平成17年度 修士論文

カーネル外スレッド et の適用範囲拡大のための追加機能の設計と実装

電気通信大学 大学院情報システム学研究科

情報システム設計学専攻

0450028 藤田 昌平

指導教員 多田 好克 教授
岡本 敏雄 教授
渡辺 俊典 教授

再提出日 平成18年2月23日

目 次

図 目 次

表 目 次

第 1 章

はじめに

最近の組み込み系計算機は, シリアルポートなどのレガシーデバイスだけではなく, USB や Ethernet など PC 並に高度な機能を持つ物が増えてきている. このような状況では, 搭載するソフトウェアを一から作成することは, 手間がかかり無駄が多くなる. そこで, Linux[2] や, FreeBSD[3] や NetBSD[4] などの UNIX 系の OS などを使用することが多くなっている. これらの OS は, 多数の CPU アーキテクチャ上に実装されており, また多数のハードウェアをサポートしているため, 作成するソフトウェアの量が少なくて済む. UNIX 系の OS には, 多くの優れたアプリケーションソフトウェアが存在するため, これらのソフトウェアを利用するために使用することも多い.

また, 組込み系ではロボット制御や情報家電など, 独自のハードウェアの制御などを行う必要が出てくる. 上記の UNIX 系 OS では, ハードウェアは仮想化されているため, ユーザプログラムではハードウェアを直接制御することが不可能である. また, CPU も仮想化され, 複数のユーザプログラムで時分割で共有されるため, 処理に実時間性がない. そこで, 本研究ではハードウェアやカーネル等の, ユーザプログラムでは制御することができない機能の制御を行うプログラムを, 簡単に作成できるような機構の構築を行う.

本研究では, 当研究室で開発された, et (external thread)[1] という機構を元にして上述した機構の開発を行う. et とは, カーネルメモリ内にロードされ, カーネルではなく独自で実行管理を行うスレッドである. このスレッドはカーネルモードで

実行されるため, ハードウェアの直接制御やカーネルパラメータの動的変更などを行うことができる. しかし, オリジナルの et はカーネル外スレッドを動作させるために必要最低限の機能しか持っておらず, 適用可能なプログラムの範囲が狭い.

そこで本研究では, et の適用範囲を広げるために以下の点について設計を行った.

- システムの導入や管理を簡単に行えるような基本設計
- スレッドの実行管理機構の設計
- ユーザプログラムとの通信機構の設計
- マルチスレッド対応設計

そして, これらの設計を et への追加機能として実装した. また, オリジナルの et は古い FreeBSD において実装されているため利用機会が少ない. そこで, 本研究では最近, 組み込み系用途などでよく利用されている Linux 2.4 上に実装を行った.

そして, 実装を行った追加機能の評価を行った. この評価は, 実際の利用状況を想定して, 一部ではサンプルプログラムを作成し, 以下の結果が得られた.

- 機能拡張により本システムの導入や利用が簡単になった
- 機能拡張により本システムの適用可能範囲が拡大した

また, スレッドが動作することによる, 他のアプリケーションへの影響について測定を行い, オーバーヘッドが大きくないことを確認した. また, 機能追加によるオーバーヘッドの測定を行い, オーバーヘッドの増加が少ないことを確認した.

第 2 章

et (external thread)

本章では, 本研究での開発のもととなった, オリジナルの et (external thread) の機能, 実装について説明する. そして, et の問題点を明らかにする.

2.1 et の機能

et の基本的な機能は, 1 つ以上の関数で構成されたスレッド (以後, このスレッドを et スレッドと呼称する) をカーネル管理外で周期的に実行させる事である. カーネル管理外で独自に実行管理を行うため, et スレッドの実行に実時間性を持たせることができる.

et スレッドはカーネルメモリ上にロードされカーネルモードで動作する. カーネルモードで動作するため, ハードウェアの制御を直接行う事や, カーネルパラメータのチューニングなどを行う事が可能である (図 2.1).

上記の, 実時間性を持った実行, ハードウェアやカーネルパラメータへの直接アクセスの機能を用いることで, et スレッドは以下のような機能を実現することができる.

- 実時間性を利用し, 時間的に連続性が要求される, マルチメディアデータや, センシングデータの処理を行う.
- 実時間性, ハードウェアへの直接アクセスを利用し, 信号線を特定タイミングで on/off してハードウェアの制御を行う.

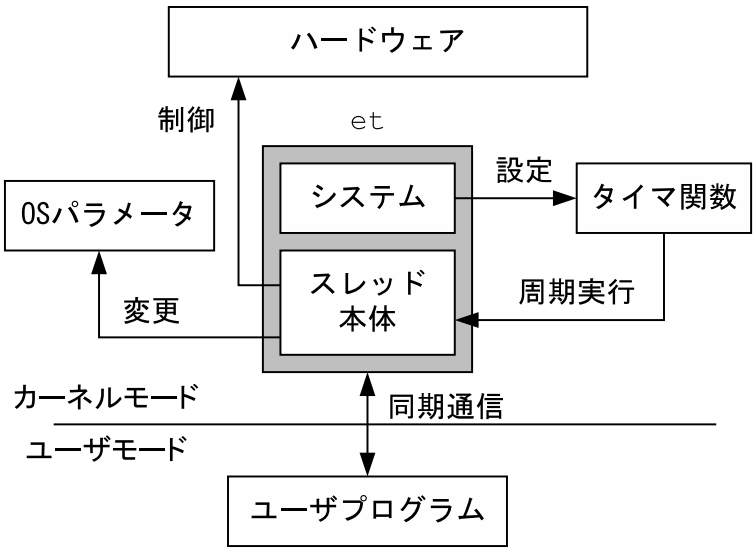

図 2.1: et の基本構造

et スレッドのプログラマはシステムの提供する関数群 (表 2.1) を用いて周期実行される関数をプログラミングする.

2.2 et の実装

オリジナルの et の実装では, et スレッドは 1 つ以上の関数で実装され, システム関数とまとめて 1 つのローダブルモジュールを構成する. モジュールロード時にシステムが初期設定を行い, 以後 et スレッドと unix を切り替えながら動作する. ここで言う unix とは, et スレッド以外のカーネルとユーザプログラムの全てを指す.

2.2.1 et スレッドと unix の切り替え

et スレッドと unix の切り替えには, *setjmp()*, *longjmp()* という大域ジャンプを用いる. この大域ジャンプは, プログラムのスタックの切り替えと処理の移行を同時に行うため, 同じメモリ空間内の別プログラムへと処理を切り替えることが可能

表 2.1: オリジナル et のシステムが提供する関数

名前	説明
et_yield	CPU 利用を放棄する
et_exit	et スレッドの動作を終了する
p_printf	コンソールへフォーマット付き文字列を出力する
p_putchar	コンソールへ 1 文字出力する
p_write	キャラクタデバイスへデータを指定サイズ出力する
p_writec	キャラクタデバイスへデータを 1 バイト出力する
p_read	キャラクタデバイスからデータを指定サイズ入力する
p_readc	キャラクタデバイスからデータを 1 バイト入力する
ET_LOCK	一時的に unix へ処理が移行しないようにロックする
ET_UNLOCK	unix へ処理が移行するようにロックを解除する

である.

モジュールロード時に, システムは et スレッドのスタック用メモリをカーネルメモリ空間に確保する. そして, そのメモリ上に et スレッドのスタックフレームを構築する. そのスタックフレーム上のリターンアドレスを et スレッドの関数のアドレスに書き換える. この状況で大域ジャンプである *longjmp()* を用いてスタックの切り替えと処理の移行を行うと, et スレッドの関数へ処理を移行することができる.

また, et スレッドへ処理を移行する前に *setjmp()* を実行しておくことで, et スレッド実行中に *longjmp()* を実行することで unix へ処理を移行することができる. 以後, この *setjmp()*, *longjmp()* を周期的に実行することで, et スレッドの周期実行を行う.

2.2.2 切り替えタイミング

et スレッドと unix の切り替えタイミングには, カーネルのタイムアウト関数を用いている. カーネルのタイムアウト関数とは, 設定された時刻が来ると登録されていたタイムアウトルーチンを呼び出す関数である. *setjmp()* と *longjmp()* を用いた et スレッドと unix の切り替えを行う関数を, タイムアウトルーチンとして設定しているため, 指定時刻に et スレッドと unix が切り替わる. 切り替え関数では, et スレッドが周期実行するような次の切り替えタイミングをタイムアウト関数に設定してから切り替えを行う.

2.3 et の問題点

オリジナルの et は, et スレッドを周期実行するための必要最低限の機能しか持っていない. また, 実装が 1 種しか存在しない. そのため, 以下のような問題が存在し, 適用可能な範囲が狭くなっている.

2.3.1 実装

オリジナルの et の実装は, FreeBSD 2.2.5 のみであり, 現在はほとんど使用されていない環境である. そのため, et が利用可能な機会が非常に少ない. また, et の実装はカーネルのバージョンに依存するため, FreeBSD 3.x 以降のバージョンではそのまま動作させることはできない.

また, 1 つの et スレッドは, システム関数と et スレッドを構成する関数の両方を含むローダブルモジュールである. そのため, 複数のローダブルモジュールにシステムが散在することになり, システムのバージョンアップなどを行う場合は, モジュール単位で全てのモジュールを再構築する必要がある.

2.3.2 実行周期

et スレッドの実行周期は, et スレッドプログラム時に静的に指定した値に固定されて動的な変更ができない. そのため, 一時的に実行周期を細かくして高い精度での制御を行ったり, 逆に実行周期を粗くして, 無駄に CPU を占有しないようにする, といった状況にあった処理が行えない.

また, CPU 利用を放棄や延長した場合, 次に et スレッドが実行される時刻が, et スレッドから unix へ切り替わった時刻を起点に一定時間経過後になる. CPU 利用の放棄を行う時刻や延長後の時刻が固定でないため, 一定間隔で et スレッドを実行したい場合には, CPU 利用の放棄や延長を利用することができない.

2.3.3 終了処理

オリジナルの et は, 外的な終了要因であるモジュール取り外しに対応した処理を et スレッドで行うことができない. このため, 外的要因により et スレッドの実行が停止される場合, 正常な終了処理を行う事ができない. 終了処理が行えない事により, デバイスの終端処理やカーネルパラメータの原状復帰などができずに, モジュー

ル取り外し後にデバイスやカーネルの動作に不具合が発生する可能性がある.

2.3.4 通信方法

ユーザプログラムとの通信方法がランデブによる同期通信 1 種類のみしか実装されていない. そのため, 非同期的な通信を行うことができず, 柔軟な et スレッドプログラミングが不可能になっている. また, 1 つの et スレッドが利用できる通信経路は 1 つのみに限定されるため, 1 つの et スレッドが複数のユーザプログラムと連携することができない.

また, et スレッドとユーザプログラムとの同期手段がランデブによる同期通信のみであり, 同期のみを行う機能を持っていない.

2.3.5 同時稼働 et スレッド数

コアとなるシステムと, et スレッドを構成する関数群が 1 つのモジュール内にまとめられている. そのため, 複数のモジュールを同時ロードすると, コアとなるシステムも複数ロードされることになり, システム同士が競合を起こしてしまう. そのため, 同時に動作することができる et スレッドが 1 つのみに限られている. この制限により, 複数機能を同時に実現することが不可能になる.

第 3 章

eET (enhanced External Thread) の設計

本章では, 前章で挙げたオリジナルの et の問題点を解決するための追加機能の設計について述べる. 以後, 機能追加された et を “eET (enhanced External Thread)” と呼称する.

3.1 設計概要

本章では, 前章で挙げた問題点への解決策として, 以下の点に分けて設計を行う.

- eET の導入, 管理, 利用が簡単に行えるようにするための “基本設計”
- 実行周期の動的変更, CPU 利用の放棄や延長時の次回実行時刻の決定, モジュール取り外し時の終了処理などのを実現するための “実行管理設計”
- ユーザプログラムとの通信を複数種実現するための “通信機能設計”
- 複数 et スレッドを同時に実行可能にするための “マルチスレッド対応設計”

3.2 基本設計

オリジナルの et では, システム部分と et スレッド部分が 1 つのローダブルモジュールとして実装され, 不可分になっていた. しかし, このような実装ではシステム部分か et スレッド部分のどちらか一方に変更が必要な場合でもローダブルモジュール全てを再構築する必要がある. また, システム部分に変更があった場合, ローダブルモジュールを再構築しなければ新しいバージョンのシステムで et スレッドを実行することができない.

そこで, コアとなるシステム部分と et スレッド部分を分離し, システム部分のモジュールと (以下, コアモジュール), et スレッド部分のモジュール (以下, スレッドモジュール) の異なるローダブルモジュールとして生成するような構造にする (図 3.1). この分離により, et スレッドごとに別々のシステムを持つ必要が無くなりシステムの一元管理が可能になる.

この他に, マルチスレッド対応の面からもシステム部分と et スレッド部分を別のモジュールに分割する. マルチスレッド対応についての設計は 3.5 章にて詳解する.

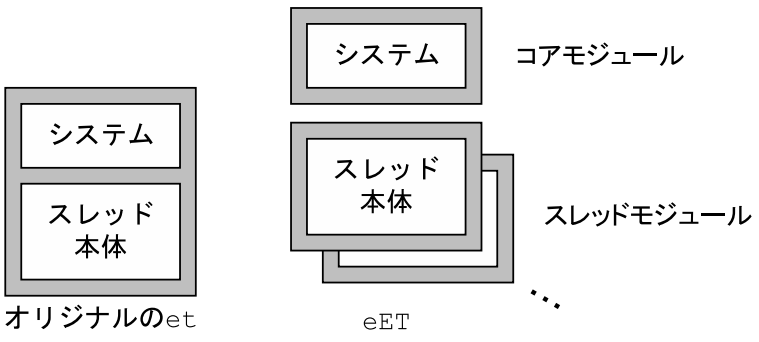

図 3.1: オリジナルの et と eET のモジュール構造の比較

しかし, ただ分割しただけではコアモジュールが実行すべき et スレッドを決定することができない. そこで, スレッドモジュールではモジュールロード時に実行す

べき et スレッドをコアモジュールに登録する．システムでは, 登録された et スレッドを周期的に実行する．

3.3 実行管理設計

3.3.1 et スレッドと unix の CPU 利用時間配分の動的変更

オリジナルの et では, et スレッドと unix の CPU 利用時間の配分が, et スレッドプログラミング時に静的に指定した値から変更することができない．このままでは, 一時的に実行周期の粒度を細かくして, 精度の高い制御へ移行したり, 逆に無駄な CPU 利用を避けるために粒度を粗くすることができない．そこで, オリジナルの et では固定であった, et スレッドと unix の CPU 利用時間配分を動的に変更可能な機構を設ける．この機能を追加することで, 状況に合わせて実行周期を調整可能な et スレッドを作成することができる．

et スレッドの CPU 利用時間コントロール機構を et スレッドとユーザプログラムのどちらにでも持たせる事を可能にするため, この時間配分の動的変更は et の et スレッドとユーザプログラムの両方から設定できるようにする．この機構により, ユーザが設定用のユーザプログラムを利用して, 手動で実行周期を変更することも可能になる．

3.3.2 et スレッドの CPU 利用の放棄と延長

CPU 利用の放棄

オリジナルの et は, et スレッドで必要な処理が終了して CPU 利用時間が必要なくなった場合, unix へと CPU を明示的に渡す機能, *et_yield()* を持っている．しかし, et スレッドが CPU を放棄した場合の次の et スレッド実行時刻は, CPU を放棄した時刻から unix の CPU 利用時間が経過したときのみとなっている (図 3.2)．こ

のような機構では, et スレッドの実行間隔を一定にすることができず, 一定間隔での処理が必要な et スレッドの作成が不可能である. そこで, eET では CPU 放棄後の次の et スレッド実行時刻を,

- オリジナルの et と同様に unix の CPU 利用時間が一定
- et スレッドの実行間隔が一定 (図 3.3)

の 2 種類から選択可能にする.

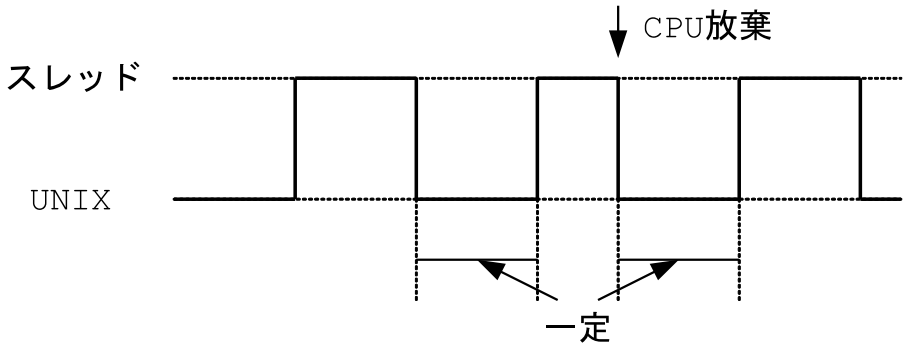

図 3.2: unix の CPU 利用間隔が一定な実行周期

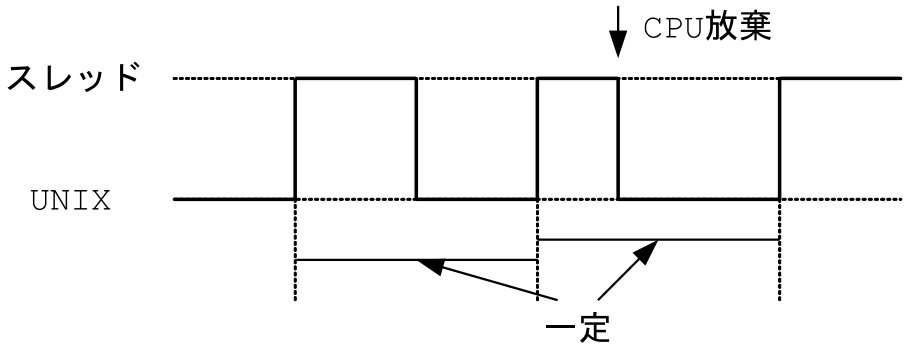

図 3.3: et スレッドの実行間隔が一定な実行周期

CPU 利用の延長

オリジナルの et では, et スレッドの要求する CPU 時間が不足した場合に, 一時的に CPU 利用を延長する機能, *ET_LOCK*, *ET_UNLOCK* を持っている. この機能は, 時間的に不可分な処理を行う場合や, 排他制御が必要なカーネル資源へのアクセスなどを行う際に, unix へ処理を移さずに et スレッドの処理を行うために用いられる.

この機能も, CPU 利用の放棄と同じく, 延長後の次の et スレッドの実行時刻を設定できず, 一定間隔での et スレッド実行が不可能になっている. そこで, CPU 利用の延長を行った場合も, 放棄の場合と同じく, 次の et スレッド実行時刻を 2 種類から選択可能にする.

3.3.3 特定イベントに対する処理

オリジナルの et はモジュール取り外しなどの OS のイベントに対応して処理を行うための機構を持っていない. ローダブルモジュールの機能として, モジュール取り外し時に処理を行う事が可能だが, その場合は et スレッドのプログラミングではなく, システムへの変更が必要になる. そのため, モジュール取り外しなどによる “外的要因による終了” に et スレッドで対応する事ができない.

そこで, 終了処理を et スレッドのメイン処理から分離し, モジュール取り外しなどの外的な要因による et スレッド終了時にシステムが実行する. システムに実行機能を持たせることで, 外的要因による終了の場合でも, et スレッドでの対応が可能になる.

さらに, 終了処理と対になった処理を行う事の多い初期処理を, 終了処理と対応させてメイン処理から分離する. こちらは, et スレッド動作開始時にシステムが実行する (図 3.4).

この初期処理と終了処理の分離により, メイン処理では周期実行すべき処理のみ

を記述することになり, et スレッドプログラム時に全体の見通しがよくなる.

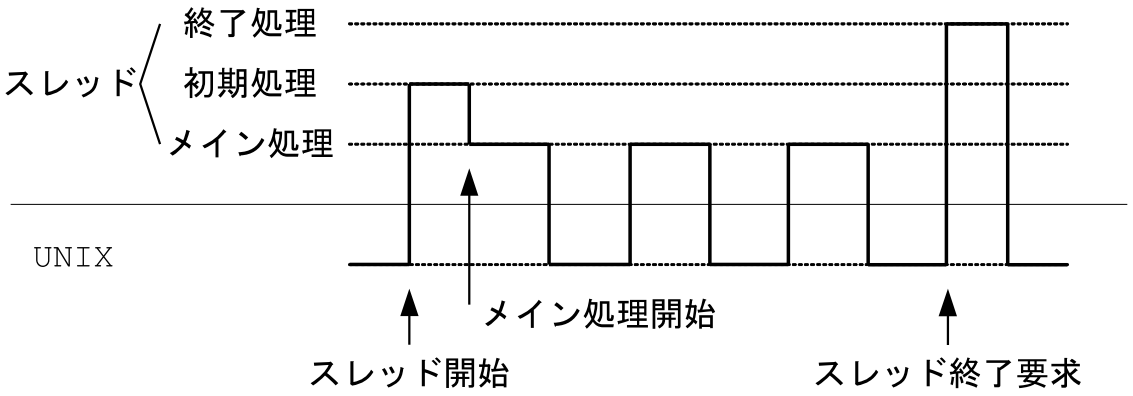

図 3.4: 初期処理と終了処理の実行タイミング

3.4 通信機能設計

本節では, et スレッドとユーザプログラムが協調動作を行うのに必要となる, 通信機能の設計について述べる.

3.4.1 通信方法

et スレッドとユーザプログラムとの通信方法には, 特殊なインタフェイスは用いず, 一般的なシステムコールを用いることで, 既存ソフトウェアを利用可能にする. et スレッドでのデータ送受信では, ユーザプログラムでのシステムコール呼び出しと同じようなインタフェイスとし, プログラミングしやすいものを目指す.

3.4.2 通信モード

オリジナルの et ではユーザプログラムとの通信はランデブによる同期通信の 1 種類のみだが, eET では複数種用意しプログラミングに柔軟性を持たせる. 本研究

では次の 3 種類を設計した.

ノンブロッキング通信

この通信モードは, ログ出力などの, 通信が et スレッドの動作に影響を与えることが好ましくない通信に用いる非同期通信である.

読み込み時は, 読み込むべきデータの有無に関わらずすぐに処理を戻す. 読み込むデータが存在しなかった場合は, 呼び出し元にその旨通知することで, 呼び出し元は通信の成否にあわせた動作が可能になる.

書き込み時も同様に, バッファが足りずにデータが溢れる場合でも, すぐに処理を戻し, 書き込めなかったことを呼び出し元に通知することで, 通信の成否にあわせた動作が可能になる.

ブロッキング通信

この通信モードは, et スレッドのやユーザプログラムの動作を一時的に停止しても, 確実にデータ通信を行う必要がある場合に用いる非同期通信である.

読み込み時に読み込みデータが存在しない場合は, 呼び出し元に処理を戻さず, 動作を中止して, データが来るまで待つ. データが来て, そのデータを読み込み, 初めて呼び出し元へ処理を戻す.

書き込み時にバッファに空きが足りず, データが溢れる場合は, 読み込み時同様呼び出し元に処理は戻さずにバッファが空くまで動作を停止する. 通信相手がバッファからデータを読み込み, バッファに空きができてデータの書き込みが可能になったならばバッファにデータを書き込み, 呼び出し元に処理を戻す.

ランデブ通信

この通信モードはオリジナル et での通信と同じく,et スレッドとユーザプログラムとで同期を取りながら通信を行いたい場合に用いる.

読み込み時でも書き込み時でも, 通信相手が現れるまで呼び出し元に処理を戻さず, 動作を停止する. もし. 呼び出しの時点で通信相手が待っていた場合は, データ受け渡しを行い, 処理を戻す.

3.4.3 同期

オリジナルの et では, ランデブ通信により et スレッドとユーザプログラムの同期をとっている. しかし, 通信を行わずに同期を取る必要がある場合にはあまり好ましくない. そこで, 本研究ではデータ通信を用いずに同期を取る機能を設計した.

本研究では et スレッドとユーザプログラムは, ランデブによる同期を行う. ランデブ通信ど同様に, 同期相手が来るまでは, 処理を戻さずに待つことで同期をとる.

3.5 マルチスレッド対応設計

本節では, 複数 et スレッドを同時に動作させる為のマルチスレッド対応機構の設計について述べる.

3.5.1 複数 et スレッド実行

オリジナルの et では, 複数の et スレッドをロードすると et スレッドと同数のシステムもロードされる事になり, システム部分が競合を起こす. そのため, 複数の et スレッドを同時に動作させることができない.

そこで, 複数の et スレッドを同時に動作可能とするために, マルチスレッド対応の設計を行った. システムの一元管理のためにコアモジュールとスレッドモジュールを分離したことで, 複数のスレッドモジュールをロードしても, システム部分は 1 つで済む. そのため, システムが競合することが無くなり, 複数の et スレッドを同時にロードすることが可能になる.

複数 et スレッドを同時に実行する場合でも, 実行すべき et スレッドを特定でき

るようにするため, スレッドモジュールが実行すべき et スレッドの登録を行う (図 3.5).

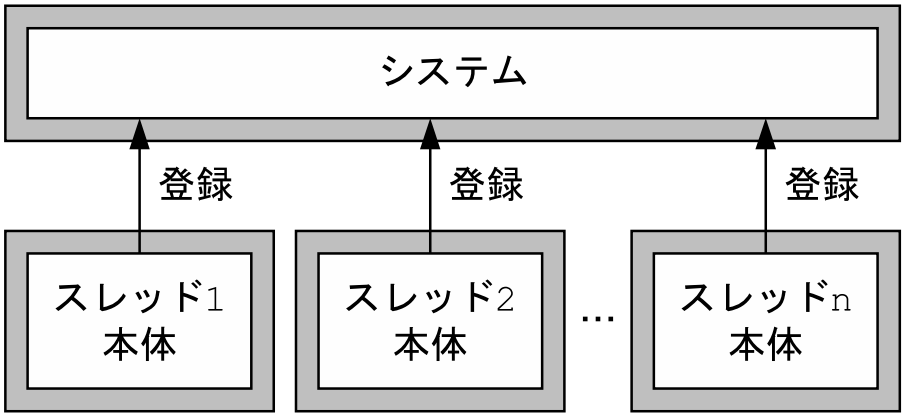

図 3.5: eET での複数 et スレッド動作時

登録はスレッドモジュール単位で行うのではなく et スレッド単位で行うため, 1 つのスレッドモジュール内に複数の et スレッドを持たせる事も可能である (図 3.6).

登録の段階で, コアシステムは et スレッドを一意に識別するためのスレッド ID を発行する. et スレッドやユーザプログラムでは, この ID を用いて CPU 利用時間の設定変更などを行う.

3.5.2 スケジューリング

複数の et スレッドが独自の実行周期で動作するため, 複数の et スレッドが要求する CPU 利用時間が競合する可能性がある. そこで, et スレッド間での CPU 利用時間のスケジューリングが必要になる.

ス eET におけるスケジューリングは, OS でのプロセススケジューリングのように, 次に実行する et スレッドを決定するのではない. CPU 利用時間の競合時に, どのように CPU 時間を割り当てるかを決定することである.

どのように et スレッドに CPU 時間を割り当てるか, et スレッドに優先度のパラ

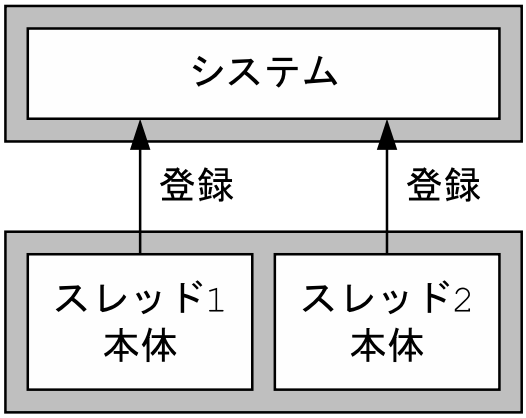

図 3.6: 複数の et スレッドを持つモジュール

メータを持たせ, その優先度によって実行する et スレッドを決定する, 優先度ベースのスケジューリングを用いる. システムでは優先度の高い et スレッドの方が優先度の低い et スレッドに比べて, 指定時刻からのズレが少なく動作させる. 優先度は, et スレッド作成時にスレッドモジュールで指定し, パラメータとしてシステムに通知する.

同じ優先度を持つ et スレッドの CPU 利用時間が競合する場合は, 利用開始時刻が早い et スレッドを優先する. さらに利用開始時刻も同じ場合はスレッド ID の小さい方を優先する.

複数の優先度の異なる et スレッドで CPU 利用時間が競合する場合, 優先度の高い et スレッドの要求を優先するように動作させる. 優先度の高い et スレッドの実行を優先するため, 優先度の低い et スレッドは優先度の高い et スレッドが CPU 利用を終了するまで実行せずに待つ. 優先度の低い et スレッドの実行時刻をずらした場合の, 次の実行時刻は以下の 2 種から選択可能とする

- スケジューリングにより変動した実行時刻のズレを無視する (図 3.7)
- スケジューリングにより変動した実行時刻のズレを考慮する (図 3.8)

次回実行時の設定は, et スレッド登録時のパラメータによって指定する.

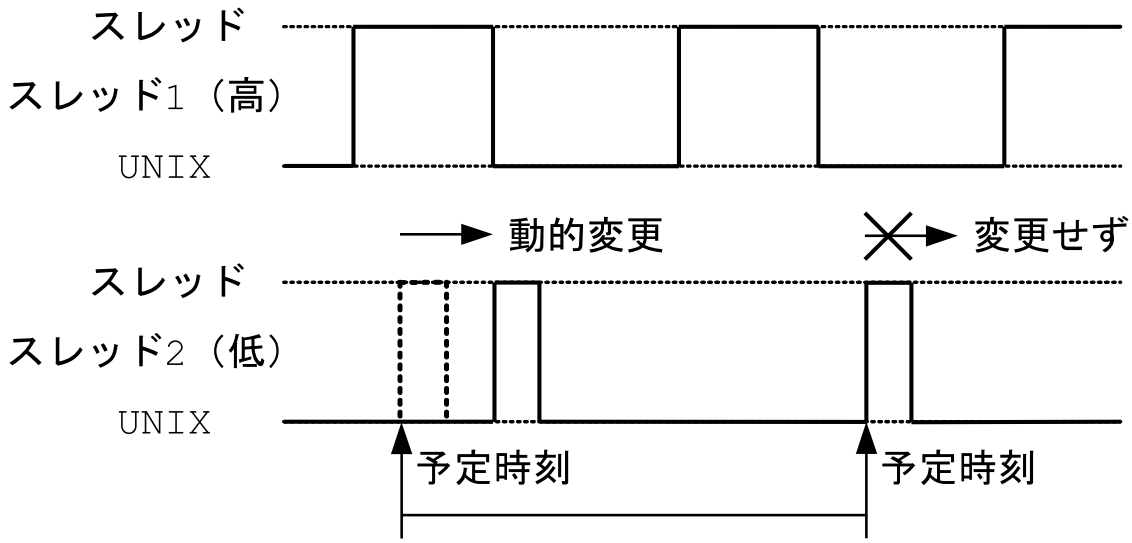

図 3.7: 次回実行時を予定通りに行う場合

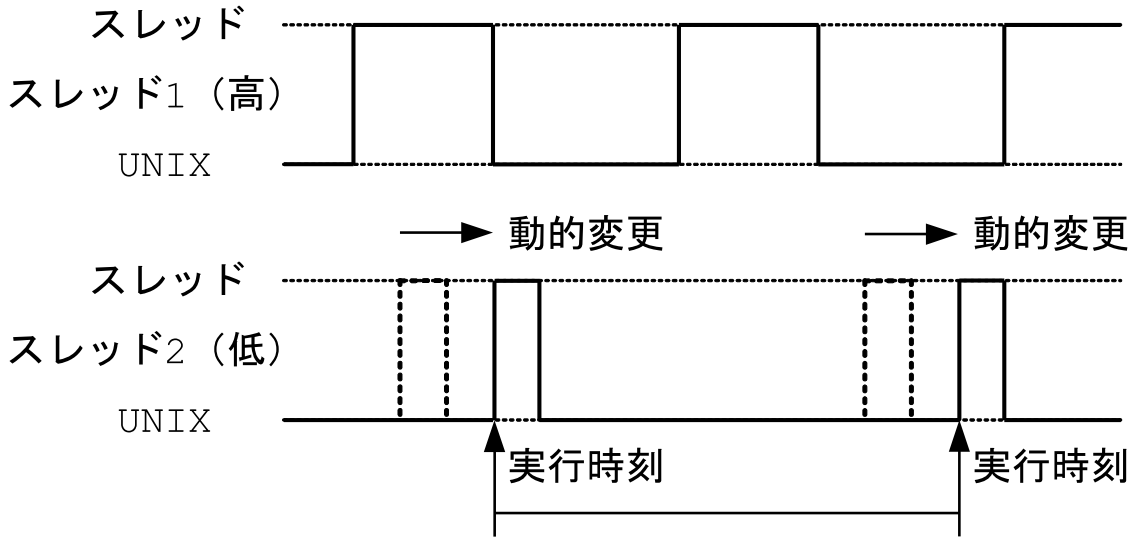

図 3.8: 次回実行時を, スケジューリングにあわせて変更する場合

3.5.3 et スレッド間通信

全ての et スレッドは同じカーネルメモリ上で動作するため, 各 et スレッドでエクスポートされた変数や関数に相互にアクセスすることが可能である. そのため, et スレッド間通信, et スレッド間同期の機能は eET のシステムでは提供せず, et スレッドプログラミングによって解決する.

et スレッド間での排他制御については, CPU 利用時間の延長に用いる切り替えロック, アンロック機構を利用する. また, et スレッドはカーネルメモリ上で動作するため, カーネル関数を利用可能である. そのため, カーネル関数による *spinlock* などの排他制御も利用可能である.

第 4 章

eETの実装

本章では, オリジナルの et へ機能追加を行った eET の実装について述べる.

本研究では, 組み込み系でよく利用されている, Linux 2.4 系列上で実装を行った. Linux 2.4 系列には以下のような特長がある.

- 対応している CPU アーキテクチャが豊富
- 対応ハードウェアが豊富
- オープンソースなので内部の動作に関する資料が豊富
- オープンソースなのでカーネル内部に手を加えることができる

4.1 Linuxへの実装

今回の実装では, Linux 2.4 系列の 1 つである, Linux 2.4.18 (i386) 上に実装を行った. まずは, オリジナルの et を機能拡張を行わずに実装した. オリジナルの et と同様の実装では実現できないため, いくつかの点について変更が必要だった. 本節では, その変更について述べる. この変更は, システムの実装上の違いであり, 他の OS へ実装した場合も同様の変更が必要になる可能性がある. しかし, 実装上の変更であり, et スレッドとのインタフェイスや動作などの eET の仕様への変更ではない.

4.1.1 切り替えトリガ

et スレッドから unix への切り替え

オリジナルの et では, 切り替えのトリガにカーネルのタイマ関数を用いている. Linux 2.4 にもタイマ関数は存在するが, タイムアウトルーチンの処理が遅延処理として実装されており, カーネルの動作が一段落した時点で動作すべき全てのタイムアウト処理を一括して処理している [5][7]. et スレッド動作中にカーネルは動作しないため, 遅延処理も動作せず, タイムアウト処理が実行されない. そのため, カーネルのタイマ関数を et スレッドから unix への切り替えトリガとして用いる事ができない. そこで, 今回の実装では, et スレッドから unix への切り替えトリガとして, カーネルのタイマ関数は用いずに独自にタイマ関数を実装して用いた.

このタイマ関数は CPU 外部のタイマデバイスによるタイマ割り込み (i386 系では IRQ0) に対応する割り込みルーチンとして実装した. カーネルが本来利用している割り込みルーチンの処理後に動作するように実装してあるため, このタイムアウト処理から et スレッドへ処理を移行してもカーネルの動作に影響を及ぼさない.

本来, i386 系の Linux ではタイマ割り込み用の割り込みルーチンは 1 つしか登録できない. そこで, カーネル内で割り込みルーチン管理に用いられているデータテーブル *irq_desc* を直接変更する. タイマ割り込みである IRQ0 には, IRQ 共有を許可するフラグ *SA_SHIRQ* が立っていないが, このフラグを直接立てることで複数個の登録を可能にした. その上で, IRQ 登録関数 *request_irq()* を用いて, 割り込みルーチン *timer_ih_handler()* を登録している (4.1). 今回の実装では, *irq_desc* の名前では直接アクセスできないため, アドレスを直接指定した別名, *irq_desc__*を用いた.

タイマ割り込み間隔を変更することで, et スレッド実行時間の粒度の限界を上げることが可能だが, 切り替え処理も多くなるのでオーバーヘッドも増加する. また, この間隔を変更するにはカーネル再構築を行う事が必要であるため, タイマの割り

```
// 割り込みルーチン管理構造体を取得
struct irqaction *irq0 = irq_desc__[0].action;
// 共有可能フラグを立てる
irq0 -> flags |= SA_SHIRQ;
// 割り込みルーチンを登録
ret = request_irq( 0,\
                   timer_ih_handler,\ // 割り込みルーチンアドレス
                   SA_SHIRQ,\         // 共有可能フラグ
                   "timer_ih",\       // ハンドラ名
                   timer_ih_handler); // デバイス識別子
```

図 4.1: タイマ割り込みルーチン登録のソースコード

込み間隔の変更は行わない．このタイマの割り込み間隔は Tick といい, Linux 2.4 標準の 10m に 1 回の間隔であり, オリジナルの et と同じである．また, オリジナルの et と同じようにカーネル再構築により間隔を変更することが可能である．

割り込みルーチンから et スレッドへと処理を移行するが, このとき割り込みルーチンの処理中であるため, 次のタイマ割り込み受け付けない．そこで, 次の割り込みを受け付けることができように, 割り込みマスクの変更を行う．

unix から et スレッドへの切り替え

unix から et スレッドへの切り替えは, カーネル資源の排他処理を不要にするためにカーネルの処理が一段落した段階での切り替えを行う．そのため, 切り替えトリガにはカーネルのタイマ関数を用いる．

4.1.2 大域ジャンプ

オリジナルの et では, et スレッドと unix の切り替えに大域ジャンプである *setjmp()*, *longjmp()* を利用している． しかし, Linux 2.4 にはカーネル関数として大域ジャンプ, *setjmp()*, *longjmp()* が存在しない．glibc には存在するが, システ

ムや et スレッドはカーネルモジュールとして実装するため利用できない. そこで, *setjmp()*, *longjmp()* の実装を行った (図 4.2). *setjmp()* では, 呼び出し時のスタックフレームを保持し, *longjmp()* では, 呼び出し時に保持したスタックフレームの復帰を行う. この *setjmp()*, *longjmp()* を用いて unix から et スレッドへの切り替えを行う.

```
setjmp:
  movl    4(%esp),%eax
  movl    %ebx,(%eax)             /* save ebx */
  movl    %esp,4(%eax)            /* save esp */
  movl    %ebp,8(%eax)            /* save ebp */
  movl    %esi,12(%eax)           /* save esi */
  movl    %edi,16(%eax)           /* save edi */
  movl    (%esp),%edx             /* get rta */
  movl    %edx,20(%eax)           /* save eip */
  xorl    %eax,%eax               /* return(0); */
  ret
longjmp:
  movl    4(%esp),%eax
  movl    (%eax),%ebx             /* restore ebx */
  movl    4(%eax),%esp            /* restore esp */
  movl    8(%eax),%ebp            /* restore ebp */
  movl    12(%eax),%esi           /* restore esi */
  movl    16(%eax),%edi           /* restore edi */
  movl    20(%eax),%edx           /* get rta */
  movl    %edx,(%esp)             /* put in return frame */
  xorl    %eax,%eax               /* return(1); */
  incl    %eax
  ret
```

図 4.2: *setjmp()*, *longjmp()* のソースコード

4.2 コアシステムと et スレッドの分離

オリジナルの et では, システム部分が全てのスレッドモジュールごとに実装されている. そこで, システムの一元管理を可能にするため, またマルチスレッドに対応するため, システム部分と et スレッド部分を分離し, 別モジュールとする. そ

れぞれをカーネルローダブルモジュールとして分離し, それぞれを “コアモジュール”, “スレッドモジュール” と呼称する. eET を利用する時は, あらかじめコアモジュールをロードしておき, その後にスレッドモジュールをロードする.

しかし, コアモジュールとスレッドモジュールを分離しただけでは, システムが実行すべき et スレッドを特定できないので, 実行すべき et スレッドの管理が必要になる. そこで, コアモジュールは表 4.1 に示す et スレッド管理に用いる関数を提供する. スレッドモジュールはこの管理用関数を用いて実行すべき et スレッドの登録, 削除を行う. スレッド ID に関しては 4.5 節で詳解する.

表 4.1: et スレッド管理用関数

名前	説明
add_et_thread	指定されたパラメータの et スレッドを新たに作成し, その et スレッドの ID を返す
del_et_thread	指定された ID を持つ et スレッドを削除する

スレッドモジュールは, ロード時にコアモジュールの提供する et スレッド登録関数 *add_et_thread()* を呼び出して登録を行う. このとき, 引数として et スレッドの実行パラメータを指定した et スレッド管理構造体 *et_thread_t* (表 4.2) を渡す. この構造体はコアシステム内で et スレッド管理にも用いているものなので, 表 4.2 以外に et スレッド生成時には必要ないメンバも存在するが, et スレッド生成時には無視する. *add_et_thread()* では, et スレッド用スタックメモリの確保, et スレッド用初期スタックフレームの生成, タイマ関数への実行登録を行い et スレッドを生成する.

不用意な et スレッドの終了を避けるため, et スレッド登録時にコアモジュールの利用カウントを増加させ, et スレッドが登録されている状態ではコアモジュールの取り外しはできないようにした.

表 4.2: et スレッド登録時に使用する, et スレッド管理構造体 *et_thread_t* のメンバ

型	名前	説明
void(*)(void)	*et_main*	et スレッドのメイン処理を行う関数へのポインタ
int(*)(void)	*et_init*	et スレッドの初期処理を行う関数へのポインタ
void(*)(void)	*et_exit*	et スレッドの終了処理を行う関数へのポインタ
unsigned int	*et_slice*	et スレッドの CPU 利用時間 (Tick で指定)
unsigned int	*unix_slice*	unix の CPU 利用時間 (Tick で指定)
unsigned int	*stack_size*	et スレッドのスタックサイズ
unsigned int	*mode*	et スレッドの動作パラメータ
unsigned int	*priority*	et スレッドの優先度
char*	*thread_name*	et スレッドの名前

4.3 実行管理

4.3.1 et スレッドと unix の時間配分の動的変更

et スレッドの実行粒度や, 1 回の実行での処理量を動的に調整可能にするため, et スレッドと unix それぞれが利用する CPU 利用時間を動的に変更可能にした.

et スレッドの動作コントロール機構を et スレッド, ユーザプログラムのどちらにでも持たせる事が可能なように, et スレッドとユーザプログラムの両方に動的変更のインタフェイスを実装した.

et スレッドからの設定変更

et スレッドからの設定変更方法はコアシステム内の関数を呼び出す形で行う. 変更, 現在値取得のための関数は表 4.3 の通りである.

et スレッドの CPU 利用時間の変更を行った場合, 変更後の値での動作は次の et スレッド実行からである. ただし, 現在値の取得を行った場合は変更後の値が取得できる.

表 4.3: et, unix の時間配分動的変更, 現在値取得関数

名前	説明
get_et_slice	指定 et スレッドの CPU 利用時間を Tick 単位で取得する
get_unix_slice	指定 et スレッドに対する unix の CPU 利用時間を Tick 単位で取得する
set_et_slice	指定 et スレッドの CPU 利用時間を Tick 単位で設定する
set_unix_slice	指定 et スレッドに対する unix の CPU 利用時間を Tick 単位で設定する

ユーザプログラムからの設定変更

ユーザプログラムからの設定変更方法としては, デバイスの設定変更に用いられる標準のシステムコールである *ioctl* を用いる. *ioctl* はデバイスファイルを経由して処理を行うため, デバイスドライバを作成する. 今回の実装では, et スレッドとユーザプログラムの通信を行う為のデバイスドライバにおいて一元的に管理されている. なお, デバイスファイル作成に関する詳細は 4.4 節に記述する. 今回実装した *ioctl* のコマンドは表 4.4 の通りである.

et スレッドに対する CPU 時間の変更を行った場合は, 変更後の値での動作は次に et スレッドが実行された後に, unix へ処理が戻ってからである. ただし, 現在値の取得を行った場合は変更後の値が取得できる.

表 4.4: et スレッド, unix の時間配分動的変更, 現在値取得 *ioctl* コマンド

名前	説明
ET_IOCTL_GET_ETSLICE	指定 et スレッドの CPU 利用時間を Tick 単位で取得する
ET_IOCTL_GET_UNIXSLICE	指定 et スレッドに対する unix の CPU 利用時間を Tick 単位で取得する
ET_IOCTL_SET_ETSLICE	指定 et スレッドの CPU 利用時間を Tick 単位で設定する
ET_IOCTL_SET_UNIXSLICE	指定 et スレッドに対する unix の CPU 利用時間を Tick 単位で設定する

4.3.2 et スレッドの CPU 利用の放棄と延長

et スレッド実行中に CPU を利用する必要が無くなった時は, 無駄な CPU の占有を避け, unix の処理を実行させるために CPU 利用の放棄を行う.

また, 時間的に不可分な処理を行うときや, カーネル資源や et スレッド間の共有資源などへのアクセスでの排他制御が必要な時は, unix や他の et スレッドへ処理が移る事を防ぐ必要がある. そこで, 現在実行中の et スレッドから, unix や他の et スレッドへ処理の切り替えが発生しないようにするための, ロックによる CPU 利用の延長機能を実装した. CPU 利用の放棄と延長のために実装した関数の一覧を表 4.5 に示す.

表 4.5: et スレッドの CPU 利用時間の放棄, 延長用関数一覧

名前	説明
et_yield	CPU 利用を放棄する
et_lock	unix への切り替えが起こらないようにロックする
et_unlock	unix への切り替えが起こるようにロックを解除する

CPU 利用放棄

CPU 利用の放棄はオリジナルの et にも実装されている. しかし, オリジナルの et では, 次の et スレッドの実行時刻は CPU の利用を放棄した時刻を基準として計算されるため, et スレッドの起動間隔を一定にしたい場合には利用できない. そこで, eET ではこの CPU 利用の放棄を拡張し, 次に et スレッドが実行されるる時刻の選択を行えるようにした.

CPU 利用放棄を行った次の et スレッド実行時刻は, et スレッド作成時のパラメータによって, 次の 2 種から決定される.

- オリジナルの et と同様に, unix の CPU 利用時間が一定 (図 3.2)
- et スレッドの実行間隔が一定 (図 3.3)

次の et スレッド起動時刻を $T_{(n+1)}$, 今回の et スレッド起動時刻を $T_{(n)}$ 現在時刻を $T_{(cur)}$, et スレッドの CPU 利用時間を $t_{(thread)}$, unix の CPU 利用時間を $t_{(unix)}$ とした場合, unix の CPU 利用時間が一定の場合は, 次の et スレッド実行時刻は,

$$T_{(n+1)} = T_{(cur)} + t_{(unix)}$$

となる. et スレッドの実行間隔が一定の場合は,

$$T_{(n+1)} = T_{(n)} + t_{(thread)} + t_{(unix)}$$

となる. 次の et スレッド実行時刻を決定するのに, 今回の et スレッド実行時刻が必要になるため, et スレッド実行再開時に記録しておく.

CPU 利用の放棄はコアシステム内の関数 *et_yield()* として実装した. *et_yield()* 内部で unix へと処理を移行するため, 次に et スレッドに処理が移行してきた時は *et_yield()* の残りの処理をし, et スレッドへ戻る.

CPU 利用延長

オリジナルの et にも, CPU 利用の延長を行う機能は実装されているが, CPU 利用放棄と同様に, 延長が起こった場合, 次に et スレッドが実行する時刻の指定ができない. 今回の実装では, CPU 利用放棄と同じく, スレッドパラメータにより

- unix の CPU 利用時間が一定
- et スレッドの実行間隔が一定

の 2 種類から選択可能にした.

CPU 利用時間の延長は, 時間的に不可分な処理の実行や, カーネル資源や et スレッド間共有資源などへのアクセスの排他制御に用いられるため, 延長する時間や unix へ処理を移行する時刻を指定するのではなく,

1. unix へ処理が移行しないようにロックする
2. クリティカルセクションの処理を行う
3. unix へ処理移行が可能なようにロックを解除する

という手順で利用する.

ロックはタイマ関数に登録されているタイムアウトルーチンを一時的に登録解除することで unix への切り替えが発生しないようにした. ロック解除は, 一時的に登録解除したタイムアウトルーチンを再度登録し直すことで unix への切り替えが発生するようにした.

ロックを解除した時点で, unix への切り替え予定時刻を過ぎていた場合, *et_yield()* を呼び出し, 即座に unix へ処理を戻す. このとき, *et_yield()* では前項の通り, et スレッドのパラメータに従って, unix の CPU 利用時間が一定になるか, et スレッドの実行間隔が一定になるように, et スレッドの次回実行時刻を設定する.

4.3.3 特定イベントに対する処理

終了処理

オリジナルの et では, モジュールの取り外しなどの外的要因による終了を et スレッドで認識することができず, 正常な終了処理を行う事ができない. ローダブルモジュールの機能としてモジュール取り外しに対応した処理を行う事は可能だが, et スレッドの一部としてプログラミングするのではなく, システム部分への変更を必要とする.

そこで, et スレッドプログラミングで対応ができるようにするため, et スレッドのメイン処理 *et_main()* から終了処理を関数 *et_exit()* として分離しする. 分離した *et_exit()* は, et スレッド登録時に終了処理を行う関数として指定する. et スレッドが外的要因により終了するときに, 登録された関数をシステムが実行する.

et スレッドが *to_exit()* を用いて明示的に終了する場合でも, 終了処理の関数は呼び出される.

初期処理

プログラミングでは, デバイスオープンとデバイスクローズのように, 終了処理と対となった処理を初期処理で行う事が多い. そこで, et スレッドプログラムの見通しをよくするため, 初期処理も関数 *et_init()* としてメインの処理から分離する. この分離により, メイン処理の関数 *et_main()* では, 周期実行される処理のみを記述するだけで済み, プログラムの可読性が増す. この関数も終了処理の関数と同様に, et スレッド登録時に初期処理の関数として指定する.

この初期処理の関数は, et スレッド登録の要求があった時点でシステムにより実行される. 初期処理が失敗した場合は, エラーを返しメインの処理を実行しないことが一般的であるため, 初期処理の関数がエラー値を返した場合は, 初期処理の失敗として et スレッド登録が失敗するようにした.

4.4 通信機能

本節では, et スレッドとユーザプログラム間のデータ通信機能の実装について述べる.

4.4.1 通信方法

ユーザプログラムから簡単に通信ができるように, また豊富にあるソフトウェア資産を有効活用できるように, ユーザプログラムからは標準のシステムコール (表 4.6) を用いて通信を行う. そのため, 通信はデバイスファイル, 今回の実装ではキャラクタデバイスを経由して通信を行う.

et スレッドからの通信はコアモジュール内の関数を呼び出して行う. et スレッド

表 4.6: ユーザプログラムの et スレッドとの通信インタフェイス

型	名前	引数	説明
int	*open*	**char***, int	通信経路を初期化し, 利用可能にする
int	*close*	int	通信経路を閉じる
ssize_t	*read*	int, void*, size_t	通信経路からデータを読む
ssize_t	*write*	int, void*, size_t	通信経路へデータを書き込む

プログラミングをしやすくするため, 関数の名前, インタフェイスは, ユーザプログラムでの通信インタフェイスに似せて実装した (表 4.7). インタフェイスに関しては以下の 2 点がユーザプログラムでの通信インタフェイスと異なる.

- *open_chrdev()* では, ファイル名を指定する必要は無く, 替わりにデータ通信に用いるバッファサイズを指定する.
- *open_chrdev()* で指定する通信モードは, eET 専用の通信モードを用いる.

バッファは読み込み用と書き込み用の 2 つが確保されるため, 使用メモリは指定したバッファサイズの 2 倍になる.

メジャー番号とマイナ番号

eET での et スレッドとユーザプログラムとの通信では, デバイスファイルであるキャラクタデバイスを用いて通信を行う. unix システムのデバイスファイルはどのドライバと通信するかを決定するための, “メジャー番号” という番号を持っている. この番号は OS 全体を通して一意に決める必要があり, 他のドライバと競合しない番号を選ぶ必要がある. そのため, システム構築の段階で固定的に決めておく事が難しい. そこで, 自動設定を用いて使用時に空いてる番号を利用する.

表 4.7: et スレッドのユーザプログラムとの通信インタフェイス

型	名前	引数	説明
int	*open_chrdev*	**unsigned int**, int	通信経路の初期化を行い, 利用可能にする
int	*close_chrdev*	int	通信経路を閉じる
ssize_t	*read_chrdev*	int, void*, size_t	通信経路からデータを読む
ssize_t	*write_chrdev*	int, void*, size_t	通信経路へデータを書き込む

メジャー番号はコアモジュールロード時にカーネル関数を用いて使用可能な番号を自動で 1 つ選ぶ. マイナ番号は, et スレッドで *open_chrdev()* を実行した時に利用可能な番号を昇順に選ぶ. ただし, マイナ番号 0 はシステムへの *ioctl* で固定的に使用する.

et スレッド側で *open_chrdev()* を実行しない限りマイナ番号が割り振られないため, 利用するデバイスファイルが確定しない. そのため, 全ての通信は et スレッド側で初期化されたあとにユーザプログラムで初期化を行う必要がある.

メジャー番号はコアモジュールをロードするたびに, マイナ番号は et スレッドから通信路の初期化を行うたびに変化する. そのため, デバイスファイルと通信路の対応を固定的に扱うことができない. そこで, カーネルからの情報提供手段である *proc* ファイルシステムを用いて対応付けの情報を提供する. 今回の実装では, */proc/et_core* から図 4.3 のように読み出せる (この例は複数スレッド対応時のものである). この対応情報は, 左から順にスレッド ID, スレッド名, 利用しているデバイスのマイナ番号全てを示す. 先頭行のスレッド ID が “S” はシステムを差し, 名前でシステムバージョンを示す, そしてマイナ番号の変わりにメジャー番号を示す. 2 行目移行はスレッドを指しす, スレッドが登録されていなければスレッド名

は “<none>” となる. キャラクタデバイスを利用していない場合は, マイナ番号は表示されない.

```
S eET-0.0.1 254
0 test-module 1 2
1 dummy
2 hoge-mod 3
3 <none>
```

図 4.3: */proc/et_core* からの読み出し例

4.4.2 通信モード

オリジナルの et では通信モードはランデブによる同期通信 1 種類しかなかったが, eET では 3 種類に拡張した. 本節では拡張した通信モードの実装について説明する.

通信モードは, *open_chrdev()* 実行時の引数で指定する. そして, この通信モードは et スレッドとユーザプログラムからの両方で利用される. そのため, ユーザプログラムの *open()* で指定される *O_NONBLOCK* などの通信モードオプションは無視される.

全ての通信において, 通信用バッファを用いる. そのため, *open_chrdev()* 実行時に必要な分のバッファサイズを指定する必要がある.

ノンブロッキング通信

ノンブロッキング通信は非同期通信であり, 通信を行った場合は通信の成功失敗にかかわらずにすぐに処理を戻す. ログ出力などの, 通信が et スレッドやユーザプログラムの動作に影響を与えることが望ましくない通信に利用する.

ノンブロッキング通信では, et スレッド, ユーザプログラムの両方において共通した動作になる.

データ読み込み時に読み込むべきデータが存在しない場合, 読み込みの再実行を促すエラーコードである “*–EAGAIN*” を返し, 即座に処理を戻す. 書き込み時に書き込みバッファにデータが入り切らない場合も同様に, “*–EAGAIN*” を返す.

エラーを受け取った呼び出し元では, データ通信の失敗を感知し, それにあわせた動作, 例えば次の機会に読み込みを再挑戦する, 通信を諦めデータを破棄する, などの対応をすることができる.

ブロッキング通信

ブロッキング通信は, バッファからのデータ読み込み, バッファへのデータ書き込みが正常に終了するまで処理を戻さない非同期通信である. 制御用のパラメータなど, 確実性が求められる通信に利用する.

データ読み込み時に読み込むべきデータが存在しない場合, et スレッドでは *et_yield()* を実行して unix へ処理を移行してユーザプログラムからのデータを待つ. このときも, スレッドパラメータにより, 次回実行時刻の設定が行われる. そして, 次の et スレッド実行時にユーザプログラムからデータが送信されていた場合は, 呼び出し元に処理を戻す. もし, 次回実行時においてもデータが存在しない場合は, 再度 *et_yield()* を実行し, データの到着を待つ.

データ書き込みの場合, バッファの空き容量が書き込むべきデータよりも少なく, 全てを書き込む事ができない場合, unix へ処理を移行し, バッファが空くまで待つ. 読み込みのときと同様に, *et_yield()* を実行して処理を unix へ移行する.

ユーザプログラムでの読み書きの場合は, カーネル関数の *wait_event_interruptible()* を用いてプロセスを一時的に停止して et スレッドの通信実行を待つ. et スレッドではデータの読み書きを行ったあと, 通信相手のユーザプログラムが停止していた場合は, カーネル関数 *wake_up_interruptible()* を用いて停止していたプロセスを起こす.

ランデブ通信

ランデブ通信は, et スレッドとユーザプログラムが同期を取りながら通信を行う.

データ読み込み時は, 前述のブロッキング通信と同様の動作を行う. ただし, et スレッドでのデータの読み込み後は, 書き込み側のユーザプログラムがデータの読み終わりを待っているので, *wake_up_interruptible()* を用いてユーザプログラムを起こす.

次に, データ書き込みにおいては, 書き込んだ後に通信相手が読み込みを行うまで待つ必要がある. 待つための原理はブロッキング通信の場合と同じで, et スレッドでは *et_yield()* を呼び出して処理を unix に移行し, ユーザプログラムがデータの読み込みを行うまで待つ. ユーザプログラムでは, *wait_event_interruptible()* を呼び出してプロセスを一時的に停止させる.

4.4.3 同期

et スレッドとユーザプログラムで同期をとって行う処理を実現するために, ランデブによる同期を実装する.

et スレッドからは関数の呼び出しとして実装し, 相手のユーザプロセスとの同期を行う. 関数の動作はランデブ通信と同様に, ユーザプログラムから同期の合図が来るまで *et_yield()* を用いて動作を停止する. すでに同期相手のユーザプログラムが待っている場合は, *wake_up_interruptible()* を用いてユーザプログラムを起こしてから処理を戻す.

ユーザプログラムからは *ioctl* の一種として実装する. 動作は, ランデブ通信の場合と同様に et スレッドが来るまでの間 *wait_event_interruptible()* を用いてプロセスを停止する. 停止している et スレッドがある場合は起こしてから処理を戻す.

ioctl を利用する場合, デバイスファイルを利用する必要がある. しかし, 通信にキャラクタデバイスを利用しない et スレッドの場合, et スレッド専用のキャラク

タデバイスを持たない. そこで, et スレッドとユーザプログラムの同期では, et スレッドの CPU 時間変更などに利用されるシステム用のキャラクタデバイス (マイナ番号 0) を用い, *ioctl* で利用する.

4.5 マルチスレッド対応

本項では, 複数の et スレッドを同時に実行するためのマルチスレッド対応機構の実装について説明する.

4.5.1 複数 et スレッド実行

複数 et スレッドを実行するために, まずは *add_et_thread()* で et スレッドを複数登録できるようにする.

まず, システムで扱う事のできる et スレッドの最大数を規定する. この値を用いてシステム内に et スレッド管理構造体 *et_thread_t* を必要分確保し, この構造体を用いて et スレッドを管理する. et スレッド数が最大値に達している場合に *add_et_thread()* を実行しても, エラーを返し登録は行わない.

この登録時に et スレッドを一意に識別するためのスレッド ID を返す. et スレッド, ユーザプログラムではこの ID を用いて et スレッドを特定して CPU 利用時間の動的な設定変更や, ランデブによる同期を行う.

4.5.2 スケジューリング

今回の実装においては, 複数 et スレッドの CPU 利用時間が競合した場合のスケジューリングにつては実装していない. そのため, システムは全ての et スレッドを実行予定時刻に実行しようとし, 複数 et スレッドの動作が細切れになりる.

しかし, CPU 利用時間に競合が発生しない起動周期を持つ et スレッド同士であれば同時に動作させることは可能である.

第 5 章

評価

本章では, 今回実装したシステムについての評価を行う. 行った評価は大別して 3 つある. 1 つ目は, 新たに追加実装した機能について行う評価である. 2 つ目は, et スレッドが他のアプリケーションへ与える影響を測定する評価である. 3 つ目は, 新たに追加実装した機能によるオーバーヘッドの評価である.

5.1 追加機能の評価

本節では, 本研究において実装された追加機能について, 以下の点について, 実際に eET を利用してシステムを構築する場合を想定して評価を行う.

- オリジナルの et に比べて適用範囲が拡大できたか
- オリジナルの et に比べて導入や利用が簡単になったか

5.1.1 基本機能

システムを構成するモジュールをコアモジュールとスレッドモジュールに分離した. この分離により, コアシステムの一元管理が可能となり, システム全体のメンテナンス性が向上した. この点から, システムが利用しやすくなったと言える.

また, 全てのシステムコードはローダブルモジュールのみで実現し, カーネルへの変更は行っていないため, 導入にあたり必要なことはモジュールのコンパイルの

みである. そのため, カーネルやドライバソースの変更や, 再構築を必要とせず, システムの導入がしやすくなったと言える.

今回の実装ではカーネルの割り込みルーチン管理テーブルである *irq_desc* へアクセスするために, カーネルコンパイル時に生成されるカーネルシンボル一覧である "*System.map*" というファイルが必要である. そのため, このファイルが存在しない場合はカーネルの再構築が必要になってしまう. しかし, この問題点は設計上のものではなく, 実装上の問題であるため, 実装により解決可能である.

5.1.2 実行管理機能

et スレッドと unix の CPU 利用時間の動的変更

オリジナルの et では, et スレッドの CPU 利用時間と unix の CPU 利用時間が固定である. そのため, et スレッドは常に固定間隔で実行される.

eET では, et スレッドの CPU 利用時間と unix の CPU 利用時間を動的に変更できる.

例えば, ロボットなどで移動速度にあわせて制御間隔を変更し, ゆっくり動いている時はあまり無駄に CPU 時間を消費しないようにし, 高速移動時は短い間隔で処理を行う. というような利用が可能になった. そのため, オリジナルの et に比べて適用可能範囲が拡大できたと言える.

CPU 利用時間の放棄と延長

CPU 利用時間の放棄と延長はオリジナルの et にも実装されている. しかし, オリジナルの et では放棄や延長を行った次のスレッド実行時刻が unix への切り替え時刻を基準に計算される. そのため, et スレッドを一定周期で実行するには, CPU 利用の放棄や延長が利用できない.

eET では, et スレッドは CPU 利用放棄や延長を行う場合でも, et スレッド実行間隔を一定にするか, unix 時間を一定にするかを決められる. そのため, CPU 利用

の放棄や延長を行う場合でも, et スレッドの実行間隔を一定にすることができる.

この追加機能により, ロボット制御や, カーネルパラメータ監視, などの一定間隔で処理が必要なプログラムにおいても, CPU 利用の放棄や延長が可能になり, 効率的なプログラムが可能になる.

終了処理と初期処理の分離

オリジナルの et では, モジュール取り外しに対応するにはシステムに変更を加える必要があった (図 5.1). このような変更では, システム部のコードの再利用ができず, システムバージョンアップのたびに変更が必要になり et スレッドプログラミングが煩雑になる.

eET では, メイン処理から関数として分離し, システムに登録するだけでよい. そのため, et スレッドプログラミング時に見通しが非常によくなる (図 5.2). この変更により, スレッドプログラミングが簡単になり, eET の利用がしやすくなる.

5.1.3 通信機能

通信方法

オリジナルの et では, 通信方法がキャラクタデバイスが 1 つだけに限定されていた. そのため, 複数のユーザプログラムとの連携が不可能であった.

eET では, 通信に使用するキャラクタデバイスを動的に追加, 削除できるため, 複数のユーザプログラムとの連携が可能になった. 例えば, 制御内容を決定するユーザプログラムから制御コマンドを受け取り処理を行うと同時に, ログ保存用のユーザプログラムにログを送信する. といった et スレッドの構築が可能になる.

そのため, オリジナルの et に比べて適用可能範囲が拡大できたと言える.

```
//モジュールロード時に実行される関数
//変更が入るので再利用不可
et_load(struct lkm_table *lkmtp, int cmd) {
  extern int etio_begin();
  int err;
  //--------------------------------------
  //    ここに初期処理を追加
  //--------------------------------------
  err = etio_begin();
  stack = (unsigned int)kmem_alloc(kmem_map, stack_size);
  sp = (int *)((stack+stack_size) & ~3);
  *--sp = (int)et_exit;
  et_buf[R_ESP] = (int)--sp;
  et_buf[R_EIP] = (int)et_main;
  et_buf[R_EBX] = R_INIT;
  et_buf[R_EBP] = R_INIT;
  et_buf[R_ESI] = R_INIT;
  et_buf[R_EDI] = R_INIT;
  timeout(to_et, (void *)0, (unix_slice<5)?5:unix_slice);
    /* Are 5 ticks enough for setting everything up? */
  uprintf("et installed.\n");
  p_printf("\n");
  return(err);
}
//モジュールアンロード時に実行される関数
//変更が入るので再利用不可
et_unload(struct lkm_table *lkmtp, int cmd) {
  extern int etio_end();
  int err;
  if ((err=etio_end()) == 0) {
    untimeout(to_et, (void *)0);
    untimeout(to_unix, (void *)0);
    //--------------------------------------
    //   ここに終了処理を追加
    //--------------------------------------
    uprintf("et removed.\n");
    p_printf("\n");
    kmem_free(kmem_map, stack, stack_size);
  }
  return(err);
}
```

図 5.1: オリジナルの et での初期処理と終了処理の実現例

```
int et_init(void)
{
  //--------------------------
  //   ここに初期処理を追加
  //--------------------------
}

void et_exit(void)
{
  //--------------------------
  //   ここに終了処理を追加
  //--------------------------
}

//スレッドモジュールロード時に実行される関数
//再利用可能
int init_module(void)
{
  struct et_thread_t thread;

  thread.et_main     = et_main;
  thread.et_init     = et_init;          //初期処理の関数を指定
  thread.et_exit     = et_exit;          //終了処理の関数を指定
  thread.et_slice    = ET_SLICE;
  thread.unix_slice  = UNIX_SLICE;
  thread.stack_size  = THREAD_STACK_SIZE;
  thread.mode        = THREAD_MODE;
  thread.priority    = THREAD_PRIORITY;
  thread.thread_name = THREAD_NAME;

  thread_id = add_et_thread(thread);     //et スレッドを登録
  if( thread_id < 0 ){
    return thread_id;
  }

  return 0;
}
```

図 5.2: eET での初期処理と終了処理の実現例

通信モード

オリジナルの et では, 通信方法として, ランデブによる同期通信 1 種類しか実装されておらず, 1 周期中に 1 回しか通信ができない. そのため, 通常の処理を行いながらデータを待ったり, ログなどの 1 周期中に複数回の通信を行う可能性のある機能を実現できない.

eET では, オリジナルの et にも実装されている, ランデブによる同期通信に加え, 2 種類の非同期通信を実装した. この 2 種類の非同期通信により, オリジナルの et では実現できなかった通信が可能となる. ノンブロッキング通信を用いることで, et スレッドの動作を止めること無くユーザプログラムへのログ送信を行う et スレッドの作成が可能になった. また, 通常の制御を行いながら, ユーザからのパラメータ変更を受け付ける, といった動作も可能になる.

今回, 適用範囲が拡大できたかどうかの確認のために, 実際に通信機能に対する機能追加を用いたサンプルプログラムを作成した. このサンプルプログラムでは, 制御などの周期実行すべき処理を行いながら, 突発的に発生するユーザプログラムからの通信の受信を行う. これは, ロボット制御やカーネルパラメータチューニング, ハードウェアからのデータ取得などを周期的に行う et スレッドにおいて, 制御パラメータの変更などをユーザプログラムからコマンドとして受け取る, という処理を想定したものである. オリジナルの et では, 通信はランデブ通信のみであるため, 他の処理を行いながら突発的に発生する通信を待つ, という動作は不可能である. また, 同時に通信可能なユーザプログラムが 1 つのみであるため, 取得したデータをユーザプログラムに送信しているとコマンドの受信はできない.

この et スレッドでは, ユーザプログラムとのノンブロッキング通信を複数利用することで, 以下の通信動作を同時に行う.

データをユーザプログラムに送信 これは, ハードウェアからの取得データ, 取得したカーネルパラメータなどの送信を想定している. 今回のサンプルプログラムでは, 内部パラメータから生成したデータの送信を行う.

非周期的に発生するユーザプログラムからの通信を受信 これは, 制御パラメータなどの変更を行うユーザプログラムからのコマンドを想定している. 今回のサンプルプログラムでは, 内部パラメータの変更に用いている. 実際に利用はしていないが, 可変長コマンドとした.

同時に複数のユーザプログラムと通信 これは, 通常のデータ通信の他に, コマンド履歴や制御状態などのログ出力を想定している. 今回のサンプルプログラムにおいてもログ出力に用いている.

図 5.3 はサンプルプログラムの *et_main()* 部分である. メインループ内では, コマンド取得, ログ出力, データ生成, データ送信を行っている.

図 5.4 はサンプルプログラムの *et_init()* と *et_exit()* である. 初期処理と終了処理では, 通信路の確保と解放, ハードウェアの初期, 終了処理を行う. このサンプルプログラムでは, ハードウェアの初期, 終了処理は何も行っていない.

図 5.5 は, ユーザプログラムからのコマンド受信用の関数と, ログ出力用の関数である. それぞれ, ノンブロッキング通信を用いているため, 一周期中に何回でも呼び出すことが可能である.

また, この et スレッドと通信するためのユーザプログラムとして, コマンド送信用のプログラムとログ保存用のプログラムを作成した. データ受信については, ログ保存用のプログラムで受信可能であるため, 流用した. それぞれを図 5.6, 5.7 に示す.

以上から, 複数のユーザプログラムと通信を行う et スレッドの作成が可能であることが確認でき, 適用範囲が拡大できたと言える.

同期

オリジナルの et ではランデブによる同期通信を実装しているため, et スレッドとユーザプログラムの間で同期を取ることが可能である. eET では, ランデブによ

```
void
et_main(void)
{
  struct com_t com; // コマンドの構造体（詳細は後述）
  char dat[0x100];
  char str[0x100];

  int command, val;

  int i;

  command = val = 0;

  while(1){           // メインループ
    sprintf(str, "Mark (%ld)\n", jiffies);
    put_log(str);     // 周期ごとにマークをログ出力

    // コマンド取得
    if( get_command( cdev_com, &com, dat) == 0 ){
      sprintf(str, "command recive (%ld): %d(%08X) len=(%d)\n",
              jiffies, com.com, com.val, com.data_length);
      put_log(str);  //前行で生成したコマンド受信履歴をログ出力
      // コマンドの可変長部分（文字列）をログに出力
      if( com.data_length > 0 ){
        sprintf(str, "<%s>\n", dat);
        put_log(str);
      }
    }
    // コマンドから内部パラメータ変更
    command = com.com;
    val     = com.val;

    // 内部パラメータからデータ生成
    // ハードウェア制御，カーネルパラメータ変更などの替わり
    for( i=0; i<val; i++){
      dat[i] = command;
    }
    dat[i] = 0x00;

    write_chrdev(cdev_data, dat, val+1); //データ送信

    et_yield();
  }
  return;
}
```

図 5.3: サンプルプログラムの *et_main()*

```
int
et_init(void)
{
  // 必要な通信経路を確保
  // データ送信用
  if( (cdev_data = open_chrdev(4000, CHRDEV_MODE_NONBLOCK)) < 0){
    return cdev_data;
  }
  // コマンド受信用
  if( (cdev_com  = open_chrdev(4000, CHRDEV_MODE_NONBLOCK)) < 0){
    close_chrdev(cdev_data);
    return cdev_com;
  }
  // ログ送信用
  if( (cdev_log  = open_chrdev(4000, CHRDEV_MODE_NONBLOCK)) < 0){
    close_chrdev(cdev_data);
    close_chrdev(cdev_com);
    return cdev_log;
  }

  put_log("thread initialization.\n");

  // ハードウェアの初期化（今回は何もしていない）
  init_hardware();

  put_log("OK.\n");
  return 0;
}

void
et_exit(void)
{
  // ハードウェアの終了処理（今回は何もしてない）
  cleanup_hardware();

  put_log("thread stop.");

  // 通信路の解放
  close_chrdev(cdev_data);
  close_chrdev(cdev_com);
  close_chrdev(cdev_log);
  return;
}
```

図 5.4: サンプルプログラムの *et_init()*, *et_exit()*

```
// コマンド送受信用構造体
struct com_t{
  int com;         // コマンド
  int val;         // 引数
  int data_length; // 可変長部分のデータサイズ
};

// コマンド受信関数
int
get_command(int cdev, struct com_t *com, char *dat)
{
  int ret;

  // 固定分のコマンドを受信
  ret = read_chrdev( cdev, com, sizeof(struct com_t));
  if( ret <= 0){
    return -1;
  }

  // 固定分のコマンドに含まれる可変長部のサイズを用いて
  // 可変長部分を受信
  if( com->data_length > 0 ){
    read_chrdev( cdev, dat, com->data_length );
  }

  return 0;
}

// ログ出力関数
int
put_log(char *string)
{
  // ログ文字列を送信
  return write_chrdev( cdev_log, string, strlen(string));
}
```

図 5.5: サンプルプログラムのコマンド受信関数とログ出力関数

```
int
main( int argc, char *argv[])
{
  int fd;
  struct com_t com;

  if( argc != 5 ){
    fprintf(stderr, "usage: %s devfile com val dat\n", argv[0]);
    exit(1);
  }

  // デバイスファイルをオープン
  fd = open( argv[1], O_RDWR );
  if( fd < 0 ){
    fprintf(stderr, " can't open %s.\n", argv[1] );
    exit(2);
  }

  // 固定分のコマンドを引数文字列から生成
  com.com         = atoi(argv[2]);
  com.val         = atoi(argv[3]);
  com.data_length = strlen(argv[4]) + 1;

  // 固定分のコマンドを送信
  write(fd, &com, sizeof(struct com_t));

  // 可変長分のコマンドを送信
  write(fd, argv[4], com.data_length);

  fprintf(stderr, "write command.\n");

  close(fd);

  return 0;
}
```

図 5.6: サンプルプログラムのと通信するコマンド送信用ユーザプログラム

```
int
main( int argc, char *argv[])
{
  int fd;
  char dat[0x100];
  int loop=1;
  int ret;

  if( argc != 2 ){
    fprintf(stderr, "usage: %s Devicefile\n", argv[0]);
    exit(1);
  }

  // デバイスファイルをオープン
  fd = open( argv[1], O_RDWR);
  if( fd < 0 ){
    fprintf(stderr, "can't open dev\n");
    exit(1);
  }


  while( loop == 1){
    // ログ受信
    ret = read(fd, dat, 0x100);
    if( ret > 0 ){
      // 書き出し
      write( 1, dat, ret);

    }else if( errno != EAGAIN ){
      // 取得データ量が0でリトライでなければ終了
      loop = 0;
    }
    sleep(1);
  }

  close(fd);

  return 0;
}
```

図 5.7: サンプルプログラムのログを保存するユーザプログラム

る同期通信の他に, 通信を行わないランデブによる同期命令を追加した. 機能的な面から見るとeETの同期機構も同等だが, 専用命令を用いる場合は, 通信用キャラクタデバイスの消費を押さえることができる.

5.1.4 マルチスレッド対応機能

オリジナルのetは同時に1つのetスレッドしか動作させることができない. そのため, 複数の機能を同時に実現することができない

今回の実装では, CPU利用の競合発生時のスケジューリング機構を実装していない. そのため, CPU利用時間に競合を起こさないetスレッド同士に限られるが, 複数のetスレッドを同時に実行することが可能となり, オリジナルのetに比べて適用可能範囲が拡大できたと言える.

また, スケジューリング機構を実装すれば起動周期の競合するetスレッドも同時に動作させることが可能になる.

5.2 他のアプリケーションへの影響

5.2.1 実験環境

実験環境を表5.1に示す. 実験では, 何もしないでCPUを5%占有するように, 図5.8示すetスレッドを表5.2のパラメータで動作させた. このパラメータでは, 切り替え処理などのオーバーヘッドが無ければユーザプログラムに5.0%の負荷がかかる.

全ての実験は5回行い, 平均値を取った.

この実験環境で, 全てのテストを5回行いその平均を元に負荷を計算した.

表 5.1: 実験環境

	スペック
CPU	MMX Pentium 266MHz
Memory	96MB
Kernel	Linux 2.4.18

```
void et_main(void)
{
  while( 1 ){
  }
  return;
}

int et_init(void)
{
  return 0;
}

void et_exit(void)
{
  return;
}
```

図 5.8: 実験用 et スレッドのソースコード

表 5.2: 負荷実験用 et スレッドのパラメータ

パラメータ	値
et_slice	5
unix_slice	100

5.2.2 CPU を多用するプログラム

主に CPU を必要とするプログラムの実行時にかかる負荷を測定する．今回は, CPU を利用するプログラムとして, 変数をカウントアップするプログラムを作成し, 実験を行った.

結果は表 5.3 の通りになった.

表 5.3: CPU 使用時の負荷 (括弧内は標準偏差)

所要時間	Linux 2.4.18 [sec]	linux 2.4.18 + eET [sec]	負荷率
real	29.70 (0.00)	31.20 (0.00)	5.06%
user	29.70 (0.00)	29.70 (0.01)	0.01%
system	0.01 (0.01)	0.00 (0.01)	-66.67%

この結果から, CPU を利用するだけのプログラムでは, ほぼ et スレッドが使用する CPU 時間の分だけ実行に負荷がかかることがわかる．system の負荷率については, 所要時間が小さい値であるため誤差が大きく出ているだけであり, 無視できる範囲である.

5.2.3 ディスクへの書き込みプログラム

主にディスク I/O を利用するプログラムの実行時にかかる負荷を測定するため, ディスクへのデータ書き込みにかかる時間の測定を行う．方法として, */dev/zero* から *dd* コマンドでデータを読み, ディスク上にファイルとしてデータの書き出す. ファイルとして書き出すため, メモリキャッシュにより実際にディスクに書き出す処理は後回しにされる可能性がある．そこで, 即座にディスクへ書き出すために *dd* コマンドの直後に *sync* コマンドを発行した．今回は 1000MB のデータをディスク

に書き出して測定を行った.

結果は表 5.4 のようになった.

表 5.4: ディスクへの書き込み時の負荷 (括弧内は標準偏差)

所要時間	Linux 2.4.18 [sec]	linux 2.4.18 + eET [sec]	負荷率
real	129.72 (0.89)	131.25 (1.59)	1.26%
user	1.41 (0.19)	1.32 (0.17)	-6.93%
system	48.86 (1.53)	42.95 (2.45)	-12.08%

この結果から, ディスク I/O を多用するプログラムでは, システムコードで長時間稼働していることがわかる. このシステムコード内で行われていることは, ディスクの処理待ちであり, CPU を必要とせず, et スレッド実行中でもディスク上での処理は続く. そのため, その間 CPU を et スレッド実行に当てることが可能となり, システムコードでの実行時間が減少する. 全体としては et スレッドが使用している CPU 時間よりも少ない負荷で動作することができる.

ユーザモードでの負荷率は, 母数が小さいため, わずかなズレが大きく現れているためであり, 誤差の範囲内である.

5.2.4 カーネルコンパイル

ディスク I/O と CPU の両方を多く利用するテストとして, Linux カーネルのコンパイルを行う. これも前述のテストと同じく, et を組み込んでいないカーネルと, 何も行わない et スレッドを動作させている環境で比較する.

結果は表 5.5 の通りである.

この結果より, CPU やディスクを取り混ぜて利用するプログラムにおいても, ほぼ理論値通りの負荷ですむ.

表 5.5: カーネルコンパイル時の負荷 (括弧内は標準偏差)

所要時間	Linux 2.4.18 [sec]	linux 2.4.18 + eET [sec]	負荷率
real	1401.87 (13.70)	1479.11 (84.87)	5.51%
user	1319.27 (12.36)	1324.03 (79.21)	0.36%
system	72.30 (0.46)	72.73 (1.29)	0.58%

5.3 機能追加によるオーバーヘッドの増加

ここでは, 機能追加を行ったことによるシステムの処理増加に伴うオーバーヘッドの増加について評価する. 評価は, 他のアプリケーションへの負荷を機能追加を行っていないシステムと機能追加を行ったシステムの両方で測定し, その差を見ることで行う. 実験環境は前項と同じである.

結果は表 5.6 の通りになった. この結果から, 機能追加によるシステムのオーバーヘッドの増加は 0.06%から 0.22%であり, 約 3.44 倍となっているが, 全体で 0.2%程度であり, 非常に小さいと言える.

表 5.6: 機能追加によるオーバーヘッド

	所要時間 [sec]	負荷率
無負荷	29.70	
機能追加なし	31.20	5.06%
機能追加あり	31.25	5.22%

第 6 章

関連研究

RTLinux

RTLinux[8] は, Linux にリアルタイム性を追加するための機能拡張を行ったものである. 割り込みに対する応答のリアルタイム性や, カーネルモードで動作する関数, ユーザモードで動作するプロセスに対してリアルタイム性を追加する.

割り込みに対する応答性は, 優れているが, その代償として, デバイスドライバがオリジナルの Linux と互換性が無い. そのため, 導入が非常に困難である. また, システムの導入にはカーネルへのパッチ, カーネル再構築を必要とする.

ART–Linux

ART–Linux[9] とは, RTLinux 同様, Linux にリアルタイム性を追加するために機能拡張を行ったものである. ART–Linux ではユーザモードプロセスをリアルタイム性を持たせて周期実行する. 導入にはカーネルへのパッチ, 再構築を必要とするが, デバイスドライバはソース互換であるため, 再コンパイルさえ行えばオリジナルの Linux のものがそのまま利用できる.

しかし, ART–Linux で周期実行されるのはユーザモードプロセスであり, カーネルモードで動作している本研究とは異なり, ハードウェアの直接制御や, OS のパラメータ変更などを行う事ができない.

第 7 章

まとめ

7.1 成果

本研究では, オリジナルの et について説明し, 多様な et スレッドをプログラミングする上での問題点を明らかにした. そして, その問題点を解決するための追加機能を持つ “eET” の以下の設計を行った.

- eET の導入, 利用が簡単に行えるようにするための “基本機能設計”
- 実行周期の動的変更, CPU 放棄や延長, モジュール取りはずしなどのイベントに対する処理を実現する為の “実行管理設計”
- ユーザプログラムとの通信を複数種実現するための “通信機能設計”
- 複数 et スレッドを同時に実行可能にする “マルチスレッド対応設計”

そして, 組み込み用途などで利用される事の多い, Linux 2.4 上で以下の項目について実装した.

- “基本機能”
  - オリジナルの et では, コアシステムと et スレッドを 1 つのモジュールに実装していたが, これではシステムのメンテナンス性に欠ける. そこで, コアシステムと et スレッドを, “コアモジュール” と “スレッドモジュール” に分割し, コアモジュールを一元管理可能にした.

- “実行管理”
  - オリジナルの et では不可能であった, et スレッドと unix の CPU 利用時間の動的変更を可能にした.
  - オリジナルの et では固定的であった, CPU 利用の放棄, 延長時の次回 et スレッド実行時刻を変更可能にした.
  - オリジナルの et ではモジュール取り外しなどの外的要因による et スレッド終了に et スレッドで対応できなかったが, 終了処理を分離することで対応可能にした. また, それにあわせて, 初期処理の分離も行い, プログラム全体の見通しがよくなった.
- “通信機能”
  - オリジナルの et にはランデブによる同期通信しか通信モードを持っていなかったが, eET ではランデブによる同期通信も含めて 3 種類の通信モードから選択可能にした.
  - オリジナルの et には通信を伴う同期手段しかないが, eET では同期のみを行う機構を追加した.
- “マルチスレッド対応” の一部
  - オリジナルの et は複数の et スレッドを同時に動作させることができないが, eET では複数の et スレッドを同時に動作させることが可能になった.

そして, これらの追加機能について, 実際の利用状況を想定した評価を行った. また, 実際の利用を想定したサンプルプログラムも作成し確認を行った. その結果, 以下の結果が得られた.

- 機能拡張により本システムの導入や利用が簡単になった
- 機能拡張により本システムの適用範囲が拡大した

さらに, 機能追加によるオーバーヘッドの測定を行い, オーバーヘッドの増加が小さいことを確認した.

7.2 今後の課題

今後の課題として, 今回実装を行わなかった, マルチスレッド時に CPU 利用時間が競合した et スレッド間でのスケジューリングについての実装を行う必要がある.

そのほかに, システムの提供する関数のマニュアルや, et スレッドの雛形プログラム, et スレッドプログラミングを容易にするためのライブラリの開発なども課題として残されている.

今回は, i386 アーキテクチャの一般的な PC 上で実装を行ったが, 組み込み系などでは, Power–PC, ARM, SH, MIPS などの様々なアーキテクチャが利用されている. これらのアーキテクチャへの対応も今後の課題として残されている.

謝辞

本研究を遂行するにあたって, いろいろな方々にお世話になりました.

まず, 指導教員の多田好克先生には日頃から熱心なご指導, そしてご鞭撻を賜わりました. そして, 佐藤喬助手からは研究方針や研究方法に多くのご指導をいただきました. お二人には, ご多忙にもかかわらず論文の草稿を丁寧に読んでくださり, 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.

また, 本研究を行えたことは, 共に研究生活を送り, 様々な議論をすることができた, 多田研並びに村山研の学生諸氏のおかげでもあります. 皆さんに感謝いたします,

最後に, 度重なる Kernel Panic に耐え抜いた計算機や, その他お世話になった備品にも感謝いたします.

参考文献

[1] 多田 好克, 福田 伸彦, 鈴鹿 倫之, 中村 嘉志, “カーネルの外部で走行するカーネルスレッドの提案とその実装法”, 電子情報通信学会論文誌 Vol.J85-D-1 No.3 (2002).

[2] Linux, http://www.kernel.org/

[3] FreeBSD, http://www.freebsd.org/

[4] NetBSD, http://www.netbsd.org/

[5] Daniel P. Bovet, Marco Cesati 著/ 高橋 浩和, 早川 仁 監訳/ 岡島 順治郎, 田宮 まや, 三浦 広志 訳, “詳解 Linux カーネル”, オライリージャパン (2001).

[6] Alessandro Rubini, Jonathan Corbet 著/ 山崎 康宏, 山崎 邦子, 長原 宏治, 長原 陽子 訳, “Linux デバイスドライバ 第 2 版”, オライリージャパン (2003).

[7] 高橋 浩和, “Linux V2.4 カーネル内部解析報告 ドラフト第 4 版”, http://japan.linux.com/kernel/internal24/index.shtml (2000).

[8] RTLinux, http://www.rtlinuxfree.com/

[9] ART–Linux, http://www.movingeye.co.jp/mi6/artlinux_feature.html