーネルは特別な命令を実行できるために最も抽象度の低いコンポーネントである. マイクロカーネルとはカーネルは最小限の機能を有し ,OS の機能の大半をアプリケーションとして実装するカーネルの設計手法である. 最小限の機能とはプロセス間通信, 仮想記憶管理, プロセス制御である. それに対してカーネルに全機能を含める設計方法をモノリシックカーネルと呼ぶ. モノリシックカーネルとマイクロカーネル OS の概念図を図 2.1 に示す.

図 2.1 にはマイクロカーネルがアプリケーション, サーバ, ドライバ, カーネルから成り立っていることが示されている. モノリシックカーネル OS と比較してマイクロカーネル OS のカーネルは小さい. これは OS の持つ機能の大半がアプリケーションとして実装されているからである. サーバとはドライバおよびカーネル機能を除いた OS の機能を担うプロセスの事であり, 機能ごとに異なるプロセスとして実装される. 図 2.1 ではモノリシックカーネルにおけるファイルシステムがファイルサーバとして動作している事が示されている. これらのプロセス間の通信にはメッセージパッシングが用いられる. マイクロカーネルがモノリシックカーネルと比較して問題になるのはこの

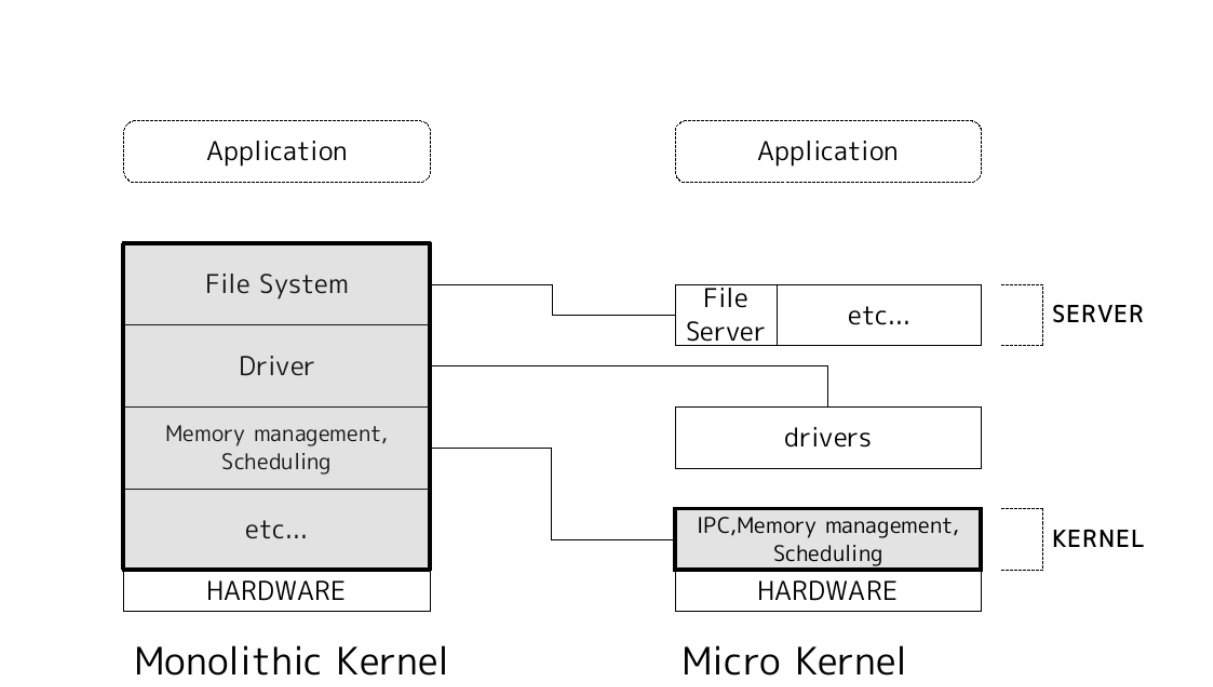

図 2.1: カーネル概念図 モノリシックカーネルとマイクロカーネル

メッセージパッシングによる処理量が大きく,パフォーマンスが低くなることである. しかし, 機能がモジュール化しやすいこと, また分散環境と親和性が高いこと, そしてコンポーネント同士が隔離されていることから堅牢性が高いなどの利点を有している. MINIX における手法はこの堅牢性が基盤となっている.

2.1.2 MINIX におけるドライバの隔離

まず OS のクラッシュを防ぐためにカーネルをドライバから隔離する方法について述べる. マイクロカーネルの設計上,MINIX のドライバはアプリケーションとして実装されている. したがってドライバはカーネル空間への直接的なアクセスを行うことができない. ゆえにメモリアクセスを通じてカーネルをクラッシュさせる可能性を低

くすることができる. しかしドライバは他のコンポーネントと異なり, ハードウェアを制御する操作が必要となり, ハードウェアの制御は危険性が高い. 例えば DMA 転送によって物理メモリで指定されたメモリ上にデータを転送できる. これはドライバが任意の物理メモリを上書きすることが可能であり, ドライバが誤った操作を行うとシステムの重要なデータが破壊され, 甚大な影響を与える危険性がある. そこでこの危険性を最小限に抑える必要がある. 危険性を最小限にするためにドライバが実行できる操作を限定する方法をとる. ユーザプロセスであるドライバは直接ハードウェアを制御することはできない. したがってドライバがハードウェアを操作するには実行レベルの高いプロセスに仲介させるか, システムコールを利用する. そこで MINIX はこれらを監視し, 各プロセスの利用可能な操作を限定し, 最小限の権限しか与えないことで危険性を最小限に留めることができる. このようにドライバをユーザプロセスとして実行することでカーネルを保護し, 同時にハードウェア操作にソフトウェアレベルの細かな制限を行うことでドライバの隔離を行っている.

2.1.3 MINIX におけるマイクロリブート

MINIX においてドライバが発生させるエラーは再生サーバという特殊なサーバが検知する. 再生サーバはすべてのプロセスの親プロセスであり, ドライバを含めたすべてのプロセスが終了した際にプロセス管理プロセスによる通知を受け取る. そして通知された情報からドライバが異常状態で終了したことを検知した場合, マイクロリブートが行われる. また終了だけではなく再生サーバは定期的な生存確認メッセージを送信し, ドライバが正常に動作しているかを確認しており, 確認がとれない場合も同様にマイクロリブートが行われる. そしてドライバのリブートは実行ファイルから行われる. マイクロリブートには状態の保存とリクエストの再試行が必要だと述べた. MINIX ではブロックデバイス, ネットワークデバイス, キャラクタデバイスに対してマイクロリブートを適用しているが, これらのドライバは状態を持たないとして状態の復元を行っていない. しかし状態の復元を行うことは可能だとしている. またリクエストの再試行は MINIX が一時的な宛先を用いたランデブー式のメッセージパッシングを行っていることから, その一時的な宛先の変更をドライバを利用しているサーバに通知し, ドラ

イバを扱うサーバ側が処理途中のリクエストを再試行することで行われる.

2.2 シャドードライバ

シャドードライバはデバイスドライバに対する耐障害性を向上する研究 [1] であり linux 上で実装されている. linux はモノリシックカーネルなのでドライバがカーネル内に存在する. よってドライバからカーネルを保護するために特別な仕組みが必要となる. ドライバがエラーを発生させた時に, シャドードライバというドライバが代わりに動作することで,ドライバを利用するコンポーネントからドライバのエラーを隠蔽する. まずドライバの隔離方法について述べた後マイクロリブートについて述べる.

2.2.1 シャドードライバにおけるドライバの隔離

シャドードライバはデバイスドライバからカーネルを保護するために Nooks[5] を用いている. Nooks ではデバイスドライバを nook と呼ばれる isolation domain に閉じ込め特定のメモリ空間以外へのアクセスを禁止することで,ドライバのバグからカーネルを保護する. これは仮想記憶を利用してページ単位で設定される. これによってドライバがバグからエラーを発生させ, 領域を侵犯するような動作をした際には例外が発生し再起動のトリガーになる.

2.2.2 シャドードライバにおけるマイクロリブート

シャドードライバにおけるドライバのマイクロリブートについて述べる. シャドードライバには Active と Passive の 2 つのモードが存在する. Passive モードの場合, ドライバの入出力を監視しドライバの状態を調べている. ドライバからのエラー通知を受けると Active モードに代わり停止させられたドライバの代わりに動作する. そしてドライバが復帰後はまた Passive モードになる. このようにしてドライバのエラー時にシャドードライバがドライバの代わりに動作することでエラーの隠蔽が行われる.

2.3 関連研究まとめ

ドライバのエラーからカーネルを保護し,かつ他のコンポーネントから透過的にドライバを復帰させる既存システムについて述べた. MINIX の方法とシャドードライバの方法をドライバの隔離について比較するとシャドードライバは依然としてカーネルレベルでデバイスドライバが動作しているので危険性がより高いと考えられる. そこで本研究ではマイクロカーネルの構造的を利用した手法に着目する.

第3章

提案システム

2章ではドライバに対する耐障害性の研究について述べたが,MINIXにおける手法はドライバだけではなくサーバプロセスに対してもマイクロリブートを適用する事が可能だとし, 今後の課題としている. しかしサーバに対してマイクロリブートを行うにはパフォーマンスが問題になる可能性が高い. なぜならばサーバはドライバと異なり, さまざまな状態を持ち, 通信を行うプロセスも多いので, 再起動後に伴う処理はそれに比例して増大するためである.

また一般的にマイクロカーネルがモノリシックカーネルに比べてパフォーマンスが低いという問題がある. これはOSの機能の大半がアプリケーションとして実装されるために, プロセス間の通信の手段であるメッセージパッシングが増加することが原因だと2章で述べた. マイクロリブートにおいてもメッセージパッシングの増大による処理量は大きく, メッセージパッシングの処理量を削減することでリブートを高速化できると考えた. そのためにはメッセージパッシングの速度を向上させるか, メッセージの送受信回数を減らす必要がある. メッセージパッシングの速度の向上は研究されてきており [6], かつマイクロカーネルの抱える一般的な問題であるので本研究対象とはしない. 本研究では特定の処理に併せてプロセス間通信の回数を削減する方法を検討する.

そこでメッセージパッシングの回数の削減を行うにあたってリブートを担うプロセス (リブートプロセス) がカーネル内のデータに対する特定の処理が必要であることに着目した. 後述するようにユーザプロセスであるリブートプロセスからカーネル内のデータになんらかの処理を行うにはメッセージパッシングを行う必要がある. そこで

リブートプロセスにカーネル内のデータに直接アクセスさせることでメッセージパッシングを削減する方法を用いることで高速化が可能である.

しかし直接アクセスには危険性が伴う. なぜならばリブートプロセスによってカーネルが破壊される可能性が発生するためである. これは本研究の懸案事項である耐障害性を低下させることになる. しかしリブートプロセスはドライバと異なり, バグや脆弱性を含みにくいと考えられる. なぜならばリブートプロセスはハードウェアを操作する必要が無く, 複雑になりにくいためである. よって直接アクセスに対して一定の制約を加えることで安全性を確保できると考える.

そこでリブートプロセスがカーネル内のデータにアクセス可能な時間と範囲を限定するという制約を用いる. 本研究はカーネル内のメモリへの直接アクセスを一時的かつ部分的に許可することによりリブートプロセスを高速化する手法を提案する.

まずカーネル内のデータに対する処理がメッセージパッシングを必要とする理由を述べる. その次にリブート時にカーネル内のデータのうちで何を必要とされるかついて述べる. そして直接アクセスに対する制限を行う方法について述べる.

3.1 アプリケーションによるカーネル内のデータの操作

提案方法を用いない場合, アプリケーションからカーネル内のデータは直接操作できないと述べた. 直接操作することが出来ない場合は間接的に操作するが, これは主に 2 通りの方法が存在する. 一つはシステムコールを使用する方法である. システムコールとは低い実行権限のコードから高い実行権限のコードを呼び出す仕組みである. システムコールを用いて, あらかじめ用意されたルーチンを呼び出し, カーネルデータへのアクセスを行うことができる. もう一つの方法はユーザプロセスがシステムプロセスに対してプロセス間通信を用いて操作を依頼する方法である. ユーザプロセスがこのシステムプロセスに対してプロセス間通信を行い操作を依頼する. システムプロセスはユーザプロセスと異なり, 実行レベルが高く, カーネル空間にアクセス可能である. ゆえにユーザプロセスが依頼した操作をシステムプロセスが代わりに行うことができる仕組みである.

いくつかの操作はシステムコールによって実現される. たとえばメッセージパッシングはシステムコールを用いて行われる. しかし大半のカーネル内のデータに対する処理はメッセージパッシングを用いて行われる方が都合がいい. なぜメッセージパッシングによってシステムプロセスが処理する方がシステムコールによって処理するよりも都合がいいのかについて述べる. システムコールによる処理はシステムコールを行った直後から開始され, 複数のプロセスが同時に同じデータを呼び出す可能性があるため排他処理が必要であり, 処理の記述が複雑になりがちである. しかしメッセージパッシングの場合はシステムプロセスが受信処理を行うまで処理が開始されない. つまり複数のプロセスが処理要求を同時に行っても 1 つ 1 つ受信し, 処理することができる. よってメッセージパッシングによる処理の方がよりシンプルに処理を記述可能であり, 都合がよい. ゆえにカーネル内のデータの処理の多くはメッセージパッシングを用いて行われる.

3.2 リブートプロセスが必要とするカーネルデータ

マイクロカーネル OS においてカーネルは自身の機能であるプロセス間通信, 仮想記憶管理, プロセス制御に関するデータを持っている. この中でリブートプロセスが必要とするのはプロセス間通信とプロセス制御に関する情報である.

2 章のはじめにマイクロリブートが適用可能なシステムはリブート中のコンポーネントへのリクエストを透過的に再試行する仕組みが必要であると述べた. そしてマイクロカーネルではリクエストはメッセージパッシングによって行われる. ゆえにリブートプロセスはメッセージパッシングの再試行を行う仕組みが必要になる. しかしメッセージパッシングに関する情報はカーネル内にある. つまりリブートプロセスがカーネル内で必要としている情報はメッセージパッシングに関する情報である. そしてメッセージパッシングはプロセスの制御情報も必要であるため, プロセスメッセージパッシングの情報とプロセスの制御情報を併せて必要とする.

3.3　アクセス許可範囲の設定

リブートプロセスに特定のカーネルメモリ領域へのアクセスを許可する方法について述べる. アクセスの許可範囲の設定は1つの構造体の範囲に限定した.この構造体については4章で詳しく述べる. アクセスを行うべき最小範囲は構造体の先頭アドレスと末尾アドレスの間である. しかし許可範囲の設定は仮想記憶を用いてページ単位で行われるので, 実際に許可を行うのは先頭番地を含むページから末尾アドレスを含むページまでとなる. このようにして求めたページをユーザ空間にマッピングすることでユーザプロセスに対してカーネルデータへのアクセスを許可する.

3.4　アクセス許可のタイミング

リブートプロセスにカーネル領域へのアクセスを許可するタイミングについて述べる. アクセス許可の開始と終了は次のようなステップによって実行される.

1. デバイスプロセスの異常終了
2. リブートプロセスへの通知
3. (開始) システムプロセスが許可範囲の設定
4. リブートプロセスがリブートを開始
5. リブート終了をシステムプロセスへ通知
6. (終了) 許可範囲の無効化

まずドライバが終了したことがリブートプロセスに通知される. プロセスマネージャはアクセスを許可の設定を行うためにシステムプロセスに対してドライバの終了を通知する. 通知を受けたシステムプロセスは3.3で述べた許可範囲の設定を行う. 設定を終えるとリブートプロセスに許可範囲の設定を終えたことを通知する. リブートプロセスはリブートを開始し,終了するとシステムプロセスにリブートの終了を通知する. システムプロセスは許可範囲の設定の無効化を行う.

3.5　提案方式まとめ

マイクロカーネルによる再起動の方法を高速化するために一時的かつ部分的なアクセス許可を行うことを提案した．リブートプロセスを高速化するために, なぜカーネルデータへのアクセスが必要なのかを述べた．またアクセス許可範囲の設定方法およびアクセス許可の開始のタイミングと終了のタイミングについて述べた．

第4章

提案方式の実装

本研究ではマイクロカーネルOSの実装を行い, 実装したOS内にドライバに対するマイクロリブートを行うシステムを実装した. 本章では実装したOSの概要を述べ, その後にマイクロリブートの実装について述べる.

4.1 実行環境

実装したOSの実行環境について述べる. 実行はプロセッサエミュレータであるQemu[7]上で行った. 対象とするCPUはintel486プロセッサである. ブートおよび2次記憶装置としてフロッピーを使用した. 実装言語はC++とNASMを使用した.

4.2 各機能毎のコードの行数

表4.1に実装を行ったOSのカーネルの各機能に対するコードの行数を示す. 全体で3589行であり, 内訳はプロセス管理が1008行, メッセージパッシングが324行, メモリ管理が666行, その他が1297行, 起動及び初期化が294行である.

4.3 提案OSの全体像

実装したOSについて概要を述べる. OS全体を図4.2に示す. 図4.2で示されるように提案OSはカーネル, ドライバ, サーバの各層で構成される. またアプリケーションを

図 4.1: カーネル機能と行数

機能	行数
全体	3589
プロセス管理	1008
メッセージパッシング	324
メモリ管理	666
起動および初期化	294
その他	1297

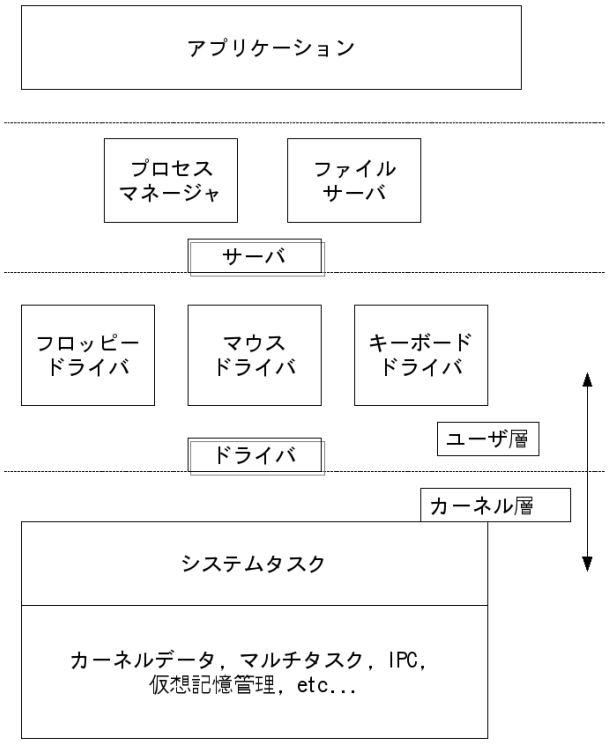

図 4.2: OS の全体像

含むシステム全体はカーネル層とユーザ層に分けられる. カーネル層にはカーネルがある. カーネルはプロセスではないが, 提案システムにおいてシステムタスクと呼ぶプロセスがカーネルの中で実行されており, カーネルにアクセスすることが可能である. ユーザ層ではドライバ, サーバ, アプリケーションがユーザプロセスとして実行されている. ドライバには実装の評価においてマイクロリブートの対象となるフロッピードライバがある. サーバにはファイルサーバとプロセスマネージャがある. ファイルサーバはフロッピードライバと通信することでフロッピーに対して読み書きを行うプロセスだが, 実装の評価のために特定のディスクの箇所を読み書きするように単純化されている. プロセスマネージャは後述するが, プロセスの抽象的な情報を管理するプロセスである. 本節ではシステムタスクとプロセスマネージャについて述べる.

4.3.1 システムタスク

図 4.2 においてシステムタスクがカーネル層に属していることが示されている. システムタスクは OS のコンパイル時に他のカーネルコンポーネントと共にリンクされる. そして実行時に高い特権レベルを与えらるプロセスである. ゆえにカーネル全体にアクセス可能なプロセスである.

システムタスクはプロセスマネージャを起動した後, メッセージを待ち受ける状態になる. 受け付けるメッセージはサーバプロセスからのメッセージと保護例外によってカーネル内のハンドラから送信されるエラーメッセージである. サーバからの要求の例として仮想アドレスから物理アドレスへの変換要求やプロセスの生成の要求がある. またエラーメッセージについては 4.4 で述べる.

4.3.2 プロセスマネージャ

プロセスマネージャが持つ機能は次の 2 つである.

- プロセス情報の管理
- ドライバのリブート処理

1つめの機能であるプロセス情報の管理とは本提案システムではドライバプロセスのPIDと実行ファイルのファイル名の対応を扱うことである. これはリブート処理に用いられる.

2つめの機能はドライバのリブート処理である. MINIXのシステムは再生サーバがリブート専用のプロセスとして実行されていたが, 本提案システムではプロセスマネージャがリブート処理を行う. これはリブート専用のプロセスを作成するよりもプロセスマネージャがリブート処理を行う方がリブート処理が高速になるからである.

なぜ高速化するのかについて述べる. まず本提案はリブートに必要な情報がカーネルに存在することで発生するメッセージパッシングを削減しリブート処理を高速化することである. これは必要な情報がカーネルとリブートを行うプロセスの間で分散しているということがメッセージパッシングを発生させる原因であり, 分散した情報を集約しメッセージパッシングを削減することで高速化可能だと述べた. これはリブートプロセスとカーネルの間だけではなく異なるプロセスの間でも成立する. あるプロセスは異なるプロセスの持つ情報に対してアクセスすることはできない. 異なるプロセスのもつ情報を処理するためにはメッセージパッシングを必要とする. つまりリブートを行うプロセスとプロセスマネージャを異なるプロセスとして実装した場合, リブートプロセスはプロセスマネージャの情報を必要とするので両者の間でメッセージパッシングが発生する. そこでプロセスマネージャがリブートプロセスの機能を兼ねることでメッセージパッシングを削減し, 高速化できる.

4.4 リブート処理の実装

図4.3にリブートの手順を示す. 図4.3よりリブートの処理は保護例外およびページフォルトが発生しハンドラが実行されるところから始まる. そしてシステムタスク (図4.3中の①), プロセスマネージャ, もう一度システムタスク (図4.3中の②) という順にそれぞれがリブート処理を行う. それぞれのリブート処理について述べる.

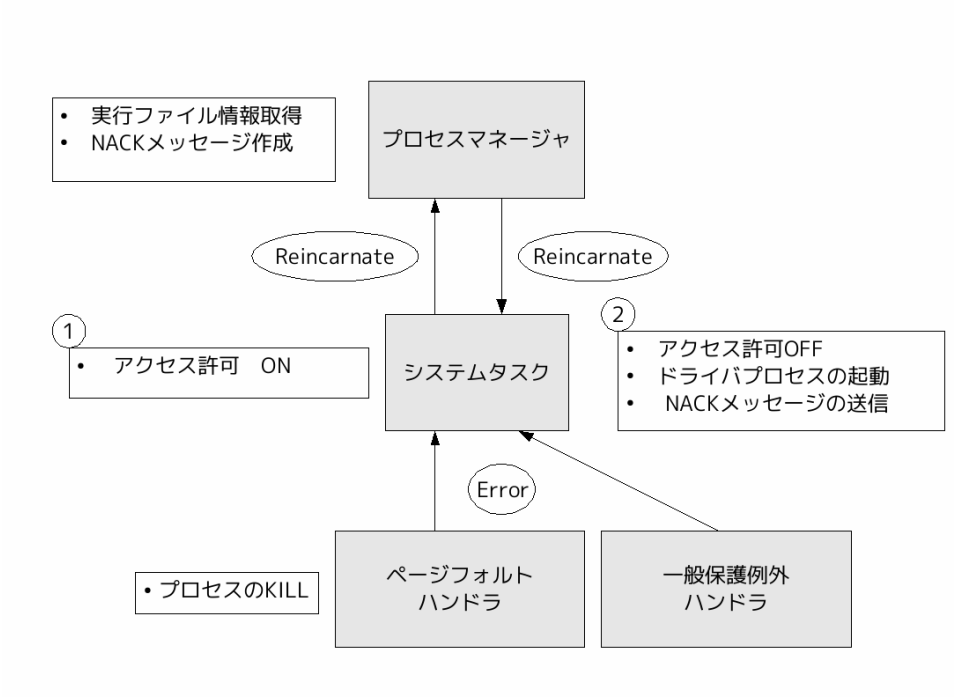

図 4.3: リブートの手順

4.4.1 ハンドラの処理

まずエラーは一般保護例外とページフォールトの発生によって検知される．これらは発生とともにハンドラが起動する．ハンドラはエラーを起こしたプロセスを停止した後, システムタスクへ通知を行う．通知にはメッセージパッシングを使用する．メッセージパッシングは図 4.5 の IPCmessage を送信するプロセスのバッファーから受信するプロセスのバッファーへコピーすることで行われる．エラーのメッセージは IPCmessage の body にエラーを表わすデータをコピーすることで作成される．そしてエラーのメッセージをシステムタスクへ送信することでハンドラの処理が終了する．

4.4.2 システムタスクの処理①

システムタスクはエラーの通知を受けて,プロセスマネージャに対してカーネルメモリ領域へのアクセス許可を行った後, プロセスマネージャにメッセージを送信する．ま

ずアクセスを許可する方法の実装について述べる.

プロセスマネージャの仮想メモリ空間は図 4.4 で表される. 仮想メモリ空間の前半はカーネルが使用する固定長のメモリ空間が存在する. カーネルが有する情報は大半がこのメモリ空間にあり, 図 4.4 に表わされるようにカーネルのプロセス管理情報もこの空間にある. カーネル固定空間を除いた仮想メモリ空間の後半はユーザ空間である. プロセスマネージャのコード, データおよびスタック領域はここにマッピングされる. そしてカーネルのプロセス管理情報をこのユーザ空間で予め確保されている未使用空間にマッピングしアクセス権限を下げることでカーネルのメモリ領域へのアクセス許可を行うことができる.

次にシステムタスクはプロセスマネージャへメッセージを送信する. このメッセージについて詳しく述べる. リブート時にシステムプロセスがプロセスマネージャに送るメッセージには図 4.6 で表わされる ReinCarnate 構造体が挿入される. ReinCarnate 構造体の 1 つめのメンバである process は図 4.4 においてカーネル固定空間からユーザ空間にマッピングされたプロセス管理情報の先頭アドレスである. 2 つめのメンバの pid はエラーを起こしたプロセスの ID である. 3 つめのメンバの mbuff は図 4.5 のメッセージと宛先情報を含む構造体の配列である. 4 つめのメンバの img はドライバの実行ファイルの情報を表わす. mbuff と img はプロセスマネージャがデータを挿入するので, この段階では空の配列である.

システムプロセスはメッセージを送りおえると通常のメッセージを待ち受ける状態に戻る. しかしリブート処理を終えたプロセスマネージャからメッセージを受信すると再びリブート処理を行う. これについては 4.4.4 で述べる.

4.4.3 プロセスマネージャの処理

プロセスマネージャはシステムタスクからメッセージを受け取るとリブート処理を行い, システムタスクにメッセージを送信する. プロセスマネージャは先に述べたようメッセージを介して ReinCarnate 構造体を受け取る. プロセスマネージャーは受け取ったメッセージからエラーを起こしたプロセスがドライバかどうかを自身の管理するプロセス情報から判別する. もしエラーを起こしたプロセスがドライバ以外のプロ

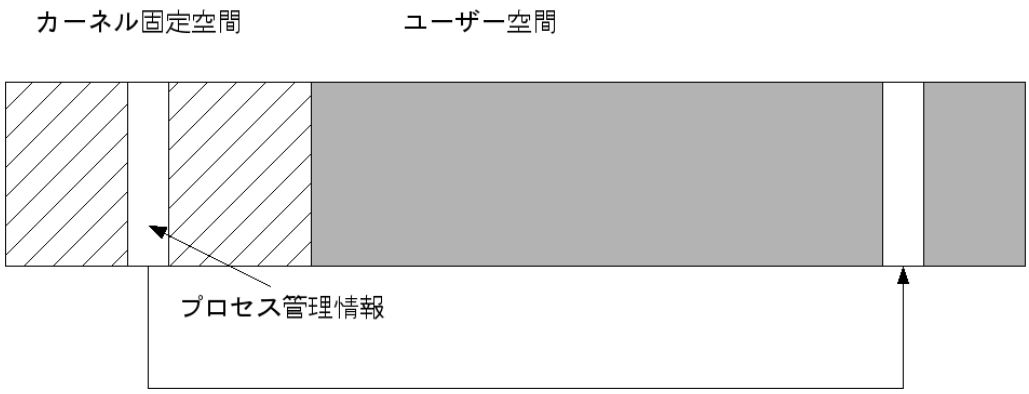

図 4.4: メモリの配置

セスである場合はプロセスマネージャはシステムプロセスから受け取ったメッセージを送り返し, 何もせず処理を終える. エラーを起こしたプロセスがドライバである場合はリブート処理を始める. プロセスマネージャが行うリブート処理は 2 つある.

- エラーを起こしたドライバに対応する実行ファイル情報の取得
- サーバプロセスに送る NACK メッセージの作成

プロセスマネージャが 1 つめの処理で取得した実行ファイル情報は ReinCarnate 構造体中の img にコピーされ,2 つめの処理で作成したメッセージは mbuff にコピーされる.

まずプロセスマネージャは実行ファイル情報の取得するが実行ファイル情報はプロセスマネージャ自身が管理している情報に存在し, プロセス ID から該当する実行ファ

イル情報を取得可能である．そこで ReinCarnate 構造体のメンバである pid からエラーを起こしたドライバの実行ファイル情報を取得する．

次に mbuff へ送信すべきメッセージの情報を挿入していく．mbuff はドライバに対する操作が失敗したことをドライバを利用していたサーバプロセスに伝えるために必要である．なぜドライバに対する操作の失敗をサーバプロセスに伝える必要があるのかについて述べるためにドライバを利用するサーバプロセスについて概略を述べる．本提案システムではフロッピードライバとファイルサーバが通信をしている．ファイルサーバはドライバに対してディスク上の特定の箇所を読み書きする様にメッセージを送る．メッセージを受け取ったドライバプロセスは IO を行い結果を得る．しかしハードウェアを操作している都合上, 操作に失敗する可能性がある．操作に失敗した場合はフロッピードライバはファイルサーバに操作が失敗したことを表わす NACK を挿入したメッセージ送信する．NACK を受け取ったファイルサーバは失敗回数が設定した数値に達するまで同じ操作をドライバに行わせる．すなわちファイルサーバは NACK を受け取ることで操作を再試行する仕組みを持っている．

リブート処理においてこの仕組みを利用することで,ドライバおよびデバイスの状態を保持する複雑な仕組みを省くことができる．複雑な仕組みとは 2 章で述べたシャドードライバ [1] の様な仕組みである．しかし複雑な状態保存を行わなくてもドライバとデバイスを同時に初期化し, サーバが操作を再試行することでほとんどのドライバの操作をやり直すことができる [3]．本システムのリブート処理ではデバイスドライバに操作を要求したサーバに対して NACK を送信することでサーバに操作を再試行させる．

プロセスマネージャは NACK メッセージを ReinCarnate 構造体にコピーする．そしてその ReinCarnate 構造体をメッセージに挿入しシステムタスクに送信し, 処理を終える．

4.4.4　システムタスクの処理②

システムタスクはプロセスマネージャから ReinCarnate 構造体が挿入されたメッセージを受け取る．まずシステムタスクはプロセスマネージャがカーネルメモリへアクセス可能な状態を解除する．次にドライバのリブートを行う．最後にプロセスマネージャ

が作成した NACK メッセージを送信する.

まずドライバのリブートについて述べる. 実行ファイルは物理メモリ上の連続する空間に保存されている. これはフロッピードライバが利用不可能になることでディスクの読み書きが行えないためディスク上に保管しても取り出すことができないからである. システムタスクはこのメモリ上の実行ファイルから新しくプロセスを生成し, 実行可能な状態にする. そして古いドライバプロセスの ID を新しいプロセスの ID につける. これによってエラーを起こしたドライバプロセスが完全に初期化された事になる. そしてドライバプロセスを実行することでドライバが復帰する.

次に NACK メッセージの送信を行う. プロセスマネージャ作成したメッセージ情報は宛先がサーバプロセス, 送信元がドライバ, メッセージが NACK という情報であり, これをシステムタスクがあたかもドライバが送信したように宛先のサーバに送信する. これによってサーバは操作を再試行し, 正常な結果を得ることができる.

```
struct IPCmessage{
MessageType type;
int length;
u8_t body[BODYSIZE];
};
```

図 4.5: メッセージ構造体

```
class ReinCarnate{
public:
Process *process;
int pid;
MessageInfo mbuff[MaxMessageNum];
        BinaryStore::Img img;
};
```

図 4.6: リブート構造体

```
int sendReinCarnate(){
  IPCmessage message;
  message.type=IPCmessage::REINCARNATE;
  ReinCarnate *rc=(ReinCarnate *)message.body;
  rc->pid=error.pageFault.pid;
  rc->process=(Process  *)(UserProcessStructAddr);
  int ret=send(P_PROCESS_MANAGER,&message);
  return ret;
}
```

図 4.7: ページフォールトの発生をプロセスマネージャに送信するコードの抜粋

4.5 実装評価

フロッピーデバイスドライバに対し, カーネル領域へのアクセスを行うバグを挿入した. これによってメモリ保護違反が発生し, リブートが行われフロッピーデバイスドライバが再起動することを確認した. またフロッピーデバイスドライバがエラーを発生させ, リブートされて復帰するまでにかかった時間の平均を表 4.8 の環境で測定した. 共有を行わない場合と共有を行う場合で比較したところ表 4.9 で表わされる結果を得た. 共有を行わない場合に比べて共有を行う場合は 26 %の処理時間を削減しており, 共有を行うことがデバイスドライバのマイクロリブートの処理量を削減することに有効であることを確認した.

この結果は MINIX における研究 [3] と単純に比較することができない. 本研究の今後の課題は, このリブート処理に必要な時間の測定を様々な環境下で行い比較評価することである.

図 4.8: 評価環境

OS	Vine Linux 5.0
カーネルバージョン	2.6.27
CPU	Pentium4 2.8GHz
メモリ	2Gbyte
Qemu バージョン	0.9.1

図 4.9: マイクロリブートの処理時間の評価

データ共有を行わなかった場合	24.294 ミリ秒
データ共有を行った場合	18.039 ミリ秒

第5章

まとめ

マイクロカーネル OS を実装し, その上にデバイスドライバのエラーに対し耐障害性を有するシステムを作成し, その高速化を提案した. まずデバイスドライバに対する耐故障性が必要であることについて述べ, 既存研究であるマイクカーネル OS である MINIX におけるドライバに耐故障システムと Linux 上で実装されているシャドードライバについて述べ, 両者の違いについて言及し, マイクカーネルの構造的な堅牢性が将来の OS にとって必要だと考えマイクロカーネルを使ったデバイスドライバの再起動システムを作成した.

マイクロカーネルは機能ごとにユーザプロセスとして実装されることによってさまざまな利点をもつ反面, パフォーマンスが低下してしまうことから, パフォーマンスが低下しにくいように, 再起動プロセスに一時的かつ部分的なカーネル空間のアクセスを許可することによって, より速く, かつなるべく安全性を損ねることなくデバイスドライバを再起動できる方法を提案し, 実装した.

今後の課題は既存システムと比較評価を行うことや, アクセス許可を他の処理に適用することができるか考察することである.

謝辞

日頃から熱心に指導して頂き, また研究課題を決める際など、親身に相談に乗って頂いた名古屋工業大学の齋藤彰一准教授に深く感謝いたします.

また, 本研究の際に多くの助言，協力をして頂いた松尾啓志教授，津邑公暁准教授，松井俊浩助教，及び齋藤研究室ならびに松尾・津邑研究室の皆様に深く感謝致します.

参考文献

[1] M. Swift, M. Annamalai, B. Bershad, and H. Levy: Recovering Device Drivers. ACM TRANSACTIONS ON COMPUTER SYSTEMS , VOL 24; NUMB 4, pp 333–360 (2006).

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

[3] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum: Failure Resilience for Device Drivers, Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, pp 41–50 (2007).

[4] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox: Microreboot–A Technique for Cheap Recovery. In Proc. 6th Symp. on Oper. Syst. Design and Impl., pp 31–44 (2004)

[5] M. Swift, B. Bershad, and H. Levy: Improving the Reliability of Commodity Operating Systems,ACM Transactions on Computer Systems, Vol 23, Issue 1, pp 77–100 (2005)

[6] Jochen Liedtke :Improving IPC by kernel design, Proceedings of the fourteenth ACM symposium on Operating systems principles, pp 175–188 (1994)

[7] Qemu: http://www.qemu.org.