平成22年度 修士論文

実時間性を確保した
ハイブリッドOSの実装

電気通信大学 大学院情報システム学研究科

情報システム基盤学専攻

0953022 湯山 圭一

指導教員 多田 好克 教授
小宮 常康 准教授
大森 匡 教授

提出日 平成23年1月27日

目 次

図目次

表目次

第 1 章

序論

本章では，本研究の概要と論文の構成について述べる．

1.1 概要

ハイブリッドオペレーティングシステム (以下: ハイブリッド OS) の一形態に，リアルタイムオペレーティングシステム (以下: RTOS) の一タスクとして Unix を動作させるものがある [1]．このようなハイブリッド OS を使用することで，実時間性と多機能性を，単一のシステムで両立できる．しかしながらハイブリッド OS では，Unix が RTOS への遷移を長時間阻害すると実時間性が損なわれるという問題がある．本研究では，この問題を解決し，実時間性を確保したハイブリッド OS を実装することを目的とする．実装には，実時間性の確保は行っていないが，ある程度ハイブリッド OS として動作する環境を使用する．このハイブリッド OS を改良し，実時間性を確保できるようにする．そして，RTOS と改良前のハイブリッド OS，改良後のハイブリッド OS の三つで，実時間性の観点から応答性能を比較する．

1.2 論文の構成

第 2 章では，現在の実時間システムやハイブリッド OS の問題点を述べる．第 3 章では，実時間性を確保したハイブリッド OS を実装する上で必要な設計を述べる．

第 4 章では，改良前のハイブリッド OS の実装を述べた後，改良後のハイブリッド OS の実装を述べる．第 5 章では，RTOS と改良前のハイブリッド OS，改良後のハイブリッド OS それぞれについて，RTOS のタスクの実時間性を調べる．第 6 章では，関連研究と本研究とを，主に実装の観点から比較する．最後に第 7 章で，本研究についてまとめる．

第 2 章

背景

近年，ディスプレイを搭載した家電製品や画像認識を行う二足歩行ロボットを製作する，といったことが行われるようになってきた．これらのシステムには，制御のために実時間性が必要になる．また，描画処理や画像認識といった多機能性も必要になる．本章では，オペレーティングシステム (以下: OS) における実時間性や多機能性，そして本研究の目的について述べる．

2.1 実時間性

まず，実時間性について述べる．Formal methods for real-time computing[2] p.1 によると，実時間システムとは，特定の時間内に結果を出力しなければならないシステムである．ここで，システムで使用する OS の実時間性について考える．

一般的な Unix は，スループットや拡張性を重視した設計となっているため，実時間性を保証していない．実時間性を妨げる要因として，以下の事柄が挙げられる．

カーネルやドライバにおいて連続で実行しなければならない時間が不定

これにより，一般のプログラムが実行されない時間を定めることができず，実時間性を保証できない．

仮想記憶

仮想記憶へスワップアウトされたデータを，メモリへスワップインするのに

かかる時間は，一般的な Unix では見積もることができない．例えば，スラッシング状態になると，スワップインするのにかかる時間は大幅に長くなる．

処理の締切時刻を考慮しない時分割スケジューリング

個々のプロセスの処理の締切時間は考慮せずに，システム全体のスループットを重視したスケジューリングを行う．

プロセスの切り替え間隔が長い

タスク切り替えのオーバヘッドを抑えるために，プロセスの切り替え間隔を長くしている．これにより，応答性能が悪くなる．

Unix の持つこれらの要因を，実時間性を確保できるように改良すれば，実時間システムの OS として使用できる．しかし，以下のニつの理由により，この改良は現実的ではない．

一つ目の理由は，一般に Unix のソースコードは量が膨大で，改良に手間がかかることである．例えば，Linux Kernel 2.6.37[3] は，1000 万行を超えている．実時間性を確保できるようにするための改良は，実装に依存するため，機械的には行うことができず，手作業で行わなければならない．膨大なソースコードの改良を手作業で行わなければならないので，実時間性を確保できるように改良することは手間がかかる．

二つ目の理由は，一般に Unix の開発には多くの人が携わっているので，実時間性を確保できるように改良した実装を，維持することは難しいからである．特にオープンソースソフトウェアである Linux Kernel[3] や NetBSD[4] では，世界中の人々が開発に携わり，日々更新されている．実時間性を確保できるように改良した実装を維持するためには，開発に携わっている人全員が，従来のスループットや拡張性を重視した実装を用意するとともに，実時間性を確保した実装も用意しなければならない．しかし，これを行うことは難しいので，実時間性を確保できるように改良した実装を維持することは難しい．

一般に，実時間性の要求されるシステムのOSとしては，VxWorks[5]や，μITRONの実装の1つであるTOPPERS[6]などの，RTOSが使用される．

2.2 多機能性

本研究では，各種デバイスドライバや，一般にUnixで実装されているようなシステムコール，各種ライブラリなどのソフトウェア資産が豊富にあることを，多機能性があるという．

Unixは個人から企業まで様々な分野で多目的に利用されている．そのため，Unixにはソフトウェア資産が豊富に蓄積されていて，Unixには多機能性がある．Unixに用意されている多機能性を利用することで，Unix上では容易にアプリケーションを実装できることが多い．しかし，RTOSは使用されている分野が限定的である．このため，ソフトウェア資産は少なく，RTOSには多機能性がない．RTOSを使用して開発を行う場合，Unixには既に用意されている機能を，一から実装しなければならないことが多い．

2.3 実時間性と多機能性の両立

実時間性と多機能性を両立するため，RTOSとUnixを別々のハードウェアで動作させ，一つのシステムとして使用する方法がある．これを図2.1に示す．このように実装する時，実時間性が必要な処理は，RTOSのタスクとして実装して，RTOSが動作しているハードウェア上で動作させる．また，多機能性があると実装が容易になる処理は，Unixのプロセスとして実装して，Unixが動作しているハードウェア上で動作させる．このように実装することにより，実時間性を確保しつつ，実装を容易にできる．

しかし，搭載するハードウェアの増加により，費用や消費電力，搭載面積，搭載容量が増加してしまう．一般に，家電製品やロボットなどでは，これらに対する厳

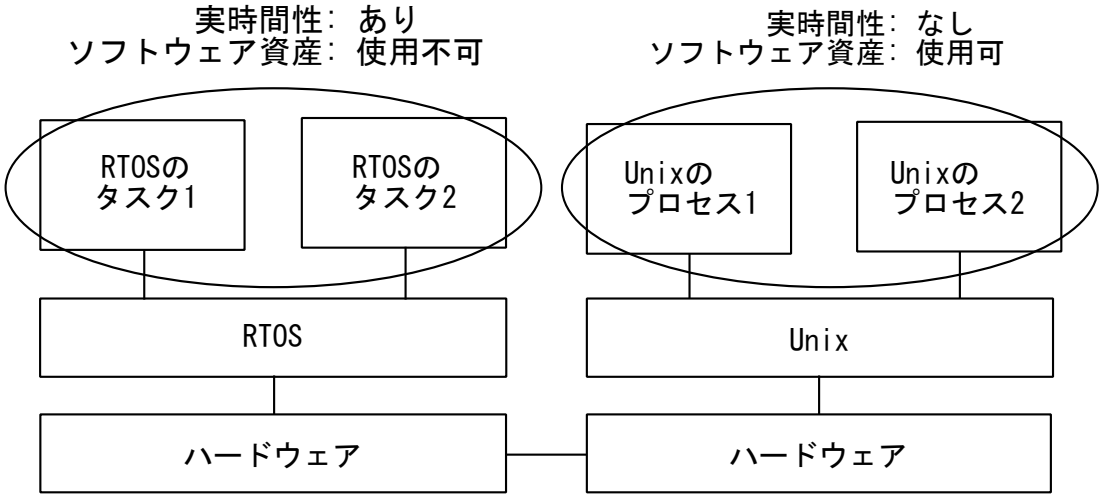

図 2.1: RTOS と Unix を別々のハードウェアで動作させて実時間性と多機能性を両立する方法

しい制約がある．また，実時間性の観点から，RTOS と Unix との間でのデータの共有も課題となる．もし，実時間性と多機能性双方を単一のハードウェア上で実現出来れば，これらの制約を避けることができる．このような計算機環境の実現方法の 1 つとして，RTOS の一タスクとして Unix を動作させるハイブリッド OS がある．

2.4 ハイブリッド OS

性質の異なる複数の OS を一つのハードウェア上で動作させるシステムを，ハイブリッド OS という．ハイブリッド OS の形態には，三種類ある．

一つ目は，下位の OS として OS–A を動作させ，OS–A の一タスクとして他の OS(OS–B) を動作させる形態 [1] である．以下，ここでは，OS–B を上位の OS と呼ぶ．この形態を図 2.2 に示す．この形態には，上位の OS の管理を比較的簡単に行うことができることや，上位の OS の管理に下位の OS の機能を活用することができるといった利点がある．下位の OS として RTOS を，上位の OS として Unix を

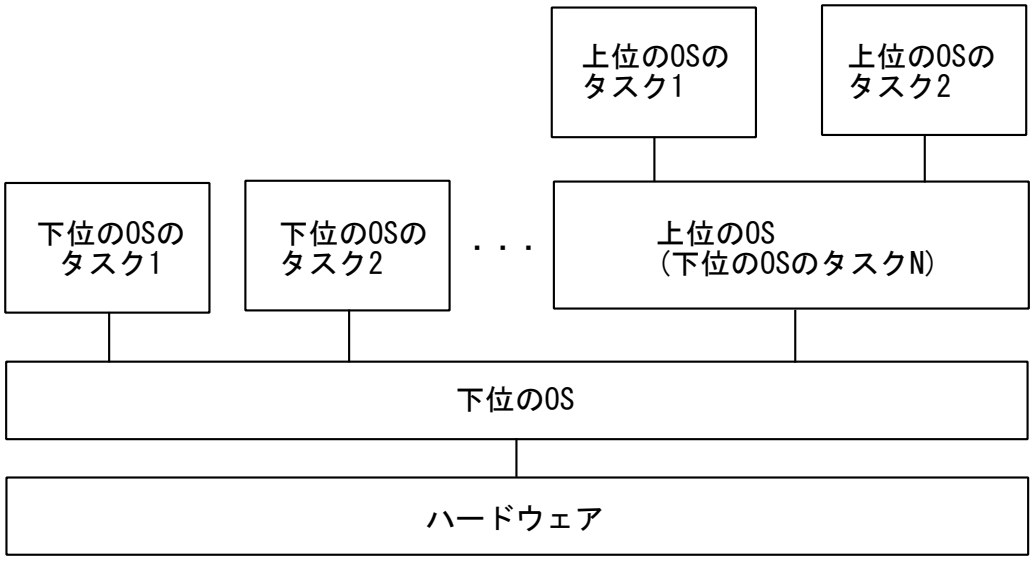

図 2.2: ハイブリッド OS の一つ目: 下位の OS の一タスクとして上位の OS を動作させる形態

動作させることにより，実時間性と多機能性の両立ができる．ただし，この形態には，本論文の 2.5 節で示すような，実時間性を損なう問題点もある．

二つ目は，下位の OS として OS–A を動作させ，他の OS(OS–B) のタスクそれぞれを，OS–A のタスクとして実行する形態 [7] である．以下，ここでは，OS-B を上位の OS と呼ぶ．この形態を図 2.3 に示す．この形態では，下位の OS が，上位の OS をタスク単位で制御できる．よって，下位の OS が上位の OS をきめ細かく制御できる利点がある．しかし，一つ目の形態に比べて，上位の OS をタスク単位で制御するので，必要なハードウェアの性能は高くなる．また，ハイブリッド OS として動作させるための各 OS への変更が多くなる．下位の OS として RTOS を，上位の OS として Unix を動作させることにより，実時間性と多機能性の両立ができる．

三つ目は，仮想マシンモニタを用意し，仮想マシンモニタの上で各 OS 用の仮想マシンを動作させ，それぞれの仮想マシンの上で OS を動作させる形態 [8] である．この形態を図 2.4 に示す．この形態は，OS への変更が少なくてすむが，別途仮想マ

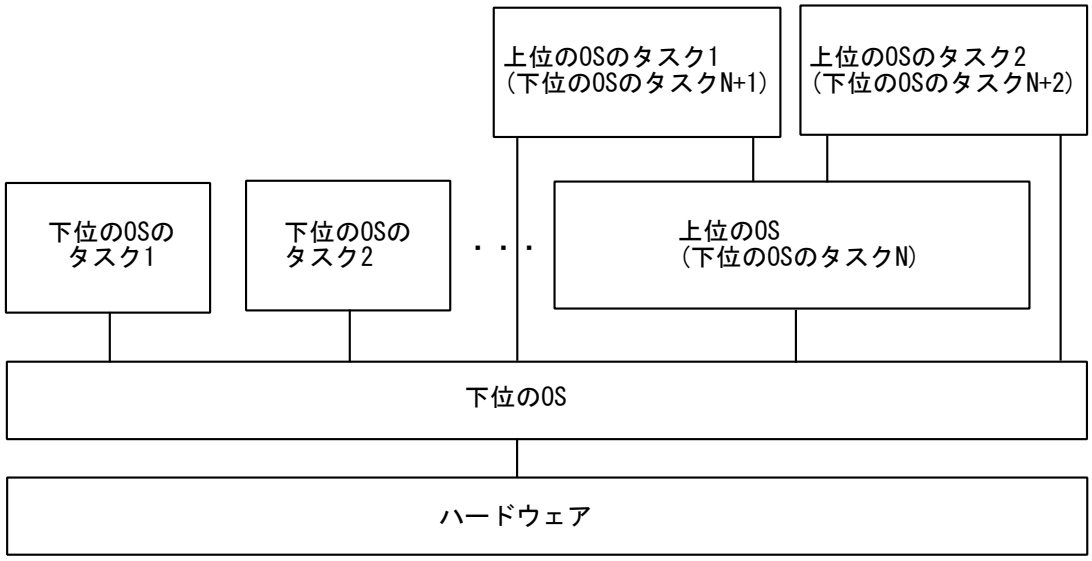

図 2.3: ハイブリッド OS の二つ目: 上位の OS のタスクそれぞれを下位の OS のタスクとして動作させる形態

シンモニタと仮想マシンの実装が必要になる．また，一つ目の形態に比べて，仮想化によるオーバヘッドが大きくなるので，必要なハードウェアの性能は高くなる．実時間性を確保できる仮想マシンモニタと仮想マシンを用意し，RTOS と Unix をこの仮想マシンモニタの仮想マシン上で動作させることにより，実時間性と多機能性の両立ができる．

本研究では，ハイブリッド OS の中でも，RTOS の一タスクとして Unix を動作させるハイブリッド OS を扱う．これ以降，単にハイブリッド OS と述べた場合は，一つ目の形態，すなわち RTOS の一タスクとして Unix を動作させるハイブリッド OS を指す．ハイブリッド OS の構成と，各 OS の上で動くアプリケーションの特徴を図 2.5 に示す．ハイブリッド OS では，実時間性が必要な処理は，RTOS のタスクとして実装する．また，多機能性があると実装が容易な処理は，Unix のプロセスとして実装する．このように実装することにより，ハードウェアのコストを抑えながら，実時間性を確保しつつ，実装を容易にできる．

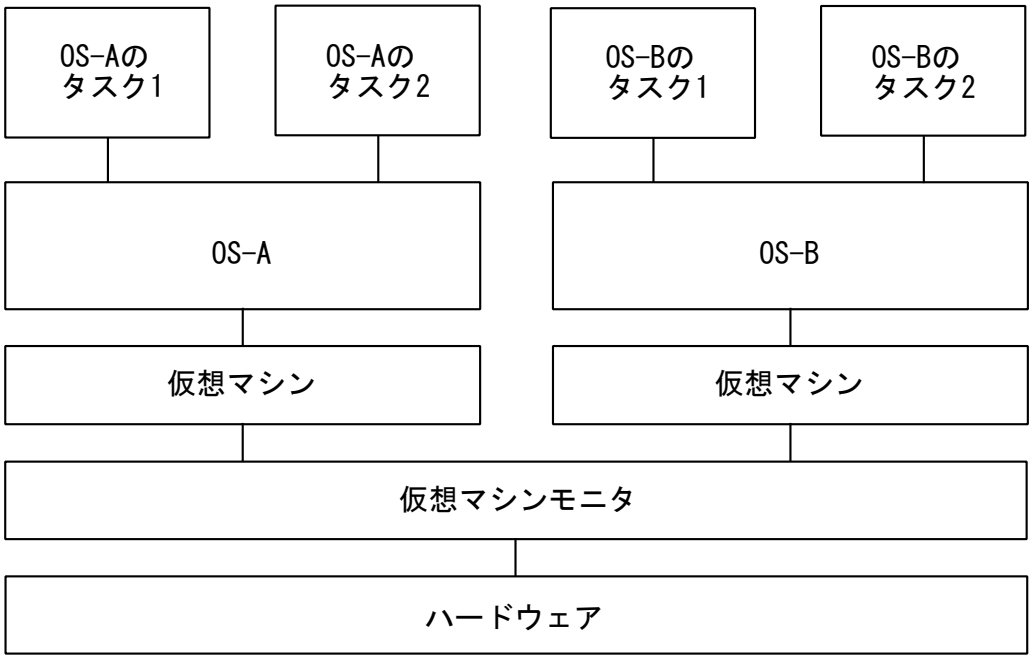

図 2.4: ハイブリッド OS の三つ目: 各仮想マシンの上で OS を動作させる形態

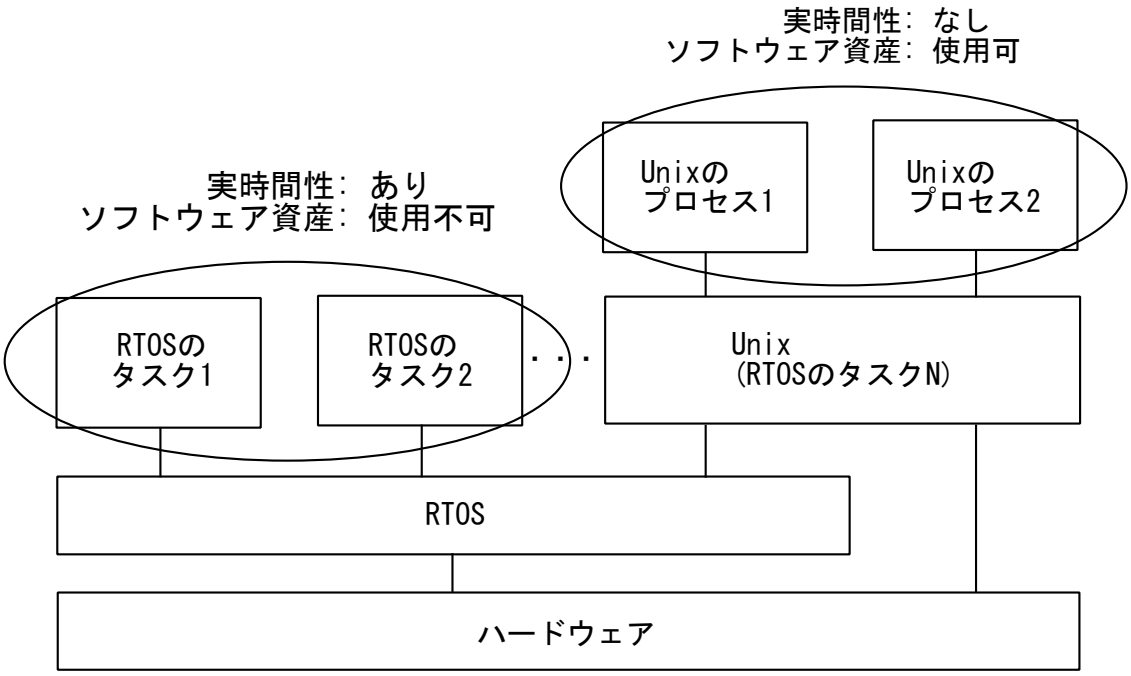

図 2.5: 本研究で扱うハイブリッド OS の構成と各 OS の上で動くアプリケーションの特徴

2.5 ハイブリッド OS の問題点

しかし，ハイブリッド OS には次の問題点がある．Unix が，RTOS への遷移を長時間阻害すると，RTOS の実時間性が損なわれるという問題である．

この問題を説明するために，タスクの切り替えについて考える．一般に RTOS のタスクは，RTOS のシステムコールを使用して自ら実行を放棄できる．あるタスクが実行を放棄したら，実行が他のタスクに切り替わる．また，RTOS のタスクが実行を放棄しない場合，RTOS ではタイマ割込みにより実行が他のタスクに切り替わる (pre-emption)．本研究では，RTOS の一タスクとして Unix を動作させるので，Unix が実行を放棄しない場合，Unix から RTOS の他のタスクへの遷移はタイマ割込みによって行われる．よって，Unix が長時間の割込み禁止を行うと，RTOS の実時間性を損ねてしまう．なぜならば，割込み禁止により，タイマ割込みが禁止され，タイマ割込みが RTOS に伝わらなくなるからである．

2.6 目的

本研究は，上記の問題点を解決し，実時間性を損なうことのないハイブリッド OS を構築することを目的とする．具体的には，Unix では割込み禁止を実行しないようにし，本来割込み禁止により行われていた Unix のプロセスやカーネル，デバイスドライバ間の排他制御は RTOS が行うようにする．これは，各カーネルのソースコードを書き換えることにより実現する．

第 3 章

設計

本章では，実時間性を確保したハイブリッド OS を実装する上で各デバイスや割込み，例外をどう扱うかを述べる．

3.1 Unix，RTOS 間でのデバイスの共有

本節では，Unix と RTOS とで，動作しているハードウェアの各デバイスをどう使用するかを述べる．

例えば，二足歩行ロボットの姿勢を検知するための加速度センサからの値の取得や，姿勢を制御するためのサーボモータへの出力は，制御のためのアプリケーションの実装は簡素だが，実時間性のある制御が必要である．また，カメラや Ethernet などのネットワークデバイスは，実時間性は必要ないが，システムにある多機能性を利用して実装すると，アプリケーションの実装が容易になる．このように，デバイスは一般的に，アプリケーションの実装は簡素であるが実時間性のある制御が必要なデバイスと，実時間性は必要ないがシステムにある多機能性を利用するとアプリケーションの実装が容易になるデバイスに分類することができる．

ここで，ハイブリッド OS の特性から，前者の制御を行うアプリケーションは RTOS 上のタスクとして実装し，後者の制御を行うアプリケーションは Unix のプロセスとして実装すると，実時間性のある処理ができると共に，アプリケーションの実装が容易になる．よって本研究では，基本的に RTOS と Unix とでデバイスを

表 3.1: 本研究で想定しているデバイスの利用形態の具体例

デバイス	利用形態
タイマ CPU メモリ	共有
加速度センサ サーボモータ 接触センサ	RTOS のアプリケーションが制御
ネットワークデバイス カメラ ディスプレイ	Unix のアプリケーションが制御

共有することは想定しない．ただし，タイマデバイス，CPU，メモリは共有する．本研究で想定しているデバイスの利用形態の具体例を，表 3.1 にまとめる．

まず，タイマデバイスの共有について述べる．両方の OS で時刻管理のためにタイマデバイスが必要なので，タイマデバイスは両方の OS で共有する．

次に，CPU の共有について述べる．本研究では，RTOS の一タスクとして Unix を動作させる．よって，RTOS の他のタスク同士と同様に，RTOS と Unix とでは，CPU を時分割で使用する．また，RTOS のタスクならびに Unix 間のスケジューリングは，RTOS のスケジューラが受け持つ．

最後に，メモリの共有について述べる．本研究では，RTOS と Unix はそれぞれ静的に割り当てられたメモリ領域を使用する．これは，両方の OS への変更が少なくてすみ，また RTOS の実時間性を確保することができるからである．逆に，RTOS

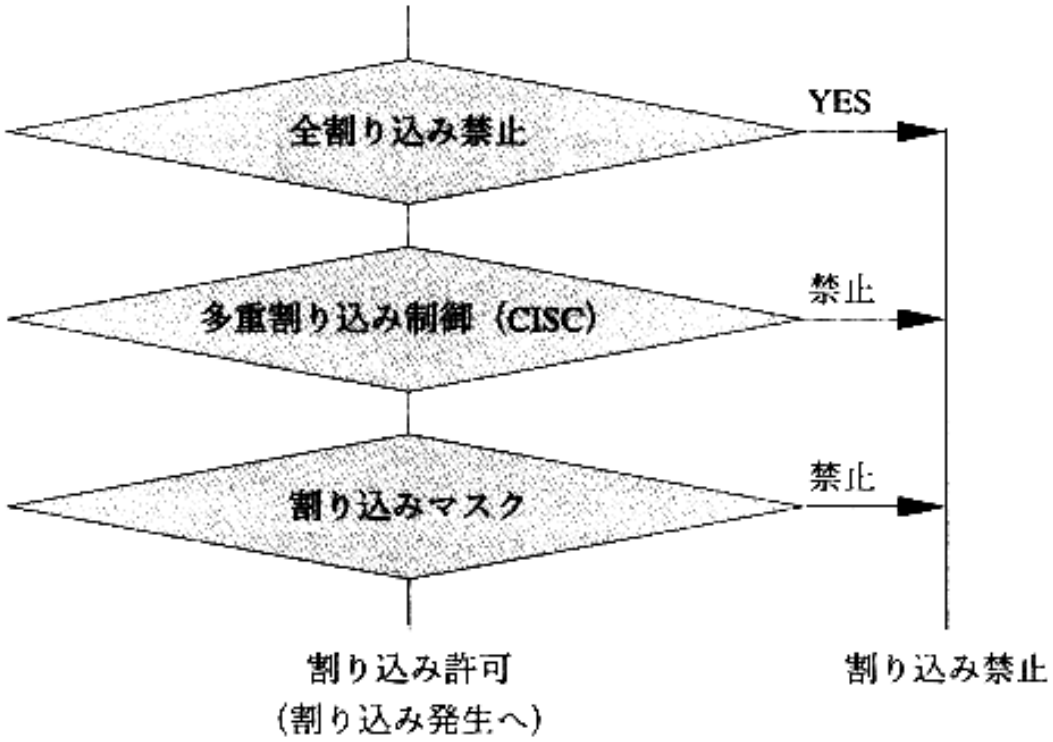

図 3.35　割り込み要求発生時のチェック

図 3.1: 割込み禁止が発生する箇所 (わかりやすい 組込システム構築技法 ソフトウェア編 [9]p. 75 図 3.35 より引用)

のメモリを動的に確保しようとすると，メモリの確保を行うときに必要な時間を見積もりにくくなるという欠点がある．よって，RTOS と Unix は，メモリを静的に分割して使用する．

3.2　割込み禁止の種類

本節では，割込み禁止について種類別に述べる．図 3.1 に示すように，一般的に割込み禁止には，全割込み禁止，多重割込み制御，割込みマスクの 3 種類ある．

全割込み禁止とは，割込みすべてを一括で禁止する方法である．全割込み禁止は，

すべての割込みを禁止したい時に使用される．ハイブリッド OS において，Unix が全割込み禁止を行うと，タイマ割込みを含めたすべての割込みが発生しなくなる．よって，Unix が全割込み禁止を行っている間は，RTOS へ遷移できなくなる．

多重割込み制御とは，割込み処理中に新たな割込みが発生した場合，この新たな割込みを制御する機構である．多重割込み制御は，割込み処理中に新たな割込みを受けてはいけない時に，新たな割込みを抑制するのに使用される．多重割込み制御により，Unix がタイマ割込みを禁止すると，タイマ割込みが発生しなくなる．よって，多重割込み制御により Unix がタイマ割込みを抑制している間は，RTOS へ遷移できなくなる．

割込みマスクとは，割込みを種類別に禁止する方法である．割込みマスクは，特定の割込みのみを禁止したい時に使用される．Unix がタイマ割込みを禁止すると，タイマ割込みが発生しなくなる．よって，Unix がタイマ割込みのマスクを行っている間は，RTOS へ遷移できなくなる．

以上より，本研究では，Unix が全割込み禁止や多重割込み制御によるタイマ割割込みの抑制，タイマ割込みのマスクを行うことのないようにし，RTOS の実時間性が損なわれることを防ぐ．

3.3 割込み，例外の仮想化

割込みや例外が発生したとき，まず RTOS か Unix のどちらかが受け取ることになる．この時，まず Unix が受け取ると，RTOS の割込みや例外の実時間性が確保できなくなってしまう．なぜならば，Unix 上で実装すると，2.1 節で述べたとおり，いつ処理が終わるかを見積もることができないからである．よって，割込みや例外は，まず RTOS が受け取る．そして，受け取った割込みや例外が，Unix への割込みや例外であった場合は，RTOS が Unix に割込みや例外の配送を行う．Unix の割込み禁止による排他制御も，この配送時に行う．詳細は，3.4 節で述べる．

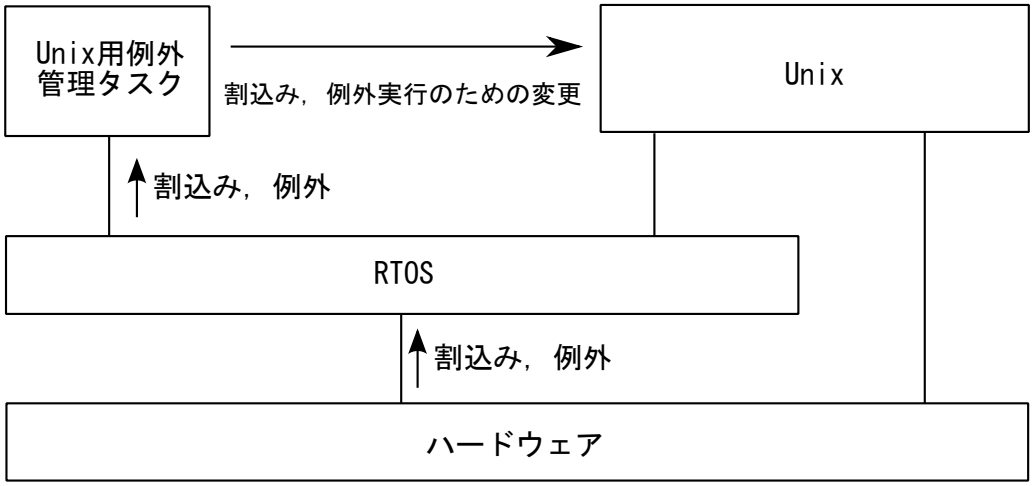

図 3.2: RTOS に用意する Unix 用のタスク

3.4 Unix 用のタスク

本節では，Unix を動作させるために RTOS に用意するタスクについて述べる．

本研究では，RTOS の一タスクとして Unix を動作させる．本研究では，このタスクを Unix タスクと呼ぶ．Unix タスクは，他の ITRON のタスクよりも優先度を低くしておく．ここで，Unix タスク以外の ITRON のタスクは，必要な処理の実行が終わった後，自ら実行を放棄するものとする．このようにすることで，実時間性の必要な他の RTOS のタスクの実行を優先させることができ，また実時間性の必要な処理を実行する必要がない時間に Unix タスクを実行できる．

また，3.3 節で述べたとおり，本研究では RTOS が Unix タスクに割込みや例外の配送を行う．この配送を RTOS のタスクとして実装する．本研究では，このタスクを Unix 用例外管理タスクと呼ぶ．Unix 用例外管理タスクは，Unix への割込みや例外が発生したことを受け，Unix タスクに割込みや例外を実行するための変更を行うタスクである．RTOS に用意する Unix 用のタスクを，図 3.2 に示す．

Unix の割込み禁止の実装を変更しないでハイブリッド OS を動作させた場合，Unix は割込み禁止命令をハードウェアに対して実行する．これを，図 3.3 に示す．

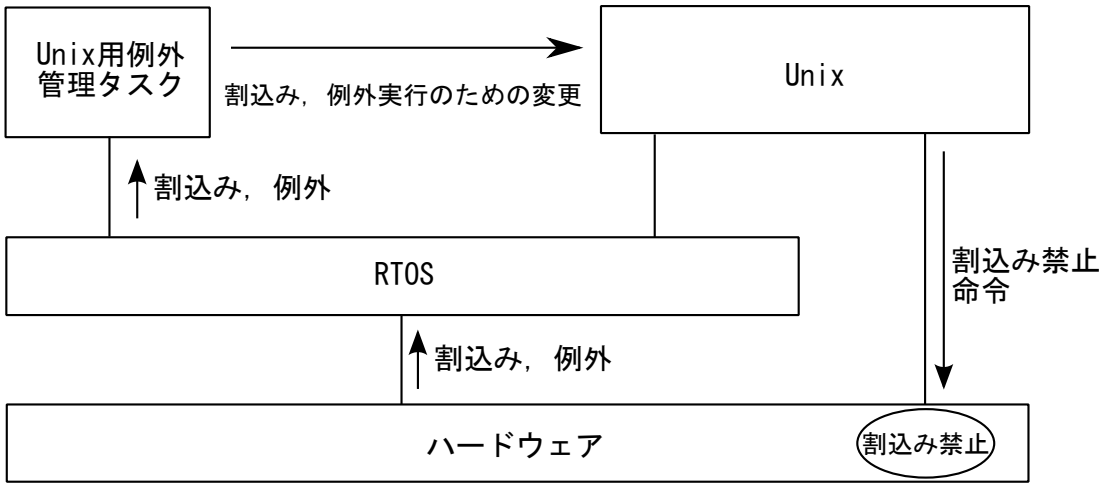

図 3.3: 割込み禁止命令の変更前のハイブリッド OS

Unix はハードウェアに対して割込み禁止を行うため，RTOS の割込みも禁止してしまい，RTOS の実時間性が確保できなくなる．そこで，本研究では Unix が割込み禁止を行わないように変更し，Unix は割込み禁止に変わる命令を Unix 用例外管理タスクに対して実行する．そして，Unix の割込み禁止に変わる排他制御は Unix 用例外管理タスクが担当する．これを，図 3.4 に示す．この排他制御は，次のように行うことで実現する．Unix が全割込み禁止や多重割込み制御，割込みマスクを行っている間に Unix への例外を受けた時は，割込みや例外を実行するための Unix タスクへの変更を行わないようにする．

ここで，Unix への割込みや例外の配送を，なぜ RTOS のカーネル内ではなくタスクとして実装するのかということについて記述する．RTOS のタスクとして実装することで，次の二つの利点が得られるからだ．

- RTOS のシステムコールを使用できる
- 実装する時に RTOS の応答性能を気にする必要がない

後者について詳しく述べる．タスクとして実装する場合，このタスクよりも優先度の高い RTOS のタスクやカーネルは，常にプリエンプト可能である．これより，実

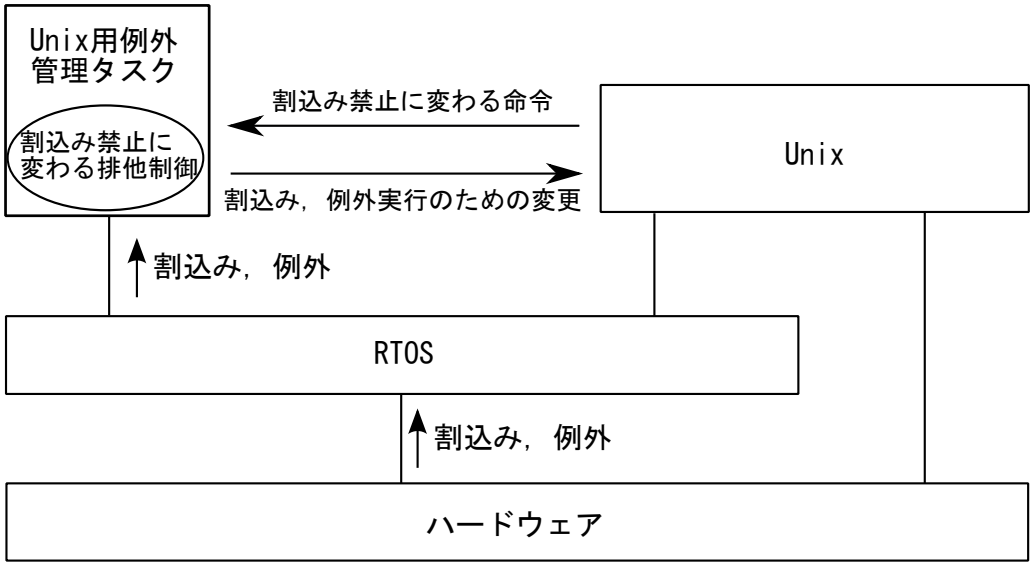

図 3.4: 割込み禁止命令の変更後のハイブリッド OS(本研究のハイブリッド OS)

時間性の必要なタスクの優先度を相対的に高くしておけば，割込みや例外の配送をタスクとして実装しても，RTOS の応答性能は変わらない．しかし，割込みや例外の配送をカーネル内に実装した場合，RTOS の応答性能が悪くなる場合がある．例えば，割込み禁止やセマフォなどの排他制御を行っている区間に実装した場合は，排他制御の実行時間が長くなるので，RTOS の応答性能は悪くなる．またそうでなくとも，実装した場所以降にあるカーネルが実行される時刻は遅くなる．よって，カーネル内に実装する場合は，実時間性に気をつけなければならない．以上より，容易に実装を行うことができるので，RTOS のタスクとして実装する．

第 4 章

実装

本研究では，ある程度ハイブリッド OS として動作する既存の実装を元に，改良を行った．この既存のハイブリッド OS では，実時間性を確保する実装は行われていない．本章では，まず本研究の開発環境を述べる．そして，既存の実装を述べ，最後に本研究で行った改良を述べる．

4.1 開発環境

開発環境は，既に研究室内に構築されていた環境を使用した．本節では，この環境について述べる．

ハイブリッド OS を動作させるターゲットの CPU として，PowerPC アーキテクチャ[10] を採用した FreeScale 社の MPC5200B[11] を使用し，ハードウェアプラットフォームとして MPC5200B を搭載した mmEyePPC を使用した．また，mmEyePPC のブータとして Das U-Boot 1.1.4[12] を使用した．ハイブリッド OS の RTOS として TOPPPERS/JSP 1.4.1[6]，Unix として NetBSD 1.6.2[4] を使用した．また，各 OS のソースコードの改良やコンパイルは，NetBSD1.6.2 を動作させた x86 アーキテクチャの計算機 (以下: ホスト計算機と呼ぶ) 上で行った．コンパイルには，NetBSD に用意されている PowerPC 用のクロスコンパイラを使用した．また，ハイブリッド OS で使用する NetBSD とコンパイルを行う NetBSD とでバージョンを合わせることで，バージョンの差違による不具合が発生しないという利点がある．

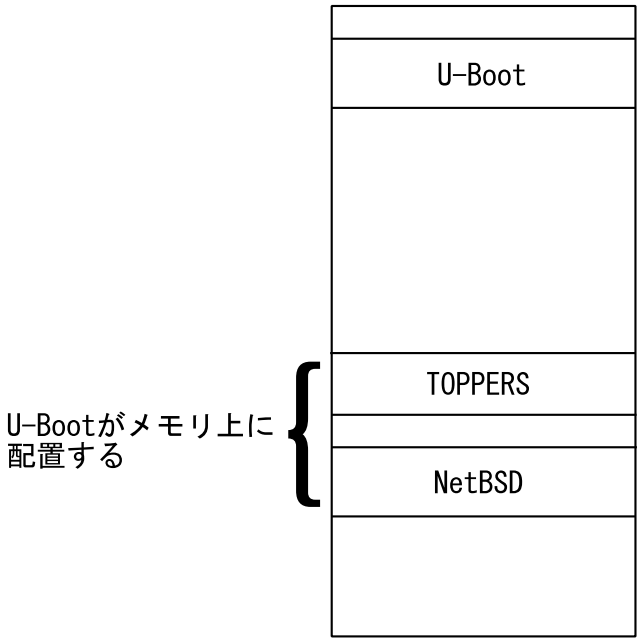

図 4.1: U-Boot により各 OS を読み込んだ時のメモリマップ

次に，コンパイルにより作成した各 OS のバイナリを mmEyePPC で読み込む方法について記述する．mmEyePPC には，CompactFlash や USB メモリなどの記憶媒体を接続することができる．しかし，OS を書き換えるたびに，OS のバイナリをこれらの記憶媒体へホスト計算機で書き込み，書き込んだ記憶媒体を mmEyePPC に接続し直して読み込むという方法は，時間と手間がかかる．そこで，mmEyePPC からネットワークを介して各 OS のバイナリを読み込む方法を用いた．ここで，U-Boot には，NFS プロトコルを使用してファイルを読み込み，メモリ上に配置する機能がある．各 OS のバイナリを読み込むために，この機能を使用した．まずホスト計算機で NFS サーバを動作させて OS のバイナリを共有しておき，U-Boot から NFS 経由で OS のバイナリを読み込んだ．各 OS を読み込んだ時のメモリマップを図 4.1 に記載した．

4.2 既存の実装

まず，既存のハイブリッド OS の実装のなかでも，ハイブリッド OS として動作させるのに必要な実装を述べる．

4.2.1 NetBSD 用の割込みハンドラ

3.3 節で述べた通り，割込みや例外はすべてまず TOPPERS が受け取る．ここで，PowerPC では，割込みは，いくつかある例外の一つとなっている．これ以降，単に例外と述べた場合は，割込みを含むすべての例外を指す．NetBSD が使用するデバイスの割込みを TOPPERS が受けるためには，NetBSD が使用するデバイスの割込みハンドラを TOPPERS に実装しなければならない．各デバイスの割込みハンドラは，4.2.2.2 節で述べる NetBSD 例外管理タスクに，割込みが発生したことを通知する．この通知には，TOPPERS のタスク間通信を使用する．タスク間通信の送信により，割込みの発生の通知を行う．

4.2.2 NetBSD 用のタスク

3.4 節で述べた Unix 用の各タスクについて，本節では，Unix タスクを NetBSD タスク，Unix 例外管理タスクを NetBSD 例外管理タスクと呼ぶ．また，既存のハイブリッド OS の実装では，NetBSD タスクのためのデクリメンタ例外を擬似的に生成するためのタスクを，TOPPERS に実装している．本研究では，このタスクを NetBSD デクリメンタ例外タスクと呼ぶ．

4.2.2.1 NetBSD デクリメンタ例外タスク

まず，NetBSD デクリメンタ例外タスクを実装する理由を以下に述べる．PowerPC には，一定時間ごとにデクリメンタ例外を発生させる機構が実装されている．既存の NetBSD の実装では，このデクリメンタ例外を使用して時刻を管理している．

ハイブリッド OS 環境では，3.3 節で述べたとおり，例外は全て TOPPERS が受け取るようにするので，別途 NetBSD タスクへデクリメンタ例外を配送する実装が必要になる．そこで，擬似的にデクリメンタ例外を生成するために，NetBSD デクリメンタ例外タスクを実装している．

次に，NetBSD デクリメンタ例外タスクの実装について述べる．既存のハイブリッド OS の実装では，NetBSD デクリメンタ例外タスクを，一定時間ごとに実行を繰り返す TOPPERS の周期タスクとして実装している．NetBSD デクリメンタ例外タスクは，擬似的なデクリメンタ例外が発生したことを，NetBSD 例外管理タスクに通知する．この通知には，TOPPERS のタスク間通信を使用する．タスク間通信の送信により，擬似的なデクリメンタ例外の通知を行う．

4.2.2.2 NetBSD 例外管理タスク

NetBSD 例外管理タスクは，次の処理を行う．TOPPERS に実装した NetBSD 用の割込みハンドラや NetBSD デクリメンタ例外タスクからの例外の通知を待つ．この通知待ちには，受信するまでタスクが待ち状態になるタスク間通信機構を利用する．受信すると，NetBSD 例外管理タスクは実行可能状態となる．タスク間通信を受信したら，NetBSD タスクが例外処理を実行するための変更を，NetBSD タスクへ行う．この変更は具体的には，プログラムカウンタの書き換えや割込み要因レジスタの設定である．NetBSD 例外管理タスクは，以上の処理を繰り返す．既存のハイブリッド OS に実装されている NetBSD 例外管理タスクの処理を図 4.2 に示す．

ここで，NetBSD 例外管理タスクの優先度は，NetBSD タスクよりも高くしておく．これにより，NetBSD タスクが実行を放棄しないことにより，NetBSD タスクへの例外実行のための変更が行われなくなることを防ぐ．また，受信待ちを行うときに待ち状態となるタスク間通信機構を使用することで，NetBSD 例外管理タスクが受信待ちを行っているときに NetBSD タスクを実行することができる．

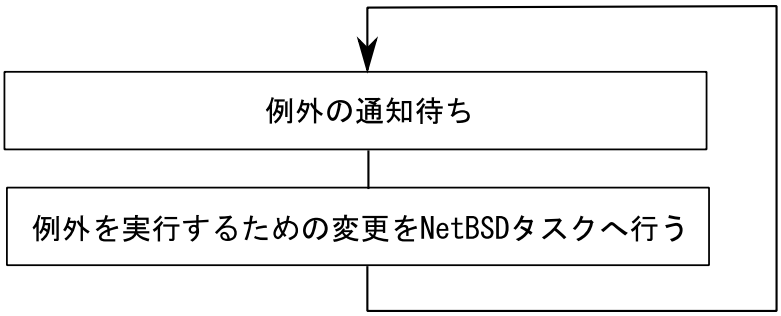

図 4.2: 既存のハイブリッド OS に実装されている NetBSD 例外管理タスクの処理

4.2.3 NetBSD からの TOPPERS のリソースの操作

ハイブリッド OS では，NetBSD から TOPPERS のリソースを操作したいことがある．例えば，既存のハイブリッド OS の実装では 4.2.6.2 節で述べる割込みのマスク制御を行うときに使用し，また本研究の実装では 4.3.1 節で述べる全割込み禁止の仮想化を行うときにセマフォの操作を行うために使用している．そこで，NetBSD から TOPPERS のリソースを操作するための機構が実装されている．この機構は，あらかじめ TOPPERS にリソースを変更するための関数を作成しておき，NetBSD から関数ポインタにより TOPPERS に実装した関数を実行することで，実現している．この機構のメモリマップを図 4.3 に示す．

このように実現する理由を以下に述べる．NetBSD では，TOPPERS 内のリソースを，NetBSD 内のリソースのようにシンボル名で参照することができない．これは，TOPPERS と NetBSD のコンパイルを別々に行うので，NetBSD から TOPPERS のシンボル名が解決できないためである．TOPPERS 内からは，TOPPERS のリソースをシンボル名で参照することができることから，リソースの変更処理は TOPPERS 内に実装する．また，このリソースの変更機能を呼び出しやすいように，TOPPERS の関数として実装しておく．そして，この関数の位置を格納する領域をメモリ上に確保しておく．関数の位置を示すのに，関数ポインタを利用する．この理由は，NetBSD で関数ポインタの実行として直接的に記述できるからである．NetBSD か

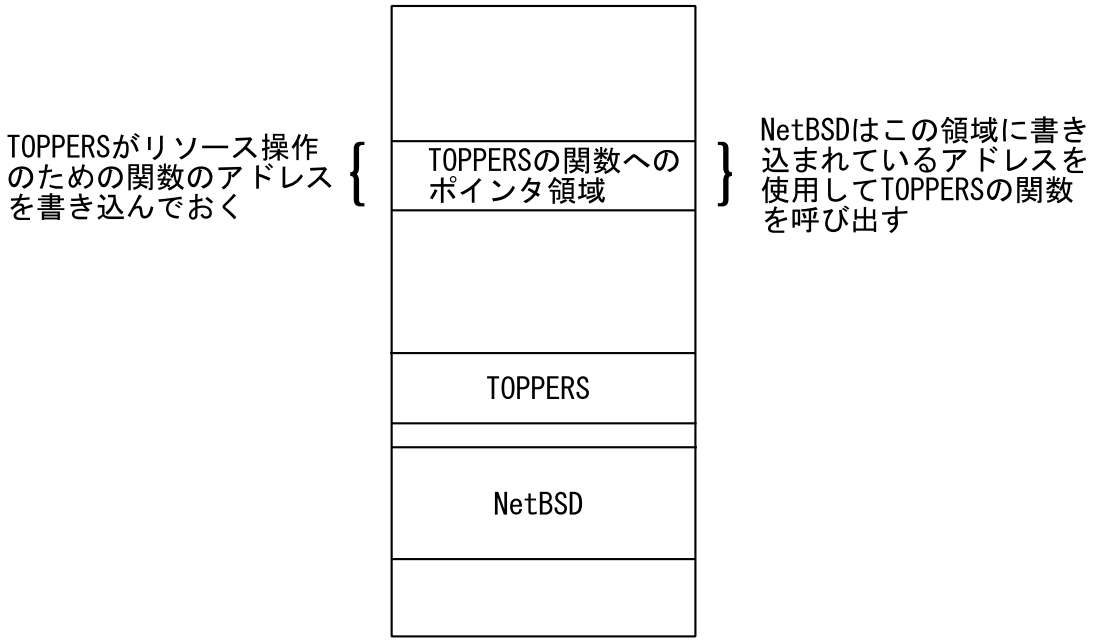

図 4.3: NetBSD から TOPPERS のリソースを操作するための機構のメモリマップ

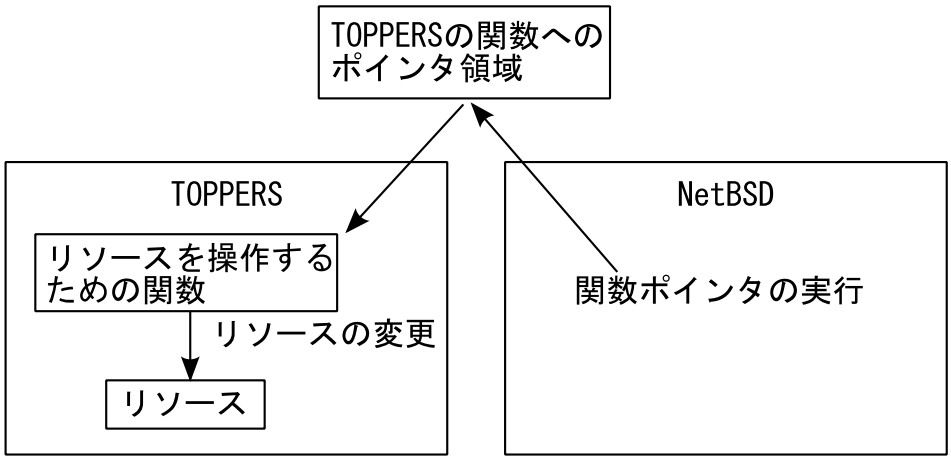

図 4.4: NetBSD からの TOPPERS のリソースの操作

らの TOPPERS のリソースの操作を図 4.4 に示す.

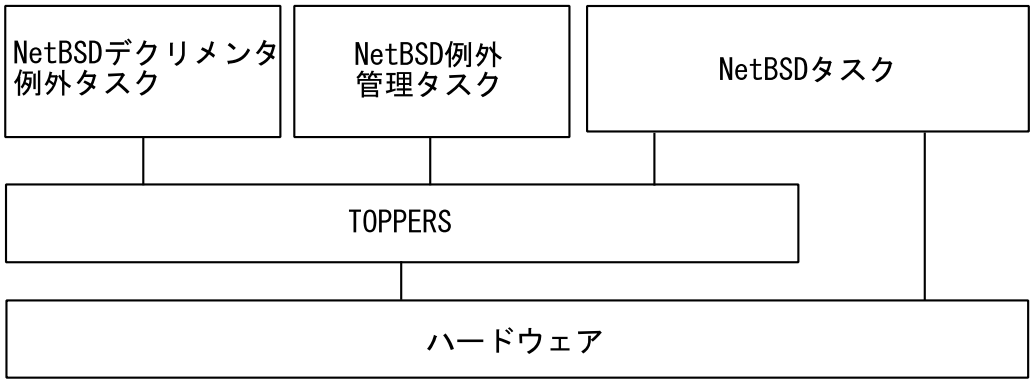

図 4.5: 本研究で改良を行うハイブリッド OS の構成

4.2.4 ハイブリッド OS の起動方法

まず TOPPERS を起動し，その後 NetBSD タスクが NetBSD の領域へジャンプすることで，ハイブリッド OS を動作させる．以下にこの詳細を述べる．4.1 節で述べたとおり，各 OS のバイナリをメモリ上に配置しておく．そして，U-Boot から TOPPERS の先頭番地へジャンプする．これにより，TOPPERS が起動する．その後，NetBSD タスクを起動し，NetBSD タスクがメモリに配置した NetBSD の先頭アドレスにジャンプすることで，NetBSD が動作する．この時に，NetSBD 例外管理タスクを同時に起動し，また NetBSD デクリメンタ例外タスクの周期実行を開始する．これにより，ハイブリッド OS として動作する．この時のハイブリッド OS の構成を図 4.5 に示す.

4.2.5 割込み，例外

PowerPC では，例外が発生したときは，特定のメモリ領域にジャンプする．そして，あらかじめこの特定のメモリ領域にプログラムを書きこんでおくことで，例外ハンドラを実現する.

TOPPERS と NetBSD は，共に例外ハンドラを使用している．4.2.4 節で述べた方法でハイブリッド OS を起動すると，後に起動する NetBSD が，例外ハンドラを上

書きし，すべての例外を NetBSD が受け取ってしまう．ここで，3.3 節では，例外は全て TOPPERS が受け取るようにすると述べた．そこで，NetBSD が TOPPERS の例外ハンドラを上書きしないようにし，また例外はすべて TOPPERS が受け取るようにするために，NetBSD が使用する各例外ハンドラのメモリ領域を移動する．

NetBSD が使用する各例外ハンドラのメモリ領域の移動は，NetBSD のソースコード内の，割込みハンドラ，例外ハンドラの書き込みを行う場所にて，書き込み先のアドレスを変更することで実現している．また，4.2.2.2 節で述べたとおり，TOPPERS が受け取った NetBSD への割込みやデクリメンタ例外の，NetBSD タスクへの配送は，NetBSD 例外管理タスクが行う．この配送は，NetBSD 例外管理タスクが，NetBSD タスクのプログラムカウンタを移動した NetBSD 割込みハンドラや例外ハンドラのアドレスに書き換えることで，実現している．なお，NetBSD のすべての例外ハンドラのメモリ領域を移動し，すべての例外を TOPPERS が受け取った上で，NetBSD へのすべての例外を TOPPERS から NetBSD に配送するのが理想的な実装である．この時のメモリマップを図 4.6 に示す．しかし今回は，実装の手間を軽減するために，NetBSD の割込みとデクリメンタ例外以外の例外ハンドラの移動は省略している．

4.2.6 割込み禁止

本節では，既存のハイブリッド OS の割込み禁止について，全割込み禁止と割込みマスクに分けて述べる．ここで，PowerPC はビッグ・エンディアンであるため，Programming Environments Manual For 32-Bit Implementations of the PowerPC Architecture[10] や MPC5200 Users Guide[11] では，最上位ビットを第 0 ビットとして説明している．これに倣い，本論文でも最上位ビットを第 0 ビットとして記述する．

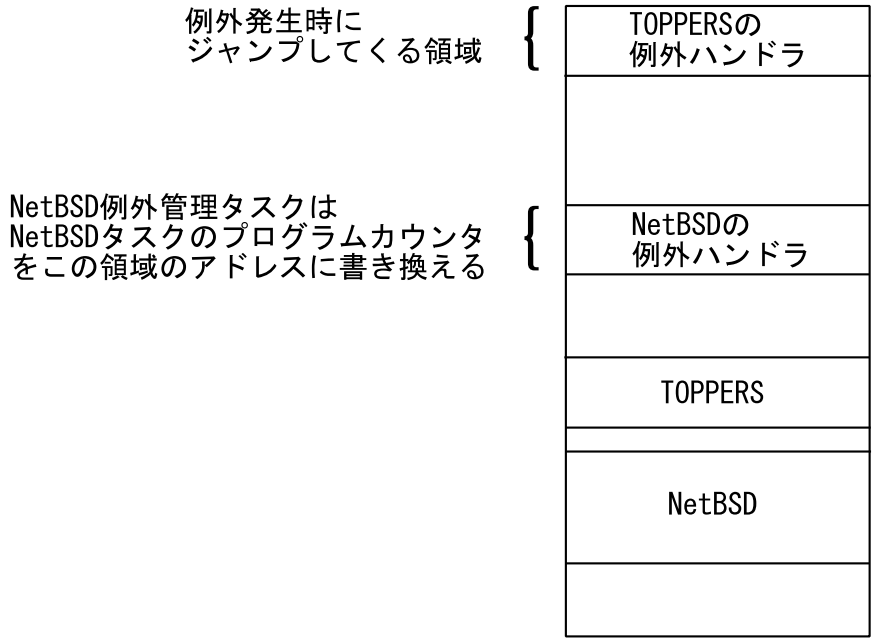

図 4.6: 各 OS の例外ハンドラを理想的に配置したときのメモリマップ

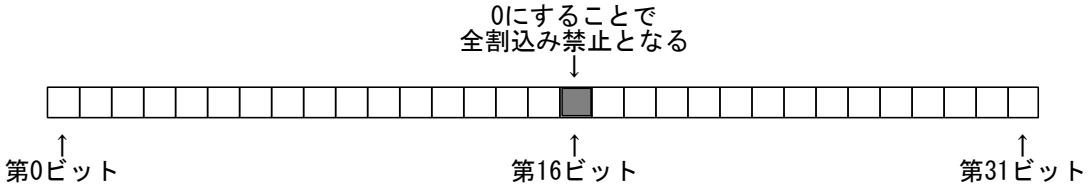

図 4.7: MSR における全割込み禁止ビット

4.2.6.1 全割込み禁止，全割込み許可

PowerPC には，Machine State Register(以下: MSR) と呼ばれる，計算機の状態を指定する 32 ビット長のレジスタがある．そして，MSR の第 16 ビットを 0 にすることで全割込み禁止状態となり，1 にすることで全割込み許可状態となる．これを図 4.7 に示す．全割込み禁止を行うことで，Non maskable interrupt 以外の割込みと例外が実行されないようになる．

また PowerPC には，MSR を直接読み書きする命令が実装されている．MSR の読み込みはアセンブラの mfmsr 命令で，MSR の書き込みはアセンブラの mtmsr 命

令で行うことができる．

既存の NetBSD の実装では，mfmsr 命令と mtmsr 命令を使って直接 MSR の第 16 ビットを書き換えることで，全割込み禁止，全割込み許可を行っている．またこの命令は，マクロを使用せず，ソースコード内の全割込み禁止，全割込み許可を行う場所に直に書かれている．

4.2.6.2 割込みマスク

MPC5200B では，Memory mapped I/O(以下: MMIO) として割り当てられているレジスタへの書き込みで，各種デバイスの制御ができる．この MMIO の一部に，各デバイスの割込みのマスク制御を行うレジスタがある．MPC5200B では，このレジスタに書き込むことで，割込みのマスク制御を行う．

既存の TOPPERS と NetBSD の実装では，両方共に割込みのマスク制御を行うためのレジスタへの書き込みにより，割込みのマスク制御を行なっている．このため，ハイブリッド OS 時には，TOPPERS と NetBSD とで割込みのマスク制御が競合してしまう．この競合を防ぐために，NetBSD が割込みのマスク制御を行う領域を，MMIO 領域から物理メモリのメモリ領域へ移動している．NetBSD では割込みのマスク制御の MMIO 領域をマクロで定義しているので，このマクロを変更することで物理メモリのメモリ領域への移動を実現している．この移動により，NetBSD は割込みのマスク制御を行わなくなる．この時のメモリマップを図 4.8 に示す．そして，競合せずに必要な割込みのマスク制御を行うように，TOPPERS の機能を利用して次のように実装している．4.2.3 節で述べた NetBSD からの TOPPERS のリソースの操作方法を利用し，TOPPERS に実装した割込みのマスク制御を行う関数を NetBSD から実行している．この関数は，TOPPERS に実装されている，割込みのマスク制御を行うレジスタを書き換える関数を実行する．既存のハイブリッド OS の実装では，NetBSD はこのようにして割込みのマスク制御を行う．

タイマ割込みの場合，割込みのマスク制御を行うことができるレジスタは二つ

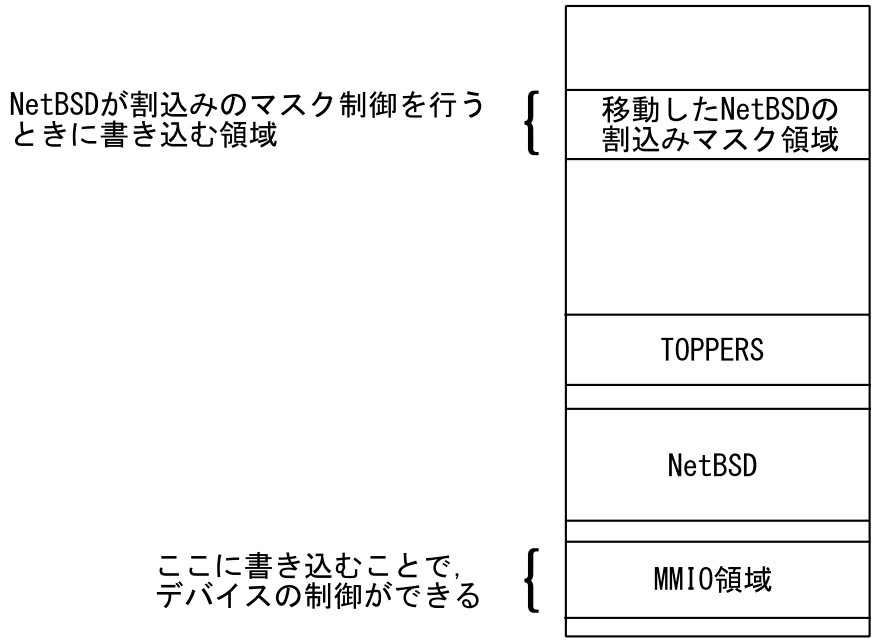

図 4.8: NetBSD の割込みのマスク制御領域を移動した時のメモリマップ

ある．タイマデバイスのレジスタと，タイマデバイスが接続されている System Integration Unit(以下: SIU) のレジスタである．既存の TOPPERS と NetBSD の実装では，両方の OS 共に SIU のレジスタへの書き込みで，タイマ割込みのマスク制御を行っている．

SIU のレジスタで行う，タイマ割込みのマスク制御の詳細を以下に述べる．MPC5200B にはタイマとして General Purpose Timer(以下: GPT) が搭載されている．GPT は，GPT0 から GPT7 まで，合計 8 個ある．各 GPT のマスク制御を行う SIU のレジスタは，32 ビット長であり，このうちの第 24 ビットが GPT0 の割込みのマスク制御ビット，第 25 ビットが GPT1 の割込みのマスク制御ビット，となっている．これを図 4.9 に示す．各ビットを 1 にすることで対応する GPT の割込みが発生しなくなり，0 にすることで割込みが発生するようになる．

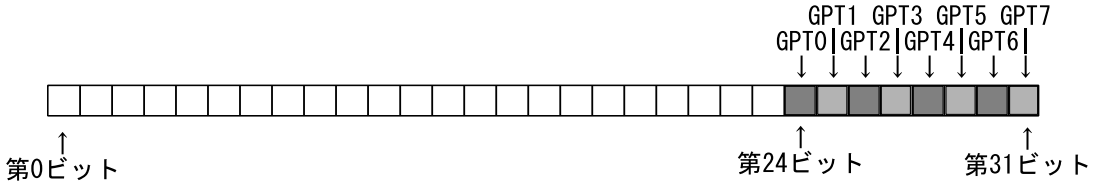

図 4.9: SIU における GPT の割込みマスクビット

4.3 本研究の実装

本研究では，NetBSD が行う割込み禁止により，TOPPERS の実時間性が損なわれないようにすることを目的としていた．これを実現するために，既存のハイブリッド OS の実装に対し，NetBSD が行う全割込み禁止と全割込み許可の仮想化と，割込みマスクの仮想化を行った．これにより，NetBSD が実際に全割込み禁止を行わないようにした．

4.3.1 全割込み禁止の仮想化

NetBSD が全割込み禁止と全割込み許可を行わないようにし，NetBSD が全割込み禁止を行っていた時は，NetBSD 例外管理タスクが行う NetBSD タスクへの例外の配送を停止するようにした．これにより，NetBSD が全割込み禁止をせずに，NetBSD が行う全割込み禁止と同等の排他制御を実現した．

ここで，実装の概要を述べる．実装には，TOPPERS のセマフォを利用した．NetBSD が行う全割込み禁止をセマフォの獲得に，全割込み許可をセマフォの解放に置き換えた．また，4.2.2.2 節で述べた NetBSD 例外管理タスクでは，例外の通知を受けた後，上記と同じセマフォを獲得するようにした．その後，例外実行のための変更を NetBSD タスクに行い，セマフォを解放し，再びタスク間通信の受信待ちを行うようにした．

以下では，全割込み禁止の仮想化について，NetBSD への変更である全割込み禁止，全割込み許可の置き換えと，NetBSD 例外管理タスクの改良に分けて述べた．

4.3.1.1 全割込み禁止，全割込み許可の置き換え

全割込み禁止，全割込み許可の置き換えについて述べる．この置き換えは，NetBSD のソースコードを直接書き換えることにより実現した．TOPPERS 内に，NetBSD が全割込み禁止を行うときに実行する関数を作成し，この関数内でセマフォの獲得を行った．NetBSD の全割込み許可に関しても同様に，TOPPERS 内に関数を作成し，この関数内でセマフォの解放を行った．そして，4.2.3 節に述べた機構を利用して，NetBSD からこれらの関数を呼び出すことにより，全割込み禁止，全割込み許可の仮想化を行った．

また，セマフォの獲得には，ノンブロッキングでセマフォの獲得を行う TOPPERS の pol_sem システムコールを使用した．この理由を以下に述べる．NetBSD が，全割込み許可をせずに，複数回連続で全割込み禁止を行うように実装されるかもしれない．もし複数回連続で全割込み禁止を行う場合，全割込み禁止の仮想化を実装すると，セマフォの獲得を複数回連続で行うことになる．セマフォの獲得時に，獲得するまでブロッキングを行うシステムコールを使用すると，二回目のセマフォの獲得時にセマフォ獲得待ちでデッドロックを起こし，NetBSD が止まってしまう．これを防ぐために，セマフォの獲得にはノンブロッキングのシステムコールを使用した．

ここで，ノンブロッキングでセマフォを獲得するシステムコールを使用すると，NetBSD 例外管理タスクがセマフォを獲得している間に NetBSD タスクが全割込み禁止区間に突入し，排他制御が成り立たなくなることが考えられるが，これは実際には起こらない．なぜならば，4.2.2.2 節で述べたとおり，NetBSD タスクよりも NetBSD 例外管理タスクの優先度を高くしてあるからだ．これにより，NetBSD 例外管理タスクの実行中に，NetBSD タスクに実行が遷移することはない．NetBSD タスクが実行されるのは，NetBSD 例外管理タスクが例外の受信待ちにより待ち状態になっている間だけである．なお，セマフォの解放には，TOPPERS の sig_sem システムコールを使用した．

本研究では，NetBSD が実行するすべての全割込み禁止，全割込み許可の置き換えを行うことはできなかった．置き換えることで NetBSD が停止してしまう箇所があった．そこで本研究の実装では，置き換えても停止しない全割込み禁止，全割込み許可のみを置き換えた．マクロ展開前のソースコード中の，全割込み禁止，または全割込み許可を 1 個と数え，全部で 29 個あるうちの，16 個を置き換えた．

置き換えることで停止してしまう原因として，MSR による全割込み禁止や全割込み許可と，本研究で実装した，仮想化した全割込み禁止や全割込み許可が混在して実行されることが考えられた．4.2.5 節で述べたとおり，NetBSD の例外ハンドラのメモリ領域は一部しか移動していない．移動していない NetBSD の割込みハンドラは，NetBSD への例外が発生したときに直接実行される．ここで，PowerPC では，次の処理がハードウェアにより行われる．例外実行時に MSR を退避した上で全割込み禁止状態になり，例外処理を終える時に行うアセンブラの rfi 命令の実行時に，退避した MSR を戻すことで，例外実行前の全割込み禁止状態もしくは全割込み許可状態を復元する．この機能により，仮想化した全割込み禁止や全割込み許可と，MSR による全割込み禁止，全割込み許可が混在している．よって，次のように実装することで，すべての全割込み禁止，全割込み許可を置き換えることができ，また正しく動作すると考えられた．NetBSD の例外ハンドラの使用するメモリ領域をすべて移動する．また，NetBSD で行われる rfi 命令を置き換えることにより，例外処理を終えるときに例外実行前の仮想化した全割込み禁止状態または全割込み許可状態に戻す．

4.3.1.2 NetBSD 例外管理タスクの改良

NetBSD 例外管理タスクの改良について述べる．全割込み禁止の仮想化を実装した時の，NetBSD 例外管理タスクの処理を図 4.10 に示した．セマフォの獲得待ちには，獲得できるまでブロッキングを行う TOPPERS の wai_sem システムコールを使用した．NetBSD タスクが全割込み許可状態である場合は，セマフォを瞬時に獲

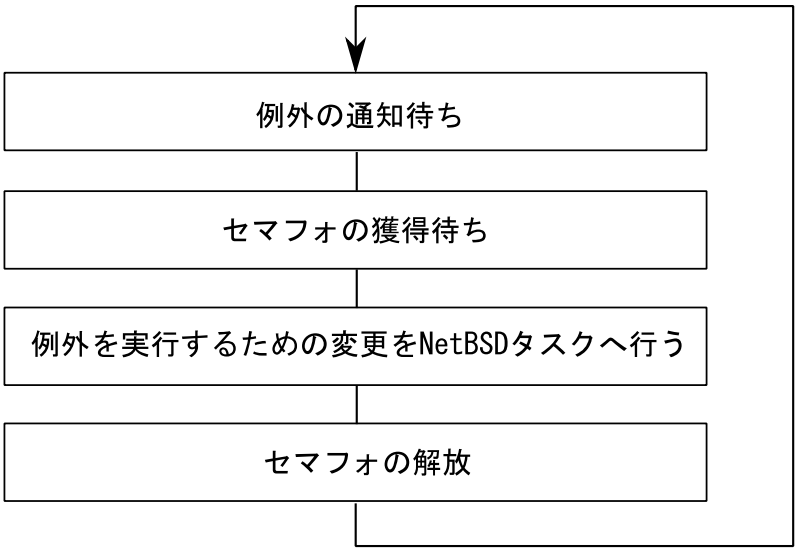

図 4.10: 全割込み禁止の仮想化を実装した時の NetBSD 例外管理タスクの処理

得し，例外実行のための NetBSD タスクへの変更を行う．一方，NetBSD タスクが全割込み禁止状態である場合は，セマフォ獲得待ちを行うときに，NetBSD 例外管理タスクが待ち状態となる．これにより，いずれ NetBSD タスクに実行が遷移し，割込み許可状態となる．全割込み許可の仮想化により，割込み許可状態になる時にはセマフォの解放が行われる．セマフォの解放が行われると，セマフォ獲得待ちをしている NetBSD 例外管理タスクに実行が遷移し，例外実行のための NetBSD タスクへの変更を行う．

4.3.2 割込みマスクの仮想化

NetBSD が割込みのマスク制御を行わないようにした．また，NetBSD 例外管理タスクでは，NetBSD の移動した割込みマスクのメモリ領域を見て，このメモリ領域で割込みがマスクされていなかった場合に割込みを行うようにした．

4.3.2.1 NetBSD が行う割込みマスク

4.2.6.2 節で述べたとおり，既存のハイブリッド OS の実装では，TOPPERS に実装した機能を実行することにより，NetBSD が割込みのマスク制御を行っていた．本研究では，NetBSD がこの機能を実行しないようにした．これは，ソースコード中の該当行を削除することにより，実現した．この改良により，NetBSD は，割込みのマスク制御の変更を物理メモリ上に書き込むだけで，実際には割込みのマスク制御を行わなくなる．

4.3.2.2 NetBSD 例外管理タスクの改良

4.3.1.2 節では，NetBSD の行う全割込み禁止と全割込み許可を仮想化するための，NetBSD 例外管理タスクの改良を述べた．本節では，この改良に加え，割込みのマスク制御を仮想化するための改良を，例外の配送部と例外の通知待ち部に分けて述べる．

まず，例外の配送部について述べる．NetBSD 例外管理タスクにおいて，全割込み禁止と全割込み許可の仮想化で使用するセマフォの獲得を行ったあと，NetBSD の移動した割込みマスクのメモリ領域を確認するようにした．そして，このメモリ領域で割込みがマスクされていない場合に，NetBSD タスクへの例外実行のための変更を行うようにした．割込みがマスクされていた場合は，割込みのマスク解除を待たずに例外の通知待ち部に戻るようにした．なぜならば，割込みのマスク解除を待つようにすると，割込みのマスク解除を待っている間は他の例外を実行することができないからだ．また，割込みがマスクされていた場合は，後で例外の配送を行うために，割込みの Interrupt Request 番号 (以下: IRQ 番号) を記録するようにした．なお以下では，割込みがマスクされていたために，割込みを行うことができなかった割込みを，マスク解除待ち割込みと呼ぶ．

次に，例外の通知待ち部について述べる．例外の通知がなくても，マスク解除待ち割込みがある場合は，例外の通知待ちで待ち状態にせずに，マスク解除待ち割込みの配送を行うようにした．これにより，マスク解除待ち割込みの割込みを行う

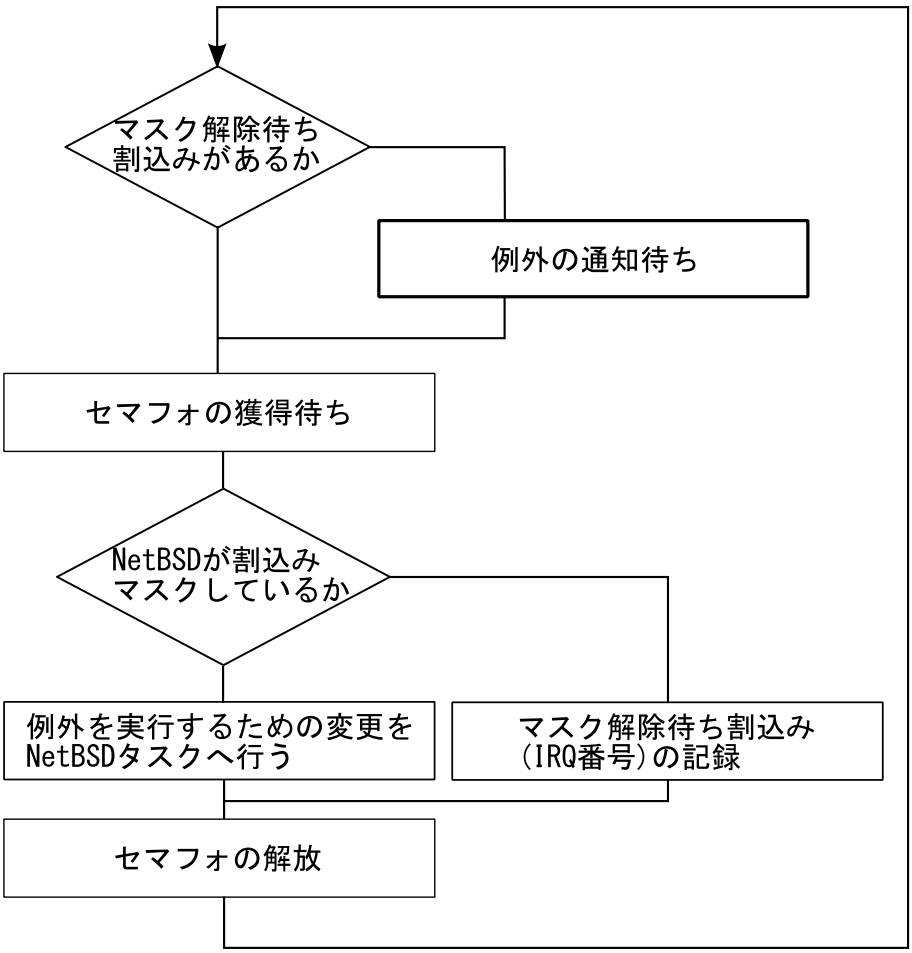

図 4.11: 割込みマスクの仮想化のための NetBSD 例外管理タスクの改良 (ただしループ状態になる場合がある)

ことができる．以上の NetBSD 例外管理タスクの改良を，図 4.11 に示した．

ただしこの改良だけでは，NetBSD タスクは実行されなくなる．以下で，この状況について考える．

NetBSD タスクがある割込みをマスクしている間に，この割込みが発生したとする．割込みハンドラが割込みの通知を行い，NetBSD 例外管理タスクがこの通知を受け取る．NetBSD 例外管理タスクは，割込み配送部で述べたとおり，IRQ 番号を記録して例外の通知待ち部に戻る．例外の通知待ち部では，この割込みがマスク解除待ち割込みとして存在するので，待ち状態にはならずに，例外の配送部を実行す

る．このように，ループ状態となる．ループ状態から抜けるためには，NetBSD タスクが割込みのマスク解除を行わなければならない．ここで，4.2.2.2 節で述べたとおり，NetBSD タスクよりも NetBSD 例外管理タスクの優先度を高くしている．よってこのループ状態のときに，NetBSD タスクが実行されることはない．以上より，NetBSD タスクは実行されなくなる．

そこで，各マスク解除待ち割込みは，マスクされていることを一度確認したら，例外の通知待ち部で停止するようにした．ここで，マスクが解除されたことを検知して，NetBSD 例外管理タスクが解除された割込みのマスク解除待ち割込みを配送するのが理想的な実装である．しかし今回は，実装の手間を軽減するために，次のように実装した．例外が配送されるたびに，例外の通知待ち部で停止するようにしたマスク解除待ち割込みを，例外の通知待ち部で停止しないようにした．このようにすることで，NetBSD 例外管理タスクに，割込みマスクの仮想化機能を実装した．以上の NetBSD 例外管理タスクの改良を，図 4.12 に示した．

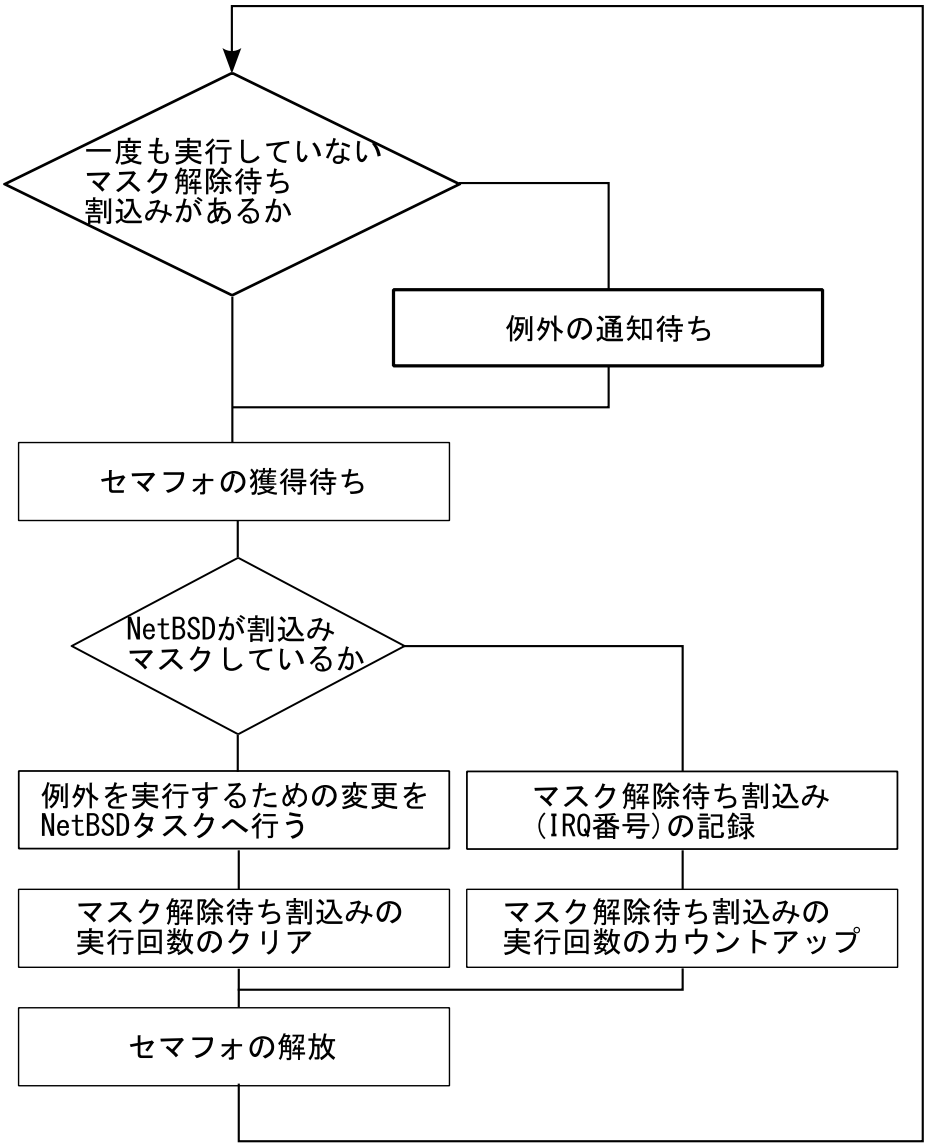

図 4.12: ループ状態になることを回避した，割込みマスクの仮想化のための NetBSD 例外管理タスクの改良

第 5 章

評価

本研究の実装により，ハイブリッド OS における TOPPERS のタスクの実時間性を確保できたかを確かめるために，次の評価実験を行った．TOPPERS のタスクを一度待ち状態にし，一定時間後に起床させた．この時の，指定した起床時刻と実際に起床した時刻との差を比較することで，TOPPERS の実時間性を調べた．

評価実験は，次のように行った．TOPPERS に評価実験用のタスクを用意し，このタスクで実時間性を調べた．評価実験用のタスクを一定時間待ち状態にし，待ち状態にする直前と起床直後とで時刻を計測した．この時刻の差により，実際に起床するまでにかかった時間を調べた．評価実験用のタスクの待ち時間には，NetBSD タスクが実行できる十分な時間を用意し，1 秒とした．また，本評価実験は，NetBSD タスクの割込み禁止やその仮想化による TOPPERS のタスクの実時間性の影響を調べることを目的としている，そこで，他のタスクの実行により起床時刻が遅れるのを防ぐために，評価実験用のタスクは，動作しているタスクの中で最高の優先度に設定した．

評価実験は，次の構成で行った．

TOPPERS のみ

ハイブリッド OS 構成にしたことによる TOPPERS のタスクの実時間性の変化を調べるために，TOPPERS のみを動作させて評価実験を行った．TOPPERS 上で評価実験用のタスクを動作させ，起床時間を計測した．ここで，ハイブリッド OS では NetBSD タスクなどの他のタスクも動作している．より近い

条件での結果を得るために，TOPPERS 上で評価実験用のタスクの他に，負荷をかけるためのタスク (以下: 負荷タスクと呼ぶ) を動作させたときの起床時間も計測した.

改良前のハイブリッド OS

本研究の改良による TOPPERS のタスクの実時間性の変化を調べるために，改良前のハイブリッド OS において評価実験を行った．改良前のハイブリッド OS にて，TOPPERS 上に新たに評価実験用のタスクを動作させ，起床時間を計測した.

NetBSD の全割込み禁止を仮想化した構成

全割込み禁止の仮想化による TOPPERS のタスクの実時間性の変化を調べるために，NetBSD の全割込み禁止を仮想化したハイブリッド OS において評価実験を行った．4.3.1 節で述べた改良を行い NetBSD の全割込み禁止を仮想化したハイブリッド OS にて，TOPPERS 上に新たに評価実験用のタスクを動作させ，起床時間を計測した.

NetBSD の全割込み禁止と割込みのマスク制御を仮想化した構成

全割込み禁止の仮想化に加え，割込みのマスク制御を仮想化したことによる TOPPERS のタスクの実時間性の実時間性の変化を調べるために，NetBSD の全割込み禁止と割込みのマスク制御を仮想化したハイブリッド OS において評価実験を行った．4.3.2 節で述べた改良を行い NetBSD の全割込み禁止と割込みのマスク制御を仮想化したハイブリッド OS にて，TOPPERS 上に新たに評価実験用のタスクを動作させ，起床時間を計測した.

また評価実験は，NetBSD の上で何も行わず負荷をかけていないときと，NetBSD の上で負荷をかけているときとで行った．本研究では，NetBSD の上からかけることのできる負荷として，搭載しているデバイスそれぞれへのアクセスがあると考えた．このアクセスによる負荷について，CPU，メモリ，周辺機器に分けて評価実

験を行った．周辺機器として，今回はネットワークデバイスを使用した．各 OS の構成において，NetBSD タスクが次の条件のときに評価実験を行った．

負荷をかけない

NetBSD 上で負荷をかけていないときの，TOPPERS のタスクの実時間性の変化を調べるために，NetBSD 上で特別に何も実行していないときに評価実験を行った．TOPPERS のみの構成では，負荷タスクを動作させずに評価実験を行った．

CPU に負荷をかける

NetBSD 上で CPU に負荷をかけたときの，TOPPERS のタスクの実時間性の変化を調べるために，NetBSD 上で空ループを実行しているときに評価実験を行った．TOPPERS のみの構成では，空ループを行う負荷タスクを動作させ，評価実験を行った．

メモリアクセスにより負荷をかける

NetBSD 上でメモリアクセスにより負荷をかけたときの，TOPPERS のタスクの実時間性の変化を調べるために，NetBSD 上で行列の乗算を実行しているときに評価実験を行った．TOPPERS のみの構成では，行列の乗算を行う負荷タスクを動作させ，評価実験を行った．

ネットワークアクセスにより負荷をかける

NetBSD 上でネットワークアクセスにより負荷をかけたときの，TOPPERS のタスクの実時間性の変化を調べるために，NetBSD 上で NFS プロトコルによる通信を行っているときに評価実験を行った．TOPPERS のみの構成では，この条件の評価実験は行わなかった．

評価実験は，4 章で述べた環境で行った．なお，MPC5200B は，400MHz で動作する．一定時間待ち状態にする機能として，TOPPERS に実装されている，自タスクを

表 5.1: 各構成における負荷をかけないときの起床時間

	最短 (μs)	最長 (μs)	平均 (μs)	分散 ((μs)^2)
TOPPERS のみ	1002204	1002204	1002204	0.00
改良前	1002226	1002228	1002231	0.24
全割込み禁止	1002224	1002226	1002228	0.12
全割込み禁止と割込みマスク	1002211	1002214	1002217	0.75

一定時間遅延させる dly_tsk システムコールを使用した．評価実験を行う TOPPERS のタスクで dly_tsk システムコールを実行した．そして，dly_tsk システムコールの前後の時刻を計測することにより，タスクが待ち状態から実行状態になるまでの時間を計測した．ここで PowerPC には，CPU の何分周かに同期してカウントアップする Time Base が実装されている．本研究で使用した mmEyePPC では，Time Base は 33MHz で動作する．時刻の計測には，この Time Base を利用した．

各構成において，各条件での計測を 100 回ずつ行った．各構成における負荷をかけないときのタスクの起床の最短時間や最長時間，平均時間，時間の分散を表 5.1 に記載した．また，各構成における各条件の起床の最短時間を表 5.2 に，最長時間を表 5.3 に記載した．各表中では，改良前のハイブリッド OS を改良前，全割込み禁止を仮想化したハイブリッド OS を全割込み禁止，全割込み禁止と割込みのマスク制御を仮想化したハイブリッド OS を全割込み禁止と割込みマスクと表記した．また，表 5.2，表 5.3 では，CPU に負荷をかけたときの結果を CPU，メモリアクセスにより負荷をかけたときの結果をメモリ，ネットワークアクセスにより負荷をかけた時の結果をネットワークと表記した．

まず，NetBSD 上で負荷をかけないときの実験結果について述べる．この時，表 5.1 より，改良前のハイブリッド OS に比べ，全割込み禁止を仮想化することによ

表 5.2: 各構成における各条件の起床の最短時間

	負荷なし (μs)	CPU(μs)	メモリ (μs)	ネットワーク (μs)
TOPPERS のみ	1002204	1002212	1002221	
改良前	1002226	1002211	1002222	1005556
全割込み禁止	1002224	1002211	1002215	1002223
全割込み禁止と割込みマスク	1002211	1002217	1002221	2564506

表 5.3: 各構成における各条件の起床の最長時間

	負荷なし (μs)	CPU(μs)	メモリ (μs)	ネットワーク (μs)
TOPPERS のみ	1002204	1002212	1002222	
改良前	1002231	1002215	1002224	1006808
全割込み禁止	1002228	1002217	1002218	1006223
全割込み禁止と割込みマスク	1002217	1002221	1002225	3217994

り，最短時間，最長時間，平均時間，分散ともに改善した．これより，次のことが考えられた．改良前のハイブリッド OS では，NetBSD タスクの全割込み禁止によりタイマ割込みの実行時刻が遅れ，TOPPERS のタスクの起床する時刻が遅くなっていた．NetBSD の全割込み禁止を仮想化することで，NetBSD タスクが全割込み禁止を行わないようになり，TOPPERS のタスクの起床時刻の遅れが改善した．また，割込みのマスク制御を仮想化することにより，最短時間，最長時間，平均時間はさらに改善した．これより，全割込み禁止と同様に，改良前のハイブリッド OS では NetBSD タスクの割込みのマスク制御により TOPPERS のタスクの起床する時刻が遅くなっていたのに対し，NetBSD タスクの割込みのマスク制御を仮想化することでこの遅れが改善したと考えられた．しかし，時間の分散は悪化した．これに関して，最短時間，最長時間を見ると，ともに平均時間との差が大きくなった．ここで，表 5.2 より，全割込み禁止と割込みのマスク制御を仮想化したハイブリッド OS の最短時間は，TOPPERS のみの構成で CPU に負荷をかけたときの最短時間よりも早かった．これより，最短時間と平均時間との差が大きくなったことに関しては，NetBSD タスクの全割込み禁止と割込みのマスク制御の割込みの仮想化により起床時刻の遅れが改善したことが原因だと考えられた．また，最長時間と平均時間との差が大きくなったことに関しては，NetBSD の一部の例外ハンドラしか移動していないことにより依然として NetBSD タスクが全割込み禁止を行うことに加え，本研究の実装によりオーバヘッドが大きくなったことが原因と考えられた．

次に，TOPPERS のみの構成と，全割込み禁止と割込みのマスク制御を仮想化したハイブリッド OS との比較について述べる．表 5.2 より，全割込み禁止と割込みのマスク制御を仮想化したハイブリッド OS の最短時間は，TOPPERS のみの構成で CPU に負荷をかけたときの最短時間よりも早かった．これより，最短時間に関しては，全割込み禁止と割込みのマスク制御を仮想化したことにより，NetBSD タスクが TOPPERS のタスクの実時間性を損なうことがなくなったといえた．しかし，TOPPERS のみの構成に対して，全割込み禁止と割込みのマスク制御を仮想化

したハイブリッド OS は，最長時間，平均時間，分散において悪化した．これは，NetBSD の一部の例外ハンドラしか移動していないことにより，依然として NetBSD タスクが全割込み禁止を行うことが原因だと考えられた．

次に，NetBSD 上で CPU に負荷をかけたときの実験結果について述べる．このとき，表 5.3 より，TOPPERS のみ，改良前のハイブリッド OS，全割込み禁止を仮想化したハイブリッド OS，全割込み禁止と割込みのマスク制御を仮想化したハイブリッド OS の順で，TOPPERS のタスクの起床時刻が遅くなった．これは，NetBSD の一部の例外ハンドラしか移動していないことにより依然として NetBSD タスクが全割込み禁止を行うことに加え，本研究の実装によりオーバヘッドが大きくなったことが原因と考えられた．

次に，NetBSD 上でメモリアクセスやネットワークアクセスにより負荷をかけたときの実験結果について述べる．表 5.3 より，これらの条件では，全割込み禁止の仮想化により TOPPERS のタスクの起床時刻が早くなった．これより，全割込みの仮想化により，TOPPERS のタスクの実時間性が改善したと考えられた．しかし，割込みのマスク制御の仮想化により TOPPERS のタスクの起床時間は遅くなった．特に，ネットワークアクセスにより負荷をかけたときは，TOPPERS のタスクの起床時刻が極端に遅くなった．これは，割込みのマスク制御の仮想化のための本研究の実装が不完全であることが原因だと考えられた．

第 6 章

関連研究

6.1 Linux on ITRON

保田らは，Linux on ITRON[1, 13] を実装している．これは，μITRON の 1 タスクとして Linux を動作させるハイブリッド OS である．この実装では，Linux の割込み処理を μITRON のタスク例外処理に置き換えることにより，Linux の割込み処理を仮想化している．これにより，Linux の全割込み禁止により μITRON に実行が遷移しない問題を解決し，実時間性を保持したハイブリッド OS を構築している．

しかし，Linux on ITRON では Linux が行う割込みマスクにより実時間性が損なわれることは考慮していない．我々の研究では，Unix が行う割込みマスクも仮想化し，Unix が行う割込みマスクにより実時間が損なわれることがないようにした．

Linux on ITRON では，Linux カーネルが行う全割込み禁止を μITRON のタスク例外禁止に，全割込み許可を μITRON のタスク例外許可に置き換え，Linux の割込み発生時にタスク例外を実行することで，Linux の割込み処理の仮想化を実装している．この実装により，我々の研究と比べて，実装するために新たにメモリ領域を確保しなくてよいという利点がある．我々の研究では，Unix が仮想化された全割込み禁止，全割込み許可を実行するために，この機能を格納しておくメモリ領域を確保している．

また Linux on ITRON では，Linux の全割込み禁止，全割込み許可から μITRON のタスク例外禁止，タスク例外許可への置き換えは，Linux のソースコード中の全

割込み禁止，全割込み許可のマクロの書き換えだけで済んでいる．我々の研究では，元にしたソースコードにて全割込み禁止，全割込み許可はマクロを使用せずに実装してあったため，全割込み禁止，全割込み許可を行っている箇所を特定し，一つ一つ置き換えるという作業が必要となった．

また，Linux on ITRON では我々の研究に比べて，Linux の割込みの仮想化の実装が難しくなるという欠点が考えられる．なぜなら，タスク例外処理は μITRON のカーネル内に実装されているので，Linux の割込みの置き換えを μITRON のカーネル内に実装するからだ．これにより，Linux の割込みの仮想化の実装では，μITRON のシステムコールを使用することはできない．また，この実装がノンプリエンプト区間内での実装であった場合，仮想化のオーバヘッドにより μITRON のタスクの実行開始が遅れる可能性がある．我々の研究では，Unix の割込みの仮想化の実装は μITRON のタスクとして実装しているので，実装の際にシステムコールを使用することができ，またいかなる時も割込みの仮想化処理を μITRON のタスクがプリエンプト可能である．

6.2 RTLinux

Barabanov らは，Linux カーネルを元に，実時間性のある RTLinux[14] を実装している．RTLinux では，独自に実時間カーネルを実装し，Linux カーネルを実時間カーネルのタスクの 1 つとして動作させている．この時，Linux の割込み処理を仮想化している．これにより，Linux の全割込み禁止により実時間カーネルに実行が遷移しない問題を解決し，実時間タスクの実時間性を保持している．

具体的には，Linux カーネルの全割込み禁止と全割込み許可を実際には行わず，特定のメモリ領域に値を書き込むようにしている．そして割込みハンドラ内では，この領域の値を確認し全割込み許可であることを確認してから，割込み処理を実行している．割込み処理を仮想化し，全割込み禁止により実時間性を損なうこと

がないようにしている点は，我々の研究と同じである．また，我々の研究ではさらに，Unix が行う割込みマスクにより実時間性を損なうことがないように対処した．

RTLinux では，独自に実時間カーネルを実装していることが，我々の研究と比べて欠点となる．例えば，既に開発されている実時間性の必要なタスクを使用する場合，RTLinux では実時間タスクを一から実装しなければならない．我々の研究では，μITRON で既に実装されているタスクは，ハイブリッド OS でも同じ μITRON が動作しているので，そのまま使用できる．また RTLinux では，独自に実時間カーネルを実装しているので，実時間タスクの開発者は実時間カーネルに実装されているシステムコールを学習しなければならない．これに対して，我々の研究では，実時間カーネルとして μITRON を使用することができる．実時間カーネルとして μITRON を使用することにより，μITRON の仕様を知っている実時間タスクの開発者は，学習なしに実時間タスクを実装できる．

また，RTLinux では優先度ベースの独自のスケジューラを実装している．Linux カーネルは最低優先度の実時間タスクとして動作させている．我々の研究では，独自にスケジューラを実装する必要はない．我々の研究では，Unix は μITRON の一タスクとして動作させるので，スケジューラは μITRON に実装されているものをそのまま使用できる．

6.3 Gandalf VMM

追川らは，RTOS と Linux を動作させることのできる Gandalf Virtual Machine Monitor(以下: Gandalf VMM)[8] を実装している．完全仮想化を行った仮想マシン上で Linux を動作させることで，Linux が RTOS の実時間性を阻害することを防いでいる．

また，Virtual Machine Monitor(以下: VMM) の上で RTOS を動作させる場合，VMM にも実時間性が必要になる．例えば，仮想マシンのスケジューリングでは，

RTOS が動作している仮想マシンを優先させなければならない．そうでない場合，例えば割込み発生時にすぐに RTOS に処理をさせないと割込みの実時間性がなくなる可能性があり，また RTOS の実行中に Linux の仮想マシンが実行をプリエンプトすると，実時間性の必要な処理の実行が許容時間内に終わらずに，実時間性がなくなる可能性がある．しかし，Gandalf VMM では，こういった VMM の実時間性は考慮していない．評価実験では，RTOS のタイマ割込みの遅延が確認されている．我々の研究では，VMM は用いていない．RTOS はハードウェア上で直接動作するので，RTOS だけに実時間性があればよい．

第 7 章

結論

本研究では，RTOS の一タスクとして Unix を動作させるハイブリッド OS を扱った．このようなハイブリッド OS では，実時間性と多機能性を単一のシステムで両立することができるが，本研究ではこの二つの性質のうちの実時間性に着目した．

このようなハイブリッド OS で，実時間性を確保できない要因を考えた．実時間性を確保できない要因として，Unix が行う全割込み禁止と割込みのマスク制御があると考えた．

実装には，実時間性の確保を考慮していない既存のハイブリッド OS を使用した．この既存のハイブリッド OS を，実時間性を確保できるように改良した．具体的には，Unix が行う全割込み禁止と全割込み許可の仮想化，そして割込みのマスク制御の仮想化を行った．Unix の全割込み禁止と全割込み許可，そして割込みのマスク制御を置き換え，Unix の割込み禁止に変わる排他制御を RTOS に実装した．

今回は，実装の簡単化のため，すべての全割込み禁止と全割込み許可を置き換えることはしなかった．しかし，改良したハイブリッド OS について評価実験を行った結果，改良したハイブリッド OS では，Unix に負荷がかかっていないときにおいて，RTOS の実時間性が向上したことが確認できた．

また，以下に，今後の課題をまとめた．

Unix は常に割込み許可状態で実行する

本研究の実装では，Unix が行う全割込み禁止と全割込み許可の一部は，仮想化機能で置き換えせずにそのまま残している．また，一部の Unix の例外ハン

ドラが直接実行されることにより，この時も全割込み禁止状態となる．これらにより，Unix が全割込み禁止状態で実行する区間が残っている．これは，RTOS の実時間性が損なわれる原因となりうる．

割込みのマスク解除の検知

本研究の実装では，RTOS から Unix への割込みの配送において，Unix が割込みのマスクを解除したことをポーリングで検知している．また，このポーリングが行われるタイミングは，未定である．これに関して，Unix の割込みのマスク解除に，RTOS のイベント通知機能を利用する方法が考えられる．割込みの配送部がこのイベントを受け取ることにより，マスク解除待ちの割込みを瞬時に配送することができる．

謝辞

本研究を遂行するにあたり，いろいろな方にお世話になりました．

まず，指導教員の多田好克教授と佐藤喬助教には日頃から熱心なご指導，ご鞭撻を賜りました．また，ご多忙中にもかかわらず論文の草稿を丁寧に読んで下さり，大変貴重なご助言をいただきました．ここに厚く御礼申し上げます．

基盤ソフトウェア学講座の小宮常康准教授には，研究方針や研究の考え方について，分かりやすく的確なご助言をいただきました．深く感謝いたします．

株式会社ブレインズの堀内岳人様には，本研究で使用した機材をお貸しいただきました．また，同ブレインズの藤田昌平様には，この機材に関するサポートをしていただきました．ご多忙中にもかかわらず，ありがとうございました．

最後に，基盤ソフトウェア学講座の学生諸君，諸君と共に研究室生活を過ごす中で，たくさんのことを学び，また経験を得ました．ありがとうございました．

参考文献

[1] 保田信長, 飯山真一, 冨山宏之, 高田広章, 中島浩: “Linux と ITRON によるハイブリッド OS の設計と実装”, 情報処理学会研究報告, システム LSI 設計技術, No.114, pp.45–50, 2004.

[2] Heitmeyer, Constance, Mandrioli, Dino and Milano, Politecnico: “Formal methods for real-time computing,” John Wiley & Sons, Inc., New York, ISBN0471958352, NY, USA, 1996.

[3] “The Linux Kernel Archives,” http://www.kernel.org/ .

[4] “The NetBSD Project,” http://www.netbsd.org/ .

[5] “Wind River VxWorks — ウインドリバー株式会社”, http://www.windriver.com/japan/products/vxworks/ .

[6] “TOPPERS プロジェクト／ INDEX”, http://www.toppers.jp/ .

[7] Robert Kaiser and Stephan Wagner: “Evolution of the PikeOS Microkernel,” Proceedings of the 1st international workshop on microkernels for embedded systems, pp.50–57, 2007.

[8] Shuichi Oikawa, Megumi Ito and Tatsuo Nakajima: “Linux/RTOS Hybrid Operating Environment on Gandalf Virtual Machine Monitor,” The 2006 IFIP International Conference on Embedded and Ubiquitous Computing ,Vol.4096, pp.287–296, 2006.

[9] 澤田 勉: “わかりやすい 組込システム構築技法 ソフトウェア編”, 共立出版株式会社, 東京, 2006.

[10] Freescale Semiconductor, Inc: “Programming Environments Manual For 32-Bit Implementations of the PowerPC Architecture,” REV 2, 2002.

[11] Freescale Semiconductor, Inc: “MPC5200 Users Guide,” Rev. 3, 2005.

[12] “WebHome < U-Boot < DENX,” http://www.denx.de/wiki/U-Boot/WebHome .

[13] Hiroaki Takada, Shin’ichi Iiyama, Tsutomu Kindaichi and Shouichi Hachiya: “Linux on ITRON: A Hybrid Operating System Architecture for Embedded Systems,” IEEE Computer Society, Applications and the Internet Workshops, IEEE/IPSJ International Symposium on, pp.4–7, 2002.

[14] Michael Barabanov and Victor Yodaiken: “Introducing Real–Time Linux,” Linux Journal, no.34, pp.19–23, 1997.