RED HAWK Linux® User's Guide

CONCURRENT

0898004-750
March 2015

Copyright 2015 by Concurrent Computer Corporation. All rights reserved.

本書は当社製品を利用する社員、顧客、エンドユーザーを対象とします。
本書に含まれる情報は、本書発行時点での正確な情報ですが、予告なく変更されることがあります。
当社は、明示的、暗示的に関わらず本書に含まれる情報に対して保障できかねます。

誤字・誤記の報告または本書の特定部分への意見は、当該ページをコピーし、コピーに修正またはコメントを記述してコンカレント日本株式会社まで郵送またはメールしてください。

http://www.ccur.co.jp/concurrent_computer_contact_us.aspx

本書はいかなる理由があろうとも当社の許可なく複製・変更することはできません。

Concurrent Computer CorporationおよびそのロゴはConcurrent Computer Corporationの登録商標です。
当社のその他すべての製品名はConcurrentの商標です。また、その他全ての製品名が各々の所有者の商標または登録商標です。
Linux®は、Linux Mark Institute(LMI)のサブライセンスに従い使用しています。

改定履歴:

Date	Level	Effective With
August 2002	000	RedHawk Linux Release 1.1
September 2002	100	RedHawk Linux Release 1.1
December 2002	200	RedHawk Linux Release 1.2
April 2003	300	RedHawk Linux Release 1.3, 1.4
December 2003	400	RedHawk Linux Release 2.0
March 2004	410	RedHawk Linux Release 2.1
July 2004	420	RedHawk Linux Release 2.2
May 2005	430	RedHawk Linux Release 2.3
March 2006	500	RedHawk Linux Release 4.1
May 2006	510	RedHawk Linux Release 4.1
May 2007	520	RedHawk Linux Release 4.2
April 2008	600	RedHawk Linux Release 5.1
June 2008	610	RedHawk Linux Release 5.1
October 2008	620	RedHawk Linux Release 5.2
December 2009	630	RedHawk Linux Release 5.4
May 2011	640	RedHawk Linux Release 6.0
March 2012	650	RedHawk Linux Release 6.0
September 2012	660	RedHawk Linux Release 6.3
February 2013	670	RedHawk Linux Release 6.3
August 2013	680	RedHawk Linux Release 6.3
May 2014	700	RedHawk Linux Release 6.5
August 2014	710	RedHawk Linux Release 6.5
March 2015	750	RedHawk Linux Release 7.0

注意事項:

本書は、Concurrent Computer Corporationより発行された「RedHawk Linux User's Guide」を日本語に翻訳した資料です。英文と表現が異なる文章については英文の内容が優先されます。

前書き

マニュアルの範囲

本書は3つのパートにより構成されます。
本書のPart 1はリアルタイム・ユーザー向け、Part 2はシステム管理者向け、Part 3は附録、用語解説、索引となります。
以下は、本書の内容の概略です。

マニュアルの構成

本書は以下のセクションで構成されます:

Part 1 – リアルタイムユーザー

- 1章:「*序文*」は、RedHawk Linux OSの手引きおよびリアルタイム機能の概要。
- 2章:「*リアルタイム性能*」は、割り込み応答、プロセス・ディスパッチ・レイテンシー(PDL : Process Dispatch Latency)およびデターミニスティック(応答時間が予測可能)なプログラムの実行を含むリアルタイム性能の実現に関する問題の検討。シールドCPUモデルの説明。
- 3章:「*リアルタイム・プロセス間通信*」は、POSIX®とSystem Vのメッセージ送受信、共有メモリ機能の使い方。
- 4章:「*プロセス・スケジューリング*」は、プロセスのスケジューリングの概要とPOSIXスケジューリングのポリシーと優先度の説明。
- 5章:「*プロセス間同期*」は、共有リソースへ同期アクセスする協同プロセス用にRedHawk Linuxより提供されるインターフェースの説明。(POSIXカウンティング･セマフォ、System Vセマフォ、再スケジューリング制御ツール、条件同期ツール)
- 6章:「*プログラム可能なクロックおよびタイマー*」は、RedHawk Linuxで利用可能なRCIMおよびPOSIXのタイミング機能の概要。
- 7章:「*システム・クロックおよびタイマー*」は、システム時間計測とCPU単位のローカルタイマーの説明。
- 8章:「*ファイル・システムとディスクI/O*」は、RedHawk Linux上でのXFSジャーナリング・ファイルシステムおよびダイレクト・ディスクIO 実行手順の説明。
- 9章:「*メモリ・マッピング*」は、プロセスが他のプロセスのアドレス空間へアクセスするためにRedHawk Linuxが提供する方法を説明。
- 10章:「*Non-Uniform Memory Access (NUMA)*」は、特定のシステム上で利用可能なNUMAサポートの説明。

Part 2 – 管理者

- 11章:「*カーネルの構成および構築*」は、RedHawk Linuxカーネルの構成および再構築方法について説明。

- 12章:「*カーネル・デバッギング*」は、**kdump**、**crash**を使ったカーネル・メモリ・イメージの保存、復元、解析のガイドラインおよび**kdb**カーネル・デバッガの基本的な使い方。
- 13章:「*Pluggable Authentication Modules (PAM)*」は、RedHawk LinuxのPAM認証機能の説明。
- 14章:「*デバイス・ドライバ*」は、RedHawkの機能とデバイスドライバの記述に関連したリアルタイムの問題の説明。
- 15章:「*PCI-to-VME サポート*」は、RedHawkがサポートするPCI-VME間ブリッジの説明。

Part 3 - 共通事項

- 付録A:「*メッセージ・キュー・プログラム例*」は、POSIXおよびSystem Vのメッセージキューの機能を解説するサンプルプログラム。
- 付録B:「*リアルタイム機能のためのカーネル・チューニング*」は、RedHawk Linuxのユニークな機能を制御するチューニング・パラメータおよびプレビルド・カーネルのデフォルト値の一覧。
- 付録C:「*ケーパビリティ*」は、RedHawk Linuxに含まれるケーパビリティと各々より提供されるパーミッション(アクセス権限)の一覧。
- 付録D:「*32bit コードから64bit コードへの移植*」は、x86_64プロセッサ上で32bitコードを64bit処理へ移植するための情報。
- 付録E:「*シールドCPU上のカーネル・レベル・デーモン*」は、シールドCPU上でカーネルレベルのデーモンを実行する方法およびパフォーマンスを向上する方法を説明。
- 付録F:「*シールドCPU上のプロセッサ間割り込み*」は、シールドCPU上でプロセッサ間割り込みを実行する方法およびパフォーマンスを向上する方法の説明。
- 付録G:「*シリアル・コンソールの設定*」は、シリアルコンソールを設定するための手順。
- 付録H:「*ブート・コマンド・ライン・パラメータ*」は、RedHawk対応のユニークなブートパラメータを説明。
- 「*用語解説*」は、本書全体で使われる用語を記述。

構文記法

本書を通して使用される表記法は以下のとおりとなります。

斜体　　ユーザーが特定する書類、参照カード、参照項目は、*斜体*にて表記します。
特殊用語も*斜体*にて表記します。

太字　　ユーザー入力は**太字**形式にて表記され、指示されたとおりに入力する必要があります。ディレクトリ名、ファイル名、コマンド、オプション、manページの引用も**太字**形式にて表記します。

`list`　　プロンプト、メッセージ、ファイルやプログラムのリストのようなオペレーティング・システムおよびプログラムの出力は`list`形式にて表記します。

`[]`　　ブラケット(大括弧)はコマンドオプションやオプションの引数を囲みます。もし、これらのオプションまたは引数を入力する場合、ブラケットをタイプする必要はありません。

ハイパーテキスト・リンク
本資料を見ている時に項、図、テーブル・ページ番号照会をクリックすると対応する本文を表示します。青字で提供されるインターネットURLをクリックするとWebブラウザを起動してそのWebサイトを表示します。赤字の出版名称および番号をクリックすると(アクセス可能であれば)対応するPDFのマニュアルを表示します。

関連図書

以下の表にRedHawk Linuxのドキュメントを記載します。
これらのドキュメントは、Concurrent Computer CorporationのWEBサイトにて参照または入手することが可能です。

http://www.ccur.com.
http://redhawk.ccur.com/docs/document.cgi?document=1redhawk

RedHawk Linux Operating System Documentation	Pub. Number
RedHawk Linux Release Notes	0898003
RedHawk Linux User's Guide	0898004
Real-Time Clock & Interrupt Module (RCIM) User's Guide	0898007
RedHawk Linux FAQ	N/A
Optional RedHawk Product Documentation	
RedHawk Linux Frequency-Based Scheduler (FBS) User's Guide	0898005

目次

前書き iii

1章 序文

概要 1-1
RedHawk Linuxカーネル 1-3
システム・アップデート 1-4
リアルタイム機能 1-4
プロセッサ・シールディング 1-4
プロセッサ・アフィニティ 1-4
ユーザー・レベル・プリエンプション制御 1-5
高速ブロック/ウェイク・サービス 1-5
RCIMドライバ 1-5
Frequency-Based Scheduler 1-5
/procの変更 1-6
カーネル・トレース機能 1-6
ptrace拡張 1-6
カーネル・プリエンプション 1-6
リアルタイム・スケジューラ 1-6
低レイテンシー拡張 1-7
優先度継承 1-7
高分解能プロセス・アカウンティング 1-7
ケーパビリティのサポート 1-7
カーネル・デバッガ 1-8
カーネルのコア・ダンプ/クラッシュ解析 1-8
ユーザー・レベル・スピン・ロック 1-8
usermapおよび/proc mmap 1-8
ハイパースレッディング 1-8
XFSジャーナリング・ファイル・システム 1-9
POSIXリアルタイム拡張 1-9
ユーザー優先度スケジューリング 1-9
メモリ常駐プロセス 1-9
メモリ・マッピングおよびデータ共有 1-10
プロセス同期 1-10
非同期入出力 1-10
同期入出力 1-10
リアルタイム・シグナルの挙動 1-11
クロックおよびタイマー 1-11
メッセージ・キュー 1-11

2章 リアルタイム性能

シールドCPUモデルの概要 2-1
デターミニズムの概要 2-2
プロセス・ディスパッチ・レイテンシー 2-2
割り込み禁止の効果 2-4
割り込みの影響 2-5
プリエンプション禁止の効果 2-8

オープン・ソース・デバイス・ドライバの影響 2-9
シールディングでリアルタイム性能を向上する方法 2-9
バックグラウンド・プロセスからのシールディング 2-9
割り込みからのシールディング 2-10
ローカル割り込みからのシールディング 2-11
CPUシールディングのインターフェース 2-12
shieldコマンド 2-12
shieldコマンド例 2-13
終了ステータス 2-13
shieldコマンド拡張機能 2-14
CPUシールディングの/procインターフェース 2-14
CPUへのプロセス割り当て 2-14
mpadviseを使ったマルチプロセッサ制御 2-15
initへのCPUアフィニティ割り当て 2-17
シールドCPUの設定例 2-17
デターミニズムを高める手順 2-20
メモリのページをロック 2-20
プログラム優先度の設定 2-21
遅延割り込み処理の優先度設定 2-21
別プロセスの起床 2-21
キャッシュ・スラッシングの回避 2-22
物理メモリの予約 2-23
NUMAノードへのバインディング 2-27
クアッドOpteronシステムのI/Oスループット 2-27
ハイパースレッディングの理解 2-28
システム構成 2-30
推奨されるCPU構成 2-30
メモリ不足状態の回避 2-34
Linuxのデターミニズムに関する既知の問題 2-34

3章 リアルタイム・プロセス間通信

概要 3-1
POSIXメッセージ・キュー 3-2
System Vメッセージ 3-3
メッセージの利用 3-4
msggetシステムコール 3-7
msgctlシステムコール 3-9
msgsndおよびmsgrcvシステムコール 3-10
メッセージの送信 3-10
メッセージの受信 3-11
POSIX共有メモリ 3-12
shm_openルーチンの利用 3-13
shm_unlinkルーチンの利用 3-15
System V共有メモリ 3-15
共有メモリの利用 3-16
shmgetシステムコール 3-19
shmctlシステムコール 3-21
共有メモリ領域をI/O空間へバインド 3-22
shmgetの利用 3-22
shmbindの利用 3-23
shmatおよびshmdtシステムコール 3-23
共有メモリ領域の結合 3-24
共有メモリ領域の分離 3-24

共有メモリ・ユーティリティ 3-25
shmdefineユーティリティ 3-25
shmconfigコマンド 3-25

4章 プロセス・スケジューリング

概要 4-1
プロセス・スケジューラの管理方法 4-2
スケジューリング・ポリシー 4-3
ファーストイン・ファーストアウト・スケジューリング(SCHED_FIFO) 4-3
ラウンドロビン・スケジューリング(SCHED_RR) 4-4
タイムシェアリング・スケジューリング(SCHED_OTHER) 4-4
バッチ・スケジューリング(SCHED_BATCH) 4-4
性能向上のための手続き 4-4
優先度設定方法 4-4
割り込みルーチン 4-5
SCHED_FIFO vs SCHED_RR 4-5
CPUをロックする固定優先度プロセス 4-6
メモリのロック 4-6
CPUアフィニティとシールド・プロセッサ 4-6
プロセス・スケジューリング・インターフェース 4-6
POSIXスケジューリング・ルーチン 4-6
sched_setschedulerルーチン 4-8
sched_getschedulerルーチン 4-9
sched_setparamルーチン 4-10
sched_getparamルーチン 4-11
sched_yieldルーチン 4-11
sched_get_priority_minルーチン 4-12
sched_get_priority_maxルーチン 4-12
sched_rr_get_intervalルーチン 4-13
runコマンド 4-14

5章 プロセス間同期

プロセス間同期の理解 5-1
再スケジューリング制御 5-3
再スケジューリング変数の理解 5-3
resched_cntlシステムコールの利用 5-4
再スケジューリング制御マクロの利用 5-5
resched_lock 5-5
resched_unlock 5-6
resched_nlocks 5-6
再スケジューリング制御ツールの適用 5-7
ビジーウェイト相互排他 5-7
spin_mutex変数の理解 5-7
spin_mutexインターフェースの利用 5-8
spin_mutexツールの適用 5-9
nopreempt_spin_mutex変数の理解 5-10
nopreempt_spin_mutexインターフェースの利用 5-10
POSIXカウンティング・セマフォ 5-12
概要 5-12
インターフェース 5-13
The sem_initルーチン 5-14
The sem_destroyルーチン 5-15
The sem_openルーチン 5-16

The sem_closeルーチン 5-17
The sem_unlinkルーチン 5-18
The sem_waitルーチン 5-19
The sem_timedwaitルーチン 5-19
The sem_trywaitルーチン 5-20
The sem_postルーチン 5-20
The sem_getvalueルーチン 5-21
POSIXミューテックスへの機能拡張 5-21
ロウバスト・ミューテックス 5-22
優先度継承 5-23
ユーザー・インターフェース 5-23
pthread_mutex_consistent_np 5-24
pthread_mutex_getunlock_np 5-24
pthread_mutex_setconsistency_np 5-24
pthread_mutex_setunlock_np 5-25
pthread_mutexattr_getfast_np 5-25
pthread_mutexattr_getprotocol 5-25
pthread_mutexattr_getrobust_np 5-26
pthread_mutexattr_getunlock_np 5-26
pthread_mutexattr_setfast_np 5-26
pthread_mutexattr_setprotocol 5-27
pthread_mutexattr_setrobust_np 5-27
pthread_mutexattr_setunlock_np 5-27
POSIXミューテックス・プログラムのコンパイル 5-27
System Vセマフォ 5-28
概要 5-28
System Vセマフォの利用 5-29
semgetシステムコール 5-31
semctlシステムコール 5-34
semopシステムコール 5-36
条件同期 5-37
postwaitシステムコール 5-37
serverシステムコール 5-39
server_block 5-39
server_wake1 5-40
server_wakevec 5-41
条件同期ツールの適用 5-42

6章 プログラム可能なクロックおよびタイマー

クロックおよびタイマーの理解 6-1
RCIMクロックおよびタイマー 6-1
POSIXクロックおよびタイマー 6-2
POSIX時間構造体の理解 6-3
POSIX clockルーチンの利用 6-4
clock_settimeルーチンの利用 6-4
clock_gettimeルーチンの利用 6-5
clock_getresルーチンの利用 6-5
POSIX timerルーチンの利用 6-6
timer_createルーチンの利用 6-6
timer_deleteルーチンの利用 6-8
timer_settimeルーチンの利用 6-8
timer_gettimeルーチンの利用 6-9
timer_getoverrunルーチンの利用 6-10

POSIX sleepルーチンの利用 6-11
nanosleepルーチンの利用 6-11
clock_nanosleepルーチンの利用 6-12

7章 システム・クロックおよびタイマー

システム時間計測 7-1
ローカル・タイマー 7-1
機能 7-2
CPUアカウンティング 7-2
プロセス実行時間および制限 7-3
インターバル・タイマーのデクリメント 7-3
システム・プロファイリング 7-3
CPU負荷バランシング 7-3
CPU再スケジューリング 7-4
POSIXタイマー 7-4
RCU処理 7-4
その他 7-4
ローカル・タイマーの禁止 7-4

8章 ファイル・システムとディスクI/O

ジャーナリング・ファイル・システム 8-1
XFSファイル・システムの作成 8-2
XFSファイル・システムのマウント 8-2
データ管理API(DMAPI) 8-2
ダイレクト・ディスクI/O 8-3

9章 メモリ・マッピング

ターゲット・プロセスのアドレス空間へのマッピングの確立 9-1
mmap(2)の利用 9-1
usermap(3)の利用 9-3
検討事項 9-4
カーネル構成パラメータ 9-4

10章 Non-Uniform Memory Access (NUMA)

概要 10-1
メモリ・ポリシー 10-2
NUMAユーザー・インターフェース 10-3
メモリ・シールドされたノード 10-3
メモリ・シールドとプリアロケート・グラフィック・ページ 10-5
run(1)を利用したNUMAサポート(プロセス用) 10-7
shmconfig(1)を利用したNUMAサポート(共有メモリ領域用) 10-9
システムコール 10-11
ライブラリ機能 10-11
情報提供ファイルおよびユーティリティ 10-11
ノード統計値 10-11
マッピングされたページのノードID 10-12
numastatを利用したNUMA成功/失敗統計値 10-13
kdbサポート 10-13
カーネル・テキスト・ページの複製 10-14
カーネル・モジュール・ページの割り当て 10-15

NUMAバランシング 10-16
NUMAバランシングの有効化 10-17
シールディングの相互作用 10-17
シールディングの制限 10-17
性能ガイドライン 10-18
タスク全体のNUMA mempolicy 10-18
共有メモリ領域 10-18
構成 10-19

11章 カーネルの設定および構成

序文 11-1
ccur-configを利用したカーネルの構成 11-2
カーネルの構築 11-4
ドライバ・モジュールの構築 11-6
プレビルトRedHawkカーネルで動的にロード可能なモジュールの構築例 11-6
追加情報 11-8

12章 カーネル・デバッギング

概要 12-1
VMcore生成イベント 12-1
vmlinuxネームリスト・ファイルの保存 12-1
VMcore Kdump構成 12-2
Kdump構成の更新 12-3
scp VMcore生成の構成 12-4
NFS VMcore生成の構成 12-6
Sysctl(1) Kdumpオプション 12-7
crashを利用したダンプの解析 12-8
ダンプ・ファイルの解析 12-8
実行中システムの解析 12-9
ヘルプの入手 12-9
カーネル・デバッガ 12-10
kgdb/kdb 12-10
NMI割り込み 12-11
NMIウォッチドッグ 12-11

13章 Pluggable Authentication Modules (PAM)

序文 13-1
PAMモジュール 13-1
サービス 13-2
ロール・ベース・アクセス制御 13-2
例 13-3
ケーパビリティの定義 13-3
例 13-4
実装詳細 13-5

14章 デバイス・ドライバ

デバイス・ドライバの種類の理解 14-1
ユーザー・レベル・デバイス・ドライバの開発 14-1
PCIリソースへのアクセス 14-1

PCI BARインターフェース 14-2
カーネル・スケルトン・ドライバ 14-6
サンプル・ドライバの機能の理解 14-6
ドライバのテスト 14-9
カーネル・レベル・ドライバの開発 14-11
ドライバ・モジュールの構築 14-11
カーネルの仮想アドレス空間 14-11
リアルタイム性能の問題 14-11
割り込みルーチン 14-11
割り込み機能の遅延(ボトム・ハーフ) 14-12
マルチスレッディングの問題 14-14
ユーザー空間I/Oドライバ(UIO) 14-14
性能の解析 14-15

15章 PCI-to-VMEサポート

概要 15-1
文書 15-2
ハードウェアのインストール 15-2
開梱 15-2
アダプター・カードの設定 15-3
PCIアダプター・カードのインストール 15-4
VMEバス・アダプター・カードのインストール 15-4
アダプター・ケーブルの接続 15-4
ソフトウェアのインストール 15-5
構成 15-6
btpモジュール 15-6
デバイス・ファイルおよびモジュール・パラメータ仕様 15-6
VMEバス・マッピング 15-7
ユーザー・インターフェース 15-7
API機能 15-8
バインド・バッファの実装 15-9
bt_get_info BT_INFO_KMALLOC_BUF. 15-9
bt_set_info BT_INFO_KMALLOC_SIZ. 15-10
bt_set_info BT_INFO_KFREE_BUF 15-10
バインド・バッファの追加情報 15-11
VMEバス空間へのマッピングおよびバインド 15-13
bt_hw_map_vme 15-13
bt_hw_unmap_vme 15-14
/procファイル・システム・インターフェース 15-15
アプリケーション例 15-17
bt_bind_mult 15-18
bt_bind_multsz 15-19
bt_hwmap 15-19
bt_hwunmap 15-19
readdma 15-20
shmat 15-20
shmbind 15-20
shmconfig-script 15-21
vme-mappings 15-21
writemem 15-21
writedma. 15-21

付録A メッセージ・キュー・プログラム例

POSIXメッセージ・キュー例 A-1
System Vメッセージ・キュー例 A-4

付録B リアルタイム機能のためのカーネル・チューニング B-1

付録C ケーパビリティ

概要 C-1
ケーパビリティ C-1

付録D 32bitコードから64bitコードへの移植

序文 D-1
手順 D-2
コーディング要件 D-3
データ型のサイズ D-3
long型 D-3
ポインタ D-3
配列 D-4
宣言 D-4
明示的なデータ・サイズ D-4
定数 D-5
API D-5
呼び出し規約 D-5
条件付コンパイル D-6
その他 D-6
コンパイル D-6
テスト/デバッグ D-6
性能問題 D-7
メモリのアライメントおよび構造体のパディング D-7

付録E シールドCPU上のカーネル・レベル・デーモン E-1

付録F シールドCPU上のプロセッサ間割り込み

概要 F-1
メモリ・タイプ・レンジ・レジスタ (MTRR)割り込み F-1
グラフィクス割り込み F-3
NVIDIA CUDA割り込み F-4
ユーザー空間でのTLBフラッシュ割り込み F-5

付録G シリアル・コンソールの設定 G-1

付録H ブート・コマンド・ライン・パラメータ H-1

用語解説 Glossary-1

Index Index-1

1
序文

本章は、RedHawk Linuxの紹介およびオペレーティング・システムに含まれているリアルタイム機能の概要を提供します。

概要

Concurrent Computer CorporationのRedHawk™ Linux® は、オープン・ソースLinuxオペレーティング・システムのリアルタイム・バージョンです。互換性およびパフォーマンスを必要とする複雑なタイムクリティカル・アプリケーションをサポートするため、標準Linuxカーネルを基に改良が行われました。RedHawkは、すべてのシステム・オペレーションを直接制御するシングル・プログラミング環境をサポートするため、シングル・カーネル設計を利用します。この設計は、デターミニスティック(レスポンス時間が予測可能)なプログラムの実行および割り込みに対するレスポンスを可能とし、更に高I/Oスループットとデターミニスティックなファイル、ネットワーキング、グラフィックI/O操作を同時に提供します。RedHawkはシミュレーション、データ収集、工業制御機器、医療画像システムが求めるデターミニスティック・アプリケーションのための理想的なLinux環境です。

RedHawkに含まれているのは、良く知られているCentOS(Community ENTerprise Operating System®)です。しかしながら、RedHawkは*全て*のCentOS互換ディストリビューションに対してインストールすることも可能であることに注意してください。

インストール・ディスクは、リアルタイム・カーネルと特定のカーネル機能にアクセスするためのライブラリを提供します。カーネルを除いて全てのRHELのコンポーネントは標準的な方法で動作します。それにはRHELで変更されていないLinuxユーティリティ、ライブラリ、コンパイラー、ツール、インストーラが含まれています。オプションであるNightStar™ RT開発ツールは、タイムクリティカルなアプリケーションの開発や周期実行、パフォーマンスをモニタリングするプロセスのスケジュールに使用出来る Frequency-Based Scheduler(FBS)およびパフォーマンス・モニタに利用することが可能です。

RedHawkカーネルは、オープン・ソースのパッチと最高水準のリアルタイム・カーネルを提供するためにConcurrentが開発した機能の両方を統合します。これらの多くの機能は、40年以上のリアルタイム・オペレーティングシステムの開発の経験に裏づけられたConcurrentが実現したリアルタイムUNIX®より派生しています。これらの特徴は、本章の 「Real-Time Features」セクションの中でもう少し詳細な情報を記載しています。

RedHawkはConcurrentのiHawkシステムに含まれています。iHawkシステムは多様なアーキテクチャや構成が利用可能な対象型マルチプロセッサ(SMP)のシステムです。

SMPシステムへの対応は高度に最適化されています。シールドCPUとして知られるユニークなコンセプトは、プロセッサーの一部を最もデターミニスティックなパフォーマンスを必要とするタスク専用とすることができます。個々のCPUは、割り込み処理、カーネルデーモン、割り込みルーチン、その他のLinuxタスクよりシールドすることが可能です。プロセッサ・シールディングは、15 μ 秒未満の割り込み応答を保証する高度なデターミニスティックな実行環境を提供します。

RedHawk Linuxは、少なくともカーネル3.Xをベースとする他のLinuxディストリビューションのようにPOSIX準拠のレベルは同等です。Concurrentは標準Linuxには存在しないPOSIXリアルタイム拡張を加えることで更なるPOSIXの互換性を付加しました。それぞれのLinux/Intel x86プラットフォーム上で動作するように設計されたパッケージアプリケーションの標準バイナリが定義されたIntel x86アーキテクチャ上のLinuxは、ConcurrentのiHawkプラットフォームで動作します。

NightStar RTは、マルチプロセッサ向けタイムクリティカル・アプリケーションの制御、監視、解析、デバッグを行うためのConcurrentの強力なツールセットです。RedHawkのカーネルには、アプリケーション実行への干渉を最小限に抑えて効果的に機能するツールの強化機能が含まれています。すべてのツールは、同一システム上でもリモートでもアプリケーション制御を邪魔することなく同じように実行されます。

NightStar RTツールには、以下のものが含まれています。詳細な情報は個々のUser's Guideを参照してください。

- NightView™ ソースレベル・デバッガー：マルチ言語、マルチプロセッサ、マルチプログラム、マルチスレッドの監視、デバッグをシングルGUIで行います。NightViewは、アプリケーションの実行速度で実行中のプログラムに修正を加えるためのホットパッチ、データ変更・修正、条件付きブレークポイント/モニタポイント/ウォッチポイントの各機能を持っています。

- NightTrace™ 実行時間アナライザー：動作中のアプリケーションの挙動を解析するために使用します。ユーザーおよびシステムの動きを高分解能タイムスタンプにて記録およびマークします。アプリケーション実行中に発生したこれらのイベントの詳細な挙動をグラフィック表示します。NightTraceは複数のプロセス、複数のプロセッサ上の挙動、分散システム上で実行されたアプリケーション、ユーザー/カーネルの相互関係を表示する理想的なツールです。その強力な機能は特定のイベントやカーネル/ユーザーのステータスを調査することが可能です。

- NightSim™ 周期スケジューラ：周期実行を必要とするアプリケーションを簡単にスケジューリングすることが可能です。開発者は連携する複数のプロセス、それらのプライオリティやCPUの割り当てを動的に制御することが可能です。NightSim は詳細かつ正確なパフォーマンスの統計値とオーバーラン発生時の様々な処理の定義を提供します。

- NightProbe™ データモニター：実行中の複数のプログラムのデータのサンプリング、記録、修正に利用します。プログラムデータはシンボルテーブルを参照して探し出します。アプリケーションページはアプリケーション実行への影響を最小限にするために物理レベルのページで共有されます。NightProbeは入出力用のGUIコントロールパネルを作成することでデバッグ、解析、エラー挿入(Fault Injection)を代用することが可能です。

- NightTune™ パフォーマンスチューナー： CPU使用状況、コンテキストスイッチ、割り込み、仮想メモリ使用状況、ネットワーク使用状況、プロセス属性、CPUシールディング等のシステムやアプリケーション性能解析のためのGUIツールです。NightTuneはポップアップダイアログまたはドラッグ＆ドロップ操作で個々のプロセスまたはグループの優先度、スケジューリング・ポリシー、CPUアフィニティを変更することができます。同時にCPUのシールディングやハイパースレッドの属性変更、個々の割り込みの割り当てを変更することも可能です。

RedHawk Linuxカーネル

RedHawk Linuxカーネルは3種類存在し、それぞれがPRTリアルタイム有りおよび無しで利用可能です。

システム管理者は、ブート・ローダーを介してどのカーネルのバージョンをロードするかを選択することが可能です。表1-1にプレビルト・カーネルのそれぞれの概要を示します。

表1-1 プレビルト・カーネル

カーネルの種類	**Generic**	**Trace**	**Debug**
カーネル名称 *	**vmlinuz-***kernelversion***-RedHawk-***x.x*	**vmlinuz-***kernelversion***-RedHawk-***x.x***-trace**	**vmlinuz-***kernelversion***-RedHawk-***x.x***-debug**
推奨使用方法	タイムクリティカル・アプリケーションの実行	NightStar RTツールを利用してパフォーマンス評価	アプリケーションまたはドライバの新規開発
概要	Genericカーネルは最も最適化されており、最高のパフォーマンスを提供しますが、NightStar RTツールの利点すべてが必要だとしても一部の機能が使えません。	TraceカーネルはGenericカーネルの全ての機能がサポートされ、NightTraceツールのカーネル・トレース機能を提供しており、多くのユーザーに推奨します。 このカーネルはシステム起動時にデフォルトでロードされます。	DebugカーネルはTraceカーネルの全ての機能がサポートされ、更に実行時間の検証が含まれ、カーネル・レベル・デバッグのサポートも提供します。 このカーネルはドライバの開発やシステムの問題をデバッグする際に推奨します。
Features			
カーネル・デバッグ	無効	無効	有効
システム・ダンプ	有効	有効	有効
カーネル・トレース (NightTraceを利用)	無効	有効	有効
高分解能プロセス・アカウンティング	有効	有効	有効
NMI Watchdog	無効	無効	有効
Frequency Based Scheduler (FBS)	モジュールがロードされたときに有効	モジュールがロードされたときに有効	モジュールがロードされたときに有効
パフォーマンス・モニタ (PM)	無効	有効	有効

***** *kernelversion* はそのカーネルをベースとするLinuxカーネル・ソースコードの公式バージョンです。
x.x はRedHawkのバージョン番号を示します。
例: vmlinuz-3.10.25-rt23-RedHawk-6.5.

システム・アップデート

「RedHawk Linux updates」はConcurrentのWebサイト「RedHawk Updates」からダウンロードすることが可能です。詳細は*RedHawk Linux Release Notes* を参照してください。

NOTE

Concurrentは「CentOS updates」のダウンロードは推奨しません。RedHawk Linuxカーネルは標準CentOSカーネルを差し替えており、CentOSディストリビューションのどのようなバージョンであっても動作する可能性は高いです。しかし、Concurrent以外からのアップグレードのインストール、特に**gcc**と**glibc**、についてはシステムが不安定となる可能性がありますので推奨しません。外部からのセキュリティに関するアップデートは必要であればインストールしても構いません。

リアルタイム機能

本セクションはオペレーティング・システムのリアルタイム処理やパフォーマンスを含む機能の簡単な説明を提供します。以下に記載された機能に関する更に詳細な情報は、本章に後にて提供します。

プロセッサ・シールディング

Concurrentは割り込みやシステムデーモンに関連した予測できない処理から選択したCPUを保護(シールド)する方法を開発しました。クリティカルなプライオリティの高いタスクを特定のCPUにバインドし、多くの割り込みやシステムデーモンを他のCPUへバインドすることにより、マルチプロセッサーシステムの特定CPU上において最高のプロセス・ディスパッチ・レイテンシー(PDL)を得ることができます。2章ではシールディングCPUの手本を紹介し、またレスポンス時間向上およびデターミニズム強化のテクニックを説明します。

プロセッサ・アフィニティ

複数のCPU上で複数のプロセスを実行するリアルタイム・アプリケーションでは、システムの全てのプロセスのCPU割り当てを明示的に制御することが望ましい。この機能はConcurrentより**mpadvise(3)**ライブラリルーチンや **run(1)**コマンドを通して提供されます。追加情報については2章およびmanページを参照してください。

ユーザー・レベル・プリエンプション制御

複数のCPU上で動作する複数のプロセスを所有するアプリケーションがプロセス間でデータを共有する動作をする時、2つ以上のプロセスの同時アクセスによる破壊を防ぐために共有データへのアクセスは保護する必要があります。共有データの保護のための最も効果的なメカニズムはスピンロックですが、スピンロックを保持している間にプリエンプトする可能性のあるアプリケーションが存在すると効果的に使用することができません。効果を維持するためにRedHawkはアプリケーションがプリエンプションを瞬時に無効にするためのメカニズムを提供します。ユーザー・レベルのプリエンプション制御に関するより詳細な情報は5章と**resched_cntl(2)**のmanページを参照してください。

高速ブロック/ウェイク・サービス

多くのリアルタイムアプリケーションは複数の協同プロセスで構成されています。これらのアプリケーションはプロセス間同期をするための効果的な方法を必要としています。Concurrentが開発した高速ブロック/ウェイク・サービスは、他の協同プロセスからのウェイク・アップ通知を待っているプロセスが瞬時に自分自身をサスペンドすることを可能とします。詳細な情報については、2章、5章および**postwait(2)**と**server_block(2)**のmanページを参照してください。

RCIMドライバ

Real-Time Clock and Interrupt Module(RCIM)をサポートするためのドライバーがインストールされています。この多目的PCIカードは以下の機能を備えています。

- 最大12個の外部デバイス割り込み
- 最大8個のシステムへの割り込み可能なリアルタイムクロック
- アプリケーションからの割り込み作成が可能な最大12個のプログラマブル割り込みジェネレータ

これらの機能はRCIMがインストールされているシステム上でローカル割り込みをすべて作成することが可能です。複数のRedHawk Linuxシステムは相互にチェーン接続することが可能で、他のRCIMがインストールされたシステムに対してローカル割り込みの配信が最大12個まで可能です。これは1つのタイマー、1つの外部割込み、もしくは1つのアプリケーション・プログラムが複数のRedHawk Linuxシステムを同期させるために同時に割り込むことを許可しています。更にRCIMには複数のシステムを共通時間で共有させることが出来る同期高分解能クロックが含まれています。更なる情報については、本書の6章と*Real-Time Clock & Interrupt Module (RCIM) User's Guide* を参照してください。

Frequency-Based Scheduler

Frequency-Based Scheduler (FBS)は、所定周期の実行パターンにより動作するアプリケーションをスケジューリングするためのメカニズムです。FBSはプログラムが実行する時間になったときにプロセスを起こすための非常にしっかりしたメカニズムも同時に提供します。更に周期アプリケーションのパフォーマンスがデッドラインを超える場合にプログラムマーが利用可能な様々なオプションにより追跡することが可能です。FBSは周期実行アプリケーションをスケジュールするためのNightSimツールの基となるカーネル・メカニズムです。更なる情報については、*Frequency-Based Scheduler (FBS) User's Guide* と*NightSim RT User's Guide* を参照してください。

/procの修正

特権を持ったプロセスが他のプロセスのアドレス空間の値を読み書きを可能にするため、修正はプロセスのアドレス空間をサポートする/procで行われます。これはNightProbeデータ・モニタリング・ツールやNightViewデバッガのサポートに利用されます。

カーネル・トレース機能

カーネルの動きをトレースする機能が追加されました。これにはカーネル・トレース・ポイントの挿入、カーネルのトレース・メモリ・バッファの読み取り、トレース・バッファの管理を行うためのメカニズムが含まれています。カーネル・トレース機能はNigthTraceにより利用できます。カーネル・トレースに関する情報は付録Dを参照してください。

ptrace拡張

Linuxのptraceデバッギング・インターフェースは、NightViewデバッガーの機能をサポートするために拡張されました。追加された機能：

- デバッガ・プロセスが停止状態のプロセス内のメモリを読み書きするための機能
- デバッガがデバッグ中のプロセスのシグナル群だけをトレースするための機能
- デバッガがデバッグ中のプロセスを新しいアドレスで再実行するための機能
- デバッガ・プロセスがデバッグ中の全ての子プロセスに自動的にアタッチするための機能

カーネル・プリエンプション

カーネル内で実行中の低優先度プロセスを高優先度プロセスがプリエンプトするための機能が提供されます。標準的なLinux下の低優先度プロセスは、カーネルから抜けるまで実行し続け、ワーストケースのプロセス・ディスパッチ・レイテンシーとなります。データ構造体を保護するメカニズムは、対称型マルチプロセッサをサポートするためにカーネルに組み込まれています。

リアルタイム・スケジューラ

リアルタイム・スケジューラは、システム内で動作中のプロセスがいくつであっても固定長のコンテキスト・スイッチ時間を提供します。また、対称型マルチプロセッサ上で動作する真のリアルタイム・スケジューリングも提供します。

低レイテンシー拡張

カーネルが使用する共有データ構造体を保護するため、カーネルはスピン・ロックとセマフォによりその共有データ構造体へアクセスするコード・パスを保護します。スピン・ロックのロック処理は、スピン・ロックが保持している間はプリエンプションや割り込みが無効となることを命じます。低レイテンシー拡張は、より良い割り込み応答時間を提供するために最悪の状況となるプリエンプションが特定されたアルゴリズムに手を加えています。

優先度継承

スリーピーウェイト相互排他メカニズムとして使用されるセマフォは優先度反転の問題を引き起こす可能性があります。クリティカル・セクション内で実行される1つ以上の低優先度プロセスが1つ以上の高優先度プロセスの動作を妨げるときに優先度反転を引き起こします。優先度継承はクリティカル・セクション内で実行中の低優先度プロセスの優先度を待機中の最高優先度プロセスへ一時的に引き上げることを生じます。これは、クリティカル・セクション内で実行中のプロセスがクリティカル・セクションから離れるまで実行し続けるために十分な優先度を持つことを確実にします。詳細な情報については5章を参照してください。

高分解能プロセス・アカウンティング

メインストリームであるkernel.orgのLinuxカーネル内では、システムはとても大雑把なメカニズムを使ってプロセスのCPU実行時間を計算しています。これは特定のプロセスが使用するCPU時間の量がとても不正確になる可能性があることを意味します。高分解能プロセス・アカウンティング機能はとても正確なCPU実行時間計算のためのアルゴリズムを提供し、優れたアプリケーションの性能モニタリングを可能にします。この機能はConcurrentが提供する全てのRedHawk Linuxプレビルト・カーネルの中に盛り込まれ、標準LinuxのCPUアカウンティング・サービスとそれらのカーネルのパフォーマンス・モニタに利用されます。CPUアカウンティング方式に関する情報は7章を参照してください。

ケーパビリティのサポート

Pluggable Authentication Module (PAM)は、ユーザーに特権を割り当てるメカニズムを提供し、認証プログラムを再コンパイルすることなく認証ポリシーを設定できます。この仕組みの下では、ルートだけが許可された特権を必要とするアプリケーションを非ルートユーザーが実行できるように設定することが可能です。例えば、メモリ内のページをロックする機能は個々のユーザーやグループに割り当て可能な所定の特権により提供されます。

特権は、設定ファイルを通して許可されます。ロールは有効なLinuxケーパビリティのセットです。定義されたロールは、予め定義されたロールのケーパビリティを継承する新しいロールと一体となって後に続くロールの基礎的要素として使用されます。ロールはシステム上でケーパビリティを定義してユーザーおよびグループに割り当てます。

PAMの機能に関する情報は13章を参照してください。

カーネル・デバッガ

メインストリームであるkernel.orgのLinuxカーネル・デバッガー**(KGDB/KDB)**はRedHawk Linuxのデバッグ・カーネルでサポートされます。

追加の情報は12章を参照してください。

カーネルのコア・ダンプ/クラッシュ解析

kexec-toolとcrushオープン・ソース・パッチが提供する**kexec**および**kdump**は、他のカーネルのクラッシュ・ダンプをロードして取り込みことを有効にし、**crash**ユーティリティはそのダンプを解析するために提供されます。クラッシュ・ダンプ解析に関する詳細な情報については12章を参照してください。

ユーザー・レベル・スピン・ロック

RedHawk Linuxのビジーウェイト相互排他ツールには低オーバーヘッドのビジーウェイト相互排他変数(スピンロック)と初期化、ロック、アンロック、クエリー・スピンロックが可能なマクロのセットが含まれます。効果を上げるためにユーザー・レベル・スピンロックはユーザー・レベル・プリエンプション・コントロールと一緒に利用する必要があります。詳細は5章を参照してください。

usermapおよび/proc mmap

libccur_rtライブラリに属する **usermap(3)**ライブラリルーチンは、簡単なCPUの読み書きを利用して現在実行中のプログラムのロケーションを効果的に監視および変更するためのアプリケーションを提供します。

/procファイルシステムの**mmap(2)**は、自分自身のアドレス空間の中に他のプロセスのアドレス空間の一部を割り当てることを許可する**usermap(3)**のための基本となるカーネルサポートです。従って、他の実行中のプログラムの監視および変更は**read(2)**および**write(2)**システムコールによる**/proc**ファイルシステムのオーバーヘッドを発生させる事なくアプリケーション自身のアドレス空間の中で簡単なCPUの読み書きとなります。詳細な情報については9章を参照してください。

ハイパースレッディング

ハイパースレッディングはIntel Pentium Xeonプロセッサーの機能です。これは1つの物理プロセッサーをオペレーティングシステムに2つの論理プロセッサーのように見せる効果があります。2つのプログラムカウンターは各々のCPUチップの中で同時に実行されるため、事実上、各々のチップはデュアルCPUとなります。物理CPUのハイパースレッディングは、キャッシュミスや特殊命令のようなものを2つのレジスターセット間で高速ハードウェアベースのコンテキスト・スイッチを利用することにより“並行”して複数のタスクを実行することが可能です。RedHawk Linuxにはハイパースレッディングのサポートが含まれています。リアルタイム環境においてこの機能を効果的に使用する詳細な情報については2章を参照してください。

XFSジャーナリング・ファイル・システム

SGIが提供するXFSジャーナリング・ファイル・システムはRedHawk Linuxに実装されています。ジャーナリング・ファイル・システムは処理を記録するためにジャーナル(ログ)を使用します。システムクラッシュの事象が発生した場合、バックグラウンド・プロセスは再起動を実行し、ジャーナルからジャーナリング・ファイル・システムへのアップデートを終了します。このようにファイルシステムのチェックの複雑さを徹底的に省くことで復旧時間を削減します。SGIは、性能と拡張性を補助するためにBツリーを広範囲にわたり利用したものをベースとしたマルチスレッド、大容量ファイルおよび大容量ファイルシステムが利用可能な64bitファイルシステム、拡張属性、可変長ブロックサイズを実装しています。詳細については8章を参照してください。

POSIXリアルタイム拡張

RedHawk Linuxは、ISO/IEC 9945-1に記述されているPOSIXリアルタイム拡張により定義された殆どのインターフェースをサポートしています。以下がサポートされている機能です。

- ユーザー優先度スケジューリング
- プロセス・メモリ・ロック
- メモリ・マップド・ファイル
- 共有メモリ
- メッセージ・キュー
- カウンティング・セマフォ
- リアルタイム・シグナル
- 非同期I/O
- 同期I/O
- タイマー(高分解能バージョン)

ユーザー優先度スケジューリング

RedHawk Linuxはユーザー優先度スケジューリングに適応しています－固定優先度POSIXスケジューリングでスケジュールされたプロセスは、実行時の状態に応じてオペレーティングシステムにより優先度が変更されることはありません。結果として生じるメリットはカーネルのオーバーヘッドの削減とユーザーコントロールの増加です。プロセス・スケジューリング機能は4章に記載されています。

メモリ常駐プロセス

ページングとスワッピングはアプリケーションプログラムに予測できないシステム・オーバーヘッド時間を付加します。パフォーマンス低下の原因となるページングとスワッピングを排除するため、RedHawk Linuxは確実なプロセスの仮想アドレス空間常駐の割り当てをユーザーに許可しています。**mlockall(2), munlockall(2)**, **mlock(2)**, **munlock(2)**のPOSIXシステムコールおよびRedHawk Linuxの**mlockall_pid(2)**システムコールは物理メモリ内のプロセス仮想アドレス空間の全てまたは一部をロックすることが可能です。詳細はmanページを参照してください。

メモリ・マッピングおよびデータ共有

RedHawk LinuxはIEEE規格1003.1b-1993およびSystem V IPCメカニズムに準拠する共有メモリおよびメモリ・マッピング機能をサポートします。POSIX機能はメモリ・オブジェクトの利用を通してプロセスがデータを共有することを許可し、1つまたはそれ以上のプロセスのアドレス空間にマップ可能な指定された記憶領域を関連したメモリと共有することを許可します。メモリ・オブジェクトにはPOSIX共有メモリオブジェクト、レギュラーファイル、いくつかのデバイス、ファイル・システム・オブジェクト(ターミナル、ネットワーク等)が含まれます。プロセスはオブジェクト上のアドレス空間の一部をにマッピングすることにより直接メモリ・オブジェクト内のデータにアクセスすることが可能です。これはカーネルとアプリケーション間のデータコピーを排除するため、**read(2)**および**write(2)**システムコールを使うよりも更に効果的です。

プロセス同期

RedHawk Linuxは協同プロセスが共有リソースへのアクセスを同期するために利用可能な多様なツールを提供します。

IEEE規格1003.1b-1993に準拠するカウンティング・セマフォは、マルチスレッド化されたプロセス内の複数のスレッドが同一リソースへのアクセスを同期することが可能です。カウンティング・セマフォはリソースの使用および割り当てが可能なタイミングを判定する値を持っています。プロセス間セマフォをサポートするSystem V IPC セマフォも利用可能です。

セマフォに加えてConcurrentが開発した一連のリアルタイム・プロセス同期ツールは、再スケジューリングの影響を受けるプロセスの制御、連続したプロセスのビジーウェイト相互排他メカニズムによるクリティカル・セクションへのアクセス、プロセス間のクライアントーサーバ相互関係の調整の各機能を提供します。これらのツールにより、優先度反転を抑制するスリーピーウェイト相互排他を提供するためのメカニズムを構成することが可能になります。

同期ツールの説明および利用手順は5章で提供されます。

非同期入出力

非同期でI/O操作を実行できるということはI/O操作の開始とブロックせずにI/O完了からの復帰が可能であることを意味します。RedHawk LinuxはIEEE規格1003.1b-1993に準拠したライブラリ・ルーチンのグループによる非同期I/Oに対応しています。これらのインターフェースはプロセスが非同期での読み書き処理の実行、シングルコールによる複数の非同期I/O操作の開始、非同期I/O操作の完了待機、待機している非同期I/O操作のキャンセル、非同期ファイルの同期実行が可能です。この"aio"機能はシステム上のinfoページ("info libc")に記載されています。

同期入出力

RedHawk LinuxはIEEE規格1003.1b-1993に準拠した同期I/O機能もサポートしています。POSIX同期I/Oはアプリケーションのデータとファイルの整合性を確実にする手段を提供します。同期出力操作は出力デバイスに書き込まれたデータの記録を確実にします。同期入力操作はデバイスから読み取ったデータと現在ディスク上に存在するデータのミラーであることを確実にします。詳細な情報についてはmanページを参照してください。

リアルタイム・シグナルの挙動

リアルタイム・シグナルの挙動はIEEE規格1003.1b-1993に含まれている、リアルタイム・シグナル番号、複数の特定シグナル発生のキューイングのサポート、複数の同種類のシグナル発生を区別するためにシグナルが作成されたときのアプリケーション定義された値の規格のサポート等によって仕様が定められています。POSIXシグナル管理機能には、シグナル受信待ち、シグナルおよびアプリケーション定義の値のキューイングが可能な **sigtimedwait(2)**, **sigwaitinfo(2)**, **sigqueue(2)**システムコールが含まれています。詳細な情報についてはmanページを参照してください。

クロックおよびタイマー

高分解能POSIXクロックおよびタイマーのサポートがRedHawk に含まれています。POSIXクロック全体はタイムスタンプやコードセグメント長の計測のような目的で使用されます。POSIXタイマーはアプリケーションが高分解能クロック上の相対または絶対時間を使用することで単発または定期的なイベントをスケジュールすることが可能です。アプリケーションは個々のプロセスで複数のタイマーを作成することが可能です。更には非常に短い時間プロセスをスリープ状態にするために利用でき、また、スリープ時間の計測に使用されるクロックを指定できる高分解能スリープのメカニズムを提供します。追加の情報は6章を参照してください。

メッセージ・キュー

IEEE規格1003.1b-1993上のPOSIXメッセージ送信機能はRedHawk Linux に含まれており、ファイルシステムとして実行されます。POSIXメッセージキュー・ライブラリ・ルーチンはメッセージキューの作成、オープン、問合せ、破棄、メッセージの送受信、送信メッセージの優先度設定、メッセージ到達時の非同期通知リクエストが可能です。POSIXメッセージキューはSystem V IPCメッセージとは関係なく動作し、System V IPCメッセージも利用できます。詳細は3章を参照してください。

2
リアルタイム性能

本章ではRedHawk Linuxでのリアルタイム性能を実現することに関連したいくつかの問題を明確にします。本章の主な焦点は、最高のリアルタイム性能を得るためにプロセスおよび割り込みをシステム内のCPUの一部に割り当てる *シールドCPUモデル* となります。

リアルタイム性能で重要なことは割り込み応答、プロセス・ディスパッチ・レイテンシー、デターミニスティック(予測可能な)プログラムの実行を明確にすることです。これらの指標上で様々なシステム動作の影響を明確にし、最適なリアルタイム性能のための手法を提供します。

シールドCPUモデルの概要

シールドCPUモデルは対称型マルチ・プロセッサー・システムにおいて最高のリアルタイム性能を得るためのアプローチです。シールドCPUモデルはリアルタイム・アプリケーションのデターミニスティック(予測可能)な実行、同じく割り込みに対してデターミニスティック(予測可能)な応答を可能にします。

コード・セグメントの実行に必要な時間が予測可能かつ一定である時、タスクはデターミニスティックな実行状態となります。同様に割り込みの応答に必要な時間が予測可能かつ一定である時、割り込み応答もデターミニスティックとなります。コード・セグメントの実行または割り込み応答を測定した時間が標準的なケースとは明らかに異なり最悪であった時、そのアプリケーション性能は *ジッター* が発生している状態と言う。リソースを共有するためのメモリ・キャッシュやメモリ・コンテンションのようなコンピュータ・アーキテクチャ機能が原因で、計測した実行時間の中に常にジッターが含まれます。それぞれのリアルタイム・アプリケーションは容認できるジッターの量を明確にする必要があります。

シールドCPUモデルでは、特定の重要なリアルタイム機能に対してハイグレードなサービスを保証する方法としてタスクと割り込みをあるCPUに割り当てます。特に高優先度タスクは、1つまたはそれ以上のシールドCPUに制限し、殆どの割り込みや低優先度タスクはそれ以外のCPUに制限します。高優先度タスクを動作させる役割を持つCPUが、割り込みに関連した予測できない処理やシステムコールを経由してカーネル空間にいる他の低優先度プロセスの動きから遮断されているこのCPUの状態を *シールドCPU* と呼びます。

シールドCPU上で実行されるべきタスクの種類の例：

- 割り込み応答時間の保証を要求するタスク
- 最速の割り込み応答時間を要求するタスク
- 高周期で実行しなければならないタスク
- デッドラインを満足するためにデターミニスティックな実行を要求するタスク
- オペレーティング・システムからの割り込みを容認できないタスク

様々なレベルのCPUシールディングは、高優先度割り込み応答しなければならない、またはデターミニスティックな実行を要求するタスクのために異なる度合いのデターミニズムを提供します。シールドCPUで可能となるシールディングのレベルを明確にする前にシステムが外部イベントにどのように応答するのか、コンピュータ・システムのいくつかの通常オペレーションが応答時間およびデターミニズムにどのような影響を与えているのかを理解する必要があります。

デターミニズムの概要

デターミニズム は一定時間内で特定のコード・パス(順に実行される命令セット)を実行するためのコンピュータシステムの能力に言及します。ある状態から他へ変化したコード・パスの実行時間の範囲はシステムでのデターミニズムの度合いを示します。

デターミニズムは、ユーザー・アプリケーションのタイム・クリティカルな部分を要求時間で実行するだけでなく、カーネル内のシステム・コードを要求時間で実行することも当てはまります。プロセス・ディスパッチ・レイテンシーのデターミニズムは、例えば、割り込みのハンドリング、ターゲット・プロセスの起床、コンテキスト・スイッチの実行、ターゲット・プロセスがカーネルから抜け出すのを許可、のような実行されなければならないコード・パスに依存します。(「Process Dispatch Latency」セクションにて用語*プロセス・ディスパッチ・レイテンシー*を明記し、マルチプロセッサーシステム内の特定CPUで最高のプロセス・ディスパッチ・レイテンシーを得るためのモデルを紹介します。)

プログラム実行のデターミニズムにおいて最大のインパクトは割り込みの受信です。これは割り込みがシステム内では常に最高優先度の機能であり、割り込みの受信が予測不可能ープログラム実行中は遅れることなく如何なるポイントでも発生する可能性があるーであるためです。重要ではない割り込みからのシールディングは、高優先度タスク実行中にデターミニズムの向上に最大のインパクトを得ることになります。

プログラム実行でのデターミニズム向上のそのほかのテクニックについては「デターミニズムを高める手順」セクションに明記されています。

プロセス・ディスパッチ・レイテンシー

リアルタイム・アプリケーションは実在イベントに応答すること、実在イベントのハンドリングに必要な処理を与えられた期限(デッドライン)内に終了することが出来なければなりません。実在イベントに応答するために必要な計算はデッドラインの前に終了していなければならず、さもなければその結果は不正確であるとみなされます。割り込みへの応答が異常に長い1つの事例は、デッドラインを超えたことが原因である可能性があります。

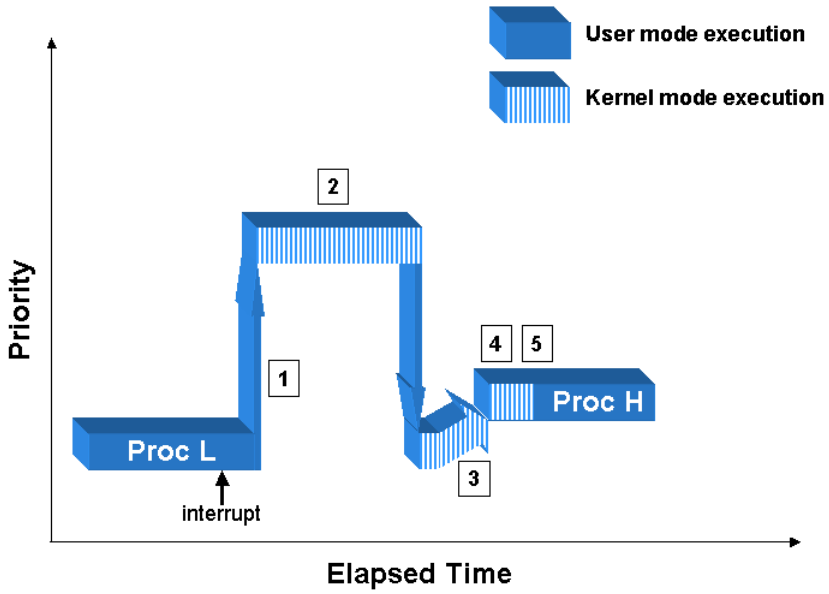
用語*プロセス・ディスパッチ・レイテンシー* は、割り込みによって通知される外部イベント発生から外部イベント待ちプロセスがユーザー・モードでの最初の命令を実行するまでの時間経過を意味します。リアルタイム・アプリケーションにとって予想される最悪のプロセス・ディスパッチ・レイテンシーは基準の鍵になります。それは、デッドラインを満足することを保証するリアルタイム・アプリケーション性能を左右する最悪の応答時間であるためです。

プロセス・ディスパッチ・レイテンシーは以下のイベント発生のシーケンスに掛かる時間で構成されます。

1. 割り込みコントローラは割り込みを通知し、CPUへの例外割り込みを作成します。
2. 割り込みルーチンが実行され、割り込みを待っている(ターゲット)プロセスが起こされます。
3. 現在実行中のプロセスは停止され、コンテキスト・スイッチが機能するためターゲット・プロセスが実行可能となります。
4. ターゲット・プロセスは割り込み待ちでブロックされていたカーネルポイントから抜けます。
5. ターゲット・プロセスはユーザー・モードで実行します。

この一連のイベントはプロセス・ディスパッチ・レイテンシーの理想のケースを表しており、図2-1に図示されています。上記1～5の番号が図2-1の中に記述されています。

図2-1 標準的なプロセス・ディスパッチ・レイテンシー



プロセス・ディスパッチ・レイテンシーは、アプリケーションが外部イベントに対して応答可能なスピードを表しているので、イベント駆動型リアルタイム・アプリケーションにとってとても重要な基準となります。殆どのリアルタイムアプリケーションの開発者が、彼らのアプリケーションが特定のタイミング制約を満たす必要があるため、予想される最悪のプロセス・ディスパッチ・レイテンシーに興味を持っています。

プロセス・ディスパッチ・レイテンシーはいくつかのオペレーティング・システムの通常操作、デバイスドライバー、ハードウェアに影響します。以下のセクションではプロセス・ディスパッチ・レイテンシーでのいくつかのジッターの原因を観察します。

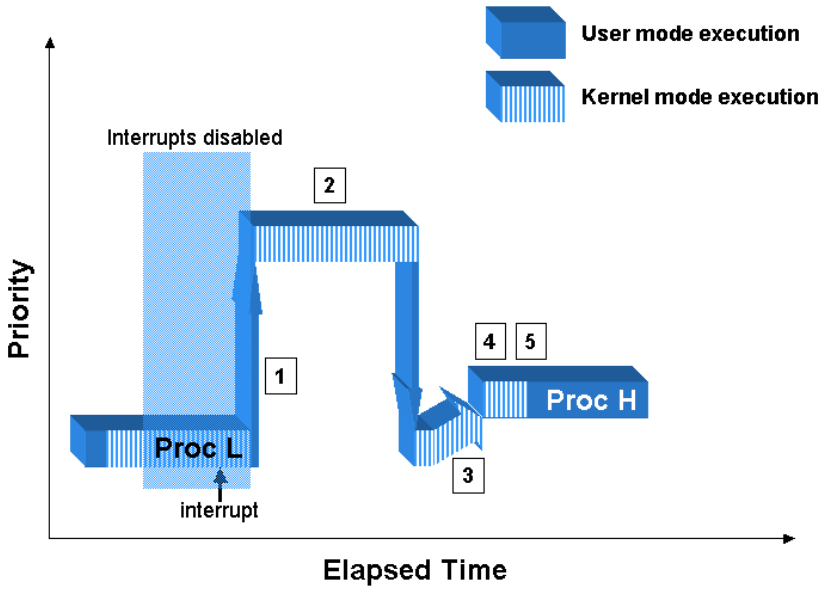
割り込み禁止の効果

オペレーティング・システムは共有データ構造体を破壊されるのを避けるために共有データ構造体へのアクセスを保護しなければなりません。割り込みレベルでデータ構造体がアクセスされることが可能な時、いつそのデータ構造体がアクセスされようとも割り込みを無効にする必要があります。これは、同じ共有データ構造体の更新最中にプログラム・レベル・コードに割り込んで共有データ構造体が破壊されることから割り込みコードを保護します。これはカーネルが短時間割り込みを無効にする主要な理由です。

割り込みが無効であるとき、応答しようとしている割り込みは再び割り込みが有効となるまでアクティブになることが出来ないため、プロセス・ディスパッチ・レイテンシーは影響を受けます。このケースでは、割り込み待機中タスクのプロセス・ディスパッチ・レイテンシーは割り込みが無効である状態が続く時間だけ延長されます。これは図2-2に図示されています。この図表内では、低優先度プロセスが割り込みを無効にするシステムコールを実行しました。割り込みが現在無効であるため、高優先度割り込みが発生した時にアクティブになることは出来ません。低優先度プロセスがクリティカル・セクションを終了した時、割り込みが有効となり、アクティブな状態となって割り込みサービスルーチンがコールされます。通常の割り込み応答のステップから完了までは普通の方法となります。図2-2に記述された1～5の番号は2-3ページで説明した通常のプロセス・ディスパッチ・レイテンシーのステップを表します。

明らかに割り込みが無効となっているオペレーティング・システム内のクリティカル・セクションは、良好なプロセス・ディスパッチ・レイテンシーを得るために最小限に抑えれなければなりません。

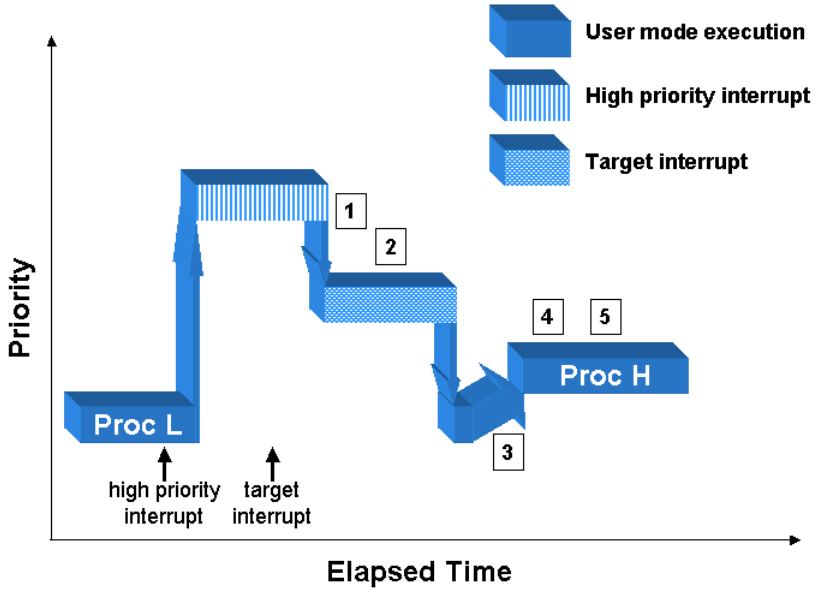
図2-2 割り込み無効によるプロセス・ディスパッチ・レイテンシーへの影響



割り込みの影響

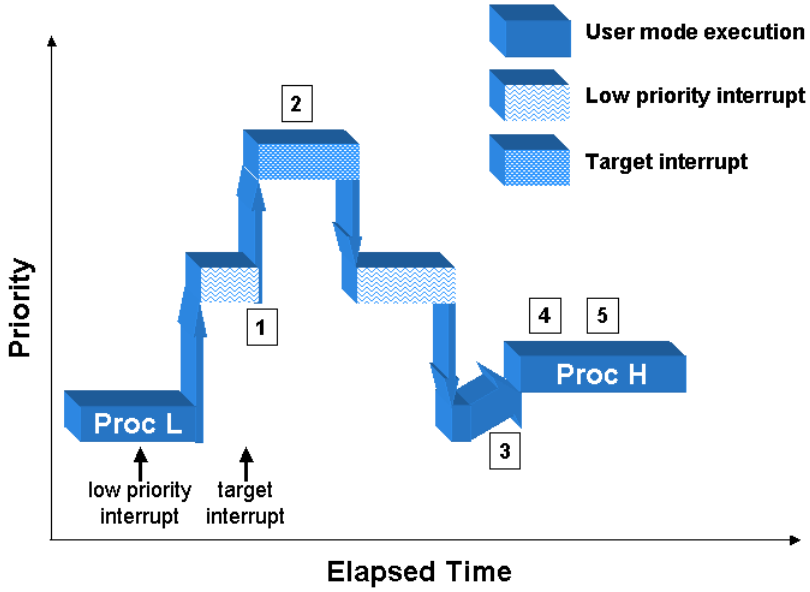
割り込みの受信は割り込みを無効にしたことと同じようにプロセス・ディスパッチ・レイテンシー影響を及ぼします。ハードウェア割り込みを受信したとき、システムは現在の割り込みよりも優先度が同等かそれ以下の割り込みをブロックします。単純なケースを図2-3に図示します。ターゲットの割り込みの前に高優先度割り込みが発生した場合、高優先度割り込みが発生するまでターゲットの割り込みの遅延を招くことになります。図2-3に記述された1～5の番号は2-3ページで説明した通常のプロセス・ディスパッチ・レイテンシーのステップを表します。

図2-3 高優先度割り込みによるプロセス・ディスパッチ・レイテンシーへの影響



割り込みの相対的な優先度はプロセス・ディスパッチ・レイテンシーに影響しません。低優先度割り込みがアクティブになる時でも、高優先度割り込みに対するプロセス・ディスパッチ・レイテンシーへの割り込みの影響は同等です。これは割り込みが常にユーザー・レベル・コードよりも高い優先度で実行されているためです。従って、我々は高優先度割り込みのための割り込みルーチンを提供するかもしれませんが、その割り込みルーチンは実行中のユーザー・レベル・コンテキストを全ての割り込みの実行が完了するまで取得することが出来ません。プロセス・ディスパッチ・レイテンシーにおけるこの低優先度割り込みの影響は図2-4に図示されています。これらの処理方法の順序は図2-3での高優先度割り込みのケースとは異なりますが、プロセス・ディスパッチ・レイテンシーにおける影響は同等です。図2-4に記述された1～5の番号は2-3ページで説明した通常のプロセス・ディスパッチ・レイテンシーのステップを表します。

図2-4 低優先度割り込みによるプロセス・ディスパッチ・レイテンシーへの影響

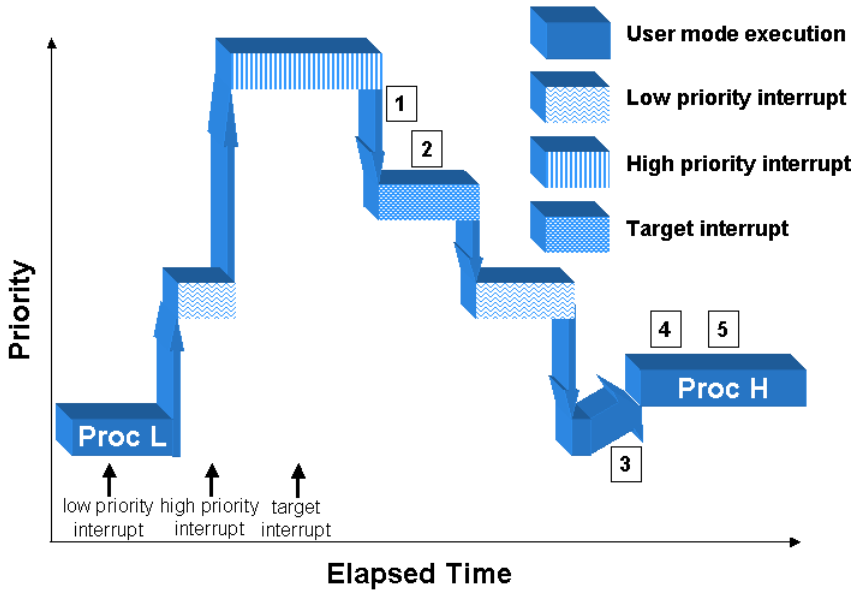


プロセス・ディスパッチ・レイテンシーに対する明確な影響で割り込みの無効化と割り込み受信とで最大の違いの1つは、割り込みが実行しているアプリケーションに対して非同期かつ予測不可能な時に発生することです。これは利用可能なシールディングの様々なレベルを理解することが重要です。

複数の割り込みが特定のCPU上で受信が可能な時、プロセス・ディスパッチ・レイテンシーへの影響は深刻となる可能性があります。これは複数の割り込み処理ルーチンが高優先度割り込みのプロセス・ディスパッチ・レイテンシーが完了する前に処理されなければならない割り込みが積み重なることが可能であるためです。図2-5は高優先度割り込みに応答しようとしている間に2つの割り込みがアクティブになるケースを示します。図2-5に記述された1～5の番号は2-3ページで説明した通常のプロセス・ディスパッチ・レイテンシーのステップを表します。CPUが割り込みを受信した時、CPUは割り込みが可能なCPUからの低優先度の割り込みを無効にします。もしこの期間に低優先度の2番目の割り込みがアクティブになったとしても、最初の割り込みがアクティブである限りブロックされます。最初の割り込みサービスが完了した時、2番目の割り込みはアクティブになりサービスが提供されます。もし2番目の割り込みが最初の割り込みよりも高優先度であった場合、その割り込みは即座にアクティブになります。2番目の割り込み処理が完了した時、最初の割り込みは再びアクティブになります。どちらのケースもユーザープロセスは、すべての保留中の割り込みのサービスが完了するまでは実行が抑制されます。

恐らく、それは割り込みがアクティブであり続け、システムが高優先度割り込みに応答することを決して許可しない異常なケースとなる可能性があります。複数の割り込みが特定のCPUに割り付けられる時、割り込みが積み重ねられることが原因でそのCPUのプロセス・ディスパッチ・レイテンシーは予測しにくくなります。

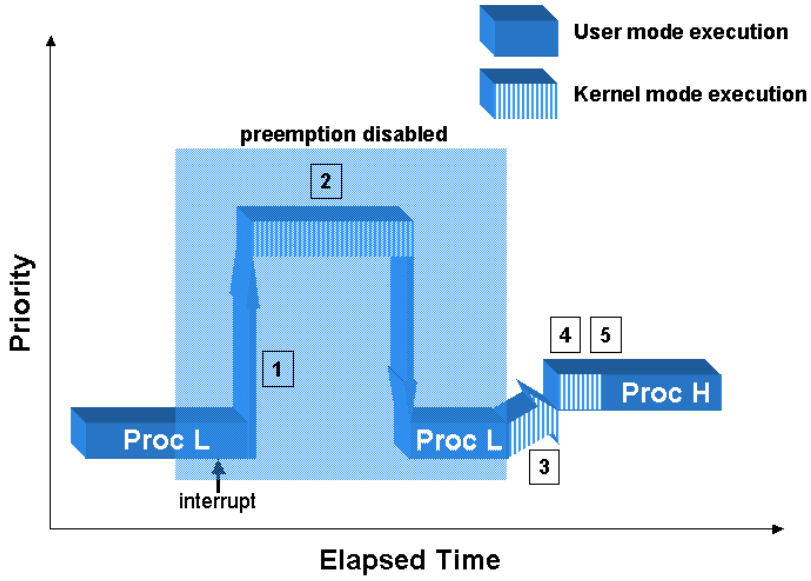
図2-5 複数割り込みによるプロセス・ディスパッチ・レイテンシーへの影響



プリエンプション禁止の効果

RedHawk Linuxには割り込みレベルでロックされない共有リソースを保護するクリティカル・セクションが存在します。このケースでは、このクリティカル・セクション間で割り込みをブロックする理由がありません。しかし、このクリティカル・セクション間で発生したプリエンプションは、もし新しいプロセスが同じクリティカル･セクションに入ってきた場合に共有リソースを破壊する原因となる可能性があります。従って、このようなクリティカル・セクションのタイプでプロセスが実行している間は、プリエンプションは無効となります。プリエンプションのブロックは割り込みの受信が遅延しません。しかし、もしその割り込みが高優先度プロセスを起こす場合は、プリエンプションが再び有効となるまでそのプロセスに切り替わる可能性はありません。同一CPUが要求されているとするならば、プロセス・ディスパッチ・レイテンシーの実際の影響はまるで割り込みが無効にされたのと同じになります。プロセス・ディスパッチ・レイテンシーにおけるプリエンプション無効の効果は図2-6に図示されています。図2-6に記述された1〜5の番号は2-3ページで説明した通常のプロセス・ディスパッチ・レイテンシーのステップを表します。

図2-6 プリエンプション無効によるプロセス・ディスパッチ・レイテンシーへの影響



オープン・ソース・デバイス・ドライバの影響

デバイス・ドライバーはスーパーバイザー・モードで実行するので、Linuxカーネルの一部となります。これは、デバイス・ドライバーは割り込みが無効またはプリエンプションが無効のLinuxの機能をコールすることが自由であることを意味します。デバイス・ドライバーも割り込みを処理しますので、割り込みレベルで過ごす時間を制御します。本章の前セクションで示したようにデバイス･ドライバーの動きは割り込み応答やプロセス・ディスパッチ・レイテンシーに影響する可能性があります。

RedHawk Linuxで有効なデバイス・ドライバーは、リアルタイム性能に不利な影響を与えないことが確かであることをテストしています。オープンソース・デバイス･ドライバーの著者は割り込みレベルで過ごす時間の最小化や割り込み時間の無効化を働きかける一方、実際には、様々な気遣いレベルでオープンソース・デバイス･ドライバーは記述されています。もし付け加えたオープンソース・デバイス･ドライバーを有効にした場合、RedHawk Linuxが提供するプロセス・ディスパッチ・レイテンシーの保証に悪影響を与えるかもしれません。

デバイス・ドライバーに関連するリアルタイムの問題の詳細な情報については「デバイス・ドライバ」章を参照してください。

シールディングでリアルタイム性能を向上する方法

本セクションは、CPUシールディングの特性の違いがユーザープロセスの割り込み応答能力(プロセス・ディスパッチ・レイテンシー)とユーザー・プロセス実行のデターミニズムをどのように向上するかを検証します。

シールディングを有効にする場合、すべてのシールディングの特性は規定値で有効になります。これはシールドCPU上における最高のデターミニスティックな実行環境を提供します。各々のシールディング特性のさらに詳細な情報は後述されています。ユーザーは各シールディング特性が与える影響、特性の一部は通常のシステム機能への副作用を持っているということを十分に理解すべきです。現在サポートされているシールディング特性の3つのカテゴリーは、

- バックグラウンド・プロセスからのシールディングs
- 割り込みからのシールディング
- ローカル割り込みからのシールディング

各々の特性はCPU単位で個別に選択可能です。各々のシールディング特性は後述されています。

バックグラウンド・プロセスからのシールディング

このシールディング特性はCPUをシステム内の一部のプロセスのために予約することを可能にします。この特性はCPUの割り込み応答を最速、予測可能であることを望むときにそのCPU上で有効にすべきです。プロセス・ディスパッチ・レイテンシー上の最高の保障は、割り込みに応答するタスクだけが割り込みを割り付けられたCPU上で実行する時に実現されます。

CPUがバックグラウンド・プロセスを実行可能である時、割り込みを割り付けたそのCPU上の極めてデターミニスティックな応答を要求される高優先度タスクのプロセス・ディスパッチ・レイテンシーに影響を与える可能性があります。これはバックグラウンド・プロセスが割り込みまたはプリエンプションを無効にできるシステムコールを呼ぶ可能性を秘めているためです。これらの処理は、本セクションの「割り込み禁止の効果」および「プリエンプション禁止の効果」で説明されているようにプロセス・ディスパッチ・レイテンシーに影響を与えます。

CPUがバックグラウンド・プロセスを実行可能である時、高優先度プロセスの実行中にデターミニズムへの影響はありません。これはバックグラウンド・プロセスの優先度が高優先度プロセスよりも低いことが想定されます。注意すべきことは、バックグラウンド・プロセスはシグナルや**server_wake1(3)**インターフェースのような他のカーネルのメカニズムを介してプロセスを起床するために必要な時間に影響を及ぼす可能性があることです。

システム内の各プロセスまたはスレッドは、CPUアフィニティ・マスクを持っています。そのCPUアフィニティ・マスクは、プロセスまたはスレッドをどのCPU上で実行するかを決定します。CPUアフィニティ・マスクは親プロセスから継承され、**mpadvise(3)**ライブラリ・ルーチンまたは**sched_setaffinity(2)**システムコールを経由して設定することが可能です。CPUがプロセスからシールドされている時、CPUはシールドCPUを含むCPUセットに明示的に設定されたCPUアフィニティを持つプロセスとスレッドのみを実行します。つまり、プロセスのCPUアフィニティ・マスクがシールドされていないCPUの場合、そのプロセスはシールドされていないCPU上でのみ実行されます。バックグラウンドプロセスからシールドされたCPU上でプロセスまたはスレッドを実行するためには、シールドCPUだけを指定したCPUアフィニティ・マスクを持っていなければなりません。

Linuxによって作成された特定のカーネル・デーモンは、システム内の各CPU上に複製されます。プロセスからシールドされているCPUは、これらの「CPU毎」のデーモンをシールドCPUから移動することはありません。これらのデーモンの影響は、カーネル構成や注意深いアプリケーション挙動の制御を通じて避けることが可能です。CPU毎カーネルデーモンからジッターを避けるための機能や方法は付録Fで説明しています。

割り込みからのシールディング

このシールディング特性はシステムが受信した割り込みの一部だけを処理するために予約することを可能にします。最も高速、最も予測可能なプロセス・ディスパッチ・レイテンシーであることが望ましい時、またはアプリケーションの実行時間にデターミニズムがあることが望ましい時、このシールディング特性を有効にするべきです。

何故なら割り込みはCPU上で常に最高優先度の機能であり、割り込みのハンドリングはプロセス・ディスパッチ・レイテンシーと高優先度タスクのコードパスの実行に必要な時間の両方に影響を与える可能性があります。これは「割り込みの影響」セクションで説明されています。

各デバイスの割り込みはIRQと結合されます。これらIRQはどのCPUで割り込みを受信するかを決定するCPUアフィニティを持っています。割り込みが特定CPUへ送られない時、割り込みコントローラはその時に発生した割り込みをハンドリングするためにIRQアフィニティ・マスクのCPUセットからCPUを選びます。IRQアフィニティは**shield(1)**コマンドまたは**/proc/irq/*N*/smp_affinity**を通して修正されます。

i386アーキテクチャ上では、**kirqd**デーモンはCPU間の割り込み負荷のバランスをとるため周期的にIRQアフィニティを調整します。このデーモンは割り込みシールドと相容れず、IRQBALANCEカーネル構成オプションにより全てのRedHawk Linuxカーネル構成ではデフォルトで無効となっています。これはSHIELDをオフにしたときに利用可能となるIRQBALANCEカーネル・パラメータで有効にすることが可能です。

もしすべてのCPU上ですべての割り込みを無効にすることが好ましい場合、推奨する手順は、1つのCPUを除いてそれ以外のすべてのCPUを割り込みからシールドし、シールドされていないCPU上で**local_irq_disable(2)**をコールすることに留意してください。詳細はmanページを参照してください。

一部の機能は割り込みがシールドCPUへ送信される原因となる可能性があります。プロセッサ間割り込みは、他のCPUにCPU毎の特定タスクをハンドルすることを強制する方法として利用されます。プロセッサ間割り込みはシールドCPUに目立つジッターを引き起こす可能性があります。完全な議論は付録Fを参照してください。

ローカル割り込みからのシールディング

ローカル割り込みは各CPUと一体となった専用タイマーのための特別な割り込みです。RedHawk Linuxでは、このタイマーはカーネル内やユーザー・レベルにおいて様々なタイムアウトのメカニズムで利用されています。この機能は7章の中で説明されいます。初期設定ではこの割り込みはシステム内のすべてのCPU上で有効となっています。

この割り込みは10ミリ秒毎に発せられ、このローカル割り込みはシステム内で最も頻繁に実行される割り込みルーチンの1つとなります。従って、このローカル割り込みはリアルタイム・アプリケーションにとってジッターの大きな原因となります。

CPUがローカル・タイマーからシールドされたとき、ローカル割り込みは事実上無効となり、そのCPUのローカル・タイマーによって提供される機能はもはや実行されません。しかしながら、ローカル・タイマーはローカル・タイマーがシールドされていない他のCPU上で実行され続けます。これらの機能のいくつかは、他の手段によって提供されている間は失われることになります。

特定CPU上でローカル割り込みが無効である時に失われる機能の1つは、CPU実行時間計算のための低分解能メカニズムです。これはCPU上で実行される各プロセスに使われるCPU時間がどの程度なのかを測定するメカニズムです。いつローカル割り込みが発生しようとも、最後のクロック・ティック分の時間は割り込まれたプロセスに加算されます。もし高分解能プロセス・アカウンティングが構成されていた場合、CPU時間はローカル割り込みが有効であるか否かは関係なく性格に計算されます。高分解能プロセス・アカウンティングは7章の「システム・クロックおよびタイマー」に明記されています。

CPUがローカル・タイマーからシールドされた時、ローカル割り込みはシールドCPUに割り付けられたプロセスによってPOSIXタイマーとnanosleepの機能のために使われ続けます。従って、もしローカル・タイマー割り込みを完全に取り除くことが重要である場合、POSIXタイマーまたはnanosleepの機能を利用しているアプリケーションをそのようなCPUに割り付けるべきではありません。もしプロセスがシールドCPU上で実行することが許されない場合、そのタイマーはプロセスの実行が許されたCPUへ移動されます。

一部の機能のためにローカル・タイマーや利用可能な手段を無効にする影響についての全ての解説は7章の「システム・クロックおよびタイマー」を参照してください。

CPUシールディングのインターフェース

本セクションは、シールドCPUを構成するために利用可能なコマンドレベルおよびプログラミング・インターフェースの両方について記述します。シールドCPUを構成するためによくある事例も記述しています。

shieldコマンド

shield(1)コマンドは選択したCPUに対して指定したシールド特性を設定します。shieldコマンドはシールドCPUとしてCPUを特徴付けるために使用されます。シールドCPUは、アプリケーション・コードの実行にかかる時間のデターミニズムを向上させるために一部のシステム機能から保護します。

shieldコマンドの実行によって作用する論理CPUのリストは、CPU番号または範囲のリストをコンマ区切りで渡します。

shieldコマンドを実行するための書式：

shield [*OPTIONS*]

オプションについては表2-1で説明します。

以下に記載されたオプションの中で、*CPULIST*は論理CPUをコンマ区切りの値または値の範囲を表しています。例えば、CPUのリスト"**0-4,7**" は、CPU番号**0,1,2,3,4,7** を指定しています。

表2-1 shield(1)コマンドのオプション

オプション	概要
--irq=*CPULIST*, **-i** *CPULIST*	割り込みから*CPULIST* のすべてのCPUをシールドします。指定されたCPU上で実行される唯一の割り込みは、他のCPU上で実行されることを防ぐためにCPUアフィニティを指定された割り込みとなります。
--loc=*CPULIST*, **-l** *CPULIST*	指定されたCPUのリストはローカル・タイマーからシールドされます。ローカル・タイマーは時間ベースのサービスをCPUに提供します。ローカル・タイマーを無効にすることは、ユーザー/システムの時間計算やラウンドロビンのような一部のシステムの機能が無効となる可能性があります。全ての解説は7章を参照してください。

表2-1 shield(1)コマンドのオプション (続き)

オプション	概要
--proc=*CPULIST***, -p** *CPULIST*	指定されたCPUのリストは無関係なプロセスからシールドされます。非シールドCPU上で実効することを許可されたアフィニティ・マスクを所有するプロセスは、非シールドCPU上で実行されるだけとなります。シールドCPU以外のいずれかのCPU上での実行が不可能となるプロセスは、シールドCPU上での実行が許可されます。
--all=*CPULIST***, -a** *CPULIST*	指定されたCPUのリストは利用可能な全てのシールド特性を所有することになります。各々のシールド特性の意味を理解するために上記の個々のシールドオプションの説明を参照してください。
--help, -h	利用可能なオプションと使用方法の解説します。
--version, -V	コマンドの現在のバージョンを印字します。
--reset, -r	全CPUに対してシールド特性をリセットします。シールドされたCPUはない状態となります。
--current, -c	アクティブな全てのCPUの現在の設定を表示します。

NUMAノードのメモリ・シールドを制御する**shield(1)**のオプションは、10章の「Non-Uniform Memory Access (NUMA)」を参照してください。

shieldコマンド例

以下のコマンドは、最初に全てのシールド特性をリセットし、次にCPUの0, 1, 2を割り込みからシールド、そしてCPUの1をローカルタイマーからシールド、CPUの2を無関係なプロセスからシールド、最後に変更後の新しい全ての設定を表示します。

```
shield -r -i 0-2 -l 1 -p 2 -c
```

以下のコマンドは、CPUの1, 2, 3を割り込み、ローカルタイマー、無関係なプロセスからシールドします。CPUの0は全ての割り込みやシールドCPUのターゲットではないプロセスをサービスする「多目的」CPUとして残します。全てのシールド特性がCPUのリストに設定されます。

```
shield --all=1-3
```

終了ステータス

通常は、終了ステータスは0です。しかし、シールドCPU属性を変更しようとしている間にエラーが発生した場合、診断メッセージが出力され終了ステータスが1で返されます。

shieldコマンド拡張機能

以下に記載された拡張機能は、経験のあるユーザーが使用されることを推奨します。

CPULIST 内で指定されるCPUは‘+’または‘-’の記号を前に置く事が可能で、そのケースのリストのCPUは既にシールドされたCPUのリストに追加 (‘+’) または除外 (‘-’) します。

オプションは複数回使用することが可能です。例えば、“shield -i 0 -c -i +1 -c” は、現在の設定にCPU0を割り込みからシールドし、現在の設定を表示した後に再度CPU1を割り込みからシールドするCPUのリストを追加することを表します。

CPUシールディングの/procインターフェース

CPUシールドのカーネル・インターフェースは以下のファイルを使用する **/proc**ファイルシステムを経由します。

/proc/shield/procs	プロセスのシールド
/proc/shield/irqs	IRQをシールド
/proc/shield/ltmrs	ローカルタイマーをシールド

全てのユーザーはこれらのファイルを読むことが可能ですが、ルートまたは**CAP_SYS_NICE**ケーパビリティを持つユーザーだけが書き換えることが可能です。

ファイルを読んだ時、8桁の16進数ASCII値が返されます。この値はシールドされたCPUのビットマスクです。設定されたビットはシールドされたCPUと一致します。設定された各々のビットのポジションはシールドされている論理CPUの番号です。

例：

```
00000001 – 0ビット目が設定されている場合はCPU0がシールド
00000002 - 1ビット目が設定されている場合はCPU1がシールド
00000004 - 2ビット目が設定されている場合はCPU2がシールド
00000006 - 1ビット目と2ビット目が設定されている場合はCPU1と2がシールド
```

ファイルへ書き込む時、8桁の16進数ASCII値が期待されています。この値は上記と同一フォームのシールドCPUのビットマスクです。その値は間もなく新しいシールドCPUセットとなります。

追加の情報は**shield(5)**のmanページを参照してください。

CPUへのプロセス割り当て

本セクションは利用可能なCPUセットへのプロセスまたはスレッドの割り付け方法を記述します。プロセスが実行を許可されたCPUセットはCPUアフィニティとして知られています。

規定値では、プロセスまたはスレッドはシステム内のどのCPU上でも実行が可能です。プロセスまたはスレッド毎にビットマスクまたはCPUアフィニティを持っており、スケジュール可能なCPUを決定します。プロセスまたはスレッドは、**fork(2)**または**clone(2)**からCPUアフィニティを継承しますが、その後アフィニティが変わる可能性はあります。

mpadvise(3)の呼び出しでMPA_PRC_SETBIASコマンドを指定、または **run(1)**コマンドの **-b** *bias* オプションを指定するすることで、1つまたは複数のプロセスまたはスレッドのCPUアフィニティを設定することが可能です。**sched_setaffinity(2)**もCPUアフィニティを設定するために使用することが可能です。

CPUアフィニティを設定するために以下の条件を満足する必要があります。

- 有効な呼び出し元プロセスのユーザーIDは、CPUアフィニティを設定しようとしている登録されているプロセスのユーザーIDと一致していなければならない、もしくは、
- 呼び出し元プロセスはCAP_SYS_NICEケーパビリティを持っている、またはルートでなければなりません

CPUにプロセスまたはスレッドのCPUアフィニティを追加するため、呼び出し元プロセスはCAP_SYS_NICEケーパビリティを持っている、またはルートでなければなりません。

CPUアフィニティは **init(8)**プロセスに割り当てることが可能です。すべての一般的なプロセスは **init**の子プロセスです。結果、一般的なプロセスのほとんどは **init** のCPUアフィニティ、もしくは **init**のCPUアフィニティの一部のCPUと同じCPUアフィニティになるはずです。(前述の)特権を持ったプロセスだけはCPUをそれらのCPUアフィニティに加えることができます。**init**への制限されたCPUアフィニティの割り当ては、すべての一般的なプロセスを **init**と同じCPUサブセットへ制限します。その例外は、適切なケーパビリティを明示的に修正したCPUアフィニティを持つプロセスです。もし**init**のCPUアフィニティを変更したいと考えるのであれば、「initへのCPUアフィニティ割り当て」セクション以下の説明を参照してください。

mpadviseライブラリ・ルーチンは「mpadviseを使ったマルチプロセッサ制御」セクション以下および**mpadvise(3)**のmanページに記述されています。**run**コマンドは4章内の「runコマンド」セクションおよび**run(1)**のmanページに記述されています。**sched_setaffinity(2)**と**sched_getaffinity(2)**に関する情報については、**sched_affinity(2)**のmanページを参照してください。

mpadviseを使ったマルチプロセッサ制御

mpadvise(3)は、様々なマルチプロセッサーの機能を実行します。CPUは 1つまたはそれ以上のCPUの組み合わせを定義した`cpuset_t` オブジェクトのポインターを指定することにより識別されます。CPUの設定に関する詳細な情報については**cpuset(3)**のmanページを参照してください。

概要

```
#include <mpadvise.h>
int mpadvise (int cmd, int which, int who, cpuset_t *setp)
gcc [options] file -lccur_rt ...
```

情報コマンド

以下のコマンドはシステム内のCPUに関する情報を取得または設定します。*which*と*who*のパラメータは無視されます。

MPA_CPU_PRESENT	システム内に物理的に存在するCPUを示すマスクを返します。**cpu(1)** コマンドによってダウンしたCPUは含まれたままとなります。
MPA_CPU_ACTIVE	バックプレーンの中にいくつ存在するかは関係なく、初期化され動作可能なCPUを示すマスクを返します。**cpu(1)**コマンドによってダウンしたCPUは含まれません。

MPA_CPU_BOOT　システムをブートしたCPUを示すマスクを返します。ブートCPUは他のCPUと共有しないいくつかの責務を持っています。

MPA_CPU_LMEMス　NUMA対応システムにおいてローカル・メモリを持つCPUを示すマスクを返します。**cpu(1)**コマンドによってダウンしたCPUは含まれたままとなります。

制御コマンド

以下のコマンドは、プロセス、スレッド、プロセス・グループもしくはユーザーによるCPU利用の制御を提供します。

MPA_PRC_GETBIAS (MPA_PID)、　指定されたプロセス内の全スレッドのCPUアフィニティもしくは指定されたスレッドに対する正確な独自バイアス(MPA_TID)に対応するCPUセットを返します。

MPA_PRC_SETBIAS (MPA_PID)、　指定されたプロセス内の全スレッドのCPUアフィニティもしくは指定されたスレッドの独自CPUアフィニティ(MPA_TID)を指定されたcpusetへ設定します。プロセスのCPUアフィニティを変更するため、現在のユーザーが **CAP_SYS_NICE**ケーパビリティを持っていない限り、有効なユーザーIDは(**exec(2)**から)登録されたプロセスのユーザーIDと一致しなければなりません。

MPA_PRC_GETRUN　指定されたスレッドが現在実行中(または実行待機中)のCPUと一致する正確なCPUを含むCPUセットを返します(MPA_TID)。MPA_PIDが指定されるとき、非スレッドプ・ログラムのCPU1つおよびマルチスレッド・プログラムの全スレッドが使用するCPU一式を返します。この値が返される頃には、CPU割り当てが既に変わっている可能性があることに注意してください。

***which*と*who* の使用方法**

which　選択基準を指定するために使用します。以下の1つを指定可能。

MPA_PID	特定のプロセスおよびその全スレッド
MPA_TID	特定のスレッド
MPA_PGID	プロセス・グループ
MPA_UID	ユーザー
MPA_LWPID	MPA_TIDと同様 (PowerMAXと互換)

who　*which* に関連する説明：

プロセスID
スレッドID
プロセス・グループID
ユーザーID

*who*の値が0である場合、呼び出し元のプロセスID、プロセス・グループID、ユーザーID が使用されます。

単独の従属(非原始)スレッドで のMPA_PIDの使用は、原始スレッドが提供されたときの処理を含んでいるプロセスを適用します。

MPA_TIDを使用する時、*who*は(gettid が返す)スレッドIDの数値でなければならず、POSIXスレッド・ライブラリに関連するpthreadのIDではありません。

initへのCPUアフィニテ割り当て

一般的な全てのプロセスは **init(8)**の子プロセスです。既定値で **init**はシステム内の全てのCPUを含むマスクを所有し、それらのCPUアフィニティの修正が可能な特殊なケーパビリティを持つ唯一の選ばれたプロセスです。もしそれがCPUのサブセットに制限された既定の全プロセスによって要求された場合、CPUアフィニティは特権を持つユーザーによって **init**プロセスへ割り付けることが可能です。この目的を達成するため、**run(1)**コマンドはシステム初期化プロセスの中で初期段階で呼び出すことが可能です。

例えば、**init**とその全ての子プロセスをCPU1,2,3へ割り付けるために以下のコマンドをシステム初期化(**inittab(5)**を参照)の初期段階で呼ばれる **/etc/rc.sysinit**スクリプトの最後に追加することが可能です。**init**プロセスはこのコマンドの中では常に1であるプロセスIDを用いて指定しています。

```
/usr/bin/run -b 1-3 -p 1
```

同じ効果が**shield(1)**コマンドを利用することにより得ることが可能です。このコマンドを利用する利点は、どのランレベルであってもコマンドラインから実行できることです。**shield**コマンドはシールドされたCPU上で既に動作しているプロセスの移動を処理します。更に**shield**コマンドを使い異なるシールドのレベルを指定することも可能です。本コマンドの詳細な情報については、「shieldコマンド」セクションまたは**shield(1)**のmanページを参照してください。

例えば、動作中のプロセスからCPU0をシールドするためには、以下のコマンドを実行します。

```
$ shield -p 0
```

CPUをシールドした後、常にシールドCPUで指定したプロセスを動作させるために**run**コマンドを利用します。

例えば、予めプロセスからシールドしたCPU0上で **mycommand**を実行するためには、以下のコマンドを実行します。

```
$ run -b 0 ./mycommand
```

シールドCPUの設定例

以下の例は、RCIMのエッジトリガー割り込みに対する最高の割り込み応答を保証するためのシールドCPUの使い方を示します。つまり、この目的はRCIM上で発生したエッジトリガー割り込み時にユーザー・レベルプロセスが起き上がるために必要な時間の最適化、およびプロセスが起き上がるときのためにデターミニスティックな実行環境を提供することです。この場合、シールドCPUはRCIMの割り込み処理とその割り込みに応答するプログラムだけを設定しなければなりません。

最初のステップは、**shield(1)**コマンドを通してシールド・プロセッサから割り込みを切り離す指示をします。最高の割り込み応答を得るためにローカル・タイマー割り込みは無効にし、バックグラウンド・プロセスは除外します。

CPU1がそれらの結果を達成するためのshieldコマンドは、

```
$ shield -a 1
```

現段階でシールドされたCPU１上の割り込みは無し、実行を許可されたプロセスもありません。CPUのシールド状況は以下の方法を利用することで確認することが可能です。

shield(1)コマンド：

```
$ shield -c
CPUID            irqs      ltmrs     procs
----------------------------------------
      0         no         no         no
      1         yes        yes        yes
      2         no         no         no
      3         no         no         no
```

cpu(1)コマンド：

```
$ cpu
cpu chip core ht ht-siblings state shielding
--- ---- ---- -- ---------- ----- ---------
0   0    -    0  2          up    none
1   3    -    0  3          up    proc irq ltmr
2   0    -    1  0          up    none
3   3    -    1  1          up    none
```

または **/proc**ファイル・システム：

```
$ cat /proc/shield/irqs
00000002
```

これは全ての割り込みがCPU1上での実行が不可能であることを表します。この例の中で、目的はシールドされたCPU上で特定の割り込みに応答することであり、CPU1にRCIMの割り込みの割り付けとCPU1上で割り込みに応答するプログラムの実行を許可する必要があります。

最初のステップはRCIMの割り込みが割り付けられたIRQを判断することです。この割り込みとIRQ間の割り当てはマザーボード上のデバイスおよび特定のPCIスロットのPCIデバイスによって変わることはありません。もしPCIボードが新しいスロットへ移動した場合はそのIRQの割り付けは変わるかもしれません。所有するデバイスのIRQを見つけるために以下のコマンドを実行します。

```
$ cat /proc/interrupts
CPU0       CPU1       CPU2       CPU3
0:     665386907          0          0          0   IO-APIC-edge  timer
4:          2720          0          0          0   IO-APIC-edge  serial
8:             1          0          0          0   IO-APIC-edge  rtc
9:             0          0          0          0   IO-APIC-level  acpi
14:       9649783          1          2          3   IO-APIC-edge  ide0
15:            31          0          0          0   IO-APIC-edge  ide1
16:     384130515          0          0          0   IO-APIC-level  eth0
17:             0          0          0          0   IO-APIC-level rcim,Intel
18:      11152391          0          0          0   IO-APIC-level aic7xxx
19:             0          0          0          0   IO-APIC-level uhci_hcd
23:             0          0          0          0   IO-APIC-level uhci_hcd
```

```
NMI:     102723410  116948412          0          0   Non-maskable  interrupts
LOC:     665262103  665259524  665264914  665262848   Local interrupts
RES:      36855410   86489991   94417799   80848546   Rescheduling interrupts
CAL:          2072       2074       2186       2119   function call interrupts
TLB:         32804      28195      21833      37493   TLB shootdowns
TRM:             0          0          0          0   Thermal event interrupts
SPU:             0          0          0          0   Spurious interrupts
ERR:             0          0          0          0   Error interrupts
MIS:             0          0          0          0   APIC errata fixups
```

上記リストの中では、RCIMはIRQ17に割り当てられています。IRQ番号が分かったら、RCIMの割り込みをIRQ17のアフィニティ・マスクを表す **/proc**ファイルを通してシールド・プロセッサへ割り付けることが可能です。IRQのアフィニティ・マスクは8桁の16進数ASCII値です。その値はCPUのビット・マスクになっています。マスク内に配置された各ビットは、この割り込みのための割り込みルーチンが処理されるかもしれないCPUを表します。各々設定されたビットの位置は割り込み処理が可能な論理CPUの番号です。以下のコマンドはCPU1にIRQ17のCPUアフィニティ・マスクを設定します。

```
$ echo 2 > /proc/irq/17/smp_affinity
```

IRQのための “**smp_affinity**” ファイルは、ルート・ユーザーだけがIRQの割り込みの割り付けが変更可能なパーミッションでデフォルトでインストールされていることに注意してください。IRQアフィニティのための **/proc**ファイルは変更が実施されたことを確認するために読み取ることも可能です。

```
$ cat /proc/irq/17/smp_affinity
00000002 user 00000002 actual
```

“user” で返された値は、IRQのCPUアフィニティのためにユーザーより指定されたビット・マスクであることに注意してください。“actual” で返された値は、存在しないCPUやシールドされたCPUがマスクから取り除かれた後に生じたアフィニティとなります。もしユーザーがシールドCPU/非シールドCPUの両方を含むアフィニティ・マスクを設定した場合、シールドCPUはIRQのアフィニティ・マスクから取り除かれただけとなることに注意してください。これは、もし割り込みを処理できるIRQのアフィニティ・マスクの中に非シールドCPUが存在しなければ、割り込みからシールドされたCPUは割り込みを処理するだけだからです。この例では、CPU1は割り込みからシールドされますが、このアフィニティ・マスクはCPU1だけが割り込みの処理を許可されたことを示すため、CPU1はIRQ17を処理します。

次のステップは、RCIMのエッジトリガー割り込みに応答するプログラムがシールドされたプロセッサー上での実行を確認することです。システム内の各プロセスはCPUアフィニティ・マスクが割り付けられています。バックグラウンド・プロセスからCPUをシールドするため、シールドされたCPUだけを指定したCPUアフィニティ・マスクを持つプロセスだけがシールドされたプロセッサー上で実行することを許可します。もしプロセスのアフィニティ・マスクの中にいくつかの非シールドCPUが存在する場合、そのプロセスは非シールドCPU上でのみ実行されることに注意してください。

以下のコマンドは、ユーザープログラム “edge-handler” をCPU1上でリアルタイム優先度で実行することを強制します。

```
$ run -s fifo -P 50 -b 1 edge-handler
```

プログラムは「mpadviseを使ったマルチプロセッサ制御」セクションで説明されている **mpadvise(3)**ライブラリ・ルーチンの呼び出しによって自身のCPUアフィニティを設定できることに注意してください。**run(1)**コマンドはプログラムのアフィニティをチェックするために利用できます。

```
$ run -i -n edge-handler
Pid   Tid  Bias Actual Policy Pri Nice Name
9326  9326 0x2  0x2    fifo   50  0    edge-handler
```

"Bias" が返す値は、ユーザーによって指定されたプロセスのCPUアフィニティのビット・マスクであることに注意してください。"actual" が返す値は、存在しないCPUやシールドされたCPUがマスクから取り除かれた後に生じたアフィニティとなります。もしユーザーがシールドCPU/非シールドCPUの両方を含むアフィニティ・マスクを設定した場合、シールドCPUはプロセスのアフィニティ・マスクから取り除かれただけとなることに注意してください。これは、もしプログラムを実行できるプロセスのアフィニティ・マスクの中に非シールドCPUが存在しなければ、バックグラウンド・プロセスからシールドされたCPUはプロセスを実行するだけだからです。この例では、CPU1はバックグラウンド・プロセスからシールドされますが、このアフィニティ・マスクはCPU1だけがプログラムの実行を許可されたことを示すため、CPU1は "edge-handler" プログラムを実行します。

デターミニズムを高める手順

以下のセクションでは、以下のテクニックを使ってパフォーマンスの向上が可能な様々な方法を説明します。

- メモリ内のプロセスのページをロック
- 適切な静的優先度割り付けの利用
- 割り込みレベルから非クリティカルな処理を排除
- 迅速なプロセスの起床
- キャッシュ・アクセスの制御
- 物理メモリの予約
- NUMAシステムにおいてプログラムからローカル・メモリへのバインド
- ハイパースレッドの慎重利用
- 低メモリ状態の回避

メモリのページをロック

オーバーヘッドに結びつくページングやスワッピングは **mlockall(2)**, **mlockall_pid(2)**, **munlockall(2)**, **munlockall_pid(2)**, **mlock(2)**, **munlock(2)**を使うことにより回避することが可能です。これらのシステムコールは物理メモリ内のプロセスの仮想アドレスの全てまたは一部をロックおよびアンロックすることを許可します。これらのインターフェースはIEEE規格1003.1b-1993に準拠しています。

これらの各コールにより、コール時点で常駐していないページはメモリに断層が生じてロックされます。**mlockall(2)**, **mlockall_pid(2)**, **munlockall(2)**, **munlockall_pid(2)**, **mlock(2)**, **munlock(2)** システムコールを使うには**CAP_IPC_LOCK**ケーパビリティを所有していなければなりません。**mlockall_pid(2)**については、呼び出し元プロセスのユーザーIDがターゲットプロセスのユーザーIDと一致しない場合、**CAP_SYS_NICE**ケーパビリティも必要となる可能性があります。ケーパビリティに関する追加の情報は13章と **pam_capability(8)**のmanページを参照してください。

メモリをロックするシステム・サービス・コールはプロセスが自分自身のアドレス空間をロックまたはアンロックする方法として提供するのに対し、**run**コマンドは更に**--lock**オプションにより他のプロセスのアドレス空間をロックまたはアンロックする機能を提供します。様々なページをロックするシステムコールを利用するための手順は、対応するmanページの中で全て説明されています。**--lock**オプションは **run(1)**のmanページの中で説明されています。

プログラム優先度の設定

RedHawk Linuxカーネルは静的優先度スケジューリングを提供します — つまり、特定のPOSIXスケジューリング・ポリシーでスケジュールされたプロセスは、実行時の動作に応じてオペレーティング・システムにより優先度が変更されることはありません。

POSIXリアルタイム・スケジューリング・ポリシーの1つによりスケジューリングされたプロセスは常に静的優先度の状態です(リアルタイム・スケジューリング・ポリシーとはSCHED_RRおよびSCHED_FIFO：これらは4章で説明されています)。プロセスのスケジューリング・ポリシーを変更するには、**sched_setscheduler(2)**と**sched_setparam(2)**のシステムコールを利用することが可能です。プロセスの優先度をより高い(より有利な)値に変更するためにこれらのシステムコールを利用するには、CAP_SYS_NICEケーパビリティを持っていなければならないことに注意してください(これらのルーチンを利用するためのケーパビリティ条件に関する全ての情報は、対応するmanページを参照してください)。

特定CPU上で実行中の最高優先度のプロセスは、最高のプロセス・ディスパッチ・レイテンシーとなります。もし、あるプロセスがCPU上で実行している他のプロセスよりも低い優先度が割り付けられている場合、このプロセス・ディスパッチ・レイテンシーは高い優先度のプロセスが実行に費やす時間に影響されます。よって、優れたプロセス・ディスパッチ・レイテンシーを必要とする1つ以上のもプロセスが存在する場合、いくつかのCPUにそれらのプロセスを分散することを推奨します。特定CPUへのプロセスの割り付け方法については、「CPUへのプロセス割り当て」セクションを参照してください。

プロセスのスケジューリングは4章ですべて説明されています。プロセスの優先度を変更するための**sched_setscheduler**および**sched_setparam**システムコールの利用手順についても説明されています。

遅延割り込み処理の優先度設定

Linuxは割り込みレベルで別に行われた処理を遅延するために割り込みルーチンで使用されるいくつかのメカニズムをサポートしています。デバイス割り込みを処理するためのその処理は2つの役割に分けます。最初の役割は割り込みレベルで実行し、割り込み完了処理の最もクリティカルな側面のみ処理します。2つ目の役割はプログラムレベルで実行するために遅延します。割り込みレベルからの非クリティカル処理を排除することにより、本セクション「割り込みの影響」の最初に説明されているようにシステムはより良い割り込み応答時間を得ることが可能となります。

割り込みルーチンの2番目の役割はカーネル・デーモンに処理され、デバイス・ドライバーで使用される割り込みを遅延する技法次第となります。システム管理者に許可された遅延した割り込み処理を扱うカーネル・デーモンの優先度を設定するためのカーネル・チューニング・パラメータが存在します。リアルタイム・タスクが遅延した割り込みを処理しているCPU上で実行する時、遅延した割り込みカーネル・デーモンの優先度を設定することが可能となり、高優先度のユーザープロセスは遅延した割り込みカーネル・デーモンよりも更に有利な優先度を所有します。これはこのリアルタイム・プロセスのために追加のデターミニスティックな応答時間を可能にします。

割り込み処理の遅延、カーネル・デーモン、カーネル・チューニング・パラメータに関する詳細な情報については、「デバイス・ドライバ」章を参照してください。

別プロセスの起床

マルチプロセス・アプリケーションでは、多くの場合に特定のタスクを実行するためにプロセスを起こす必要があります。システムの応答性の1つの基準は、1つのプロセスが別のプロセスを起こすことができる速度です。他のタスクへの切り替えを実行するために使用できる最速の方法は、**postwait(2)**システムコールを使用することです。レガシー・コードとの互換性のために**server_block(2)**と**server_wake1(2)**の機能がRedHawk Linuxにて提供されます。

これらの機能を使用する方法は5章の中で説明されています。

キャッシュ・スラッシングの回避

アプリケーションが異なるCPU上で複数の実行スレッド間で共有されるアドレス空間の一部を所有する場合、あるスレッドに頻繁に使用される変数(例えば *i*)と、それとは別のスレッドに頻繁に使用される変数(例えば *j*)が、同じキャッシュ・ラインに配置されているメモリ内に互いに接近して配置されないことを確保することが重要です。もし *i* と *j* が同じキャッシュ・ラインに配置されている場合、それぞれのスレッドにより *i* と *j* への参照が行われる時にそのキャッシュ・ラインは2つのCPU間であわただしく動くことになり、キャッシュ性能が低下します。

逆に1つのスレッドが頻繁に複数の変数(例えば *i, j, k*)を使う場合、同じキャッシュ・ラインに *i, j, k* を配置しようとすることがむしろ望ましいのです。同じキャッシュ・ラインに *i, j, k* が配置されている場合、*i, j, k* のいづれかを参照する時に3つの変数全てが余計な性能低下なしに利用可能となります。

配列を使用するアプリケーションは更なる制約があり、配列のサイズをシステムのキャッシュ・サイズと比較する方法を理解することが重要となります。例えば、配列が1.2Mbyteのメモリを必要とするのにシステムがたった1Mbyteのキャッシュを提供する場合、キャッシュ内で完全に実行される配列を持つことの利点を得ること無く、配列操作は他のどの変数もキャッシュの利用から完全に除外します。この場合の唯一の解決策は、より大きなキャッシュを持つシステムを購入するか、より小さな配列を使用できるように配列を使用するアルゴリズムを再設計することです。

今日の殆どのシステムはNUMAシステムであることに注意して下さい。NUMAシステム上のCPUはグループに編成され、各グループはいくつかの(通常、非キャッシュ)ローカル・メモリが利用可能となります。データがキャッシュ内に無くメモリから読む必要がある時、メモリ操作が高速かつ最もデターミニスティックな状態となるように同一NUMAノード・グループのCPU上で大量のデータを共有するどの実行スレッドも実行されることを確保することが重要となります。

もう一つのNUMAシステムの重要な特徴は、各NUMAノードは通常ローカルIOバスを持っているということです。一般的にシステム・デバイス(例えば、ディスク、CD/DVDドライブ、ネットワーク・カード他)は特定のNUMAノードに対してはローカルとなり、他のNUMAノードのCPU上で実行しているスレッドに対してはリモートとなります。任意のシステムにとっては、どのNUMAノードをどのIOデバイスと連動させるかを決定するために役に立ちます。ディスクを集中的に使用するスレッドは、ディスク・コントローラに属するNUMAノード内のCPU上において最高パフォーマンスで実行されることとなります。ネットワークを集中的に使用するスレッドは、ネットワーク・コントローラに属するNUMAノード内のCPU上において最高パフォーマンスで実行されることとなります。

システムを購入または構成する時、どのNUMAノードがハードウェア上のどのデバイスに属しているかを理解することは重要となります。お手持ちのアプリケーションのリソースの使用形態を理解することもまた重要となります。例えば、ディスクとネットワークを集中的に使用するアプリケーションを所有している場合、パフォーマンスを最適化するためにネットワーク・コントローラとディスク・コントローラの両方に属するNUMAノードを持つハードウェアを選択します。

物理メモリの予約

物理メモリは**/boot/grub2/grub.cfg**ファイル内のコマンドライン引数の利用することにより予約することが可能です。

このタイプの割り当ては、ローカル・デバイスで必要となるDMAバッファまたは PCI-to-VMEアダプタのようなものを介してiHawkのメモリにマッピングされた外部ホストのデバイスに利用することが可能です。それは動的な仮想メモリ・アロケーションが提供するページ・アロケーションのランダム性を持たないデータ空間を提供するために使用することが可能です。これは大きなデータ配列のキャッシュ衝突を一定以上にすることでアプリケーションの性能を向上させ、連続したプロセスの実行による実行時間の不一致を減らします。

grub.cfgファイル内でのメモリのカスタム・マッピングによって、RAMの予約された区域を獲得することが可能です。System Vの**shmop(2)**の機能は物理メモリのこの区域へのアクセスに使用することが可能です。**shmconfig(1)**, **shmbind(2)**, **shmop(2)**の各機能はこのメモリ区域の作成およびアタッチに利用することが可能です。

利用可能な物理RAMの量は、i386上では以下に示すように **/proc/iomem**の内容を調べることにより見ることが可能です。

```
$ cat /proc/iomem

00000000-0009ffff : System RAM
  00000000-00000000 : Crash kernel
000a0000-000bffff : Video RAM area
000c0000-000cefff : Video ROM
000d0800-000d3fff : Adapter ROM
000f0000-000fffff : System ROM
00100000-7fe8abff : System RAM
  00100000-004f58a5 : Kernel code
  004f58a6-00698577 : Kernel data
7fe8ac00-7fe8cbff : ACPI Non-volatile Storage
7fe8cc00-7fe8ebff : ACPI Tables
7fe8ec00-7fffffff : reserved
        :
                    (このサンプルからI/O情報は削除しています)
```

“System RAM” と記述された箇所は、割り付け可能な物理メモリを表しています。

物理RAMを予約する方法を説明する**/boot/grub2/grub.cfg**の例を16進数で以下に示します (10進数での例は後に続きます) 。**grub.cfg**に設定されたコマンドは、メモリ・マッピングを作成するために起動時に処理されます。

“memmap=exactmap” エントリは正確なBIOSマップが使用されることを指定します。

残りのエントリは領域を定義するために指定します。そのコマンドの書式は、

memmap=*size*<*op*>*address*

<*op*>の場所にシステムRAMは ‘@’、予約は ‘$’、ACPIは ‘#’ を指定します。RedHawk 7において‘$’は予約語となりますので、'¥'(英語環境ではバックスラッシュ)を追加して「¥$」として下さい。

以下の例では、丁度1Gのアドレスより下に32MBを予約します。

```
default=0
timeout=10
splashimage=(hd0,0)/grub/ccur.xpm.gz
```

```
title RedHawk Linux 6.3.3 (Trace=Yes, Debug=No)
    root (hd0,0)
    kernel /vmlinuz-3.5.7-RedHawk-6.3.3-trace ro root=/dev/sda2 vmalloc=256M ¥
    memmap=exactmap ¥
    memmap=0xa0000@0x0 ¥
    memmap=0x3df00000@0x100000 ¥
    memmap=0x2000000¥$0x3e000000 ¥
    memmap=0x3fe8ac00@0x40000000 ¥
    memmap=0x2000#0x7fe8cc00
```

grubコマンド行は256byteに制限されることに注意する必要があります。grubコマンド行へのパラメータ追加はこの制限を越えてはいけません。

上記のエントリは**memexact(1)**ユーティリティを使って取得し、その後**/boot/grub2/grub.cfg**コマンド行へコピーされます。**memexact**はコマンド・オプションを処理し、**/proc/iomem**の内容もしくはコマンド行で指定されたファイルに準じて適切なメモリ予約コマンドを実行します。

```
# /usr/bin/memexact -x -MS=32M,U=1G
memmap=exactmap memmap=0xa0000@0 memmap=0x3df00000@0x100000 memmap=0xa0000@0
memmap=0x3df00000@0x100000 memmap=0x2000000¥$0x3e000000
memmap=0x3fe8ac00@0x40000000 memmap=0x2000#0x7fe8cc00
```

オプション：

-x	16進数での出力を指定します
-M	複数のエントリを陳列します
-S	予約サイズを指定します
-U	予約の上限を指定します

この領域の予約は、**/proc/iomem**内の "System RAM" で確認できるメモリの領域、かつカーネルのアドレスを含んでいない限り任意に選ぶことが可能です。"Adapter ROM"、 "System ROM"、 "ACP"、 "reserved" の各領域は、これらのコマンドを使って再マップしてはいけません。**memexact(1)**は**/proc/iomem**およびコマンド行オプションで与えられた内容に準じて予約するために適切な領域を選びます。

CAUTION

これらのエントリの中で発生した既に予約された領域(例. System RAM等)の重複のようなエラーは、カーネルの起動に致命的なエラーの原因となる可能性があります。

以下の例は10進数のアドレスを使用します。これは上述の16進数の例と同一の機能で同一の結果を生じます。

```
# memexact -MS=32M,U=1G
memmap=exactmap memmap=640K@0 memmap=991M@1M memmap=32M¥$992M
memmap=1047083K@1G memmap=8K#2095667K
```

以下は、記10進数のエントリが追加された **grub.cfg**ファイルに相当します。

```
default=0
timeout=10
splashimage=(hd0,0)/grub/ccur.xpm.gz
title RedHawk Linux 6.3.3 (Trace=Yes, Debug=No)
    root (hd0,0)
    kernel /vmlinuz-3.5.7-RedHawk-6.3.3-trace ro root=/dev/sda2 vmalloc=256M ¥
    memmap=exactmap ¥
    memmap=640K@0 ¥
    memmap=991M@1M ¥
```

```
memmap=32M¥$992M ¥
memmap=1047083K@1G ¥
memmap=8K#2095667K
```

以下は上述の例が予約を実行する前後のメモリの比較です。「予約後」の“reserved”と記述された 0x3e000000の領域は新たに予約された32MBの領域です。

予約前の **/proc/iomem**	予約後の **/proc/iomem**

00000000-0009ffff : System RAM
  00000000-00000000 : Crash kernel
000a0000-000bffff : Video RAM area
000c0000-000cefff : Video ROM
000d0800-000d3fff : Adapter ROM
000f0000-000fffff : System ROM
00100000-7fe8abff : System RAM
  00100000-004f58a5 : Kernel code
  004f58a6-00698577 : Kernel data
7fe8ac00-7fe8cbff : ACPI Non-volatile Storage
7fe8cc00-7fe8ebff : ACPI Tables
7fe8ec00-7fffffff : reserved
  .
  .
	00000000-0009ffff : System RAM
	  00000000-00000000 : Crash kernel
	000a0000-000bffff : Video RAM area
	000c0000-000cefff : Video ROM
	000d0800-000d3fff : Adapter ROM
	000f0000-000fffff : System ROM
	00100000-3dffffff : System RAM
	  00100000-004f58a5 : Kernel code
	  004f58a6-00698577 : Kernel data
	3e000000-3fffffff : reserved
	40000000-7fe8abff : System RAM
	7fe8cc00-7fe8ebff : ACPI Tables
	  .
	  .

(このサンプルからI/O情報は削除しています)	

次の例は、x86_64システム上の4GBを超える2つのシステムメモリ領域の間でメモリ領域を予約するために**grub.cfg**の中に設定するコマンドを説明します。予約前の**/proc/iomem**の出力を以下に示します。

x86_64システムにおいて“mm”は“memmap”の別名で“ex”は“exactmap”の別名であることに注意してください。これらの短い別名は、grubコマンド行毎におよそ256文字の制限があるため、予約エリアを設定するのに必要な文字数を減らすために使用する必要があります。

```
mm=ex ¥
mm=0x9fc00@0x0 ¥
mm=0x400@0x9fc00 ¥
mm=0x20000$0xe0000 ¥
mm=0xcfef0000@0x100000 ¥
mm=0x10000#0xcfff0000 ¥
mm=0x840000¥$0xff7c0000 ¥
mm=512M@0x100000000 ¥
mm=512M$4608M ¥
mm=1G@5G
```

以下は上述の例が予約を実行する前後のメモリの比較です。「予約後」の“reserved”と記述された 0x0000000120000000 の領域は新たに予約された領域です。

予約前の**/proc/iomem**	予約後の**/proc/iomem**
0000000000000000-000000000009fbff : System RAM 000000000009fc00-000000000009ffff : reserved 00000000000a0000-00000000000bffff : Video RAM area 00000000000c0000-00000000000c7fff : Video ROM 00000000000c8000-00000000000cbfff : Adapter ROM 00000000000f0000-00000000000fffff : System ROM 0000000000100000-00000000d7feffff : System RAM 0000000000100000-00000000005c9521 : Kernel code 00000000005c9522-0000000000954137 : Kernel data 00000000d7ff0000-00000000d7ffefff : ACPI Tables 00000000d7fff000-00000000d7ffffff : ACPI Non-volatile Storage 00000000ff7c0000-00000000ffffffff : reserved 0000000100000000-000000017fffffff : System RAM . .	0000000000000000-000000000009fbff : System RAM 000000000009fc00-000000000009ffff : System RAM 00000000000a0000-00000000000bffff : Video RAM area 00000000000c0000-00000000000c7fff : Video ROM 00000000000c8000-00000000000cbfff : Adapter ROM 00000000000f0000-00000000000fffff : System ROM 0000000000100000-00000000cffeffff : System RAM 0000000000100000-00000000005c9521 : Kernel code 00000000005c9522-0000000000954137 : Kernel data 00000000cfff0000-00000000cfffffff : ACPI Tables 00000000ff7c0000-00000000ffffffff : reserved 0000000100000000-000000011fffffff : System RAM 0000000120000000-000000013fffffff : reserved 0000000140000000-000000017fffffff : System RAM . .
(このサンプルからI/O情報は削除しています)	

shmconfig(1)または**shmbind(2)**は予約済み物理メモリにパーティションを作成するために使用されます。System Vの共有メモリ操作**shmop(2)**はアクセス領域を増やすアプリケーションで使用することが可能です。

以下の例は、本セクション内で提示したi386システム上の最初の例に基づき、物理アドレス0x3e000000にアクセス制限なしおよび6602のキーによる32MBのSystem Vメモリ・パーティションを作成します。

```
#  usr/bin/shmconfig -s 0x2000000 -p 0x3e000000 -m 0777 6602
```

このコマンドは共有メモリパーティションの作成を自動化するために **/etc/rc.local**へ設定しても構いません。この例では符号化された6602のキーを使うと同時に、キーとして**/dev/***MyDevice* のようなパスを使用して、**ftok(3)**の機能を使ってアタッチするために使うキーを得ることをアプリケーションに許可します。

以下のコードの断片は動的に共有メモリ・パーティションを作成するために使用することも可能です。

```
.
.
paddr = 0x3e000000 ;
shmkey = ftok( pathname ) ;
shmid = shmget ( shmkey, sizeof ( <shared_region> ) ,
                      SHM_R | SHM_W | IPC_CREAT) ;
shmstat = shmbind ( shmid , paddr ) ;
pstart = shmat ( shmid , NULL , SHM_RND ) ;
.
.
```

システム上の共有メモリ・セグメントは **ipcs(8)** (-m オプション) を使って、もしくは**/proc/sysvipc/shm**ファイルを通して見ることが出来ます。

```
#  cat /proc/sysvipc/shm
```

```
  key  shmid          perms      size   cpid  lpid   nattch  uid  gid  cuid  cgid
atime  dtime          ctime  physaddr
 6602      0            777  33554432   4349     0        0    0    0     0     0
    0      0     1153750799  3e000000

# ipcs -m
------ Shared Memory Segments --------
         key    shmid     owner     perms       bytes     nattch     status
  0x000019ca        0      root       777    33554432          0
```

これらの機能やユーティリティの利用に関する詳細な情報については、manページまたは3章を参照してください。

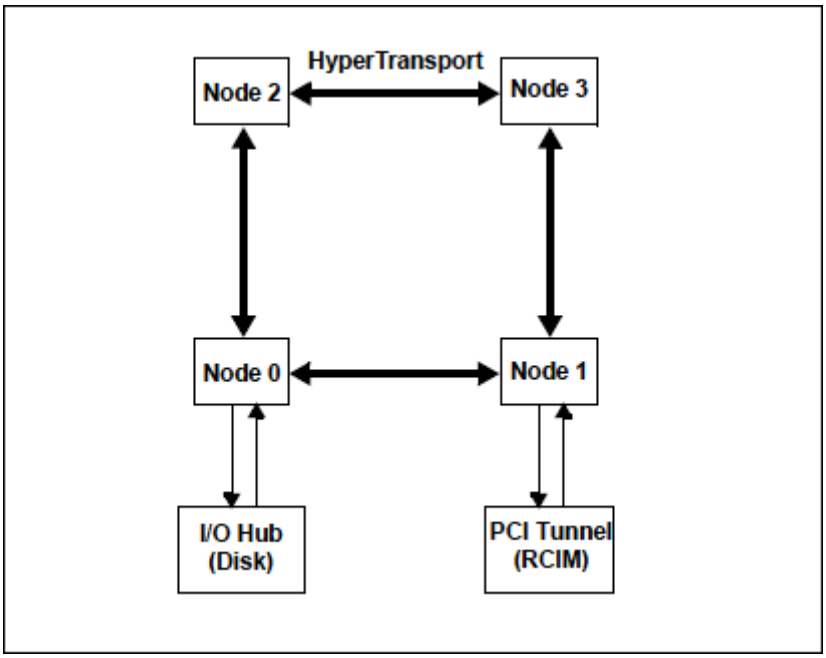
NUMAノードへのバインディング

iHawk Opteronシステムのような<u>N</u>on-<u>U</u>niform <u>M</u>emory <u>A</u>ccess (NUMA)のシステム上では、ほかのシステムよりも一部のメモリ領域へのアクセスに時間が掛かります。NUMAシステム上のメモリはノードに分けられ、ノードはメモリ領域とNUMAノードのメモリ領域と同じ物理バス上に存在する全てのCPUで定義されます。もしこのタイプのシステム上で実行中のプログラムがNUMA対応でない場合、十分に実行することが出来ません。
デフォルトで、ページは(プログラムが実行された)ローカルCPUに存在するノードから割り付けられますが、タスクまたはタスク内の仮想領域は最良のデターミニズムと制御のために特定のノードからのページ割り付けを指定することが可能です。NUMAに関する情報は10章を参照してください。

クアッドOpteronシステムのI/Oスループット

クアッドOpteron対称型マルチプロセッサ・システムにおいて、各プロセッサはプロセッサに直結するユニークなメモリ・バンクを持っています。システム内の全てのメモリはHyperTransport™ を介してどのCPUからもアクセスすることが可能ですが、プロセッサに直結したメモリはそのプロセッサ上で動作している実行スレッドに最速のアクセス時間を提供します。このレイアウトを図2-7に図示します。

図2-7 クアッドOpteronのI/Oスループット・レイアウト



Opteronシステムでの I/Oデバイスへのアクセスも同様に完全な対象型ではありません。I/O HubとPCI Tunnelは、システム内の特定ノードに直結しています。図2-7の中でI/O Hub はNode 0に接続し、PCI Tunnel はNode 1 に接続しています。試験においてプログラム化されたI/Oの時間は、プログラム化したI/Oを実行しているプログラムがデバイスが存在するI/Oバスへ接続したノード上で実行している時、最速かつよりデターミニスティックであることを示しました。I/O性能に対するこの影響は、ほかのプログラムがI/Oまたは非ローカルメモリ操作を実行中であるため、HyperTransport相互接続により競合するときに特に目立ちます。

これは、アプリケーションがデータミニスティックな高速プログラム化I/Oを要求する場合、そのようなI/Oを実行しているそのプログラムは、デバイスが存在するI/Oバスに最も近いプロセッサ上で実行せざるを得ないことを意味します。

I/Oブリッジに結合するノードはシステム構成図を見ることにより、もしくは試験することにより明らかにすることが可能です。

ハイパースレッディングの理解

ハイパースレッディングは、iHawk 860システムのIntel Pentium Xeonプロセッサの機能です。これは1つの物理プロセッサでアプリケーション・ソフトウェアの複数スレッドの同時実行を可能にします。これはプロセッサの実行リソース一式を共有すると同時に各プロセッサ上に2つの構造形態を持つことによって実現します。この構造形態はプログラムまたはスレッドの流れを追跡し、実行リソースは(Add, Multiply, Load等の)作業をするプロセッサ上のユニットになります。ハイパースレッド付き物理CPUの2つの各々の構造形態は、“論理”CPUとして考えることが可能です。“Sibling CPU”という言葉は、同一物理CPU上に存在する1対の論理CPUとは別のCPUを指します。

スレッドがスケジューリングされた時、オペレーティング・システムは1つの物理CPU上の2つの論理CPUをあたかも別個のプロセッサのように扱います。**ps(1)**もしくは**shield(1)**のようなコマンドは各論理CPUを識別します。これはマルチプロセッサ対応ソフトウェアが倍の数の論理CPU上で修正せずに実行することを可能にします。ハイパースレッド技術は2個目の物理プロセッサを追加することにより得られる性能レベルの度合いを供与しない一方、いくつかのベンチマーク・テストは並列アプリケーションが30%ほどの性能増加を体験できることを示しています。リアルタイム・アプリケーションにとってハイパースレッディングを利用する最善の操作方法案は「推奨されるCPU構成」セクションを参照してください。

2つの論理CPUを持つ1つの物理CPUは実行リソースを効果的に利用できるため、ハイパースレッディングによる性能増加が発生します。非ハイパースレッドCPU上の標準的なプログラムの実行中において、チップ上の実行リソースは多くの場合入力待ちで遊んでいます。2つの論理CPUが実行リソース一式を共有しているため、2番目の論理CPU上で実行しているスレッドは1つのスレッドだけが実行中で遊んでいる他のリソースを使うことが出来ます。例えば、1つの論理CPUが終了するためにメモリからフェッチを待って停止している間、そのシブリングは命令ストリームを処理し続けることが可能です。プロセッサとメモリの速度が全く等しくないため、プロセッサはメモリからのデータ転送を待つことに大部分の時間を費やす可能性があります。従って、特定の並列アプリケーションのためのハイパースレッディングは著しい性能向上を提供します。他の並列処理の例は、他の加算とロード処理を実行中の浮動小数点演算を実行する1つの論理プロセッサです。チップ上の異なるプロセッサ実行ユニットを利用するため、これらの演算は並列に実行されます。

ハイパースレッディングはマルチスレッドの作業負荷に対して通常高速実行を提供する一方、リアルタイム・アプリケーションにとって問題となる可能性があります。これはスレッド実行のデターミニズムに対する影響によるものです。ハイパースレッドCPUは別スレッドと一体となっているプロセッサの実行ユニットを共有するため、ハイパースレッドCPU上でスレッドを実行したときに実行ユニット自身が他のリソースレベルで競合します。ハイパースレッド上の高優先度のプロセスが命令の実行を使用としたときに実行ユニットは常に利用可能ではないため、ハイパースレッド上のコード・セグメントの実行に必要な時間は非ハイパースレッドCPU上と同様に予測できません。

並列リアルタイム・アプリケーションの設計者は、アプリケーションにとってハイパースレッディングが意味があるのかどうかを判定しなければなりません。タスクをハイパースレッドCPU上で並列実行することが、連続的に実行することと比較してアプリケーションの利益となるでしょうか？もしそうなのであれば、開発者はハイパースレッド上で実行することにより重要な高優先度スレッドの実行速度にどれくらいのジッターをもたらすのかを判断するために測定することが可能です。

容認できるジッターのレベルはアプリケーションに大いに依存します。もしハイパースレッディングが原因で容認できないジッター量をリアルタイム・アプリケーションにもたらす場合、影響したタスクは**cpu(1)**コマンドによりシブリングCPUをダウン状態(アイドル状態)にしたシールドCPU上で実行されなければなりません。CPUをダウン状態にしたシステムの例は本章で後述します。特定のプロセッサ間割り込みは、ダウン状態(詳細は**cpu(1)**のmanページを参照してください)のCPU上で処理され続けることに注意しなければなりません。もし必要であれば、ハイパースレッディングはシステム全体で無効にすることが可能です。詳細は「システム構成」セクションを参照してください。

ハイパースレッディング技術はシステムの各プロセッサ内で並列処理を提供することによりマルチプロセッシングを補いますが、デュアルもしくはマルチプロセッサに置き換わるものではありません。システムに利用可能な2つの論理CPUがあっても、同じ量の実行リソースを共有したままです。従って、専用の実行リソース一式を所有するもう1つの物理プロセッサの性能の利点は、より大きな性能レベルを提供することです。これはデターミニスティックな実行環境を獲得するためにシールドCPUを利用するアプリケーションにとっては特に有効となります。

上述したように各論理CPUは完全な構造状態一式を維持します。この(シブリングCPUによって共有されていない)構造状態は汎用レジスタ、制御レジスタ、高度プログラマブル割り込みコントローラ(APIC)レジスタ、いくつかのマシン・ステータス・レジスタ(MSR)で構成されます。論理プロセッサはキャッシュ、実行ユニット、分岐予測、制御ロジック、バスのような物理プロセッサ上の殆どのリソースを共有します。各論理プロセッサはそれぞれの割り込みコントローラもしくはAPICを持っています。ハイパースレッディングが有効であるか無効であるかは関係なく、特定の論理CPUに送信する割り込みはその論理CPUだけで処理されます。

システム構成

以下の項目はハイパースレッドの可用性がシステム全体に影響を及ぼします。

- システムは Intel Pentium Xeonプロセッサが構成されている必要があります。
- カーネル構成GUI上の「Processor Type and Features」にあるカーネル・チューニング・パラメータ X86_HTを通してハイパースレッディングを有効にする設定でなければなりません。ハイパースレッディングはすべてのRedHawk i386プレデファイン・カーネルでは規定値で有効となっています。
- ハイパースレッディングを利用するためにBIOSで有効にする必要があります。必要に応じてBIOSの設定に関するハードウェアの資料を参照してください。

ハイパースレッディングは、シブリングCPUをダウン状態にするための**cpu(1)**コマンドを使ってCPU毎に無効にすることが可能です。詳細は**cpu(1)**のmanページを参照してください。

ハイパースレッディングを有効である場合、**top(1)**や**run(1)**のようなコマンドは、以前に存在したハイパースレッディングをサポートしていないRedHawk Linux Release 1.3より前のバージョンが動作しているシステムのCPU数の2倍レポートすることに注意してください。システム全体でハイパースレッディングが無効になっているとき、論理CPUの数は物理CPUの数と等しくなります。

推奨される**CPU**構成

ハイパースレッディング技術は並列アプリケーションに性能向上の可能性を提供します。しかし、CPUリソースが1つの物理CPUを論理CPU間で共有されている様式であるため、様々なアプリケーションの混合はパフォーマンスの結果が異なることとなります。これはアプリケーションにデターミニスティックな実行時間を必要とするリアルタイム要件がある時に特に当てはまります。従って、最適な性能を判断するために様々なCPU構成でアプリケーションの性能テストをすることがとても重要になります。例えば、もし1組のシブリングCPU上で並列に実行可能な2つのタスクが存在する場合、両方のシブリングCPUを使って並列にそれらのタスクを実行するために必要な時間に対してシブリングCPUの1つをダウン状態にしてそれらのタスクを連続で実行するために必要な時間を必ず比較してください。ハイパースレッディングによって提供されるユニークな並列処理の優位性をそれら2つのタスクが得られるかどうかを判断できるでしょう。

以下は、リアルタイム・アプリケーションのためにハイパースレッドCPUを含むSMPシステムを構成する方法を提案します。これらのサンプルには、様々な性能特性を持つアプリケーションにとって最高に作用するかもしれない構成に関するヒントが含まれています。

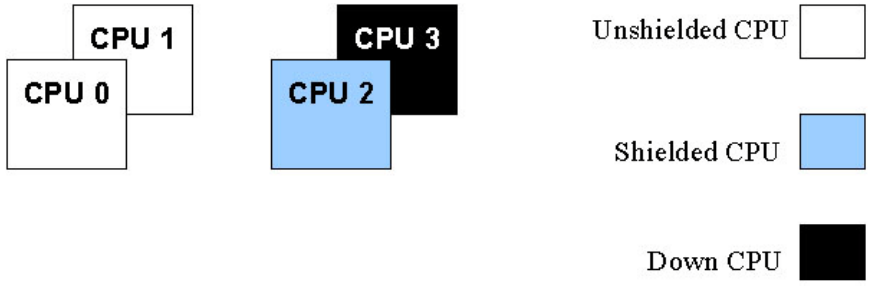
標準的なシールドCPUモデル

このモデルは、プログラム実行においてデータミニズムを非常に厳しく要求するアプリケーションに利用できるでしょう。シールドCPUは、これらの種類のタスクに最もデターミニスティックな環境を提供します(シールドCPUに関する詳細な情報については「シールディングでリアルタイム性能を向上する方法」セクションを参照してください)。シールドCPUのデターミニズムを最大限にするために物理CPU上のハイパースレッディングは無効にします。これは**cpu(1)**コマンドを使ってシールドCPUのシブリング論理CPUをダウン状態にすることで完了します。

標準的なシールドCPUモデルでは、非シールドCPUはハイパースレッディングは有効の状態です。一般的にハイパースレッディングはより多くのCPUリソースを利用されることを許可するため、これらのCPUは非クリティカルな作業負荷に使用されます。

2つの物理CPU(4つの論理CPU(を持つシステム上の標準的なシールドCPUモデルを図2-8に図示します。この例の中では、CPU3はダウン状態となりCPU2は割り込み、プロセス、ハイパースレッディングからシールドされています。高優先度割り込みとその割り込みに応答するプログラムは、割り込みに対して最高のデターミニスティックな応答をするためCPU2に割り付けます。

図2-8 標準的なシールドCPUモデル



この構成を設定するコマンドは、

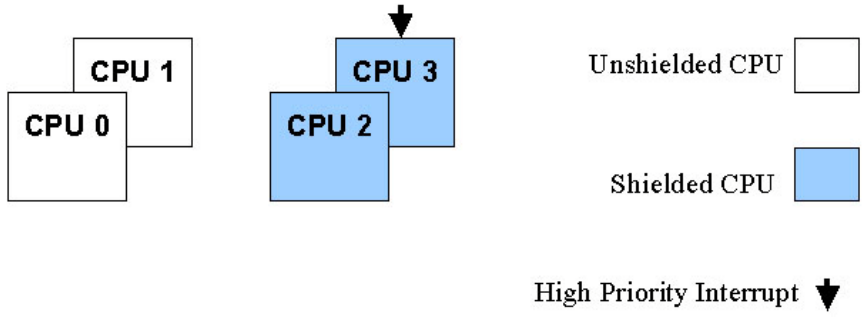
```
$ shield -a 2
$ cpu -d 3
```

割り込みを分離したシールディング

このモデルは標準的なシールドCPUモデルに非常に似ています。しかし、このケースではすべての論理CPUは使用されており、1つもダウン状態ではありません。標準的なシールドCPUモデルのように論理CPUの1つの集合はシールドされています。しかし、シールドCPUのシブリングをダウン状態にすることよりむしろ、それらのCPUをシールドしてデターミニスティックな割り込み応答を要求する高優先度割り込みを処理するために専念させます。これはプロセスと割り込みからシブリングCPUをシールドし、更にそのシブリングCPUへ特定の割り込みのCPUアフィニティを設定することにより完了します。

割り込みを分離したシールディングを図2-9に図示します。

Figure 2-9 割り込みを分離したシールディング



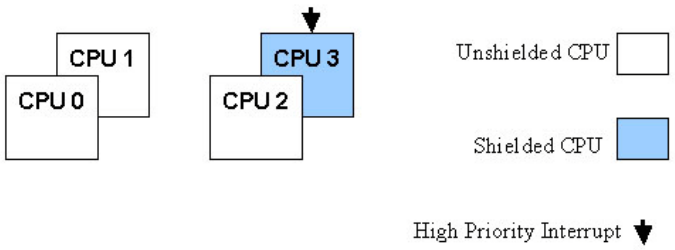
このアプローチの利点は、(CPU3上で動作する)割り込みルーチンとシブリングCPU上の高優先度タスク(CPU2上で動作する割り込み待ちプログラム)の実行との間で僅かながらの並列処理を提供することです。割り込みルーチンがCPU3上で実行している唯一のコードあるため、この割り込みルーチンは通常L1キャッシュに完全に保持され、そのコードはキャッシュの中に留まって、割り込みルーチンに対し最適な実行時間を提供します。その割り込みルーチンはシブリングCPU上の割り込み待ちタスクを起こすためにプロセッサ間割り込みを送信する必要があるため、小さな代償を払うことにはなります。この余分なオーバーヘッドは2μ秒未満です。

割り込み分離によるシールディングの利用によるもう一つの潜在的効果は、デバイスのI/Oスループットを向上させることです。デバイスの割り込み処理にCPUが専念しているため、この割り込みはI/O操作が完了したときに常に可能な限り迅速に完了します。これは割り込みルーチンが即座に次のI/O操作の開始を可能にし、より良いI/Oスループットを提供します。

ハイパースレッドのシールディング

この構成は標準的なシールドCPUモデルとは別のバリエーションです。このケースでは、一つのシブリングCPUがもう一方のシブリングCPUが通常のタスクの実行を許可されている状態でシールドされています。シールドCPUはもう一方のシブリングCPU上の動作状況によってデターミニズムに影響を与えます。また一方で、この優位性はアプリケーションがより多くの物理CPUのCPUパワーを利用することが可能であることです。ハイパースレッドのシールディング構成を図2-10に図示します。

図2-10 ハイパースレッドのシールディング



このサンプルでは、CPU3はシールドされ、高優先度の割り込みとその割り込みに応答するプログラムだけの実行を許可しています。CPU2はシールドされていないために通常の利用が可能、つまり特定のタスク一式を実行するために構成されています。プリエンプションや割り込みブロックが無効の時、CPU3上で実行中の高優先度割り込みやタスクには影響がないため、CPU2上で動作するタスクは直接割り込み応答時間へ影響を与えることはありません。しかし、チップのリソースレベルではCPU3上での実行のデターミニズムには影響を与える競合が存在します。その影響の度合いはアプリケーションにとても依存します。

浮動小数点演算 / 整数演算の共有

この構成はアプリケーションが主に浮動小数点演算を実行するいくつかのプログラムおよび整数算術演算を実行するいくつかのプログラムを持っているときに利用することが可能です。ハイパースレッドCPUの両方のシブリングは特定のタスクを実行するために使用されます。浮動小数点を集約したプログラムを1つのシブリングCPUに割り付け、主に整数計算を実行するプログラムをもう一方のシブリングCPUに割り付けます。この構成の優位点は浮動少数点演算と整数演算は異なるチップのリソースを利用することです。これはチップレベルでの利用が可能な並列処理であるため、ハイパースレッド型並列処理を十分に活用することが可能となります。コンテキスト・スイッチ間で浮動小数点レジスタの Save/Restore がないため、整数演算だけを実行するCPU上のアプリケーションはコンテキスト・スイッチ時間が高速に見えることに注意すべきです。

共有データ・キャッシュ

この構成はアプリケーションが生産者/消費者型プリケーションの時に利用することが可能です。言い換えると、1つのプロセス(消費者)はもう一方のプロセス(生産者)から渡されたデータで動作しています。このケースでは、生産者と消費者の各スレッドはハイパースレッドCPUのしうリングに割り付ける必要があります。2つのシブリングCPUがデータ・キャッシュを共有するため、生産者プロセスによって生産されるデータは消費者プロセスが生産者タスクから渡されたデータをアクセスしたときにデータ・キャッシュ内に留まっている可能性があります。このように2つのシブリングCPUを利用することは生産者と消費者の各タスクが並列に実行することを可能にし、またそれらの間で渡されるデータは基本的に高速キャッシュ・メモリを介して渡されます。これはハイパースレッド型並列処理を利用するために重要な機会を提供します。

もう1つのこのモデルの潜在的な利用法は、ハイパースレッドCPU上の1つのシブリングCPU上のプロセスのためにもう一つのシブリングCPUで実行中のプロセスで使われるデータ・キャッシュの中にデータをプリフェッチすることです。

シールド・ユニプロセッサ

この構成はハイパースレッド・シールディング構成の1つのバリエーションです。唯一の違いは、SMPシステム内の一つの物理プロセッサよりもむしろ、ユニプロセッサにこの技術を適用していることです。物理メモリには現在2つの論理CPUが含まれているため、ユニプロセッサは現在シールドCPUを作るために使用することが可能です。このケースでは、CPUの1つをシールドの設定して、一方他のCPUはバックグラウンド処理を実行するために使用します。このタイプのシールドCPUのデターミニズムは、異なる物理CPUでのCPUシールディングの利用ほど確実ではありませんが、シールドされていないよりは明らかに良くなります。

メモリ不足状態の回避

所有するシステムが適切な物理メモリを搭載していることを確認してください。Concurrentのリアルタイム保証は、リアルタイム・アプリケーションが利用するために十分なRAMが搭載されているシステムが正しく構成されていることを前提とします。メモリが少ない状況では、システムの完全性をより確実にし、適切なシステムの挙動を維持するためにリアルタイムのデッドラインを犠牲にするかもしれません。Linux はメモリが不足する時、無作為にメモリを開放するために終了するプロセスを選ぶことで、他のプロセスを起動することが可能になります。
メモリの使用状況は **/proc/meminfo**, **free(1), vmstat(8)**に含まれるいくつかのツールの利用で監視することが可能です。

Linuxのデターミニズムに関する既知の問題

以下は、リアルタイム性能にネガティブな影響を与えることが知られている標準的なLinux の問題です。システムがリアルタイム・アプリケーションを実行中は、これらの行為は本質的に一般的な管理用であり実行してはいけません。

- **hdparm(1)**ユーティリティは、IDEまたはSCSIディスク用の特別なパラメータを有効にするためのコマンド・ライン・インターフェースです。本ユーティリティは非常に長い時間割り込みを無効にすることで知られています。
- **blkdev_close(2)**インターフェースは、RAWブロック・デバイスへ書き込むためにブート・ローダーに使用されます。これは非常に長い時間割り込みを無効にすることで知られています。
- フレームバッファ(fb)コンソールがスクロールすることを避けてください。これは非常に長い時間割り込みを無効にすることで知られています。
- 仮想コンソールを使用する時、コンソールは切り替えないでください。これは非常に長い時間割り込みを無効にすることで知られています。
- CDのマウント/アンマウントおよびファイルシステムのアンマウントは避けてください。これらのアクションは長い待ち時間を引き起こします。
- CDの自動マウントは止めてください。これはポーリングのインターフェースで周期的なポーリングが長い待ち時間を引き起こします。
- **haldaemon**サービスはリアルタイム性能に干渉する事が明らかであり、デフォルトでOFFになっています。

一方、これはファイルのコンテキスト・メニューからCDまたはDVDにファイル(例：ISO)を焼き付けるには実行させる必要があります。ディスクにファイルを焼き付けるには、最初に**haldaemon**サービスを開始して下さい：

```
$ service haldaemon start
```

その後コピー処理が完了したらサービスを停止して下さい：

```
$ service haldaemon stop
```

- カーネル・モジュールをアンロードすることは避けてください。この行為はCPUに不必要なジッターが増す可能性のある**kmodule**デーモンをCPU毎に作成し破棄します。
- **ksoftirqd**カーネル・デーモンにより定期的にフラッシュされるIPルート・キャッシュ・テーブルは、利用可能なメモリの量に基づいて動的に大きさを設定します (例：メモリ4GBのシステムに対して128Kエントリ)。もしネットワークのデターミニズムに問題がある場合、特にシングルCPUシステムにおいてはフラッシュに必要な時間が問題となる可能性があります。過剰なksoftirqd の実行を減らすため、IPルート・キャッシュ・テーブルはGRUBコマンド **rhash_entries=***n* (n はテーブル・エントリの数)を利用して固定サイズに設定することが可能です。例：
  `rhash_entries=4096` (エントリ数を4Kに設定)
- シールドCPU上でタイムクリティカル・アプリケーションの実行中、Xサーバの開始および停止する時にリアルタイムに問題が発生する可能性があります。システムで使われているグラフィックカードの種類によっては、多くのプロセッサ間割り込みの性能低下という結果になるかもしれません。もしこのような経験があるのであれば、これらの割り込みを減らすために付録Fを参照してください。
- カーネル内にDELETE_SOFTLOCKUPが設定されていないことを確認してください。本オプションは、全てのCPU上の過度なCPU毎デーモンを終了することによるCPUシールディング、および保持されている再スケジューリング変数がソフト・ロックアップとしてsoftlockup機構により誤って判断されるために再スケジュール変数に干渉します。
- カーネル内にSCHED_SMT_IDLEが設定されていないことを確認してください。他のスレッドがSCHED_RRまたはSCHED_FIFOタスクとして実行中にもしSCHED_OTHERタスクを実行するためにスケジュールされた場合、このパラメータは一組のSMT CPUのスレッドが強制的にアイドルになることを回避します。
- **mount(1)**オプションの**noatime**は、ファイルシステムにアクセスする毎にiノードのアクセス時間の不必要なアップデートを排除するために**/etc/fstab**内で定義することを推奨します。

3
リアルタイム・プロセス間通信

本章ではRedHawk Linuxがサポートする POSIXのリアルタイム・プロセス間通信、およびSystem Vのメッセージ送受信と共有メモリ機能について説明します。
付録AにはPOSIXとSystem Vのメッセージ・キュー機能の使用方法を説明したプログラム例が含まれています。

概要

RedHawk Linuxはプロセスがデータをやり取りすることを許可するいくつかのメカニズムを提供します。それらのメカニズムにはIEEE規格1003.1b-1993に準拠するメッセージ・キュー、共有メモリ、セマフォの他にSystem Vの Interprocess Communication (IPC) パッケージに含まれるそれらの機能も含まれています。メッセージ・キューと共有メモリは本章の中で解説し、セマフォは5章の「プロセス間同期」で解説しています。

メッセージ・キュー は1つ以上の読み取りプロセスにより読まれるメッセージを1つ以上のプロセスが書き込むことが可能です。この機能はメッセージ・キューの作成、オープン、問い合わせ、破棄、送信、メッセージ・キューからのメッセージ受信、送信メッセージの優先度指定、メッセージ到達時の非同期通知リクエストを提供します。

POSIXとSystem Vのメッセージング機能はお互い独立して動作します。推奨するメッセージ送受信メカニズムは、効率性と可搬性の理由によりPOSIXメッセージ・キュー機能です。本章の「POSIXメッセージ・キュー」と「System Vメッセージ」のセクションでこれらの機能を説明しています。

共有メモリ は協同するプロセスがメモリの共通エリアを通してデータを共有することが可能です。1つ以上のプロセスがメモリの一部に接続することが可能で、故にそこに置かれたどんなデータでも共有することが可能です。

メッセージング同様、POSIXとSystem Vの共有メモリ機能はお互いに独立して動作します。アプリケーション内で共有メモリに置いたデータは一時的なものでシステム再起動後に存在する必要がないSystem V 共有メモリエリアの使用を推奨します。System V共有メモリ内のデータはメモリにのみ保持されます。ディスク・ファイルはそのメモリと関連しておらず、結果、**sync(2)** システムコールによるディスク・トラフィックが生じることもありません。また、System V共有メモリは共有メモリセグメントを物理メモリ領域にバインドさせることが可能です。この機能についての情報は「System V共有メモリ」セクションを参照してください。

System V共有メモリの使用の代替えとして、**/dev/mem**ファイルの一部をマッピングする **mmap(2)**システムコールを使用します。**mmap**システムコールに関する情報は、9章の「メモリ・マッピング」を参照してください。**/dev/mem**ファイルに関する情報は、**mem(4)**のmanページを参照してください。

POSIX共有メモリのインターフェースは**/var/tmp**ディレクトリ内のディスク・ファイルにマッピングされます。もしこのディレクトリが**memfs**ファイルシステム上にマウントされている場合、**sync**システムコール中の共有データのフラッシュによる余計なディスク・トラフィックは発生しません。もしこのディレクトリが通常のディスク・パーティション上にマウントされている場合、マッピングされたディスク・ファイル内の共有データを更新し続けるために**sync**システムコール中はディスク・トラフィックが発生します。

POSIX共有メモリのデータがファイルに保存されたかどうかに関係なく、それらのデータはシステム再起動後は保持されません。POSIX共有メモリ機能は、本章の「POSIX共有メモリ」セクションで説明しています。

POSIXメッセージ・キュー

アプリケーションは複数の協同プロセスから構成され、おそらく別個のプロセッサ上で動作することになります。これらのプロセスは効果的に通信しそれらの動作を調整するためにシステム全体でPOSIXメッセージ・キューを使用します。

POSIXメッセージ・キューの主な用途は、プロセス間でデータ送受信するためです。対照的に同一プロセス内スレッドは既にアドレス空間全体を共有しているため、同一プロセス内の協同スレッド間のデータ送受信機能としてはほとんど必要ありません。しかし、1つ以上のプロセス内のスレッド間でデータ送受信するためにアプリケーションがメッセージ・キューを利用することは避けられません。

メッセージ・キューは**mq_open(3)**を使って作成およびオープンされます。この機能はコールした後にオープン・メッセージ・キューを参照するために使用するメッセージ・キュー記述子(`mqd_t`)を返します。各メッセージ・キューは*/somename* の形式の名称によって識別されます。2つのプロセスが**mq_open**に同じ名前を渡すことによって同じキューを操作することが可能となります。

メッセージは、**mq_send(3)**と**mq_receive(3)**を使ってキューとの受け渡しを行います。プロセスがキューの使用を終了した時、**mq_close(3)**を使ってキューを閉じ、キューが既に必要ではない時、**mq_unlink(3)**を使って削除することが可能です。キューの属性は、**mq_getattr(3)**と**mq_setattr(3)**を使って取得および(場合によっては)修正することが可能です。プロセスは**mq_notify(3)**を使って空のキューへのメッセージ到達の非同期通知をリクエストすることが可能です。

メッセージ・キュー記述子は、オープン・メッセージ・キュー記述の参照先です(**open(2)**を参照)。**fork(2)**実行後、子プロセスは親のメッセージ・キュー記述子のコピーを継承し、それらの記述子は親プロセスと一致する記述子と同じオープン・メッセージ・キュー記述を参照します。一致する記述子は2つのプロセスは、オープン・メッセージ・キュー記述に関連するフラグ(`mq_flags`)を共有します。

各メッセージは優先度を持っており、メッセージは常に最高優先度の受信プロセスが先に配信されます。

メッセージ・キューは仮想ファイル・システム内に作成されます。このファイル・システムは以下のコマンドを使ってマウントすることが可能です。

```
$ mkdir /dev/mqueue
$ mount -t mqueue none /dev/mqueue
```

ファイル・システムがマウントされた後、システム上のメッセージ・キューは通常ファイルのために使用されるコマンド(例：**ls(1), rm(1)**)を使って見ることおよび操作することが可能となります。

POSIXメッセージ・キューのサポートはカーネル構成パラメータPOSIX_MQUEUEを介して構成可能です。このオプションはデフォルトで有効となっています。サンプルプログラムは付録Aで提供されます。

メッセージ・キュー・ライブラリ・ルーチンをコールするすべてのアプリケーションはリアルタイム・ライブラリに静的または動的のどちらでもリンクしなければなりません。以下の例は典型的なコマンドラインの書式を示します。

gcc [*options...*] *file* **-lrt** ...

System Vメッセージ

System Vのプロセス間通信(IPC: Interprocess Communication)型メッセージは、プロセス(実行中のプログラム)がバッファに格納されたデータの交換を通して通信することを可能にします。このデータはメッセージと呼ばれる別々の部分の中でプロセス間に送信されます。このIPC型を利用するプロセスはメッセージの送信および受信が可能です。

プロセスがメッセージを送受信する前にオペレーティング・システムはこれらの操作を処理するためにソフトウェアのメカニズムを作成する必要があります。プロセスは**msgget(2)**システムコールを利用して処理します。こうすることでプロセスはメッセージの所有者/作成者になり、それ自体を含む全てのプロセスに対して初期操作の権限を指定します。その後、所有者/作成者は所有権の放棄または**msgctl(2)**システムコールを使って操作権限を変更することが可能となります。しかし、作成者は機能が存在する限り依然として作成者のままです。権限を持つほかのプロセスは様々な他の制御機能を実行するために**msgctl**を使うことが可能です。

もし操作の実行に失敗した場合、権限を持っておりメッセージの送受信を行おうとしているプロセスは実行を停止することが可能です。これは、メッセージの送信をしようとしているプロセスは指定されたメッセージ・キューに対してメッセージを送信することが可能になるまで待つことが出来ます。この受信プロセスは影響を及ぼすことはなく(間接的を除く：例えば、もし消費者が消滅していなければ、そのキューのスペースは最終的には空になります)、逆も同じとなります。実行の停止を指示するプロセスは*ブロッキング・メッセージ操作* を実行します。実行の停止を許可されないプロセスは*ノンブロッキング・メッセージ操作* を実行します。

ブロッキング・メッセージ操作を実行するプロセスは、3つの条件の1つが発生するまで停止することが可能です。

- 操作が成功
- プロセスがシグナルを受信
- システムからメッセージ・キューが削除

システムコールはプロセスに利用可能なこれらのメッセージ・ケーパビリティを作ります。呼び出し元プロセスはシステムコールに引数を渡し、システムコールはその機能が成功または失敗のどちらかとなります。もしそのシステムコールが成功した場合、その機能は実行され適切な情報を返します。そうではない場合、プロセスに-1が返され、それに応じた`errno`が設定されます。

メッセージの利用

メッセージが送信または受信する前にユニークな識別されたメッセージ・キューとデータ構造体を作成する必要があります。そのユニークな識別子はメッセージ・キュー識別子(*msqid*)と呼ばれます(これは関連するメッセージ・キューやデータ構造体を確認もしくは参照するために使用されます)。この識別子は通常のアクセス制限下にあるシステムのあらゆるプロセスよりアクセス可能です。

メッセージ・キューの対応するカーネルデータ構造体は送信されるもしくは受信される各メッセージに関する情報を保持するために使用されます。システム内部で使用されるこの情報は、以下の各メッセージが含まれます。

- メッセージの種類
- テキスト・メッセージのサイズ
- テキスト・メッセージのアドレス

ユニークな識別されたメッセージ・キュー`msqid_ds`のために1つの関連するデータ構造体が存在します。このデータ構造体はメッセージ・キューに関連する以下の情報を含んでいます。

- データの権限操作 (構造の権限操作)
- キュー上の現在のバイト数
- キュー上のメッセージの数
- キュー上の最大バイト数
- 最後にメッセージを送信したプロセス識別番号 (PID)
- 最後にメッセージを受信したPID
- 最後にメッセージを送信した時間
- 最後にメッセージを受信した時間
- 最後に変更した時間

NOTE

本章で説明するすべてのC言語のヘッダーファイルは、**/usr/include**サブディレクトリにあります。

関連するメッセージ・キュー・データ構造体`msqid_ds`の定義は図3-1に示すメンバーに含まれています。

図3-1 msqid_ds構造体の定義

```
struct ipc_perm msg_perm;/* structure describing operation permission */
__time_t msg_stime; /* time of last msgsnd command */
__time_t msg_rtime; /* time of last msgrcv command */
__time_t msg_ctime; /* time of last change */
unsigned long int __msg_cbytes; /* current number of bytes on queue */
msgqnum_t msg_qnum; /* number of messages currently on queue */
msglen_t msg_qbytes;/* max number of bytes allowed on queue */
__pid_t msg_lspid; /* pid of last msgsnd() */
__pid_t msg_lrpid; /* pid of last msgrcv() */
```

C言語のデータ構造体`msqid_ds`の定義は、実際にはこの構造体は**<bits/msq.h>** に定義されていますが、**<sys/msg.h>**ヘッダーファイルをインクルードすることにより取得する必要があります。

プロセス間通信許可データ構造体`ipc_perm` の定義は、図3-2に示すメンバーに含まれています。

図3-2 ipc_perm構造体の定義

```
__key_t __key; /* Key. */
__uid_t uid; /* Owner's user ID. */
__gid_t gid; /* Owner's group ID. */
__uid_t cuid; /* Creator's user ID. */
__gid_t cgid; /* Creator's group ID. */
unsigned short int mode; /* Read/write permission. */
unsigned short int __seq; /* Sequence number. */
```

C言語のデータ構造体`ipc_perm`の定義は、実際にはこの構造体は**<bits/ipc.h>**に定義されていますが、**<sys/ipc.h>**ヘッダーファイルをインクルードすることにより取得する必要があります。<**sys/ipc.h**>は一般的に全てのIPC機能に使用されることに注意してください。

msgget(2)システムコールは2つのタスクの1つを実行します。

- 新しいメッセージ・キュー識別子を作成し、それに対応するメッセージ・キューとデータ構造体を作成します
- 既にメッセージ・キューとデータ構造体に対応した既存のメッセージ・キュー識別子を確認します

両方のタスクは**msgget**システムコールに渡す引数*key* が必要です。もし*key* が既存のメッセージ・キューに使用されていない場合、システム・チューニング・パラメータを超えない条件で新しい識別子は*key* に対応するメッセージ・キューとデータ構造体を作成して返します。

key の値がゼロ(IPC_PRIVATE)を指定するための条件もあります。IPC_PRIVATEが指定されたとき、新しい識別子はメッセージ・キュー最大数(MSGMNI)のシステム制限を超えない限り、常に対応するメッセージ・キューとデータ構造体を作成して返します。**ipcs(8)**コマンドは全てゼロで*msqid* のための*key* フィールドを表示します。

もしメッセージ・キュー識別子が指定された*key* が存在する場合、既存の識別子の値が返されます。もし既存のメッセージ・キュー識別子を返して欲しくない場合、制御コマンド(IPC_EXCL)をシステムコールに渡す*msgflg* 引数に設定することが可能です(本システムコールの詳細は、「msggetシステムコール」を参照してください)。

メッセージ・キューが作成される時、**msgget**をコールしたプロセスは所有者/作成者になり、対応するデータ構造体はそれに応じて初期化されます。所有権を変更することは可能ですが、作成されるプロセスは常に作成者のままであることを覚えておいてください。メッセージ・キュー作成者もまたそれのために初期操作権限を決定します。

一旦ユニークなメッセージ・キュー識別子が作成された、もしくは既存の識別子が見つかたら、**msgop(2)**(メッセージ操作)と**msgctl(2)**(メッセージ制御)を使用することが可能です。

前述のようにメッセージ操作はメッセージの送信と受信で構成されます。**msgsnd**と**msgrcv**のシステムコールは各々の操作のために提供されます(これらのシステムコールの詳細は「msgsndおよびmsgrcvシステムコール」を参照してください)。

msgctlシステムコールは以下の方法によりメッセージ機能を制御することを許可します。

- メッセージ・キュー識別子に対応するデータ構造体の取得 (IPC_STAT)
- メッセージ・キュー権限の変更操作 (IPC_SET)
- 特定メッセージ・キューのメッセージ・キューサイズ(`msg_qbytes`)の変更(IPC_SET)
- 対応するメッセージ・キューとデータ構造体と共にオペレーティング・システムから特定メッセージ・キュー識別子の削除 (IPC_RMID)

msgctlシステムコールの詳細は「msgctlシステムコール」セクションを参照してください。

System Vメッセージ・キューを利用したサンプルプログラムに関しては、付録Aを参照してください。更なるサンプルプログラムは、各System Vシステムコールを深く掘り下げた使い方の説明をインターネットで見つけることが可能です。これらはシステムコールを説明する本章のセクションの中で記載されています。

msggetシステムコール

msgget(2)は新しいメッセージ・キューを作成または既存のメッセージ・キューを取得します。

本セクションでは**msgget**システムコールを説明します。より詳細な情報は**msgget(2)**のmanページを参照してください。この呼び出しの使用を説明しているプログラムは、**README.msgget.txt** 内に提供された多数のコメントと共に**/usr/share/doc/ccur/examples/msgget.c**で見つけることが可能です。

概要

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget (key_t key, int msgflg);
```

上記の全てのインクルードファイルは、オペレーティング・システムの**/usr/include**サブディレクトリにあります。

`key_t` はヘッダーファイル**<bits/types.h>**の中で整数型にするために`typedef`によって定義されています**(**このヘッダーファイルは**<sys/types.h>**内部に含まれています**)**。正常終了した場合にこの機能から返される整数はユニークなメッセージ・キュー識別子*msqid* です**(***msqid* は本章の 「メッセージの利用」セクション内で説明されています**)**。失敗した場合、外部変数**errno**に失敗の理由を知らせる値が設定され、**-1**が返されます。

メッセージ・キューとデータ構造体に対応する新しい*msqid* は以下の条件に1つでも該当する場合に作成されます。

- *key* がIPC_PRIVATE
- メッセージ・キューとデータ構造体に対応する*msqid* が存在しない*key*、かつ*msgflg*とIPC_CREATの論理積がゼロではない

msgflg 値の組み合わせ：

- 制御コマンド (フラグ)
- 操作パーミッション

制御コマンドはあらかじめ定義された定数です。以下の制御コマンドは**msgget**システムコールに適用され、**<sys/ipc.h>**ヘッダーファイル内部に含まれる**<bits/ipc.h>**ヘッダーファイル内に定義されています。

IPC_CREAT	新しいセグメントを作成するために使用されます。もし使用されない場合、**msgget**は*key* に対応するメッセージ・キューの検出、アクセス許可の確認、セグメントに破棄マークがないことを確認します。
IPC_EXCL	IPC_CREATと一緒の使用は、指定された*key* に対応するメッセージ・キュー識別子が既に存在している場合、このシステムコールはエラーを引き起こします。これは新しい**(**ユニークな**)**識別子を受け取らなかった時に受け取ったと考えてしまうことからプロセスを守るために必要です。

操作パーミッションは、対応するメッセージ・キュー上で実行することを許可されたプロセスの操作を決定します。「読み取り」許可はメッセージを受信するため、または**msgctl**のIPC_STAT

操作によってキューのステータスを決定するために必要です。“書き込み”許可はメッセージを送信するために必要です。

表3-1は有効な操作パーミッション・コードの(8進数で示す)数値を示します。

表3-1 メッセージ・キューの操作パーミッション・コード

操作パーミッション	8進数値
Read by User	00400
Write by User	00200
Read by Group	00040
Write by Group	00020
Read by Others	00004
Write by Others	00002

特有の値は、必要とする操作パーミッションのために8進数値を追加もしくはビット単位の論理和によって生成されます。これが、もし「Read by User」と「Read/Write by Others」を要求された場合、コードの値は00406(00400＋00006)となります。

msgflg 値は、フラグ名称と8進数の操作パーミッション値を一緒に使用して簡単に設定することが可能です。
使用例：

```
msqid = msgget (key, (IPC_CREAT | 0400));
msqid = msgget (key, (IPC_CREAT | IPC_EXCL | 0400));
```

システムコールを常に企てられます。MSGMNIの制限を超えると常に失敗を引き起こします。MSGMNIの制限値は、その時々で使用されている可能性のあるシステム全体のユニークなメッセージ・キューの数で決定します。この制限値は**<linux/msg.h>**の中にある固定された定義値です。

メッセージ・キュー制限値のリストは以下のオプションを使って**ipcs(8)**コマンドで取得することができます。さらに詳細はmanページを参照してください。

```
ipcs -q -l
```

特定の関連したデータ構造体の初期化だけでなく特定のエラー条件に関して**msgget(2)**のmanページを参照してください。

msgctlシステムコール

msgctl(2)はメッセージ・キュー上の制御操作を実行するために使用されます。

本セクションでは**msgctl(2)**システムコールを説明します。さらに詳細な情報は**msgctl(2)**のmanページを参照してください。この呼び出しの使用を説明しているプログラムは、**README.msgctl.txt**内に提供された多くのコメントと共に**/usr/share/doc/ccur/examples/msgctl.c**で見つけることが可能です。

概要

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl (int msqid, int cmd, struct msqid_ds *buf);
```

上記の全てのインクルードファイルは、オペレーティング・システムの**/usr/include**サブディレクトリにあります。

msgctlシステムコールは正常終了で0、それ以外で-1の整数値を返します。

msqid 変数は**msgget**システムコールを使って作成された有効な負ではない整数値でなければなりません。

cmd 引数は以下の値のいずれかとなります。

IPC_STAT	指定されたメッセージ・キュー識別子に対応するデータ構造体、ポインタ*buf* によって指し示されるユーザーメモリ空間のデータ構造体の場所を含むステータス情報を返します。「Read」許可が必要です。
IPC_SET	有効なユーザーIDとグループID、操作パーミッション、メッセージ・キューのバイト数を含むポインタ*buf* によって指し示されるユーザーメモリ空間のデータ構造体を書き込みます。
IPC_RMID	指定されたメッセージ・キューと共にそれに対応するデータ構造体を削除します。

NOTE

msgctl(2)サービスはIPC_INFO, MSG_STAT, MSG_INFOコマンドもサポートします。しかし、これらのコマンドは**ipcs(8)**ユーティリティで使用するためだけに意図されているので、これらのコマンドについての説明はありません。

IPC_SETまたはIPC_RMID制御コマンドを実行するため、プロセスは以下の条件を1つ以上満たしていなければなりません。

- 有効なOWNER のユーザーIDを所有
- 有効なCREATOR のユーザーIDを所有
- スーパーユーザー
- CAP_SYS_ADMINケーパビリティを所有

さらにMSGMNB(**<linux/msg.h>**で定義)の値を超えて`msg_qbytes`のサイズを増やすIPC_SET制御コマンドを実行する時、プロセスはCAP_SYS_RESOURCEケーパビリティを所有していなければなりません。

メッセージ･キューは、**-q** *msgid*(メッセージ・キュー識別子)または**-Q** *msgkey*(対応するメッセージ・キューのキー)オプション指定による**ipcrm(8)**コマンドの利用で削除される可能性もあることに注意してください。このコマンドを使用するため、ユーザーは同じ有効なユーザーIDもしくはIPC_RMID 制御コマンドの実行に必要なケーパビリティを持っている必要があります。このコマンドの使用に関して更なる情報は**ipcrm(8)**のmanページを参照してください。

msgsndおよびmsgrcvシステムコール

msgsndおよび**msgrcv**のメッセージ操作システムコールは、メッセージの送受信するために使用されます。

本セクションでは**msgsnd**と**msgrcv**システムコールを説明します。より詳細な情報は**msgop(2)**のmanページを参照してください。この呼び出しの使用を説明しているプログラムは、**README.msgop.txt** 内に提供された多数のコメントと共に**/usr/share/doc/ccur/examples/msgop.c**で見つけることが可能です。

概要

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd (int msqid, void *msgp, size_t msgsz, int msgflg);

int msgrcv (int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

上記の全てのインクルードファイルは、オペレーティング・システムの**/usr/include**サブディレクトリにあります。

メッセージの送信

msgsndシステムコールは正常終了で0、それ以外で-1の整数値を返します。

msqid 変数は**msgget**システムコールを使って作成された有効な負ではない整数値でなければなりません。

msgp 引数はメッセージの形式および送信するメッセージを含むユーザーメモリ空間内の構造体のポインタです。

msgsz 引数は*msgp* 引数によって指し示されるデータ構造体の文字配列の長さを指定します。これはメッセージの長さになります。配列の最大サイズはの中で定義されるMSGMAXによって決定されます。

msgflg 引数は、IPC_NOWAITフラグが設定されていない ((*msgflg* & IPC_NOWAIT)= = 0)場合はメッセージ操作の実行はブロックされ、指定されたメッセージ・キュー上に割り当てられた合計バイト数が使用されている場合 (`msg_qbytes`) は操作はブロックされます。IPC_NOWAITフラグがセットされている場合、システムコールは失敗し、-1を返します。

メッセージの受信

msgrcvシステムコールが成功した時は受信したバイト数を返し、失敗した時は-1を返します。

msqid 引数は有効な負ではない整数値でなければなりません。つまり、*msqid* 引数は**msgget**システムコールを使って作成された整数値でなければなりません。

msgp 引数はメッセージの形式やメッセージテキストを受信するユーザー空間内の構造体へのポインタです。

msgsz 引数は受信するメッセージの長さを指定します。もしこの値がメッセージの配列よりも小さい場合、必要であればエラーを返すことが可能です。(下の*msgflg* 引数を参照してください)

msgtyp 引数は指定された特定の形式のメッセージ・キュー上の最初のメッセージを選ぶために使用されます。

- *msgtyp* がゼロの場合、キューの最初のメッセージを受信
- *msgtyp* がゼロよりも大きく*msgflg* にMSG_EXCEPTが**設定さていない**場合、同じ型式の最初のメッセージを受信
- *msgtyp* がゼロよりも大きく*msgflg* にMSG_EXCEPTが**設定さている**場合、*msgflg* **ではない**キューの最初のメッセージを受信
- *msgtyp* がゼロよりも小さい場合、*msgtyp* の絶対値以下で最も小さいメッセージの型式を受信

msgflg 引数は、IPC_NOWAITフラグが設定されていない((*msgflg* & IPC_NOWAIT)= = 0)場合はメッセージ操作の実行はブロックされ、指定されたメッセージ・キュー上に割り当てられた合計バイト数が使用されている場合 (`msg_qbytes`) は操作はブロックされます。IPC_NOWAITフラグがセットされている場合、システムコールは失敗し、-1を返します。そして、前述したとおり、MSG_EXCEPTフラグが*msgflg* 引数に設定されて*msgtyp* 引数がゼロより大きい場合、*msgtyp* 引数とは異なるメッセージの型式のキューの最初のメッセージを受信します。

IPC_NOWAITフラグが設定された場合、キュー上に必要とする型式のメッセージがない時にシステムコールは即座に失敗します。*msgflg* はメッセージが受信するサイズよりも長い場合にシステムコールが失敗するように指定します。これは*msgflg* 引数にMSG_NOERRORを設定しない ((*msgflg* & MSG_NOERROR)) == 0) ことによってなされます。もしMSG_NOERRORフラグを設定した場合、メッセージは**msgrcv**の*msgsz* 引数で指定された長さに切り捨てられます。

POSIX共有メモリ

POSIX共有メモリ・インターフェースは、協同プロセスがデータを共有することおよび共有メモリ・オブジェクトの利用を通してより効率的に通信することを可能にします。*共有メモリ・オブジェクト* はファイル・システムから独立してストレージの名前つき領域として定義され、関連メモリを共有するために1つ以上のプロセスのアドレス空間にマッピングすることが可能です。

以下にインターフェースを簡単に記述します。

shm_open	共有メモリ・オブジェクトの作成および共有メモリ・オブジェクトとファイル記述子間の接続を確立
shm_unlink	共有メモリ・オブジェクトの名前を削除

shm_openルーチンを利用する手順は「shm_openルーチンの利用」の中で紹介されています。**shm_unlink**ルーチンを利用する手順は「shm_unlinkルーチンの利用」の中で紹介されています。

データ共有のために協同プロセスがこれらのインターフェースを使用するためには、1つのプロセスは以下のステップを完了します。紹介するステップの手順は標準的なもので、利用可能な唯一の手順ではないことに注意してください。

STEP 1:	**shm_open**ライブラリルーチンの呼び出し、ユニークな名前の指定、読み書きする共有メモリ・オブジェクトをオープンするためのO_CREATとO_RDWRビットの設定により共有メモリ・オブジェクトの作成およびそのオブジェクトとファイル記述子間の接続を確立します。
STEP 2:	**ftruncate(2)**システムコールの呼び出し、STEP 1で取得したファイル記述子の指定により共有メモリ・オブジェクトのサイズを設定します。このシステムコールは書き込み用にメモリ・オブジェクトがオープンされている必要があります。**ftruncate(2)**に関する更なる情報は対応するmanページを参照してください。
STEP 3:	**mmap(2)**システムコールの呼び出し、およびSTEP 1で取得したファイル記述子の指定により、プロセスの仮想アドレス空間の一部を共有メモリ・オブジェクトにマッピングします。(本システムコールの解説は「メモリ・マッピング」章を参照してください)

共有メモリ・オブジェクトを使用するため、他の協同プロセスは以下のステップを完了します。紹介するステップの手順は標準的なもので、利用可能な唯一の手順ではないことに注意してください。

STEP 1:	最初のプロセスによって作成された共有メモリ・オブジェクトと**shm_open**ライブラリルーチンの呼び出し、オブジェクトの作成に使用した同じ名前の指定によりファイル記述子間の接続を確立します。
STEP 2:	もし共有メモリ・オブジェクトのサイズがわからない場合、**fstat(2)**システムコールの呼び出し、STEP 1で取得したファイル記述子と`stat`構造体(<**sys/stat.h**>で定義)へのポインタの指定により共有メモリ・オブジェクトのサイズを取得します。

オブジェクトのサイズは`stat` 構造体の`st_size`領域の中に返されます。オブジェクトに対応するアクセス許可は`st_modes`領域の中に返されます。**fstat(2)**に関する更なる情報は対応するシステム・マニュアルのページを参照してください。

STEP 3: **mmap**の呼び出し、およびSTEP 1で取得したファイル記述子により、プロセスの仮想アドレス空間の一部を共有メモリ・オブジェクトにマッピングします(本システムコールの解説は「メモリ・マッピング」章を参照してください)。

shm_openルーチンの利用

shm_open(3) ルーチンは、呼び出し元プロセスのPOSIX共有メモリ・オブジェクトの作成、およびオブジェクトとファイル記述子間接続の確立が可能です。プロセスは続いて**ftruncate(2)**, **fstat(2)**, **mmap(2)**を呼び出して共有メモリ・オブジェクトを参照するために**shm_open**が返したファイル記述子を使います。プロセスが共有メモリ・オブジェクトを作成した後、他のプロセスは共有メモリ・オブジェクトと**shm_open**の呼び出し、同じ名前の指定によるファイル記述子間の接続を確立することが可能になります。

共有メモリ・オブジェクトが作成された後、共有メモリ・オブジェクト内の全データは**munmap(2)**, **exec(2)**, **exit(2)**の呼び出し、および1つのプロセスが**shm_unlink(3)**を呼び出して共有メモリ・オブジェクトの名前を削除することにより全てのプロセスがアドレス空間と共有メモリ・オブジェクト間のマッピングを削除するまで残ります。お使いのシステムを再起動した後は、共有メモリ・オブジェクトもその名前も有効ではありません。

概要

```
#include <sys/types.h>
#include <sys/mman.h>

int shm_open(const char *name, int oflag, mode_t mode);
```

引数は以下のように定義されます：

name 共有メモリ・オブジェクトの名前を指定するNULLで終わる文字列のポインタ。この文字列は最大255文字に制限される可能性があることに注意してください。これに先頭のスラッシュ(/)文字を含めることが可能ですが、途中にスラッシュ文字を含めてはいけません。この名前はファイル・システムの一部ではないことに注意してください：先頭のスラッシュも現在の作業ディレクトリも名前の解釈に影響を及ぼしません(**/shared_obj** と**shared_obj**は同一の名前として解釈します)。もしPOSIXインターフェースをサポートするあらゆるシステムに移植できるコードを書きたいと考えているのであれば、*name* はスラッシュ文字で始めることを推奨します。

oflag 以下のビットをつ以上設定した整数値。

O_RDONLYとO_RDWRは相互排他的なビットであり、どちらか一方が設定されている必要があることに注意してください。

	O_RDONLY	共有メモリ・オブジェクトを読み取り専用でオープンします。
	O_RDWR	共有メモリ・オブジェクトを読み書き用にオープンします。共有メモリ・オブジェクトを作成するプロセスは**ftruncate(2)**の呼び出しによってそのサイズを設定できるようにするために書き込み用でオープンしなければならないことに注意してください。
	O_CREAT	存在しない場合、*name* で指定された共有メモリ・オブジェクトを作成します。メモリ・オブジェクトのユーザーIDは呼び出したプロセスの有効なユーザーIDに設定され、このグループIDは呼び出したプロセスの有効なグループIDに設定し、*mode* 引数により指定された許可ビットが設定されます。 *name* で指定された共有メモリ・オブジェクトが存在する場合、O_EXCLを目的として記述されている以外は、設定されたO_CREATは効力がありません。
	O_EXCL	もしO_CREATが設定され*name* で指定された共有メモリ・オブジェクトが存在する場合、**shm_open**は失敗します。もしO_CREATが設定されていない場合、このビットは無視されます。もしオブジェクトが存在し、読み書き用にオープンされた場合、O_TRUNCは*name* で指定された共有メモリ・オブジェクトの長さはゼロに切り詰められます。所有者と共有メモリ・オブジェクトのモードは変更されません。
mode		次の例外と共に*name* で指定された共有メモリ・オブジェクトの許可ビットが設定された整数値：プロセスのファイル・モード作成マスクに設定されたビットは共有メモリ・オブジェクトのモード(更なる情報は**umask(2)**と**chmod(2)**のmanページを参照してください)の中でクリアされます。もし*mode* に許可ビット以外のビットが設定されている場合、それらは無視されます。共有メモリ・オブジェクトを作成している時のみ、プロセスは*mode* 引数を指定します。

もし呼び出しが成功した場合、**shm_open**はサイズがゼロの共有メモリ・オブジェクトを作成し、呼び出し元プロセスに対してオープンしていないファイル記述子を返します。FD_CLOEXECファイル記述子フラグは新しいファイル記述子のために設定され、このフラグは共有メモリ・オブジェクトを識別するファイル記述子が**exec(2)**システムコール(更なる情報は**fcntl(2)** のシステム・マニュアルのページを参照してください)の実行でクローズされることを示します。

戻り値-1はエラーが発生したことを示し、`errno`はエラーを示すために設定されます。発生する可能性のあるエラーのタイプのリストについては**shm_open(3)**のmanページを参照してください。

shm_unlinkルーチンの利用

shm_unlink(3)ルーチンは呼び出し元プロセスが共有メモリ・オブジェクトの名前を削除することを許可します。もし1つ以上のプロセスが呼び出した時点で共有メモリ・オブジェクトにマッピングされたアドレス空間の一部を所有している場合、**shm_unlink**が返す前に名前は削除されますが、共有メモリ・オブジェクトの中のデータは最後のプロセスがマッピングしたオブジェクトを削除するまで削除されません。もしプロセスが**munmap(2)**, **exec(2)**, **exit(2)**を呼び出した場合、マッピングは削除されます。

概要

```
#include <sys/types.h>
#include <sys/mman.h>

int shm_unlink(const char *name);
```

引数は以下のように定義されます：

name 削除する共有メモリ・オブジェクト名を指定するNULLで終わる文字列のポインタ。この文字列は最大255文字に制限される可能性があることに注意してください。これに先頭のスラッシュ**(** /)文字を含めることが可能ですが、途中にスラッシュ文字を含めてはいけません。この名前はファイル・システムの一部ではないことに注意してください：先頭のスラッシュも現在の作業ディレクトリも名前の解釈に影響を及ぼしません**(/shared_obj**と**shared_obj**は同一の名前として解釈します)。もしPOSIXインターフェースをサポートするあらゆるシステムに移植できるコードを書きたいと考えているのであれば、*name* はスラッシュ文字で始めることを推奨します。

戻り値0は**shm_unlink**の呼び出しが成功したことを示します。戻り値-1はエラーが発生したことを示します。`errno`はエラーを示すために設定されます。発生する可能性のあるエラーのタイプのリストについては**shm_ unlink(3)**のmanページを参照してください。もしエラーが発生した場合、**shm_unlink**の呼び出しは名前付き共有メモリ・オブジェクトを変更しません。

System V共有メモリ

共有メモリは2つ以上のプロセスがメモリ、つまりその中に格納されているデータを共有することを可能にします。これは共通の仮想メモリアドレス空間へのアクセスの設定を許可することによって行われます。この共有は領域ベースで存在し、それはハードウェア依存のメモリ管理となります。

プロセスは最初に**shmget(2)**システムコールを使って共有メモリ領域を作成します。作成に関し、プロセスは共有メモリ領域のために全体的な操作許可を設定して、サイズをバイトで設定し、共有メモリ領域を参照専用**(** 読み取り専用)で結合するように指定することが可能です。

もしメモリ領域が参照専用として指定されていない場合、適切な操作許可を持つ他の全てのプロセスはメモリ領域の読み取り、または書き込みが可能です。

システム上の共有メモリ領域は**/proc/sysvipc/shm**ファイルおよび**-m**オプションを使用して**ipcs(8)**を介して見ることができます。

共有メモリの操作、**shmat(2)**(共有メモリの結合)と**shmdt(2)**(共有メモリの分離)は、共有メモリ領域上で実行することが可能です。もしプロセスがパーミッションを所有している場合、**shmat**はプロセスが共有メモリ領域に結合することを許可します。その後、許可されて読み取りまたは書き込みが可能になります。**shmdt**はプロセスが共有メモリ領域から分離することを許可します。その結果、共有メモリ領域への読み書きの機能を失います。

共有メモリ領域の最初の所有者/作成者は、**shmctl(2)**システムコールを使って他のプロセスへ所有権を放棄することが可能です。しかし、機能が削除される、もしくはシステムが最初期かされるまで作成されたプロセスは作成者のままとなります。パーミッションを持つ他のプロセスは、**shmctl**システムコールを使って共有メモリ領域上の他の機能を実行することが可能です。

プロセスは**shmbind(2)**システムコールを使ってI/Oメモリ領域へ共有メモリ領域をバインドすることが可能です。**shmbind**システムコールの詳細は「共有メモリ領域をI/O空間へバインド」セクションを参照してください。

協同プログラムによって共有メモリの利用を容易にするため、**shmdefine(1)**と呼ばれるユーティリティが提供されます。このユーティリティの利用手順は「shmdefineユーティリティ」で説明されています。共有メモリ領域の作成と物理メモリの一部へのバインドを援助するため、**shmconfig(1)** と呼ばれるユーティリティも提供されます。このユーティリティの利用手順は「shmconfigコマンド」で説明されています。

共有メモリの利用

プロセス間のメモリ共有は仮想領域ベース上に存在します。常にオペレーティング・システム内に存在する個々の共有メモリ領域のコピーが1つだけ存在します。

メモリの共有が稼働される前にユニークに識別された共有メモリ領域とデータ構造体が作成される必要があります。作成されたユニークな識別子は共有メモリ識別子(*shmid*)と呼ばれ、これは対応するデータ構造体を特定する、または参照するために使用されます。通常のアクセス制限を条件として、この識別子はシステム内のどのプロセスにも利用可能です。

各共有メモリ領域用に以下がデータ構造体に含まれます。

- 操作パーミッション
- 領域サイズ
- セグメント記述子 (内部システムのためだけに使用)
- 最後に操作を実行したPID
- 作成者のPID
- 現在結合しているプロセスの数
- 最後に結合した時間
- 最後に切り離した時間
- 最後に変更した時間

対応する共有メモリ領域データ構造体`shmid_ds`の定義は、図3-3に示すメンバー含みます。

図3-3 shmid_ds 構造体の定義

```
struct shmid_ds {
struct ipc_perm shm_perm; /* operation perms */
int shm_segsz; /* size of segment (bytes) */
time_t shm_atime; /* last attach time */
time_t shm_dtime; /* last detach time */
time_t shm_ctime; /* last change time */
unsigned short shm_cpid; /* pid of creator */
unsigned short shm_lpid; /* pid of last operator */
short shm_nattch; /* no. of current attaches */
};
```

共有メモリ領域のデータ構造体`shmid_ds`用のC言語データ構造体の定義は、**<sys/shm.h>**ヘッダファイルの中にあります。

この構造体の`shm_perm`メンバはテンプレートして`ipc_perm`を使うことに注意してください。`IPC`機能のために`ipc_perm`データ構造体は全て同じで、これは**<sys/ipc.h>**ヘッダファイルの中にあります。

shmget(2)システムコールは2つの仕事を実行：

- 新しい共有メモリ識別子を取得し、対応する共有メモリ領域データ構造体を作成します
- 対応する共有メモリ領域データ構造体を持っている既存の共有メモリ識別子を返します

実行されるタスクは、**shmget**システムコールに渡す*key* 引数の値によって決定されます。

key は選択した整数、もしくは**ftok**サブルーチンの使用により生成した整数にすることが可能です。**ftok**サブルーチンは提供されたパス名と識別子をベースとするキーを生成します。**ftok**を使用することで、ユニークなキーを取得することが可能になり、パス名に関連するファイルへのアクセス制限でキーへのユーザーのアクセス制御が可能となります。もし協同プロセスだけが使用可能なキーを確保したい場合、**ftok**を使用することを推奨します。このサブルーチンは以下のように指定されます。

```
key_t ftok( path_name, id )
```

path_name 引数は呼び出し元プロセスが利用可能である既存のファイルのパス名のポインタを指定します。*id* 引数は協同プロセスグループを独自に特定する文字を指定します。**ftok**は指定された*path_name* と*id* に基づくキーを返します。**ftok**の使用に関する追加情報は**ftok(3)**のmanページの中で提供されます。

もし*key* が既に既存の共有メモリ識別子に使用されておらず*shmflg* にIPC_CREATフラグ設定されている場合、新しい識別子はシステム・チューニング・パラメータを超えない条件で作成された共有メモリ領域データ構造体と一緒に返されます。

プライベート・キー(IPC_PRIVATE) として知られている値がゼロの*key* を指定するための条件も存在し、プライベート・キーを指定された時、新しい*shmid* はシステム・チューニング・パラメータを超えない限り、常に作成された共有メモリ領域データ構造体と一緒に返されます。**ipcs(8)**コマンドは*shmid* のための*key* フィールドは全てゼロで表示します。

もし指定された*key* の*shmid* が存在する場合、既存の*shmid* の値が返されます。もし既存の*shmid* を返す必要が無い場合、制御コマンド(IPC_EXCL) はシステムコールに渡される*shmflg* 引数の中に指定(設定)することが可能です。

新しい共有メモリ領域が作成された時**shmget**をコールしたプロセスは所有者/作成者となり、それに応じて対応するデータ構造体は初期化されます。所有権は変更される可能性がありますが、作成されたプロセスは常に作成者のままであることを覚えておいてください(「shmctlシステムコール」を参照してください)。共有メモリ領域の作成者はそれのために最初の操作パーミッションも決定します。

一旦識別されたユニークな共有メモリ領域データ構造体が作成されると、**shmbind**, **shmctl**, 共有メモリ操作(**shmop**)が利用可能となります。

shmbindシステムコールは、I/Oメモリの一部に共有メモリ領域をバインドすることが可能です。システムコールの詳細は「共有メモリ領域をI/O空間へバインド」セクションを参照してください。

shmctl(2)システムコールは以下の方法で共有メモリ機能を制御することを許可します。

- 共有メモリ領域に関わるデータ構造体の取得 (IPC_STAT)
- 共有メモリ領域用操作パーミッションの変更 (IPC_SET)
- 対応する共有メモリ領域データ構造体と共にオペレーティング・システムより特定の共有メモリ領域を削除 (IPC_RMID)
- メモリ上の共有メモリ領域のロック (SHM_LOCK)
- 共有メモリ領域のアンロック (SHM_UNLOCK)

shmctlシステムコールの詳細は「shmctlシステムコール」セクションを参照してください。

共有メモリ領域操作(**shmop**)は共有メモリ領域の結合と分離で構成されます。**shmat**と**shmdt**はそれらの操作の各々のために提供されます(**shmat**と**shmdt**システムコールの詳細は「shmatおよびshmdtシステムコール」セクションを参照してください)。

shmdefine(1)と**shmconfig(1)**ユーティリティは共有メモリ領域を作成することが可能なことに注意することは重要です。これらのユーティリティに関する情報は「共有メモリ・ユーティリティ」セクションを参照してください。

shmgetシステムコール

shmget(2)は新しい共有メモリ領域を作成または既存の共有メモリ領域を特定します。

本セクションでは**shmget**システムコールを説明します。より詳細な情報は**shmget (2)**のmanページを参照してください。この呼び出しの使用を説明しているプログラムは、**README.shmget.txt** 内に提供された多数のコメントと共に**/usr/share/doc/ccur/examples/shmget.c**で見つけることが可能です。

概要

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget (key_t key, size_t size, int shmflg);
```

上記の全てのインクルードファイルは、オペレーティング・システムの**/usr/include**サブディレクトリにあります。

`key_t` はヘッダーファイル**<bits/sys/types.h>**の中で整数型にするために`typedef`によって定義されています**(**このヘッダーファイルは**<sys/types.h>**内部に含まれています**)**。正常終了した場合にこのシステムコールから返される整数は、*key* の値に対応する共有メモリ領域識別子(*shmid*) です**(***shmid* は本章の「共有メモリの利用」セクション内で説明されています**)**。失敗した場合、外部変数**errno**に失敗の理由を知らせる値が設定され、**-1** が返されます。

共有メモリデータ構造体に対応する新しい*shmid* は以下の条件に1つでも該当する場合に作成されます。

- *key* が IPC_PRIVATE
- 共有メモリデータ構造体に対応する*shmid* が存在しない*key* 、かつ*shmflg*とIPC_CREATの論理積がゼロではない

shmflg 値の組み合わせ：

- 制御コマンド (フラグ)
- 操作パーミッション

制御コマンドはあらかじめ定義された定数です。以下の制御コマンドは**shmget**システムコールに適用され、**<sys/ipc.h>**ヘッダーファイル内部に含まれる**<bits/ipc.h>**ヘッダーファイル内に定義されています。

IPC_CREAT	新しいセグメントを作成するために使用されます。もし使用されない場合、**shmget**は*key* に対応するセグメントの検出、アクセス許可の確認、セグメントに破棄マークがないことを確認します。
IPC_EXCL	IPC_CREATと一緒の使用は、指定された*key* に対応する共有メモリ識別子が既に存在している場合、このシステムコールはエラーを引き起こします。これは新しい(ユニークな)識別子を受け取らなかった時に受け取ったと考えてしまうことからプロセスを守るために必要です。

パーミッション操作はユーザー、グループ、その他のために読み取り/書き込み属性を定義します。
表3-2は有効な操作パーミッションコードの(8進数で示す)数値を示します。

表3-2 共有メモリ操作パーミッション・コード

操作パーミッション	8進数値
Read by User	00400
Write by User	00200
Read by Group	00040
Write by Group	00020
Read by Others	00004
Write by Others	00002

特有の値は、必要とする操作パーミッションのために8進数値を追加もしくはビット単位の論理和によって生成されます。これが、もし「Read by User」と「Read/Write by Others」を要求された場合、コードの値は00406 (00400＋00006)となります。**<sys/shm.h>**の中にある定数SHM_RとSHM_Wは所有者のために読み書きパーミッションを定義するために使用することが可能です。

shmflg 値は、フラグ名称と8進数の操作パーミッション値を一緒に使用して簡単に設定することが可能です。
使用例：

```
shmid = shmget (key, size, (IPC_CREAT | 0400));
shmid = shmget (key, size, (IPC_CREAT | IPC_EXCL | 0400));
```

以下の値は**<sys/shm.h>**の中で定義されています。これらの値を超えると常に失敗の原因となります。

SHMMNI	いつでも利用可能なユニークな共有メモリ領域(*shmids*)の最大数
SHMMIN	最小共有メモリ領域サイズ
SHMMAX	最大共有メモリ領域サイズ
SHMALL	最大共有メモリ・ページ数

共有メモリ制限値のリストは以下のオプションの使用により**ipcs(8)**コマンドで取得することが可能です。詳細はmanページを参照してください。

```
ipcs -m -l
```

特定の関連するデータ構造体の初期化および特定のエラー条件については**shmget(2)**のmanページを参照してください。

shmctlシステムコール

shmctl(2)は共有メモリ領域の制御操作を実行するために使用されます。

本セクションでは**shmctl**システムコールを説明します。さらに詳細な情報は**shmctl(2)**のmanページを参照してください。この呼び出しの使用を説明しているプログラムは、**README.shmctl.txt**内に提供された多くのコメントと共に**/usr/share/doc/ccur/examples/shmctl.c**で見つけることが可能です。

概要

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl (int shmid, int cmd, struct shmid_ds *buf);
```

上記の全てのインクルードファイルは、オペレーティング・システムの**/usr/include**サブディレクトリにあります。

shmctlシステムコールは正常終了で0、それ以外で-1の整数値を返します。

shmid 変数は**shmget**システムコールを使って作成された有効な負ではない整数値でなければなりません。

cmd 引数は以下の値のいずれかとなります。

IPC_STAT	指定された*shmid* に対応するデータ構造体、ポインタ*buf* によって指し示されるユーザーメモリ空間のデータ構造体の場所を含むステータス情報を返します。「Read」許可が必要です。
IPC_SET	指定されたに*shmid* 対応する有効なユーザーIDとグループID、パーミッション操作を設定します。
IPC_RMID	指定された*shmid* と共にそれに対応するデータ構造体を削除します。
SHM_LOCK	共有メモリ領域のスワップを防ぎます。ユーザーはロックが有効になった後、存在することを要するどのページもフォールトする必要があります。プロセスはこの操作を実行するためにスーパーユーザもしくはCAP_IPC_LOCK権限を持っている必要があります。
SHM_UNLOCK	メモリから共有メモリ領域をアンロックします。プロセスはこの操作を実行するためにスーパーユーザもしくはCAP_IPC_LOCK権限を持っている必要があります。

NOTE

msgctl(2)サービスはIPC_INFO, SHM_STAT, SHM_INFOコマンドもサポートします。しかし、これらのコマンドは**ipcs(8)**ユーティリティで使用するためだけに意図されているので、これらのコマンドについての説明はありません。

IPC_SETまたはIPC_RMID制御コマンドを実行するため、プロセスは以下の条件を1つ以上満たしていなければなりません。

- 有効なOWNERのユーザーIDを所有
- 有効なCREATORのユーザーIDを所有
- スーパーユーザー
- CAP_SYS_ADMINケーパビリティを所有

共有メモリ領域は、**-m** *shmid*(共有メモリ領域識別子)または**-M** *shmkey*(対応する領域のキー)オプション指定による**ipcrm(8)**コマンドの利用で削除される可能性もあることに注意してください。このコマンドを使用するため、プロセスはIPC_RMID 制御コマンドの実行に必要となるのと同じ権限を持っている必要があります。このコマンドの使用に関して更なる情報は**ipcrm(8)**のmanページを参照してください。

共有メモリ領域を**I/O**空間へバインド

RedHawk Linuxは共有メモリ領域をI/O空間の一部にバインドすることが可能です。そうするための手順は以下となります。

1. 共有メモリ領域を作成 (**shmget(2)**).
2. PCI BAR スキャン・ルーチンを使用してI/O領域の物理アドレスを取得
3. 領域をI/Oメモリにバインド (**shmbind(2)**).
4. 領域をユーザーの仮想アドレス空間に結合 (**shmat(2)**).

コマンドレベルで、**shmconfig(1)**ユーティリティは共有メモリ領域を作成して、それを物理メモリへバインドするために使用することが可能です。詳細は「共有メモリ・ユーティリティ」セクションを参照してください。

shmatと**shmdt**のシステムコールを使用することにより、共有メモリ領域とユーザーの仮想アドレス空間との結合および分離が可能となります。これらのシステムコールの使用手順は「shmatおよびshmdtシステムコール」の中で説明されています。

shmgetの利用

shmget(2)システムコールは共有メモリ領域を作成するために最初に呼び出されます。呼び出しの正常終了によって、*size* バイトの共有メモリ領域が作成され、領域の識別子が返されます。

I/O空間にバインドした時、領域のサイズはPCI BARスキャン・ルーチンを使用して取得することが可能です。(**bar_scan_open(3)**を参照してください)

shmgetの使用に関する全ての情報は「shmgetシステムコール」の中で提供されます。

shmbindの利用

共有メモリ領域を作成した後、**shmbind(2)**システムコールを使ってI/O空間の一部にそれをバインドすることが可能です。この呼び出しは、ルートまたはCAP_SYS_RAWIO の権限を持っている必要があることに注意してください。

shmbindは最初のプロセスが領域へ結合する前に呼び出される必要があります。その後、**shmat()**を介して領域へ結合するため、呼び出し元プロセスの仮想アドレス空間を物理アドレス空間の一部にマッピングを効果的に作成します。

I/O空間の一部は、その開始アドレスおよび抑制された共有メモリ領域のサイズによって定義されます。開始アドレスはページ・バウンダリ(境界線)に揃える必要があります。共有メモリ領域のサイズは**shmget**の呼び出しに使用する*size* 引数により確立されます。もし1024バイトの共有メモリ領域を作成し、例えば、開始位置0x2000000 (16進数表示)で物理メモリの一部へバインドしたい場合、物理メモリの境界部分はメモリ位置0x2000000から0x2000BFFを含むことになります。

デバイス用の物理アドレスはシステム内のハードウェア変更が原因で変わる可能性があることに注意してください。確実にデバイスを参照するために物理アドレスをPCI BAR スキャンルーチンを使って取得する必要があります。**bar_scan_open(3)**のmanページを参照してください。

shmbindを呼び出すために必要とされる仕様は以下のとおりです。

```
int shmbind(int shmid, unsigned long paddr)
```

引数は以下のように定義されます：

shmid	物理メモリの一部へバインドしたい共有メモリ領域の識別子
paddr	指定した共有メモリ領域をバインドしたいメモリの開始物理アドレス

shmatおよび**shmdt**システムコール

共有メモリ操作のシステムコール**shmat**と**shmdt**は、呼び出し元プロセスのアドレス空間へ共有メモリ領域の結合および分離をするために使用されます。本セクションは**shmat**と**shmdt**のシステムコールを説明します。更なる詳細な情報は**shmop(2)**のmanページを参照してください。これらの呼び出しの使用を説明しているプログラムは、**README.shmop.txt**内に提供された多くのコメントと共に**/usr/share/doc/ccur/examples/shmop.c**で見つけることが可能です。

概要

```
#include <sys/types.h>
#include <sys/shm.h>

void *shmat (int shmid, const void *shmaddr, int shmflg);
int shmdt (const void *shmaddr);
```

上記の全てのインクルードファイルは、オペレーティング・システムの**/usr/include**サブディレクトリにあります。

共有メモリ領域の結合

shmatシステムコールは*shmid* で指定された呼び出し元プロセスのアドレス空間に共有メモリ領域を結合します。これは文字列のポインタ値を返します。正常終了でその値はプロセスが共有メモリ領域に結合されているメモリのアドレスになり、失敗時は値が-1となります。

shmid 引数は有効な負ではない整数値でなければなりません。これは前述の**shmget**システムコールを使って作成されている必要があります。

shmaddr 引数を**shmat**システムコールへ渡す際にゼロもしくはユーザー指定とすることが可能です。もしそれがゼロの場合、オペレーティング・システムは共有メモリ領域が結合されるアドレスを選びます。もしそれがユーザー指定の場合、そのアドレスはプログラムのアドレス空間内の有効なページ境界アドレスである必要があります。以下に典型的なアドレスの範囲を例示します。

```
0xc00c0000
0xc00e0000
0xc0100000
0xc0120000
```

オペレーティング・システムがアドレスを選ぶことを許可することは移植性を向上させます。

shmflg 引数は**shmat**システムコールへSHM_RND(切り捨て)やSHM_RDONLY(読み取り専用)フラグを渡すために使用されます。

共有メモリ領域の分離

shmdtシステムコールは呼び出し元プロセスから*shmaddr* により指定されたアドレスにある共有メモリ領域を分離します。これは正常終了で0、それ以外で-1の整数値を返します。

共有メモリ・ユーティリティ

RedHawk Linuxは共有メモリ領域の利用を容易にする2つのユーティリティを提供します。**shmdefine(1)**ユーティリティは協同プロセスが使用する1つ以上の共有メモリ領域を作成することが可能です。**shmconfig(1)**コマンドは共有メモリ領域を作成し、物理メモリの一部へバインドすることが可能です。これらのユーティリティはこの後のセクションで説明します。

shmdefineユーティリティ

shmdefineユーティリティは一連の協同プロセスが共有メモリの利用を容易にするために設計されました。例え1つ以上の共有メモリ領域を協同する多数のプログラムがあったとしても、1度だけユーティリティを呼び出すことが必要です。**shmdefine**はソース・オブジェクト・ファイルへリンクする必要のあるオブジェクト・ファイルを生成するため、リンクする以前に呼び出す必要があります。

RedHawk Linuxシステム上で実行するプログラム用に**shmdefine**は現在GNU C, Fortran, Adaコンパイラ(`gcc`, `g77` `GNAT`)で動作します。

このユーティリティの使用に関する詳細は、*Quick Reference for shmdefine* (文書番号0898010) と**shmdefine(1)**のmanページを参照してください。

shmconfigコマンド

shmconfig(1)コマンドは特定のキーに対応する共有メモリ領域を作成し、特定I/Oメモリの一部へ任意にバインドすることを支援します。

コマンドの構文：

/usr/bin/shmconfig -i *DEVSTR*
/usr/bin/shmconfig -b *BARSTR* [**-s** *SIZE*] [**-g** *GROUP*] [**-m** *MODE*] [**-u** *USER*]
{*key* | **-t** *FNAME*}
/usr/bin/shmconfig -s *SIZE* [**-p** *ADDR*] [**-g** *GROUP*] [**-m** *MODE*] [**-u** *USER*]
{*key* | **-t** *FNAME*}

共有メモリ領域へ割り当てるNUMAメモリ・ポリシーに関する情報については、10章または**shmconfig(1)**のmanページを参照してください。

オプションは表3-3で説明しています。

表3-3 shmconfig(1) コマンドのオプション

Option	Description
--info=*DEVSTR*, **-i** *DEVSTR*	以下で構成される*DEVSTR* にマッチしている各デバイス上の各メモリ領域に関する情報を出力 *vendor_id*:*device_id* **--bind**を使用すると役に立ちます。 *DEVSTR* 上の情報に関しては**--bind**を参照してください。
--bind=*BARSTR*, **-b** *BARSTR*	共有領域へバインドするためにメモリ内のI/O領域を特定します。*BARSTR* は以下で構成されます。 *vendor_id*:*device_id*:*bar_no*[:*dev_no*] *vendor_id* と *device_id* はハードウェア・デバイスを特定し、通常2つの16進数の値をコロンで区切って表します(例 8086:100f)。ベンダーのマニュアル、**/usr/share/hwdata/pci.ids**、**lspci –ns**より取得することが可能です。これらのIDを指定する時、接頭語"0x"を必要とします(例 0x8086:0x100f)。後述の「見本」を参照してください。 *bar_no* はバインドするメモリ領域を特定します。この値を取得するために**-i** オプションを使用します(出力は"Region *bar_no*: Memory at ..." と表示されます)。メモリ領域だけをバインすることが可能です。 *dev_no* は任意、ベンダーIDとデバイスIDがマッチしている複数のボード間を識別するためだけに必要です。この値を取得するために**-i**オプションを使用します(出力は"Logical device: *dev_no*:" と表示されます)。 このオプションを使用するためにユーザーはCAP_SYS_RAWIO権限を持っている必要があります。
--size=*SIZE*, **-s** *SIZE*	バイトで領域のサイズを指定します。**--bind**は必須ではなく、デフォルトは全てのメモリ領域です。
--physical=*ADDR*, **-p** *ADDR*	領域をバインドする物理I/Oメモリの一部の開始アドレスとして*ADDR* を指定します。このオプションは廃止されましたので、**--bind**を使用してください。このオプションを使用するためにユーザーはCAP_SYS_RAWIO権限を持っている必要があります。
--user=*USER*, **-u** *USER*	共有メモリ領域所有者のログイン名称を指定します。
--group=*GROUP*, **-g** *GROUP*	領域へのグループアクセスを適用するグループ名称を指定します。
--mode=*MODE*, **-m** *MODE*	共有メモリ領域へのアクセスを管理するパーミッションのセットとして*mode* を指定します。パーミッションを指定するために8進数を使用する必要があります。
--help, -h	利用可能なオプションと使用方法について説明します。
--version, -v	コマンドのバージョンを印字します。

/procと**/sys**ファイルシステムはこのコマンドを使用するためにマウントされている必要があります。

-s引数により指定された領域のサイズは、そこに配置されるデータのサイズと一致している必要があることに注意することは重要です。もし**shmdefine**が使用されている場合、領域のサイズは共有メモリの一部であると宣言されている変数のサイズと一致している必要があります。より大きなサイズの指定でも機能します(**shmdefine**に関する情報は、「shmdefineユーティリティ」を参照してください)。

ユーザーとグループに関連する領域を識別し、アクセスを制御するパーミッションを設定するために**-u**, **-g**, **-m**オプションを指定することを推奨します。もし指定されていない場合、領域のデフォルトのユーザーIDおよびグループIDはそれらの所有者で、デフォルトのモードは0644です。

key 引数は共有メモリ領域用にユーザーが選択した識別子を表します。この識別子は整数もしくは既存ファイルを参照する標準的なパス名称とすることが可能です。パス名称が提供される時、`ftok(key,0)` は**shmget(2)**コールのためにキーとなるパラメータとして使用されます。

--tmpfs=*FNAME* / **-t** *FNAME* はキーの代わりにtmpfsファイルシステムのファイル名称を指定するために使用することが可能です。**-u, -g, -m**オプションはこの領域のファイル属性を設定もしくは変更するために使用することが可能です。

shmconfigが実行される時、内部のデータ構造体と共有メモリ領域は指定されたキーに対して作成され、もし**-p**オプションが使用される場合、共有メモリ領域はI/Oメモリの連続する領域にバインドされます。

shmconfigで作成された共有メモリ領域へのアクセスするため、プロセスは領域の識別子を取得するために最初に**shmget(2)**を呼び出す必要があります。この識別子は共有メモリ領域を操作する他のシステムコールで必要になります。**shmget**の仕様は以下のとおりです。

int shmget(*key***,** *size***, 0)**

key の値は**shmconfig**で指定された*key* の値によって決定されます。もし*key* の値が整数だった場合、その整数は**shmget**の呼び出しで*key* に指定される必要があります。もし*key* の値がパス名称だった場合、**shmget**の呼び出しで*key* に指定したパス名称に基づく整数値を取得するために最初に**ftok**サブルーチンを呼び出す必要があります。パス名称からキーへ変更するときに**shmconfig**は*id* がゼロの**ftok**を呼び出すため、**ftok**の呼び出しにおける*id* 引数の値はゼロである必要があることに注意することが重要です。*size* の値は**shmconfig**の**-s**引数で指定したバイト数と等しくする必要があります。共有メモリ領域が既に作成されたため、ゼロの値は*flag* 引数として指定されます。

shmgetに関するすべての情報は「shmgetシステムコール」を参照してください。**ftok**の使用方法については、「共有メモリの利用」と**ftok(3)**のmanページを参照してください。グローバル・リソースとして処理するためにマッピングされたメモリの領域を作成する時、**shmconfig**を呼び出すために**/etc/init.d**ディレクトリ内の**shmconfig**スクリプトへ行を追加することにより有用であると感じるかもしれません。そうすることで、非協同プロセスがそれを使用する機会を得る前にIPCキーを予約することが可能となり、共同プロセスが領域の使用が必要となる前に共有メモリ領域と物理メモリ間のバインドを確立することが可能となります。以下の例のような行を追加してください。

```
/usr/bin/shmconfig -p 0xf00000 -s 0x10000 -u root -g sys -m 0666 key
```

見本

この見本では、RCIM上の物理メモリ領域を**lspci(8)**使って確認し、共有メモリ領域へバインドします。**lspci**を使用するためにはルートである必要があることに注意してください。もしルート権限を持っていない場合、**/usr/share/hwdata/ pci.ids**を見てデバイス名称(RCIM)を探すことが可能で、IDの値はベンダー/デバイスの記述の左側に列挙されます。2つ以上のデバイスIDが同一デバイスとして列挙されている時、どれを使用するかを決めるために列挙された各*device_id*で**shmconfig –i**を実行します。

1. RCIMボードの*bus:slot.func* 識別子を見つけます：

```
# lspci -v | grep -i rcim
0d:06.0 System peripheral: Concurrent Computer Corp RCIM II
Realtime Clock ...
```

2. *vendor_id:device_id* 番号を取得するためにRCIM識別子を使用します：

```
# lspci -ns 0d:06.0
0d:06.0 Class 0880: 1542:9260 (rev 01)
```

3. このデバイスのメモリ領域を見つけます。**lspci**は*vendor_id:device_id* の値を接頭語" 0x" なしの16進数形式(1542:9260) で出力しますが、**shmconfig**はベース識別子(0x1542:0x9260)を必要とすることに注意してください。

```
# shmconfig -i 0x1542:0x9260
Region 0: Memory at f8d04000 (non-prefetchable) [size=256]
/proc/bus/pci0/bus13/dev6/fn0/bar0
Region 1: I/O ports at 7c00 [size=256]
/proc/bus/pci0/bus13/dev6/fn0/bar1
Region 2: Memory at f8d00000 (non-prefetchable) [size=16384]
/proc/bus/pci0/bus13/dev6/fn0/bar2
```

4. RCIMメモリ領域#2 へバインドします：

```
# shmconfig -b 0x1542:0x9260:2 -m 0644 -u me -g mygroup 42
```

5. システム上のIPC共有メモリ領域を確認します。`physaddr` はバインドした物理アドレスを
表し、上述のステップ3の**shmconfig –i**コマンドにより出力されたアドレスと一致することに注意してください。

```
# cat /proc/sysvipc/shm
     key     shmid perms        size   cpid   lpid  nattch    uid
gid  cuid  cgid       atime       dtime        ctime  physaddr
      42         0   644       16384   1734      0       0   5388
100     0     0           0           0   1087227538  f8d00000
```

4
プロセス・スケジューリング

本章ではRedHawk Linuxシステム上におけるプロセス・スケジューリングの概要を提供します。どのようにプロセス・スケジューラが次に実行するプロセスを決定するのかを説明し、POSIXスケジューリング・ポリシーと優先度を説明します。

概要

RedHawk Linux OSの中で、スケジュール可能な存在は常にプロセスです。スケジューリング優先度とスケジューリング・ポリシーはプロセスの属性です。システム・スケジューラはプロセスが実行される時に決定します。それは構成パラメータ、プロセスの性質、ユーザー要求に基づいて優先度を保持し、CPUへプロセスを割り当てるためにこれらの優先度と同様にCPUアフィニティを使用します。

スケジューラは4つの異なるスケジューリング・ポリシー、1つは非クリティカルなプロセス用(SCHED_OTHER)、1つはバックグランドでCPUに負荷をかけるプロセス用(SCHED_RATCH)、2つはリアルタイム・アプリケーション用に固定優先度ポリシー(SCHED_FIFO とSCHED_RR) を提供します。これらのポリシーは、4-3ページの「スケジューリング・ポリシー」セクションで詳細が説明されています。

デフォルトでは、スケジューラはタイムシェアリング・ポリシーのSCHED_OTHERを使います。SCHED_OTHERポリシーの中のプロセスに対し、双方向プロセスには優れた応答時間、CPU集中型プロセスには優れたスループットを提供しようとするため、スケジューラは実行可能なプロセスの優先度を動的に操作します。

固定優先度スケジューリングはプロセス毎を基準に静的優先度を設定することが可能です。スケジューラは固定優先度スケジューリング・ポリシーを使用するプロセスの優先度を決して変更しません。例え他のプロセスが実行可能であるとしても、最も高いリアルタイム固定優先度プロセスは常に実行可能なCPUを直ぐに確保します。従って、設定されているプロセス優先度に応じてプロセスが動作する正確な順番をアプリケーションは指定することが可能です。

リアルタイム性能を必要としないシステム環境では、デフォルトのスケジューラの設定は十分に機能し、固定優先度プロセスは必要とされません。しかし、リアルタイム・アプリケーションもしくは厳格なタイミングな制約を持つアプリケーションのために固定優先度プロセスはクリティカルなアプリケーションの要求が満たされることを保証する唯一の方法です。特定のプログラムが非常にデターミニスティックな応答時間を要求する時、固定優先度スケジューリング・ポリシーを使用する必要があり、最もデターミニスティックな応答が必要なタスクは最も適した優先度を割り付ける必要があります。

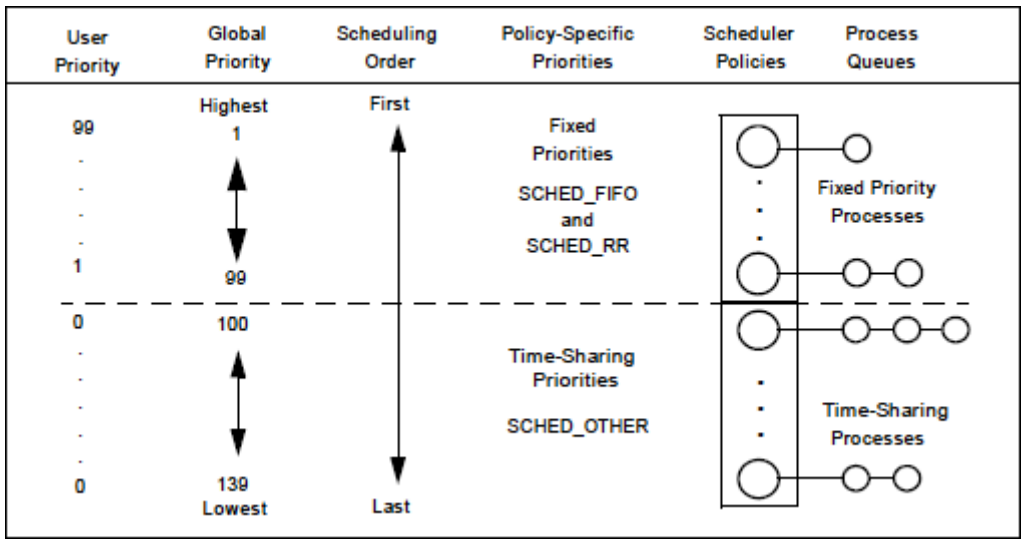
IEEE規格1003.1bに基づくシステムコール一式は、プロセスのスケジューリング・ポリシーおよび優先度へのダイレクトなアクセスを提供します。このシステムコール一式に含まれているのは、プロセスがスケジューリング・ポリシーおよび優先度を取得もしくは設定することを許可するシステムコールで、特定のスケジューリング・ポリシーに関連する優先度の最小値・最大値を取得し、ラウンドロビン(SCHED_RR)・スケジューリング・ポリシーに基づいてスケジュールされたプロセスのタイム・クォンタムを取得。

run(1)コマンドの使用により、コマンド・レベルでプロセスのスケジューリング・ポリシーと優先度を変更することが可能となります。システムコールと**run**コマンドは効果的な使用のための手順とヒントと共に本章で後述されています。

プロセス・スケジューラの管理方法

図4-1にスケジューラの操作方法を図示します。

図4-1 スケジューラ



プロセスが作成される時、ポリシーの範囲内でスケジューリング・ポリシーと優先度を含むスケジューリング・パラメータを継承します。デフォルトの構成では、プロセスはSCHED_OTHERポリシーでスケジュールされたタイムシェアリング・プロセスとして開始します。プロセスが固定優先度ポリシーで正しくスケジュールされるためには、ユーザー要求がシステムコールもしくは**run(1)**コマンドを介して行われる必要があります。

プロセスの優先度を設定する時、プロセスは"User Priority(ユーザー優先度)" を設定します。これはユーザーが現在の優先度を取り出すときに呼び出す**sched_getparam(2)**によって報告される優先度でもあります。移動可能なアプリケーションは特定のスケジューリング・ポリシー用の有効な優先度の埴を判断するために**sched_get_priority_min()**と**sched_get_priority_max()**のインターフェースを使用する必要があります。ユーザー優先度の値(`sched_priority`) は各プロセスに割り当てられます。SCHED_OTHERプロセスは0のユーザー優先度が割り当てられるだけです。SCHED_FIFOとSCHED_RRプロセスは1から99の範囲内のユーザ優先度を持っています。

スケジューラはポリシー固有優先度(Policy-Specific Priorities)からグローバル優先度(Global Priorities)へスケジューリングを変更します。グローバル優先度はカーネル内部で使用されるスケジューリング・ポリシーの値です。スケジューラは見込まれる各グローバル優先度の値に対して実行可能なプロセスの一覧を保持します。SCHED_OTHER スケジューリング・ポリシーに対応する40個のグローバル・スケジューリング優先度で、固定優先度スケジューリング・ポリシー(SCHED_RR and SCHED_FIFO) に対応する99個のグローバル・スケジューリング優先度。スケジューラは空ではない最も高いグローバル優先度のリストを探して、現在のCPU上で実行するためにそのリストの先頭のプロセスを選びます。

スケジューリング・ポリシーは、リスト内のプロセスがブロックされるもしくは実行可能となる時、リスト内でユーザー優先度とプロセスの相対位置が等しいプロセスのリストへ挿入される各プロセスについて決定します。

固定優先度プロセスが特定CPUですぐに実行可能である間は、タイムシェアリング・プロセスがそのCPU上で実行することはありません。

一度スケジューラがCPUへプロセスを割り付けたら、プロセスはそのタイム・クォンタム使い切る、スリープする、高優先度プロセスによりブロックもしくまプリエンプトされるまで実行されます。

ps(1)と**top(1)**により表示される優先度は内部で計算された値でユーザーに設定された優先度を間接的に反映するだけであることに注意してください。

スケジューリング・ポリシー

POSIXはプロセスをスケジュールする方法を制御する3種類のスケジューリング・ポリシーを定義：

SCHED_FIFO	ファーストイン・ファーストアウト(FIFO)
SCHED_RR	ラウンドロビン (RR)
SCHED_OTHER	既定値のタイムシェアリング

ファーストイン・ファーストアウト・スケジューリング(SCHED_FIFO)

SCHED_FIFOは0より高いユーザー優先度でのみ使用することが可能です。これはSCHED_FIFOプロセスが実行可能となった時、現在実行中のどのようなSCHED_OTHERプロセスであっても常に即座にプリエンプトすることを意味します。SCHED_FIFOはタイム・スライシングのない単純なスケジューリングのアルゴリズムです。SCHED_FIFO優先度でスケジュールされたプロセスに対し、次のルールが適用されます：高優先度の他のプロセスにプリエンプトされたSCHED_FIFOプロセスはその優先度リストの先頭に留まり、全ての高優先度プロセスが再びブロックされたら直ぐに実行を再開します。SCHED_FIFOプロセスが実行可能となった時、その優先度リストの最後尾に挿入されます。もし実行可能であった場合、**sched_setscheduler(2)**もしくは**sched_setparam(2)**の呼び出しはリストの最後尾にあるPIDに一致するSCHED_FIFOプロセスを配置します。**sched_yield(2)**を呼び出すプロセスはその優先度リストの最後尾へ配置されます。その他のイベントはユーザー優先度が等しい実行可能なプロセス待ちリストの中でSCHED_FIFO優先度でスケジュールされたプロセスは移動しません。SCHED_FIFOプロセスは、I/O要求によりブロック、高優先度プロセスによるプリエンプト、**sched_yield**を呼び出すまで実行されます。

ラウンドロビン・スケジューリング(SCHED_RR)

SCHED_RRはSCHED_FIFOの単純な拡張機能です。各プロセスはタイム・クォンタムを最大限使って実行することが許可されていることを除いては、上述のSCHED_FIFO全てがSCHED_RRに適用されます。もしSCHED_RRプロセスが周期時間分もしくはタイム・クォンタムより長く実行している場合、その優先度リストの最後尾へ配置されます。高優先度プロセスにプリエンプトされ、その後実行プロセスとして実行を再開するSCHED_RRプロセスは、割り当てられたそのラウンドロビンのタイム・クォンタムを使い切らずに終了します。タイム・クォンタムの長さは**sched_rr_get_interval(2)**で取り出すことが可能です。タイム・クォンタムの長さはSCHED_RRスケジューリング・ポリシーでスケジューリングされたプロセスに対応するナイス値に影響されます。高いナイス値は大きなタイム・クォンタムを割り当てられます。

タイムシェアリング・スケジューリング(SCHED_OTHER)

SCHED_OTHERはユーザー優先度0でのみ使用することが可能です。SCHED_OTHERは特別なリアルタイム・メカニズムのユーザー優先度を必要としない全てのプロセスを対象とする一般的なタイムシェアリングのスケジューラ・ポリシーです。実行されるプロセスは、リストの中だけで決定される動的な優先度に基づくユーザー優先度0のリストから選ばれます。動的な優先度はナイス・レベル(**nice(2)**もしくは**setpriority(2)**システムコールにより設定されます)に基づいており、実行可能な各プロセスのタイム・クォンタムのために増やされますが、スケジューラにより実行を拒否されます。これは全てのSCHED_OTHERプロセス間で公平な進行を保証します。例えば、I/O操作の実行によりプロセス自身が自主的にブロックする回数といったようなその他の要因も考慮します。

バッチ・スケジューリング(SCHED_BATCH)

SCHED_BATCHは静的優先度0でのみ使用することが可能です。このポリシーは自身の(ナイス値に基づく)動的優先度に応じてプロセスをスケジュールするSCHED_OTHERに類似しています。違いはSCHED_BATCHはプロセスがCPUに負荷をかけるものとスケジューラが常に仮定することです。その結果、スケジューラは起動する動作に対して小さなスケジューリング・ペナルティを適用するので、このプロセスはスケジューリングの決定において少し冷遇されます。

SCHED_BATCHはナイス値を下げたくない場合以外は非対話型の負荷、および(負荷のあるタスク間で)相互に余計なプリエンプションを引き起こすことがないデターミニスティックなスケジューリング・ポリシーが必要な負荷に対して便利です。

性能向上のための手続き

優先度設定方法

次の部分的なコードは現在のプロセスを60の固定優先度でSCHED_RR固定優先度スケジューリング・ポリシーに配置します。POSIXスケジューリング・ルーチンに関する情報は本章の「プロセス・スケジューリング・インターフェース」セクションを参照してください。

```
#include <sched.h>
...
struct sched_param sparms;

sparms.sched_priority = 60;
if (sched_setscheduler(0, SCHED_RR, &sparms) < 0)
{
    perror("sched_setsched");
    exit(1);
}
```

割り込みルーチン

固定優先度スケジューリング・ポリシーの1つにスケジュールされたプロセスは、ソフトIRQやタスクレットに関連する処理よりも高い優先度に割り付けられます。これらの割り込みルーチンは与えられたCPU上で実行した割り込みルーチンの代わりに作業を実行します。実際の割り込みルーチンはハードウェア割り込みレベルで実行され、(固定優先度スケジューリング・ポリシーの1つにスケジュールされたプロセスを含む)CPU上の全ての機能にプリエンプトします。Linuxのデバイス・ドライバ作成者は、デバイスが割り込みをハンドルされたと確信させるためにデバイスとのやり取りに要求される仕事量を最小限で実行することを奨励します。デバイス・ドライバはデバイス割り込みルーチンに関連する作業の残りを処理するための割り込みメカニズムの1つを起動することが出来ます。固定優先度プロセスはそれらの割り込みルーチンより上の優先度で実行されているため、この割り込み構造は固定優先度プロセスが割り込みルーチンから見込まれる最小限のジッター量を得ることが可能となります。デバイス・ドライバーの割り込みルーチンに関する詳細な情報については「デバイス・ドライバ」章を参照してください。

SCHED_FIFO vs SCHED_RR

2つの固定優先度スケジューリング・ポリシーはその性質がとても似ており、殆どの条件下で同一の作法で動作します。SCHED_RRがプロセスで使えるタイム・クォンタムを所有している間にタイム・クォンタムを使い切った時、もし固定優先度スケジューリング・ポリシーの1つの中に優先度の等しい実行可能な状態のプロセスが存在する場合、プロセスはCPUを放棄するだけであることを覚えることが重要です。もし優先度の等しい実行可能な状態のプロセスが無い場合、スケジューラは当初のSCHED_RRプロセスがそのCPU上で実行可能な最高優先度プロセスで有り続け、同一プロセスが実行のために再度選択されることが確定します。

これは、全く同じスケジューリング優先度にて固定優先度スケジューリング・ポリシーの1つにスケジュールされた実行中のプロセスが複数存在する場合、SCHED_FIFOとSCHED_RRでスケジュールされたプロセス間の違いが唯一時間だけであることを意味します。この場合、SCHED_RRはプロセスに割り当てられたタイム・クォンタムに従いCPUを共有することをそれらのプロセスに許可します。プロセスのタイム・クォンタムは**nice(2)**システムコールにより影響を受けることに注意してください。より高いナイス値を持つプロセスは大きなタイム・クォンタムが割り当てられます。プロセスのタイム・クォンタムは**run(1)**コマンドを介して変更することも可能です(詳細は本章の「runコマンド」を参照してください)。

CPUをロックする固定優先度プロセス

SCHED_FIFOとSCHED_RRのスケジューリング・ポリシーでスケジュールされたプロセスの非ブロック無限ループはすべての低優先度プロセスを無期限にブロックします。このシナリオが完全に他のプロセスのCPUを奪うため、予防策としてこれを避ける必要があります。

ソフトウェア開発中、プログラマはテスト中のアプリケーションよりも高いユーザー優先度にスケジュールされたシェルをコンソール上で利用可能な状態を保つことにより、このような無限ループを中断することができます。これは予想通りにブロックしないもしくは終了しないテスト中のリアルタイム・アプリケーションの緊急停止を可能にします。SCHED_FIFOおよびSCHED_RRプロセスは絶えず他のプロセスをプリエンプトすることが可能であるため、ルート・プロセスもしくはCAP_SYS_NICEケーパビリティを持つプロセスだけはそれらのポリシーを有効にすることが許可されます。

メモリのロック

ページングとスワッピングは大抵の場合、アプリケーション・プログラムに予測不可能な量のシステム・オーバヘッド時間を付加します。ページングとスワッピングが原因の性能ロスを排除するため、物理メモリ内のプロセスの仮想アドレス空間全てもしくは一部をロックするために **mlockall(2)**, **munlockall(2)**, **mlock(2), munlock(2)**各システムコールおよびRedHawk Linuxの **mlockall_pid(2)**システムコールを使用します。

CPUアフィニティとシールド・プロセッサ

システム内の各プロセスはCPUアフィニティ・マスクを持っています。CPUアフィニティ・マスクはどのCPU上でプロセスの実行を許可するかを決定します。CPUがプロセスからシールドされている時、そのCPUでは、シールドされたCPUだけを含むCPUセットとなるCPUアフィニティが明示的に設定されたプロセスだけを実行します。これらのテクニックの利用は、プロセスの実行をどこでどのように制御するかが更に加わります。詳細な情報については「リアルタイム性能」章を参照してください。

プロセス・スケジューリング・インターフェース

IEEE規格1003.1bに基づくシステムコール一式はプロセスのスケジューリング・ポリシーおよび優先度への直接アクセスを提供します。**run(1)**コマンドの使用によりコマンド・レベルでプロセスのスケジューリング・ポリシーおよび優先度を変更しても構いません。システムコールについては後述します。**run**コマンドについては4-14ページに詳述されています。

POSIXスケジューリング・ルーチン

後に続くセクションでPOSIXのスケジューリング・システムコールの使用手順を説明します。これらのシステムコールを以下で簡単に説明します。

スケジューリング・ポリシー：

sched_setscheduler プロセスのスケジューリング・ポリシーと優先度を設定
sched_getscheduler プロセスのスケジューリング・ポリシーを取得

スケジューリング優先度：

sched_setparam プロセスのスケジューリング優先度を変更
sched_getparam プロセスのスケジューリング優先度を取得

CPUの放棄：

sched_yield CPUの放棄

最低/最高優先度：

sched_get_priority_min	スケジューリング・ポリシーに対応する最低優先度を取得
sched_get_priority_max	スケジューリング・ポリシーに対応する最高優先度を取得

ラウンドロビン・ポリシー：

sched_rr_get_interval	SCHED_RRスケジューリング・ポリシースケジュールされたプロセスに対応するタイム・クォンタムを取得

sched_setschedulerルーチン

sched_setscheduler(2)システムコールはスケジューリング・ポリシーと関連するパラメータをプロセスへ設定することが可能です。

sched_setschedulerは、(1)プロセスのスケジューリング・ポリシーをSCHED_FIFOもしくはCHED_RRへ変更する、もしくは、(2)SCHED_FIFOもしくはCHED_RRにスケジューリングされたプロセスの優先度を変更するために呼び出して使用すること、以下の条件の1つを満たす必要があることに注意することが重要です：

- 呼び出し元プロセスはルート権限を所有している必要がある
- 呼び出し元プロセスの有効ユーザーID(uid) はターゲットプロセス(スケジューリング・ポリシーと優先度が設定されているプロセス)の有効ユーザーIDと一致している必要がある、もしくは呼び出し元プロセスはユーパーユーザーもしくはCAP_SYS_NICE ケーパビリティを所有している必要がある

概要

```
#include <sched.h>

int sched_setscheduler(pid_t pid, int policy, const struct sched_param *p);

struct sched_param {
    ...
    int sched_priority;
    ...
};
```

引数は以下のように定義されます：

pid スケジューリング・ポリシーと優先度が設定さているプロセスのプロセス識別番号(PID)。現在のプロセスを指定するには*pid* の値をゼロに設定します。

policy <**sched.h**>ファイル内に定義されているスケジューリング・ポリシー。*policy* の値は以下の1つである必要があります：

SCHED_FIFO	ファーストイン・ファーストアウト(FIFO)
SCHED_RR	ラウンドロビン(RR)
SCHED_OTHER	タイムシェアリング

p *pid* で識別されるプロセスのスケジューリング優先度を指定する構造体へのポインタ。優先度は、指定されたポリシーに対応するスケジューラ・クラスに定義される優先度の範囲内にある整数値です。次のシステムコールの1つを呼び出すことにより対応するポリシーの優先度の範囲を判断することが可能です：**sched_get_priority_min**もしくは**sched_get_priority_max**(これらのシステムコールの説明については4-12ページを参照してください)。

もし指定したプロセスのスケジューリング・ポリシーと優先度の正常に設定された場合、**sched_setscheduler**システムコールはプロセスの以前のスケジューリング・ポリシーを返します。戻り値-1はエラーが発生したことを示し、`errno`はエラーを知らせるため設定されます。発生する可能性のあるエラーの種類の一覧は**sched_setscheduler(2)**のmanページを参照してください。もしエラーが発生した場合、プロセスのスケジューリング・ポリシーと優先度は変更されません。

プロセスのスケジューリング・ポリシーを変更した時、その新しいタイム・クォンタムもポリシーと優先度に関連するスケジューラに定義されているデフォルトのタイム・クォンタムへ変更される事に注意することが重要です。**run(1)**コマンドの使用によりコマンド・レベルでSCHED_RRスケジューリング・ポリシーにスケジュールされたプロセスのタイム・クォンタムを変更することが可能です(このコマンドの情報については4-14ページを参照してください)。

sched_getschedulerルーチン

sched_getscheduler(2)システムコールは指定したプロセスのスケジューリング・ポリシーを取得することが可能です。スケジューリング・ポリシーは次のように<**sched.h**>ファイル内に定義されています：

SCHED_FIFO	ファーストイン・ファーストアウト(FIFO)
SCHED_RR	ラウンドロビン(RR)
SCHED_OTHER	タイムシェアリング

概要

```
#include <sched.h>

int sched_getscheduler(pid_t pid);
```

引数は以下のように定義されます：

pid スケジューリング・ポリシーを取得したいプロセスのプロセス識別番号(PID)。現在のプロセスを指定するには*pid* の値をゼロに設定します。

もし呼び出しが成功した場合、**sched_getscheduler**は指定されたプロセスのスケジューリング・ポリシーを返します。戻り値-1はエラーが発生したことを示し、`errno` はエラーを知らせるため設定されます。発生する可能性のあるエラーの種類の一覧は**sched_getscheduler(2)**のmanページを参照してください。

sched_setparamルーチン

sched_setparam(2)システムコールは指定されたプロセスのスケジューリング・ポリシーと関連するスケジューリング・パラメータを設定することが可能です。

sched_setparamは、SCHED_FIFOもしくはSCHED_RRにスケジュールされたプロセスのスケジューリング優先度を変更するために呼び出して使用すること、以下の条件の1つを満たす必要があることに注意することが重要です：

- 呼び出し元プロセスはルート権限を所有している必要がある
- 呼び出し元プロセスの有効ユーザーID(uid) はターゲットプロセス(スケジューリング・ポリシーと優先度が設定されているプロセス)の有効ユーザーIDと一致している必要がある、もしくは呼び出し元プロセスはユーパーユーザーもしくはCAP_SYS_NICE ケーパビリティを所有している必要がある

概要

```
#include <sched.h>

int sched_setparam(pid_t pid, const struct sched_param *p);

struct sched_param {
    ...
    int sched_priority;
    ...
};
```

引数は以下のように定義されます：

pid スケジューリング優先度を変更するプロセスのプロセス識別番号(PID)。現在のプロセスを指定するには*pid* の値をゼロに設定します。

p *pid* で識別されるプロセスのスケジューリング優先度を指定する構造体へのポインタ。優先度は、プロセスの現在のスケジューリング・ポリシーに対応する優先度の範囲内にある整数値です。高い数値はより有利な優先度とスケジューリングを表します。

sched_getscheduler(2)システムコールの呼び出しによりプロセスのスケジューリング・ポリシーを取得することが可能です(このシステムコールの説明は4-8ページを参照してください)。**sched_get_priority_min(2)**および**sched_get_priority_max(2)**システムコールを呼び出すことにより対応するポリシーの優先度の範囲を判断することが可能です。(これらのシステムコールの説明は4-12ページを参照してください)

戻り値0は指定したプロセスのスケジューリング優先度の変更が成功したことを表します。戻り値-1はエラーが発生したことを示し、errnoはエラーを知らせるため設定されます。発生する可能性のあるエラーの種類の一覧は**sched_setparam(2)**のmanページを参照してください。もしエラーが発生した場合、プロセスのスケジューリング優先度は変更されません。

sched_getparamルーチン

sched_getparam(2)システムコールは指定したプロセスのスケジューリング・パラメータを取得します。

概要

```
#include <sched.h>

int sched_getparam(pid_t pid, struct sched_param *p);

struct sched_param {
    ...
    int sched_priority;
    ...
};
```

引数は以下のように定義されます：

pid スケジューリング優先度を取得したいプロセスのプロセス識別番号(PID)。現在のプロセスを指定するには*pid* の値をゼロに設定します。

p *pid* で識別されるプロセスのスケジューリング優先度を返す構造体へのポインタ。

戻り値0は**sched_getparam**の呼び出しが成功したことを表します。指定したプロセスのスケジューリング優先度は、*p* が示す構造体の中に返されます。発生する可能性のあるエラーの種類の一覧は、**sched_getparam(2)**のmanページを参照してください。

sched_yieldルーチン

sched_yield(2)システムコールは、呼び出し元プロセスが再び実行可能な状態の最高優先度プロセスになるまでCPUを放棄することを許可します。**sched_yield**の呼び出しは、呼び出し元プロセスと優先度が等しいプロセスが実行可能な状態である場合にのみ有効であることに注意してください。このシステムコールは、呼び出し元プロセスよりも低い優先度のプロセスの実行を許可するために使用することは出来ません。

概要

```
#include <sched.h>

int sched_yield(void);
```

戻り値0は**sched_yield**の呼び出しが成功したことを表します。戻り値-1はエラーが発生したことを示します。`errno`はエラーを知らせるため設定されます。

sched_get_priority_minルーチン

sched_get_priority_min(2)システムコールは指定したスケジューリング・ポリシーに対応する最も低い優先度を取得することが可能です。

概要

```
#include <sched.h>

int sched_get_priority_min(int policy);
```

引数は以下のように定義されます：

policy ファイル内に定義されるスケジューリング・ポリシー。*policy* の値は以下の1つである必要があります。

SCHED_FIFO	ファーストイン・ファーストアウト(FIFO)
SCHED_RR	ラウンドロビン(RR)
SCHED_OTHER	タイムシェアリング

数字的に高い優先度値を持つプロセスは数字的に低い優先度値を持つプロセスよりも前にスケジュールされます。**sched_get_priority_min**より返される値は、**sched_get_priority_max**より返される値よりも小さくなります。

RedHawk Linuxは、ユーザー優先度値の範囲がSCHED_FIFOとSCHED_RRに対しては1から99、そしてSCHED_OTHERに対しては優先度0 が許可されます。

もし呼び出しに成功した場合、**sched_get_priority_min**は指定したスケジューリング・ポリシーに対応する最も低い優先度を返します。戻り値-1はエラーが発生したことを示し、`errno`はエラーを知らせるため設定されます。発生する可能性のあるエラーの一覧を取得するには**sched_get_priority_min(2)**のmanページを参照してください。

sched_get_priority_maxルーチン

sched_get_priority_max(2)システムコールは指定したスケジューリング・ポリシーに対応する最も高い優先度を取得することが可能です。

概要

```
#include <sched.h>

int sched_get_priority_max(int policy);
```

引数は以下のように定義されます：

policy ファイル内に定義されるスケジューリング・ポリシー。*policy* の値は以下の1つである必要があります。

SCHED_FIFO	ファーストイン・ファーストアウト(FIFO)
SCHED_RR	ラウンドロビン(RR)
SCHED_OTHER	タイムシェアリング

数字的に高い優先度値を持つプロセスは数字的に低い優先度値を持つプロセスよりも前にスケジュールされます。**sched_get_priority_max**より返される値は、**sched_get_priority_min**より返される値よりも大きくなります。

RedHawk Linuxは、ユーザー優先度値の範囲がSCHED_FIFOとSCHED_RRに対しては1から99、そしてSCHED_OTHERに対しては優先度0が許可されます。

もし呼び出しに成功した場合、**sched_get_priority_max**は指定したスケジューリング・ポリシーに対応する最も高い優先度を返します。戻り値-1はエラーが発生したことを示し、`errno`はエラーを知らせるため設定されます。発生する可能性のあるエラーの一覧を取得するには**sched_get_priority_max(2)**のmanページを参照してください。

sched_rr_get_intervalルーチン

sched_rr_get_interval(2)システムコールはSCHED_RRスケジューリング・ポリシーでスケジュールされたプロセスのタイム・クォンタムを取得することが可能です。タイム・クォンタムとはカーネルがプロセスにCPUを割り当てる一定の時間です。CPUが割り当てられたプロセスがそのタイム・クォンタム分を実行しているとき、スケジューリングの決定が行われます。もし同じ優先度の他のプロセスが実行可能な状態の場合、そのプロセスがスケジュールされます。もしそうでない場合は、他のプロセスが実行され続けます。

概要

```
include <sched.h>

int sched_rr_get_interval(pid_t pid, struct timespec *tp);

struct timespec {
    time_t tv_sec; /* seconds */
    long tv_nsec; /* nanoseconds */
};
```

引数は以下のように定義されます：

pid タイム・クォンタムを取得したいプロセスのプロセス識別番号(PID)。現在のプロセスを指定するには*pid* の値をゼロに設定します。

tp *pid* で識別されるプロセスのラウンドロビン・タイム・クォンタムが返されるtimespec 構造体へのポインタ。識別されたプロセスはSCHED_RRスケジューリング・ポリシーで実行している必要があります。

戻り値0は**sched_rr_get_interval**の呼び出しが成功したことを表します。指定したプロセスのタイム・クォンタムは*tp* が示す構造体の中に返されます。戻り値-1はエラーが発生したことを示し、`errno`はエラーを知らせるため設定されます。発生する可能性のあるエラーの一覧は**sched_rr_get_interval(2)**のmanページを参照してください。

runコマンド

run(1)コマンドは、プロセス・スケジューラ属性とCPUアフィニティを制御するために使用することが可能です。このコマンドの構文は、

run [*OPTIONS*] {*COMMAND* [*ARGS*] | *PROCESS/THREAD_SPECIFIER*}

runコマンドは、オプションのリストが記述された環境で指定のコマンドを実行し、コマンドの終了値を伴って終了します。もし指定子(*SPECIFIER*)が与えられた場合、**run**は指定子のプロセス/スレッド一式の環境を変更します。指定子は以下で定義します。コマンドは同じコマンド・ラインの実施で指定子を組み合わせることはできません。

runコマンドは指定したPOSIXスケジューリング・ポリシーおよび指定した優先度でプログラムを実行することが可能です。(POSIXスケジューリング・ポリシーの説明は4-3ページを参照してください)SCHED_RRポリシーでスケジュールされたプログラムのタイム・クォンタムもやはり設定することが可能です。

プログラムのスケジューリング・ポリシーと優先度を設定するため、シェルから**run**コマンドを呼び出し、**--policy=***policy* もしくは**–s** *policy* オプションおよび**--priority=***priority* もしくは**-P** *priority* オプションの両方を指定します。*policy* の有効なキーワードは、

SCHED_FIFO or fifo	ファーストイン・ファーストアウト・スケジューリング
SCHED_RR or rr	ラウンドロビン・スケジューリング
SCHED_OTHER or other	タイムシェア・スケジューリング
SCHED_BATCH or batch	対話型ジョブを除く長期実行
SCHED_IDLE or idle	CPUがアイドルになった時に実行

priority の値は、指定するスケジューリング・ポリシー(もし**-s**オプションが使用されていない場合は現在のスケジューリング・ポリシー)が有効な整数値である必要があります。より高い数値は、より有利なスケジューリング優先度を表します。

SCHED_RRスケジューリング・ポリシーにスケジュールされたプログラムのタイム・クォンタムを設定するため、**--quantum=***quantum* もしくは**-q** *quantum* オプションも指定します。*quantum* は、-20から19までをナイス値として指定(スライス時間は-20が最も長く19が最も短い)、もしくはナイス値に対応するミリ秒の値として指定します。**--quantum=list**オプションはナイス値と対応するミリ秒の値を表示します。

スケジューリング・ポリシーを設定する時にSCHED_RESET_ON_FORK属性を適用するには、**--resetonfork**もしくは**–r**オプションを使用します。スケジューリング・ポリシーがSCHED_FIFOもしくはSCHED_RRのどちらかでこのオプションを**--policy**オプションと共に使用される時、指定されたプロセスもしくはコマンドによって続いて作成される子プロセスは、親プロセスのリアルタイム・スケジューリング・ポリシーは継承しませんが、その代わりに子プロセスはSCHED_OTHERスケジューリング・ポリシーが割り当てられます。また、**--resetonfork**オプションが使用される時、親プロセスのスケジューリング・ポリシーに関係なく、もし親プロセスのナイス値が0以下であっても子プロセスは0のナイス値が割り当てられます。**--resetonfork**オプションは**--policy**オプションと一緒に使用される時にのみ有効です。

--bias=*list* もしくは**-b** *list* オプションを使用することでCPUアフィニティを設定することが可能です。*list* は論理CPU番号のカンマ区切りリストもしくは範囲です(例：“0, 2-4, 6”)。アクティブな全てのプロセッサもしくはブート・プロセッサをそれぞれ指定するため、*list* は文字列で“active” もしくは“boot”と指定することも可能です。CAP_SYS_NICEケーパビリティは更にCPUをアフィニティへ追加するためには必要となります。

--negateもしくは**-N**オプションはCPUバイアス・リストを無効な状態にします。**--negate**オプションが指定される時、バイアス・リスト・オプションも指定される必要があります。指定されるバイアスはバイアス・リスト内に指定されたものを除くシステム上の全てのCPUを含みます。

--copies=*count* もしくは**-c** *count* オプションは、コマンドの同一コピーを指定した回数実行することが可能です。

その他のオプションで、情報の表示やバックグラウンドでのコマンド実行に利用することが可能です。NUMAメモリ・ポリシーを設定するためのオプションは10章に記述されています。詳細な情報については**run(1)**のmanページを参照してください。

PROCESS/THREAD_SPECIFIER

このパラメータは対象となるプロセスまたはスレッドを指定するために使用します。以下の1つだけを指定することが出来ます。複数のコンマ区切りの値は全ての*list* で指定することが可能で範囲は必要に応じて許可されます。

--all, **-a**	既存のプロセスとスレッドを全て指定します。
--pid=*list*, **-p** *list*	変更する既存のPIDのリストを指定します。全てのスケジューラ操作は、全てのサブスレッドを含むリストアップされた全てのプロセス・セットに限定します。
--tid=*list*, **-t** *lis*	変更する既存のTIDのリストを指定します。全てのスケジューラ操作は、リストアップされたプロセスのスレッドと不特定ではないシブリング・スレッドだけに限定します。**-l** *list* はPowerMAXとの互換性のために使用することが可能です。
--group=*list*, **-g** *list*	変更するプロセス・グループのリストを指定し、リストアップされたプロセス・グループ内の既存のプロセス全てが変更されます。
--user=*list*, **-u** *list*	変更するユーザーのリストを指定し、リストアップされたユーザが所有する既存のプロセス全てが変更されます。リスト内の各ユーザは有効なユーザーIDの数値もしくはログインIDの文字のどちらかになります。
--name=*list*, **-n** *list*	変更する既存のプロセス名称のリストを指定します。

見本

1. 次のコマンドは、**make(1)**を既定優先度の既定SCHED_OTHERスケジューリング・ポリシーでCPU0-3のいずれかのバックグラウンドにて実行します。

   ```
   run --bias=0-3 make &
   ```

2. 次のコマンドは、**date(1)**を優先度10のSCHED_RR(ラウンドロビン)スケジューリング・ポリシーにて実行します。

   ```
   run -s SCHED_RR -P 10 date
   ```

3. 次のコマンドは、プロセスIDが987のスケジューリング優先度をレベル32へ変更します。

   ```
   run --priority=32 -p 987
   ```

4. 次のコマンドは、プロセス・グループが1456の全てのプロセスをCPU3へ移動します。

   ```
   run -b 3 -g 1456
   ```

5. 次のコマンドは、名称が"pilot" の全てのプロセスを優先度21のSCHED_FIFOスケジューリング・ポリシーで実行するために設定します。

   ```
   run -s fifo -P 21 -n pilot
   ```

更なる情報は**run(1)**のmanページを参照してください。

5
プロセス間同期

本章ではRedHawk Linuxがプロセス間同期のニーズに対応するために提供するツールについて説明します。ここで説明する全てのインターフェースは、共有リソースへのアクセスを同期する協同プロセスのための手段を提供します。

マルチプロセッサ・システム内の複数のプログラムによる共有データへのアクセスを同期させるために最も効果的なメカニズムは、スピンロックを使用することです。しかし、スピンロック保持中のプリエンプションから保護するために使用している再スケジューリング変数もなしにユーザー・レベルからスピンロックを使用することは安全ではありません。

もし移植性が効率性よりも大きな問題である場合、POSIXカウンティング・セマフォとミューテックスは共有データへの同期アクセスにとって次善の選択です。プロセスがセマフォの値の交換を通じて通信することを許可するSystem V セマフォも提供されます。多くのアプリケーションが複数のセマフォの利用を必要とするため、この機能はセマフォの集合もしくは配列を作ることが可能となります。

同期する協同プロセスの共有メモリ内データへのアクセスに関する問題は、Concurrentがこれらの問題に対する解決策を提供するために開発したツールも加えて説明します。

プロセス間同期の理解

マルチプロセスのリアルタイム・アプリケーションは、同じリソース一式—例えば、I/Oバッファ、ハードウェア・デバイス・ユニット、クリティカル・セクション・コード—へのアクセスの調整を協同プロセスに許可する同期メカニズムを必要とします

RedHawk Linuxは数々のプロセス間同期ツールを提供します。それには、再スケジューリングに対するプロセスの脆弱性の制御、ビジーウェイト相互排他メカニズムプロセスのアクセスを含むクリティカル・セクションへのプロセスのアクセスの整列、クリティカル・セクションに対する相互排他のためのセマフォ、プロセス間双方向通信の調整のためのツールが含まれます。

共有メモリの利用を通して仮想メモリ空間の一部を共有する2つ以上のプロセスからなるアプリケーション・プログラムは、効率的に共有メモリへのアクセスを調整できる必要があります。同期の2つの基本的な方法(*相互排他*と*条件同期*)は、共有メモリへのプロセスのアクセスを調整するために使用されます。相互排他メカニズムは共有リソースへの協同プロセスのアクセスを順番に並べます。条件同期メカニズムはアプリケーションが定義する条件が満足するまでプロセスの進行を延ばします。

相互排除メカニズムは協同プロセスがクリティカル・セクションで同時に実行することができるのは1つだけであることを保証します。3種類のメカニズムは通常は相互排他を提供するために使用されます—ビジーウェイト、スリーピーウェイト、プロセスがロックされたクリティカル・セクションへ入ろうとする時に2つの組み合わせを必要とします。*スピンロック*として知られるビジーウェイト・メカニズムは、テスト&セット操作をサポートしたハードウェアを使用してロックを取得するロッキング手法を使用します。もしプロセスが現在ロックされた状態でビジーウェイト・ロックを取得しようとする場合、

ロックしているプロセスは、テスト&セット操作をプロセスが現在保有しているロックがクリアされテスト&セット操作が成功するまでリトライし続けます。対照的にセマフォのようなスリーピーウェイト・メカニズムは、もしそれが現在ロックされた状態でロックを取得しようとするのであればプロセスをスリープ状態にします。

ビジーウェイト・メカニズムは、ロックを取得する試みの殆どが成功する時に非常に効果的です。これは単純なテスト&セット操作がビジーウェイト・ロックを取得するために必要とされる全てであるからです。ビジーウェイト・メカニズムは、ロックが保持される時間が短い時にリソースを保護するために適しています。それには次の2つの理由があります：1) ロックの保持時間が短い時、ロック中のプロセスがアンロック状態でロックを取得するので、ロック・メカニズムのオーバーヘッドも最小限となる可能性があり、2) ロックの保持時間が短い時、ロックを取得する遅れも短くなることが予想されます。ビジーウェイト・メカニズムはロックからアンロック状態となるの待っている間にCPUリソースを消費しようとするため、ロック取得の遅れが短時間で保たれるビジーウェイト相互排他を使用する場合は重要となります。一般的なルールとして、もしロックを保持する時間が2つのコンテキスト・スイッチの実行に要する時間よりも全て少ない場合、ビジーウェイト・メカニズムは適切です。ビジーウェイト相互排他を実行するためのツールは、「ビジーウェイト相互排他」セクションで説明しています。

クリティカル・セクションは大抵は非常に短時間です。同期のコストを比較的小さく保つため、クリティカル・セクションの入口/出口で実行される同期処理をカーネルへ入れることは出来ません。クリティカル・セクションの入場および退場に関連する実行オーバーヘッドがクリティカル・セクション自体の長さよりも長くなることは好ましくありません。

効果的な相互排他ツールとしてスピンロックを使用するため、あるプロセスが他のプロセスがロックをリリースするのを待つためにスピンする予想時間は、短時間だけでなく予測可能である必要があります。ロック中プロセスのページ・フォルト、シグナル、プリエンプションのような予測不可能なイベントは、クリティカル・セクション内の本当の経過時間が期待される実行時間を著しく超える原因となります。せいぜい、これらのクリティカル・セクション内部の予測不可能な遅れは、他のCPUの遅れが予想されるよりも長くなる原因となる可能性があり、最悪の場合、それらはデッドロックを引き起こす可能性があります。メモリ内のページ・ロックはクリティカル・セクションへ入る時間に影響を与えないためにプログラムの初期化中に完了させることが可能です。再スケジューリング制御のメカニズムは、低オーバーヘッドのシグナル制御とプロセス・プリエンプションの手法を提供します。再スケジューリング制御のためのツールは「再スケジューリング制御」で説明されています。

セマフォは相互排他を提供するためのもう1つのメカニズムです。既にロックされているセマフォをロックしようとするプロセスはブロックされるもしくはスリープ状態となるため、セマフォはスリーピー・ウェイト型の相互排他となります。POSIXのカウンティング・セマフォは移植可能な共有リソースへのアクセス制御の手段を提供します。カウンティング・セマフォは整数値とそれに対して定義される操作の制限セットを持つオブジェクトです。カウンティング・セマフォは、ロックとアンロック操作で最速性能を得るために実装される単純なインターフェースを提供します。POSIXのカウンティング・セマフォは、「POSIXカウンティング・セマフォ」セクションの中で説明されています。System Vのセマフォは、多くの追加機能(例えば、セマフォ上でいくつくらい待ちがあるのかを調べる機能、もしくは一連のセマフォを操作する機能)を許可する複雑なデータ型です。System Vのセマフォは「System Vセマフォ」セクションで説明されています。

ミューテックスはプログラム内の複数のスレッドが同時ではありませんが同じリソースを共有することを可能にします。ミューテックスを作成し、リソースを必要とするどのスレッドもリソースを使用している間は他のスレッドからミューテックスをロックする必要があり、もう必要とされない時にそれをアンロックします。POSIXのミューテックスは、特にリアルタイム・アプリケーションにとって便利でミューテックス単位に個々に設定可能な2つの機能(*ロウバスト・ミューテックス*と*優先度継承ミューテックス*)を持っています。ロウバスト性(堅牢性)はもしアプリケーションのスレッドの1つがミューテックス保持中に死んだ場合、回復する機会をアプリケーションに与えます。

優先度継承ミューテックスを使用するアプリケーションは、時々引き上げられるミューテックスの所有者の優先度を検出することが可能です。これらは「POSIXミューテックスへの機能拡張」セクションで説明されています。

再スケジューリング制御

マルチプロセス、リアルタイム・アプリケーションは頻繁に短い時間CPUの再スケジューリングを伸ばすことを望みます。効果的にビジーウェイト相互排他を使うため、スピンロックの保持時間は小さくかつ予測可能である必要があります。

CPU再スケジューリングとシグナル処理は予測不可能である主要な原因です。プロセスはスピンロックを得るときは再スケジューリングに自分自身が影響を受けないようにし、ロックを開放する時は再び被害を受けやすくなります。システムコールは呼び出し元の優先度をシステムの中で最高に引き上げることは可能ですが、それをするときのオーバーヘッドは法外となります。

再スケジューリング変数は再スケジューリングとシグナル処理のための制御を提供します。アプリケーション内の変数を登録し、アプリケーションから直接それを操作します。最スケジューリング無効、クォンタム終了、プリエンプション、特定タイプのシグナルである間は保持されます。

システムコールと一連のマクロは再スケジューリング変数の使用を提供します。次のセクションで変数、システムコール、マクロを記述し、それらの使用方法について説明します。

ここに記述される基礎的なものは低オーバーヘッドのCPU再スケジューリング制御とシグナル配信を提供します。

再スケジューリング変数の理解

再スケジューリング変数は、再スケジューリングに対するシングル・プロセスの脆弱性を制御する**<sys/rescntl.h>**の中で定義されるデータ構造体：

```
struct resched_var {
    pid_t rv_pid;
    ...
    volatile int rv_nlocks;
    ...
};
```

これはカーネルではなくアプリケーションによりプロセス単位もしくはスレッド単位に割り付けられます。**resched_cntl(2)**システムコールは変数を登録し、カーネルは最スケジューリングを決定する前に変数を調べます。

resched_cntlシステムコールの使用は「resched_cntlシステムコールの利用」で説明されています。再スケジューリング制御マクロ一式はアプリケーションから変数の操作を可能にします。それらのマクロの使用は「再スケジューリング制御マクロの利用」で説明されています。

各スレッドはそれぞれの再スケジューリング変数を登録する必要があります。最スケジューリング変数は、再スケジューリング変数のロケーションを登録するプロセスもしくはスレッドに対してのみ有効です。現在の実装においては、再スケジューリング変数はシングル・スレッドのプロセスでのみ使用することを推奨します。再スケジューリング変数を使用するマルチスレッド・プログラムでのフォークは回避する必要があります。

resched_cntlシステムコールの利用

resched_cntlシステムコールは様々な再スケジューリング制御操作を実行することが可能です。それらには再スケジューリング変数の登録と初期化、ロケーションの取得、延長可能な再スケジューリング時間長の制限設定が含まれています。

概要

```
#include <sys/rescntl.h>

int resched_cntl(cmd, arg)

int cmd;
char *arg;
```

gcc [*options*] *file* **-lccur_rt ...**

引数は以下のように定義されます：

cmd　　実行される操作

arg　　*cmd* の値に依存する引数の値へのポインタ

cmd は以下のいずれかとなります。各コマンドに関連する*arg* の値が表示されています。

RESCHED_SET_VARIABLE

このコマンドは呼び出し元の再スケジューリング変数を登録します。再スケジューリング変数は、MAP_SHAREDにてマップされた共有メモリの領域もしくはファイル内のページを除くプロセスのプライベート・ページの中にある必要があります。

同一プロセスの2つのスレッドはそれらの再スケジューリング変数として同じアドレスを登録してはいけません。もし*arg* がNULLでない場合、それが指す`struct resched_var`は初期化され、物理メモリ内へロックされます。もし*arg* がNULLの場合、呼び出し元は既存の変数から分離し、適当なページがアンロックされます。

fork(2)の後、子プロセスはその親プロセスから再スケジューリング変数を継承します。子プロセスの再スケジューリング変数の`rv_pid`フィールドは子プロセスIDに更新されます。

もし子プロセスが再スケジューリング変数を継承した後に1つ以上の子プロセスをフォークした場合、それらの子プロセスは`rv_pid`フィールドが更新された再スケジューリング変数を継承します。

fork, **vfork(2), clone(2)**を呼び出したその時にもし再スケジューリング変数が親プロセスの中でロックされた場合、再スケジューリング変数は停止します。このコマンドの使用はルート権限もしくはCAP_IPC_LOCKとCAP_SYS_RAWIOケーパビリティを必要とします。

RESCHED_SET_LIMIT
: このコマンドはデバッグ・ツールです。もし*arg* がNULLでない場合、呼び出し元が望む再スケジューリング延長の最大時間長を指定する`struct timeval`を指定します。もしCPUのローカル・タイマーが有効である場合、この制限を超える時にSIGABRTシグナルが呼び出し元へ送信されます。もし*arg* がNULLの場合、以前のどのような制限も無視されます。

RESCHED_GET_VARIABLE
: このコマンドは再スケジューリング変数のロケーションを返します。*arg* は再スケジューリング変数のポインタを指定する必要があります。もし呼び出し元が再スケジューリング変数を持っていない場合は*arg* が参照するポインタにはNULLが設定され、そうでなければ再スケジューリング変数のロケーションが設定されます。

RESCHED_SERVE
: このコマンドはペンディング中のシグナルとコンテキスト・スイッチを提供するために**resched_unlock**で使用されます。アプリケーションはこのコマンドを直接使用する必要はありません。*arg* は0です。

再スケジューリング制御マクロの利用

一連の再スケジューリング制御マクロは再スケジューリング変数のロックとアンロックおよび有効な再スケジューリングのロック数を決定することが可能です。これらのマクロを以下で簡単に説明します：

resched_lock
: 呼び出し元プロセスが保持する再スケジューリングのロック数を増やします

resched_unlock
: 呼び出し元プロセスが保持する再スケジューリングのロック数を減らします

resched_nlocks
: 有効な現在の再スケジューリングのロック数を返します

resched_lock

概要

```
#include <sys/rescntl.h>

void resched_lock(r);

struct resched_var *r;
```

引数は以下のように定義されます：

r
: 呼び出し元プロセスの再スケジューリング変数へのポインタ

resched_lockは値を返しません。これは呼び出し元プロセスが保持する再スケジューリングのロック数を増やします。プロセスがカーネルに入らない限り、クォンタム終了、プリエンプション、いくつかのシグナル配信は全ての再スケジューリングのロックが開放されるまで延長されます。

しかし、もしプロセスが例外(例：ページ・フォルト)を発生もしくはシステムコールを行う場合、シグナルを受信する、さもなければ再スケジューリングのロック数に関係なくコンテキスト・スイッチを保持する可能性があります。次のシグナルはエラー状態を表し、再スケジューリングのロックに影響されません：SIGILL, SIGTRAP, SIGFPE, SIGKILL, SIGBUS, SIGSEGV, SIGABRT, SIGSYS, SIGPIPE, SIGXCPU, SIGXFSZ

再スケジューリング変数がロックされている間にシステムコールを行うことは可能ですが、推奨できません。また一方、呼び出し元プロセスが再スケジューリング変数がロックされている間スリープする状態となるシステムコールを行うのは有効ではありません。

resched_unlock

概要

```
#include <sys/rescntl.h>

void resched_unlock(r);

struct resched_var *r;
```

引数は以下のように定義されます：

r　　呼び出し元プロセスの再スケジューリング変数へのポインタ

resched_unlockは値を返しません。もしデクリメントやコンテキスト・スイッチの後に未処理のロックが存在しない、もしくはシグナルが保留中の場合、それらは即座に提供されます。

NOTE

`rv_nlocks`フィールドはロックがアクティブであると判断させるために正の整数である必要があります。従って、もしこのフィールドがゼロもしくは負の値であった場合、アンロックである判断されます。

resched_nlocks

概要

```
#include <sys/rescntl.h>

int resched_nlocks(r);

struct resched_var *r;
```

引数は以下のように定義されます：

r　　呼び出し元プロセスの再スケジューリング変数へのポインタ

resched_nlocksは有効な現在の再スケジューリングのロック数を返します。これらのマクロの使用に関する更なる情報は、**resched_cntl(2)**のmanページを参照してください。

再スケジューリング制御ツールの適用

以下のCプログラムの断片は、前のセクションで説明したツールを使って再スケジューリングを制御するための手順を説明しています。このプログラムの断片はグローバル変数として再スケジューリング変数(rv)を定義し、**resched_cntl**を呼び出して変数の登録と初期化を行い、そして**resched_lock**と**resched_unlock**をそれぞれ呼び出して再スケジューリング変数をロックおよびアンロックします。

```
static struct resched_var rv;

int main (int argc, char *argv[])
{
    resched_cntl (RESCHED_SET_VARIABLE, (char *)&rv);
    resched_lock (&rv);
    /* nonpreemptible code */
    ...
    resched_unlock (&rv);

    return 0;
}
```

ビジーウェイト相互排他

ビジーウェイト相互排他は、共有リソースの同期変数を関連付けることにより達成します。プロセスもしくはスレッドがリソースへのアクセス増加を望む時、同期変数をロックします。リソースの使用が終了する時、同期変数をアンロックします。最初のプロセスもしくはスレッドがリソースをロックしている間にもし他のプロセスもしくはスレッドがリソースへのアクセスを増やそうとした時、そのプロセスもしくはスレッドはロックの状態を繰り返し検査することにより遅らせる必要があります。この同期の形式は、ユーザー・モードから直接アクセス可能である同期変数およびロックとアンロック操作が非常に低オーバーヘッドであることを要求します。

RedHawk Linuxは2種類の低オーバーヘッドのビジーウェイト相互排他変数(**spin_mutex** と **nopreempt_spin_mutex**)を提供します。**nopreempt_spin_mutex**はミューテックスを保持している間、スレッドもしくはプロセスを非プリエンプト状態にするために再スケジューリング変数を自動的に使用しますが、**spin_mutex**はそうではありません。

後に続くセクションでは、変数とインターフェースを定義し、それらの使用手順を説明します。

spin_mutex変数の理解

ビジーウェイト相互排他変数はスピンロックとして知られるデータ構造体です。spin_mutex変数は以下のように**<spin.h>**の中で定義されています。

```
typedef struct spin_mutex {
    volatile int count;
} spin_mutex_t;
```

スピンロックは2つの状態(ロックとアンロック)を持っています。初期化される時、スピンロックはアンロック状態にあります。もし共有リソースへのアクセスを調整するためにスピンロックを使用したいと考えている場合、アプリケーション・プログラムの中にそれらを割り当てて同期したいプロセスまたはスレッドが共有するメモリの中にそれらを配置する必要があります。「spin_mutexインターフェースの利用」で説明されているインターフェースを使うことによりそれらを操作することが可能です。

spin_mutexインターフェースの利用

このビジーウェイト相互排他インターフェース一式は、スピンロックの初期化、ロック、アンロックおよび特定のスピンロックがロックされているかどうかを判断することが可能です。以下で簡単に説明します：

spin_init	スピンロックをアンロック状態に初期化します
spin_lock	スピンロックがロックされるまでスピンします
spin_trylock	指定されたスピンロックのロックを試みます
spin_islock	指定されたスピンロックがロックされているかどうかを確認します
spin_unlock	指定されたスピンロックをアンロックします

これらのインターフェースのいずれも無条件にスピンロックをロックすることが可能なものはないことに注意することが重要です。提供されるツールを使用することによりこの機能を構築することが可能です。

CAUTION

スピンロック上の操作は再起的ではありませんが、もし既にロックされたスピンロックを再ロック使用とする場合、プロセスまたはスレッドはデッドロックとなる可能性があります。

これらを使用する前に**spin_init**の呼び出しによりスピンロックを初期化する必要があります。各スピンロックに対して1度だけ**spin_init**を呼び出します。もし指定するスピンロックがロックされている場合、**spin_init**は効果的にアンロックしますが、これは値を返しません。**spin_init**インターフェースは以下のように指定されます：

```
#include <spin.h>

void spin_init(spin_mutex_t *m);
```

引数は以下のように定義されます：

m　　スピンロックの開始アドレス

spin_lockはスピンロックがロックされるまでスピンします。これは値を返しません。このインターフェースは以下のように指定されます：

```
#include <spin.h>

void spin_lock(spin_mutex_t *m);
```

もし呼び出し元プロセスまたはスレッドがスピンロックのロックに成功した場合、**spin_trylock**はtrue を返し、もし成功しなかった場合はfalse を返します。**spin_trylock**は呼び出し元プロセスまたはスレッドをブロックしません。このインターフェースは以下のように指定されます：

```
#include <spin.h>
int spin_trylock(spin_mutex_t *m);
```

もし指定されたスピンロックがロックされている場合、**spin_islock**はtrue を返します。もしアンロックされている場合はfalse が返します。これはスピンロックをロックしません。このインターフェースは以下のように指定されます：

```
#include <spin.h>
int spin_islock(spin_mutex_t *m);
```

spin_unlockはスピンロックをアンロックします。これは値を返しません。このインターフェースは以下のように指定されます：

```
#include <spin.h>
void spin_unlock(spin_mutex_t *m);
```

spin_lock, **spin_trylock, spin_unlock**はNightTrace RTで監視するためにトレース・イベントを記録することが可能です。アプリケーションは**<spin.h>**より前にSPIN_TRACEを定義することにより、これらのトレース・イベントを有効にすることが可能です。例：

```
#define SPIN_TRACE
#include <spin.h>
```

もし**-lpthread**がリンクされる場合、アプリケーションは**-lntrace**もしく**-lntrace_thr**もリンクされる必要があります。

これらのインターフェースの使用に関する更なる情報は、**spin_init(3)**のmanページを参照してください。

spin_mutexツールの適用

ビジーウェイト相互排他のためのspin_mutex ツールの使用手順は、以下のコードの断片で説明します。最初の部分は、スピンロックを取得するために再スケジューリング制御と一緒にこれらのツールを使用する方法を示し、次頁はスピンロックを開放する方法を示します。これらのコードの断片にシステムコールもプロシージャコールも含まれていないことに注意してください。

_m 引数はスピンロックを指し、引数は呼び出し元プロセスもしくはスレッドの再スケジューリング変数を指します。これはスピンロックが共有メモリ内にあることを前提としています。ページングやスワッピングに関連するオーバーヘッドを回避するため、クリティカル・セクション内部で参照されるページは物理メモリにロックすることを推奨します。(**mlock(2)** および **shmctl(2)**システムコールを参照してください)

```
#define spin_acquire(_m,_r) ¥
{ ¥
    resched_lock(_r); ¥
    while (!spin_trylock(_m)) { ¥
        resched_unlock(_r); ¥
        while (spin_islock(_m)); ¥
        resched_lock(_r); ¥
    } ¥
}
```

```
#define spin_release(_m,_r) ¥
{ ¥
    spin_unlock(_m); ¥
    resched_unlock(_r); ¥
}
```

前頁の断片では、**spin_trylock**と**spin_islock**のインターフェースの使用に注意してください。もしスピンロックをロックしようとしているプロセスもしくはスレッドがロックされているスピンロックを見つけた場合、**spin_islock**の呼び出しによりロックが開放されるまで待ちます。このシーケンスは直接**spin_trylock**でポーリングするよりも効率的です。**spin_trylock**インターフェースはテスト&セットの原始的なスピンロックを実行するための特別な命令を含みます。これらの命令は**spin_islock**による単純なメモリ読み取りの実行よりも効果は小さくなります。

再スケジューリング制御インターフェースの使用もまた注意してください。デッドロックを回避するため、プロセスもしくはスレッドはスピンロックのロックの前に再スケジューリングを無効にし、スピンロックのアンロック後にそれを再度有効にします。プロセスもしくはスレッドは**spin_islock**の呼び出しの直前で再スケジューリングを再度有効にするので、再スケジューリングが必要以上に長くなることはありません。

nopreempt_spin_mutex変数の理解

nopreempt_spin_mutex は、共有リソースへの同期アクセスを複数のスレッドもしくはプロセスに許可するビジーウェイト・ミューテックスです。再スケジューリング変数はミューテックスがロックされている間に非プリエンプトなスレッドもしくはプロセスにするために使用されます。スレッドもしくはプロセスは複数のミューテックスのロックを安全に重ねることが可能です。nopreempt_spin_mutex は、以下のように**<nopreempt_spin.h>**の中で定義されています：

```
typedef struct nopreempt_spin_mutex {
    spin_mutex_t spr_mux;
} nopreempt_spin_mutex_t;
```

スピンロックは2つの状態(ロックとアンロック)を持っています。初期化される時、スピンロックはアンロック状態にあります。

もし共有リソースへのアクセスを調整するために非プリエンプト・スピンロックを使用したいと考えている場合、アプリケーション・プログラムの中にそれらを割り当てて同期したいプロセスまたはスレッドが共有するメモリの中にそれらを配置する必要があります。「nopreempt_spin_mutexインターフェースの利用」で説明されているインターフェースを使うことによりそれらを操作することが可能です。

nopreempt_spin_mutexインターフェースの利用

このビジーウェイト相互排他インターフェース一式は非プリエンプト・スピンロックのロック、アンロックが可能です。再スケジューリング変数はロックされたミューテックスを保持する間に非プリエンプトなスレッドもしくはプロセスにするために使用されます。以下で簡単に説明します：

nopreempt_spin_init	スピンロックをアンロック状態に初期化します
nopreempt_spin_init_thread	プリエンプションがブロックさることを保証します
nopreempt_spin_lock	スピンロックがロックされるまでスピンします

nopreempt_spin_trylock	指定されたスピンロックのロックを試みます
nopreempt_spin_islock	指定されたスピンロックがロックされているかどうかを確認します
nopreempt_spin_unlock	指定されたスピンロックをアンロックします

これらを使用する前に**nopreempt_spin_init**の呼び出しによりスピンロックを初期化する必要があります。各スピンロックに対して1度だけこのインターフェースを呼び出します。もし指定するスピンロックがロックされている場合、**nopreempt_spin_init**は効果的にアンロックしますが、これは値を返しません。このインターフェースは以下のように指定されます：

```
#include <nopreempt_spin.h>
void nopreempt_spin_init(nopreempt_spin_mutex_t *m);
```

引数は以下のように定義されます：

m	スピンロックの開始アドレス

nopreempt_spin_lockと**nopreempt_spin_trylock**が呼び出された時、**nopreempt_spin_init_thread**はプリエンプションがブロックされることを保証します。nopreempt_spin_mutexがマルチ・スレッド・プロセスで使用される時、プロセスは**-lpthread**がリンクされる必要があり、各スレッドは**nopreempt_spin_init_thread**を少なくても1回は呼び出す必要があります。もしプロセスがマルチ・スレッド化されていない場合、このルーチンを少なくても1回は呼び出す必要があります。このルーチンは、プロセスもしくはスレッドが何個ミューテックスを使用しているかに関係なく1回は呼び出される必要があります。もしプリエンプションのブロックが保証される場合ゼロが返りますが、そうではない場合**errno**が設定されて-1が返ります。このインターフェースは以下のように指定されます：

```
#include <nopreempt_spin.h>
int nopreempt_spin_init_thread(void)
```

nopreempt_spin_lockはスピンロックがロックされるまでスピンします。これは値を返しません。このインターフェースは以下のように指定されます：

```
#include <nopreempt_spin.h>
void nopreempt_spin_lock(nopreempt_spin_mutex_t *m);
```

もし呼び出し元プロセスもしくはスレッドがスピンロックのロックに成功した場合、**nopreempt_spin_trylock**はtrue を返しますが、もし成功しなかった場合はfalseを返します。**nopreempt_spin_trylock**は呼び出し元プロセスもしくはスレッドをブロックしません。このインターフェースは以下のように指定されます：

```
#include <nopreempt_spin.h>
int nopreempt_spin_trylock(nopreempt_spin_mutex_t *m);
```

もし指定されたスピンロックがロックされている場合、**nopreempt_spin_islock**はtrueを返します。もしアンロックされている場合はfalse を返します。このインターフェースは以下のように指定されます：

```
#include <nopreempt_spin.h>
int nopreempt_spin_islock(nopreempt_spin_mutex_t *m);
```

nopreempt_spin_unlockはスピンロックをアンロックします。これは値を返しません。このインターフェースは以下のように指定されます：

```
#include <nopreempt_spin.h>
void nopreempt_spin_unlock(nopreempt_spin_mutex_t *m);
```

nopreempt_spin_lock, **nopreempt_spin_trylock, nopreempt_spin_unlock**はNightTrace RTで監視するためにトレース・イベントを記録することが可能です。アプリケーションは**<nopreempt_spin.h>**より前にSPIN_TRACEを定義することにより、これらのトレース・イベントを有効にすることが可能です。例：

```
#define SPIN_TRACE
#include <nopreempt_spin.h>
```

もし**-lpthread**がリンクされる場合、アプリケーションは**-lntrace**もしく**-lntrace_thr**もリンクされる必要があります。

これらのインターフェースの使用に関する更なる情報は、**nopreempt_spin_init(3)**のmanページを参照してください。

POSIXカウンティング・セマフォ

概要

カウンティング・セマフォはロックとアンロック操作のための最速性能を達成するために実装可能な単純なインターフェースを提供します。カウンティング・セマフォは整数値とそれに対して定義される操作の制限セットを持つオブジェクトです。これらの操作と対応するPOSIXインタフェースは以下を含みます：

- セマフォをゼロもしく正の値に設定する初期化操作— **sem_init**もしくは**sem_open**
- セマフォの値をデクリメントするロック操作— **sem_wait**もしくは**sem_timedwait**。結果の値が負の場合、操作を実行しているタスクはブロックします
- セマフォの値をインクリメントするアンロック操作— **sem_post**。もし結果の値がゼロ以下の場合、セマフォ上でブロックされているタスクの1つが起こされます。もし結果の値がゼロを超える場合、セマフォ上でブロックされたタスクはありません。
- 値が正の場合のみセマフォの値をデクリメントする条件付きロック操作—**sem_trywait**。もし値がゼロもしくは負の場合、操作は失敗します。
- セマフォの値のスナップショットを提供するクエリ操作—**sem_getvalue**

ロック、アンロック、条件付きロック操作は強力です(一連の操作が同時に実行され、それらが全て同時に完了できる場合のみ)。

カウンティング・セマフォは複数の協同プロセスで使用できるあらゆるリソースへのアクセスを制御するために使用することが可能です。カウンティング・セマフォは名前付きでも名前なしでも可能です。

スレッドは**sem_init(3)**ルーチンの呼び出しを通して名前なしセマフォを作成し初期化します。このセマフォは呼び出しで指定される値に初期化します。アプリケーション内の全スレッドは、**sem_init**ルーチンの呼び出しで作成された名前なしセマフォにアクセスします。

スレッドは**sem_open**ルーチンの呼び出しおよびユニークな名前(分かりやすいNULLで終了する文字列)の指定することにより名前付きセマフォを作成します。セマフォは、セマフォを作成するための**sem_open**呼び出しで供給される値に初期化されます。**sem_open**ルーチンはプロセスの仮想アドレス空間にセマフォを含めますので、名前付きセマフォのためにプロセスが空間を割り当てることはありません。他のプロセスは**sem_open**の呼び出しおよび同じ名前の指定により名前付きセマフォへアクセスすることが可能です。

名前なしもしくは名前付きのセマフォを初期化する時、その値は利用可能なリソースの数に設定する必要があります。相互排他を提供するためにカウンティング・セマフォを使うには、セマフォの値は1 に設定する必要があります。

クリティカルなリソースへのアクセスを望む協同タスクは、そのリソースを保護するセマフォをロックする必要があります。タスクがセマフォをロックする時、それはシステム内の他の協同タスクから干渉されることなくリソースが使用可能であることを知っています。リソースを保護するセマフォを取得した後にリソースがアクセスされるようにアプリケーションが書かれている必要があります。

セマフォの値が正である限りリソースは利用可能で、リソースの1つはそれを取得しようとしている次のタスクに割り当てられます。セマフォの値がゼロの時、利用可能なリソースは1つもなく、リソースを取得しようとしているタスクは利用可能となる1になるまで待つ必要があります。もしセマフォの値が負である場合、リソースの1つを取得するためにブロックされているもしくは待機しているタスクが1つ以上存在する可能性があります。タスクがリソースの使用を完了する時、それはセマフォをアンロックし、リソースを他のタスクが使用可能であることを知らせます。

所有権の概念はカウンティング・セマフォには当てはまりません。あるタスクがセマフォをロックすることが可能で、他のタスクはそれをアンロックすることが可能です。

セマフォのアンロック操作は安全な非同期シグナルで、これはタスクがデッドロックを引き起こすことなくシグナル・ハンドリング・ルーチンからセマフォをアンロックすることが可能です。

所有権の欠如は優先度の継承を不可能にします。何故ならタスクがセマフォをロックする時にタスクはセマフォの所有者にはならないため、タスクは同じセマフォをロックしようとするのをブロックする高優先度タスクの優先度を一時的に継承することが出来ません。その結果、無限の優先度逆転が生じる可能性があります

インターフェース

以降のセクションでPOSIXカウンティング・セマフォ・インターフェースの使用手順を説明します。これらのインターフェースを以下で簡単に説明します：

sem_init	名前なしカウンティング・セマフォを初期化します
sem_destroy	名前なしカウンティング・セマフォを削除します
sem_open	名前付きカウンティング・セマフォの作成、初期化、接続の確立を行います
sem_close	名前付きカウンティング・セマフォへのアクセスを放棄します
sem_unlink	名前付きカウンティング・セマフォの名前を削除します
sem_wait	カウンティング・セマフォをロックします
sem_trywait	カウンティング・セマフォがアンロックである場合ロックします
sem_timedwait	カウンティング・セマフォをタイムアウト付きでロックします
sem_post	カウンティング・セマフォをアンロックします
sem_getvalue	カウンティング・セマフォをの値を取得します

これらのインターフェースを使用るするため、Pスレッド・ライブラリをアプリケーションにリンクする必要があることに注意してください。以下のサンプルは動的に共有ライブラリとリンクする時に実施するコマンド・ラインを示します。ネイティブPOSIXライブラリ(NPTL) はデフォルトで使用されます。

gcc [*options*] *file*.**c -lpthread**

同じアプリケーションを以下のコマンド・ラインで静的にリンクさせることが可能です。これはLinuxスレッド・ライブラリを使用します。

gcc [*options*] **-static** *file*.**c -lpthread**

プロセス共有セマフォのサポートがLinuxスレッド・ライブラリにはないことに注意してください。

The sem_initルーチン

sem_init(3)ライブラリ・ルーチンは、セマフォによって保護されている利用可能なリソースの数にセマフォの値を設定することにより、呼び出し元プロセスが名前なしカウンティング・セマフォを初期化することが可能です。相互排他のためにカウンティング・セマフォを使用するには、プロセスは値を1に設定します。

NPTLライブラリを使用して動的にリンクされたプログラムは、*pshared* パラメータがゼロではない値に設定される時にプロセス間でセマフォを共有することが可能です。もし*pshared* にゼロが設定された場合、セマフォは同じプロセス内のスレッド間だけで共有します。

Linuxスレッド・ライブラリを使用して静的にリンクされたプログラムは、同じプロセス内のスレッド間で共有するカウンティング・セマフォを所有することのみ可能です(*pshared* は0を設定する必要があります)。プロセス内の1つのスレッドがセマフォを作成、初期化した後、同一プロセスの他の協同スレッドはこのセマフォを操作することが可能です。**fork(2)**システムコールにより作成される子プロセスは、親プロセスが既に初期化したセマフォへのアクセスを継承しません。**exec(3)**もしくは**exit(2)**システムコールを呼び出した後、プロセスもまたセマフォへのアクセスを失います。

sem_wait, **sem_timedwait**, **sem_trywait**, **sem_post**, **sem_getvalue** ライブラリ・ルーチンはセマフォを操作するために使用されます。名前なしカウンティング・セマフォは**sem_destroy** ルーチンの呼び出しにより削除されます。これらのルーチンはこの後のセクションで説明します。

CAUTION

IEEE 1003.1b-1993 規格は、複数のプロセスが同一セマフォに対して**sem_init**を呼び出した時に発生することを示していません。現在、RedHawk Linuxの実装は、単に最初の**sem_init**呼び出しの後に行われる**sem_init**の呼び出しで指定される値にセマフォを再初期化します。

アプリケーション・コードがPOSIXインターフェース(将来のConcurrentのシステムを含む)をサポートするどのようなシステムにも移植することが出来ることを確実にするため、**sem_init**を使う協同プロセスはシングル・プロセスが特定のセマフォの初期化が1度だけ行われることを守る必要があります。

もし**sem_init**の呼び出しの前に既に初期化され、この同じセマフォを待機中のスレッドが複数存在している後に**sem_init**が呼び出された場合、これらのスレッドは**sem_wait**の呼び出しから返ることは決してなく、明示的に終了させることが必要となります。

概要

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

引数は以下のように定義されます：

sem 初期化する名前なしカウンティングセマフォを表すsem_t 構造体へのポインタ

pshared セマフォを他のプロセスが共有するかどうかを示す整数値。もし*pshared* にゼロ以外の値が設定されている場合、セマフォはプロセス間で共有されます。もし*pshared* にゼロが設定されている場合、セマフォは同一プロセス内のスレッド間でのみ共有されます。Linuxスレッド・ライブラリを使って静的にリンクされたプログラムは、プロセス間で共有するセマフォを使用することは出来ず、*pshared* にゼロを設定する必要があり、もしゼロが設定されていない場合は**sem_init**は-1を返し、**errno**に**ENOSYS**が設定されます。

value ゼロもしくは、セマフォの値を現在利用可能なリソースの数へ初期化する正の整数値。この数はSEM_VALUE_MAXの値を超えることができません(この値を確認するには<**semaphore.h**>を参照してください)。

戻り値0は**sem_init**の呼び出しが成功したことを示します。戻り値-1はエラーが発生したことを示し、errnoはエラーを知らせるため設定されます。発生する可能性のあるエラーの種類の一覧は**sem_init(3)**のmanページを参照してください。

The sem_destroyルーチン

CAUTION

名前なしカウンティング・セマフォは、どのプロセスもセマフォを操作する必要がなくなり、現在セマフォをブロックするプロセスが存在しなくなるまで削除してはいけません。

概要

```
#include <semaphore.h>
int sem_destroy(sem_t *sem);
```

引数は以下のように定義されます：

sem 削除する名前なしカウンティング・セマフォへのポインタ。**sem_init(3)**の呼び出しで作成されたカウンティング・セマフォだけを**sem_destroy**の呼び出しで削除することが可能です。

戻り値0は**sem_destroy**の呼び出しが成功したことを示します。戻り値-1はエラーが発生したことを示し、errnoはエラーを知らせるため設定されます。発生する可能性のあるエラーの種類の一覧は**sem_destroy(3)**のmanページを参照してください。

The sem_openルーチン

sem_open(3)ライブラリ・ルーチンは、呼び出し元プロセスが名前付きカウンティング・セマフォの作成、初期化、接続を確立することが可能です。プロセスが名前付きカウンティング・セマフォを作成する時、ユニークな名前をセマフォへ関連付けます。これもやはりセマフォに保護されている利用可能なリソースの数にセマフォの値を設定します。相互排他のために名前付きカウンティング・セマフォを使用するには、プロセスは値を1に設定します。

名前付きセマフォを作成した後、他のプロセスは**sem_open**の呼び出しおよび同じ名前の指定によりそのセマフォへの接続を確立することが可能となります。正常に完了すると**sem_open**ルーチンは名前付きカウンティング・セマフォのアドレスを返します。プロセスはその後、**sem_wait**, **sem_trywait**と**sem_post**の呼び出しでセマフォを参照するためにそのアドレスを使用します。プロセスは**sem_close**ルーチンもしくは**exec(2), _exit(2)**システムコールを呼び出すまで名前付きセマフォを操作し続ける可能性があります。**exec**もしくは**exit**の呼び出しで名前付きセマフォは**sem_close**の呼び出しであるかのように終了します。**fork(2)**システムコールにより作成される子プロセスは親プロセスが確立した名前付きセマフォへのアクセスを継承します。

もしシングル・プロセスが**sem_open**を同じ名前を指定して複数呼び出しを行う場合、(1)プロセス自身が**sem_close**の呼び出しを通してセマフォを閉じていない、もしくは、(2)いくつかのプロセスが**sem_unlink**の呼び出しを通して名前を削除していない限り同じアドレスが各々の呼び出し元に返されます。

もし複数のプロセスが**sem_open**を同じ名前を指定して複数呼び出しを行う場合、いくつかのプロセスが**sem_unlink**の呼び出しを通して名前を削除していない限り、同じセマフォ・オブジェクトのアドレスが各々の呼び出し元に返されます(各呼び出しにおいて必ずしも同じアドレスが返さるわけではないことに注意してください)。もしプロセスが**sem_unlink**の呼び出しを通して名前を削除した場合、セマフォ・オブジェクトの新しいインスタンスのアドレスが返されます。

概要

```
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflag[, mode_t mode, unsigned int value ]);
```

引数は以下のように定義されます：

name セマフォの名前を指定するNULLで終了する文字列。接頭語"sem" は*name* の前に付加され、セマフォは**/dev/shm**にデータファイルとして作成されます。先頭のスラッシュ(/)文字は可能(移植性のあるアプリケーションのために推奨)ですがスラッシュを途中に埋め込めません。先頭のスラッシュ文字も現在の作業ディレクトリも名前の解釈に影響を与えません。例えば、**/mysem**と**mysem**は両方とも**/dev/shm/sem.mysem**として解釈されます。接頭語4文字を含むこの文字列は**/usr/include/limits.h**で定義されるNAME_MAX未満で構成されている事に注意が必要です。

oflag 呼び出し元プロセスが名前付きカウンティング・セマフォもしくは既存の名前付きカウンティング・セマフォへの接続の確率かどうかを示す整数値。以下のビットが設定することが可能です：

O_CREAT *name* で指定されるカウンティング・セマフォが存在しない場合、作成されます。セマフォのユーザーIDは呼び出し元プロセスの有効なユーザーIDに設定され、そのグループIDは呼び出し元プロセスの有効なグループIDに設定され、そのパーミッション・ビットは*mode* 引数で指定されたとおりに設定されます。セマフォの初期値は*value* 引数で指定されたとおりに設定されます。このビットを設定する時、*mode* と*value* 引数の両方を指定する必要があることに注意してください。

もし*name* で指定されるカウンティング・セマフォが存在する場合、O_EXCLに記述されている事以外は設定されたO_CREATは効力を持ちません。

O_EXCL もしO_CREATが設定され、*name* で指定されたカウンティング・セマフォが存在する場合、**sem_open**は失敗します。もしO_CREATが設定されていない場合、このビットは無視されます。

もしO_CREATとO_EXCL以外のフラグ・ビットが*oflag* 引数に設定されている場合、 **sem_open**ルーチンはエラーを返すことに注意してください。

mode 次の例外を含む*name* で指定されるカウンティング・セマフォのパーミッション・ビットを設定する整数値：プロセスのファイル・モード作成マスクに設定されたビットはカウンティング・セマフォのモードでクリアされます(更なる情報については**umask(2)**と**chmod(2)**のmanページを参照してください)。パーミッション・ビット以外のビットが*mode* に設定されている場合、それらは無視されます。プロセスは名前付きカウンティング・セマフォを作成するときのみ*mode* 引数を指定します。

value ゼロもしくは現在利用可能なリソースの数にセマフォの値を初期化する正の整数値。この数は<**limits.h**>で定義されるSEM_VALUE_MAXの値を超えることは出来ません。プロセスは名前付きカウンティング・セマフォを作成するときのみ*value* 引数を指定します。

もし呼び出しが成功した場合、**sem_open**は名前付きカウンティング・セマフォのアドレスを返します。SEM_FAILEDの戻り値はエラーが発生したことを示し、`errno`はエラーを示すために設定されます。発生する可能性のあるエラーのタイプのリストについては**sem_open(3)**のmanページを参照してください。

The sem_closeルーチン

sem_close(3)ライブラリ・ルーチンは呼び出し元プロセスが名前付きカウンティング・セマフォへのアクセスを放棄することが可能です。**sem_close**ルーチンはセマフォを利用するプロセスに割り当てられたシステム・リソースを開放します。その後、セマフォを操作しようとするプロセスがSIGSEGVシグナルの配信を招く結果となる可能性があります。

セマフォに関連するカウントはプロセスの**sem_close**呼び出しに影響を受けません。

概要

```
#include <semaphore.h>
int sem_close(sem_t *sem);
```

引数は以下のように定義されます：

sem アクセスを開放する名前付きカウンティング・セマフォへのポインタ。**sem_open(3)**の呼び出しを通して確立したカウンティング・セマフォとの接続のみを指定することが可能です。

戻り値0は**sem_close**の呼び出しが成功したことを示します。戻り値-1はエラーが発生したことを示し、`errno`はエラーを示すために設定されます。発生する可能性のあるエラーのタイプのリストについては**sem_close(3)**のmanページを参照してください。

The sem_unlinkルーチン

sem_unlink(3)ライブラリ・ルーチンは呼び出し元プロセスがカウンティング・セマフォの名前を削除することが可能です。その後同じ名前を使用してセマフォへの接続を確立しようとするプロセスはセマフォの異なるインスタンスに対し接続を確立します。呼び出し時点でセマフォを参照しているプロセスは、**sem_close(3), exec(2), exit(2)**システムコールを呼び出すまでセマフォを使用し続けることが可能です。

概要

```
#include <semaphore.h>
int sem_unlink(const char *name);
```

引数は以下のように定義されます：

name セマフォの名前を指定するNULLで終了する文字列。接頭語"sem" は*name* の前に付加され、セマフォは**/dev/shm**にデータファイルとして作成されます。先頭のスラッシュ(/)文字は可能(移植性のあるアプリケーションのために推奨)ですがスラッシュを途中に埋め込めません。先頭のスラッシュ文字も現在の作業ディレクトリも名前の解釈に影響を与えません。例えば、**/mysem**と**mysem**は両方とも**/dev/shm/sem.mysem**として解釈されます。接頭語4文字を含むこの文字列は**/usr/include/limits.h**で定義されるNAME_MAX 未満で構成されている事に注意が必要です。

戻り値0は**sem_unlink**の呼び出しが成功したことを示します。戻り値-1はエラーが発生したことを示し、`errno`はエラーを示すために設定されます。発生する可能性のあるエラーのタイプのリストについては**sem_unlink(3)**のmanページを参照してください。

The sem_waitルーチン

sem_wait(3)ライブラリ・ルーチンは呼び出し元プロセスが名前なしカウンティング・セマフォをロックすることが可能です。もしセマフォの値がゼロである場合、セマフォは既にロックされています。この場合、プロセスはシグナルもしくはセマフォがアンロックされるまでブロックします。もしセマフォの値がゼロより大きい場合、プロセスはセマフォをロックし続けます。いずれにせよ、セマフォの値は減少します。

概要

```
#include <semaphore.h>
int sem_wait(sem_t *sem);
```

引数は以下のように定義されます：

sem　　ロックする名前なしカウンティング・セマフォへのポインタ

戻り値0はプロセスが指定したセマフォのロックに成功したことを示します。戻り値-1はエラーが発生したことを示し、errnoはエラーを示すために設定されます。発生する可能性のあるエラーのタイプのリストについては**sem_wait(3)**のmanページを参照してください。

The sem_timedwaitルーチン

sem_timedwait(3)ライブラリ・ルーチンは呼び出し元プロセスが名前なしカウンティング・セマフォをロックすることが可能ですが、もし**sem_post**を介してアンロックする他のプロセスもしくはスレッドを待つことなしにセマフォがロックできない場合、指定されたタイムアウトの期限が切れた時に待機は終了します。

概要

```
#include <semaphore.h>
#include <time.h>
int sem_timedwait(sem_t *sem, const struct timespec *ts);
```

引数は以下のように定義されます：

sem　　ロックする名前なしカウンティング・セマフォへのポインタ

ts　　待機が終了する単一時間を秒とナノ秒で指定した**<time.h>**に定義されているtimespec 構造体へのポインタ
例：

```
ts.tv_sec = (NULL)+2
ts.tv_nsec = 0
```

2秒のタイムアウトを設定。POSIX時間構造体に関する詳細な情報については、6章の「POSIX時間構造体の理解」を参照してください。

戻り値0はプロセスが指定したセマフォのロックに成功したことを示します。戻り値-1はエラーが発生したことを示し、errno はエラーを示すために設定されます。発生する可能性のあるエラーのタイプのリストについては**sem_wait(3)**のmanページを参照してください。

The sem_trywaitルーチン

sem_trywait(3)ライブラリ・ルーチンはセマフォがアンロックされていることを示すセマフォの値が0より大きい場合のみ、呼び出し元プロセスがカウンティング・セマフォをロックすることが可能です。もしセマフォの値がゼロである場合、セマフォを既にロックされており、**sem_trywait**の呼び出しは失敗します。もしプロセスがセマフォのロックに成功する場合、セマフォの値は減少し、そうでない場合は変わりません。

概要

```
#include <semaphore.h>

int sem_trywait(sem_t *sem);
```

引数は以下のように定義されます：

sem 呼び出し元プロセスがロックする名前なしカウンティング・セマフォへのポインタ

戻り値0は呼び出し元プロセスが指定したセマフォのロックに成功したことを示します。戻り値-1はエラーが発生したことを示し、`errno` はエラーを示すために設定されます。発生する可能性のあるエラーのタイプのリストについては**sem_trywait(3)**のmanページを参照してください。

The sem_postルーチン

sem_post(3)ライブラリ・ルーチンは呼び出し元プロセスがカウンティング・セマフォをアンロックすることが可能です。もし1つ以上のプロセスがセマフォを待ってブロックしている場合、最高優先度の待機中プロセスがセマフォがアンロックされた時に起こされます。

概要

```
#include <semaphore.h>

int sem_post(sem_t *sem);
```

引数は以下のように定義されます：

sem アンロックする名前なしカウンティング・セマフォへのポインタ

戻り値0は**sem_post**の呼び出しが成功したことを示します。もし正しくないセマフォ記述子が提供された場合、セグメンテーション・フォルトが生じます。戻り値-1はエラーが発生したことを示し、`errno`はエラーを示すために設定されます。発生する可能性のあるエラーのタイプのリストについては**sem_post(3)**のmanページを参照してください。

The sem_getvalueルーチン

sem_getvalue(3)ライブラリ・ルーチンは呼び出し元プロセスが名前なしカウンティング・セマフォの値を取得することが可能です。

概要

```
#include <semaphore.h>

int sem_getvalue(sem_t *sem, int *sval);
```

引数は以下のように定義されます：

sem 値を取得したい名前なしカウンティング・セマフォへのポインタ

sval 名前なしカウンティング・セマフォの値が返される場所へのポインタ。返される値はコール中のあるタイミングでのセマフォの実際の値を表します。この値は呼び出しから戻るその時点でのセマフォの実際値ではないかもしれない事に注意することが重要です。

戻り値0は**sem_getvalue**の呼び出しに成功したことを示します。戻り値-1はエラーが発生したことを示し、`errno`はエラーを示すために設定されます。発生する可能性のあるエラーのタイプのリストについては**sem_getvalue(3)**のmanページを参照してください。

POSIXミューテックスへの機能拡張

ミューテックスは同時更新やクリティカル・セクションの実行から共有データ構造体を保護するために便利な相互排他デバイスです。ミューテックスはアンロック(どのスレッドにも所有されていない)とロック(1つのスレッドが所有)の2つの状態を持っています。他のスレッドが既にロックしているミューテックスをロックしようとするスレッドは、まず所有しているスレッドがミューテックスをアンロックするまで停止します。

RedHawkで利用可能な標準的なPOSIXのPスレッド・ミューテックス機能には以下のサービスが含まれます。これらのサービスのすべての情報はmanページを参照してください。

pthread_mutex_init(3)	ミューテックスを初期化
pthread_mutex_lock(3)	ミューテックスをロック
pthread_mutex_trylock(3)	ミューテックスのロックを試す
pthread_mutex_unlock(3)	ミューテックスをアンロック
pthread_mutex_destroy(3)	ミューテックスを破棄
pthread_mutexattr_init(3)	ミューテックスの属性オブジェクトを初期化
pthread_mutexattr_destroy(3)	ミューテックスの属性オブジェクトを破棄
pthread_mutexattr_settype(3)	ミューテックスの属性タイプを設定
pthread_mutexattr_gettype(3)	ミューテックスの属性タイプを取得

それらのサービスに加え、RedHawkにはロウバスト性(堅牢性)と優先度継承を提供する以下のPOSIXのPスレッド拡張が含まれます。*ロウバスト性* はもしアプリケーションのスレッドの1つがミューテックス保持中に死んだ場合、回復する機会をアプリケーションに与えます。「*優先度継承* 」は、ミューテックスを所有するどのスレッドも直接的または間接的にスレッドの優先度のスケジューリングをスリープ中の最高優先度スレッドの優先度へ自動的に引き上げます。これらの条件の詳細を以下に記述します。

サービスは以降のセクションおよびmanページで説明されています。

pthread_mutex_consistent_np(3)	矛盾するミューテックスの矛盾をなくす
pthread_mutex_getunlock_np(3)	ミューテックスのアンロック・ポリシーを返す
pthread_mutex_setconsistency_np(3)	ミューテックスの整合性状態を設定
pthread_mutex_setunlock_np(3)	ミューテックスのアンロック・ポリシーを設定
pthread_mutexattr_getfast_np(3)	操作モードを返す
pthread_mutexattr_getprotocol(3)	プロトコルを返す
pthread_mutexattr_getrobust_np(3)	ロウバスト・レベルを返す
pthread_mutexattr_getunlock_np(3)	アンロック・ポリシーを返す
pthread_mutexattr_setfast_np(3)	操作モードを設定
pthread_mutexattr_setprotocol(3)	プロトコルを設定
pthread_mutexattr_setrobust_np(3)	ロウバスト・レベルを設定
pthread_mutexattr_setunlock_np(3)	アンロック・ポリシーを設定

ロウバスト・ミューテックス

ロウバスト・ミューテックスを使用するアプリケーションは、ミューテックスを保持中に前のミューテックス所有者が終了したかどうかを検出することが可能です。新しい所有者はミューテックスに保護されている状態を除去しようとし、もしそれが出来た場合、再び正常なミューテックスをマークするすることが可能となります。もし状態の除去が出来なかった場合、ミューテックスをロックしようとする際に回復不可能であることを表すステータスを取得するようにするためにミューテックスは回復不可能とマークする可能性があります。

これを実装するため、EOWNERDEADとENOTRECOVERABLEの2つの新しい`errno`コードを利用できます。ミューテックス保持中にスレッドが死んだ時、ミューテックスは自動的にアンロックし死んだとマークされます。デッド・ミューテックスにおいて各々の成功したロックが成功ではなくEOWNERDEADエラーを返すことを除いては、デッド・ロックは通常のロックのように動作します。

従って、ロウバストに関係するアプリケーションは戻された全ロック要求のステータスを調べる必要があります。EOWNERDEADである時、アプリケーションはそれを無視することが可能で、所有者が死んだおよび矛盾(正常)がマークされたことに起因するアプリケーションの不正を何でも回復、もし回復出来なかった場合、回復不可能をマークします。

回復不可能をマークされたミューテックスはENOTRECOVERABLEエラーを伴うミューテックスの将来全ての操作を拒否します。唯一の例外はミューテックスを最初期化するサービスとミューテックスの状態を問い合わせるサービスです。回復不可能になるミューテックスでスリープしているスレッドはENOTRECOVERABLEエラーを伴い直ぐに起き上がります。

優先度継承

優先度継承ミューテックスを使用するアプリケーションは、その時々に一時的に引き上げられる優先度を検出することが可能です。引き上げはミューテックスを取得したそれらのスレッドで発生し、他の高優先度スレッドはそのミューテックスを待ってスリープ状態に入ります。この場合、スリープしているスレッドの優先度は所有者がロックを保持する間は一時的にロック所有者に移されます。

それらのスリープしているスレッドは他のミューテックスを順に所有することができるため、それら自身が優先度を引き上げ、最大機能はどの優先度へ引き上げるかを決定する際に優先度を引き上げられるスリープ・スレッドを使用して解決します。

ユーザー・インターフェース

ここに記載されたサービスの完全な説明は後に続くセクションおよび対応するオンラインのmanページで提供されます。

以下のサービスはミューテックスの状態で操作します。

pthread_mutex_consistent_np(3)	矛盾するミューテックスの矛盾をなくす
pthread_mutex_getunlock_np(3)	ミューテックスのアンロック・ポリシーを返す
pthread_mutex_setconsistency_np(3)	ミューテックスの整合性状態を設定
pthread_mutex_setunlock_np(3)	ミューテックスのアンロック・ポリシーを設定

以下に記載されたサービスはミューテックス属性オブジェクト内に格納されている属性に関して修正もしくは問い合わせを行います。「*ミューテックス属性オブジェクト*」は属性オブジェクトを伴い作成されたミューテックス内で利用可能であるミューテックスの機能を定義するデータ構造体です。ミューテックスは多くの機能を持っているので、ミューテックス属性オブジェクトはアプリケーションが1つのミューテックス属性オブジェクト内で要求されるすべての属性を定義するためにそれの使い勝手を良くして、一連の属性オブジェクトを持つことになる全てのミューテックスを作成します。

更にミューテックスの寿命のために固定される必要のあるそれらの属性は、ミューテックス属性オブジェクトを通してのみ定義することが可能です。同様にミューテックスの寿命を変更可能な属性は、ミューテックス属性オブジェクトを通して最初の定義を与えることが可能で、その後、対応するミューテックス自身の属性操作を介して変更することが可能です。

属性の取得：

pthread_mutexattr_getfast_np(3)	操作モードを返す
pthread_mutexattr_getprotocol(3)	プロトコルを返す
pthread_mutexattr_getrobust_np(3)	ロウバスト・レベルを返す
pthread_mutexattr_getunlock_np(3)	アンロック・ポリシーを返す

属性の設定：

pthread_mutexattr_setfast_np(3)	操作モードを設定
pthread_mutexattr_setprotocol(3)	プロトコルを設定
pthread_mutexattr_setrobust_np(3)	ロウバスト・レベルを設定
pthread_mutexattr_setunlock_np(3)	アンロック・ポリシーを設定

pthread_mutex_consistent_np

このサービスは矛盾するミューテックスの矛盾をなくします。

概要

```
int pthread_mutex_consistent_np (pthread_mutex_t *mutex)
```

もしミューテックスの所有者がそれを保持中に死んだ場合に矛盾のないミューテックスは矛盾することになります。更に、所有者の死の検出においては、まるで**pthread_mutex_unlock**が実行されたかのようにミューテックスはアンロック状態となります。後続の所有者が所有権を与えられた**pthread_mutex_lock**から戻りEOWNERDEADエラーを受信することを除いて、ロックは通常のように機能し続けます。これは取得したミューテックスが矛盾していることを新しい所有者へ知らせています。

このサービスは矛盾するミューテックスの所有者に呼ばれることのみ可能です。

pthread_mutex_getunlock_np

このサービスはミューテックスのアンロック・ポリシーを返します。

```
int pthread_mutex_getunlock_np(const pthread_mutex_t *mutex, int *policy)
```

アンロック・ポリシーは**policy* に次の値が設定されて返されます：

PTHREAD_MUTEX_UNLOCK_SERIAL_NP
: **pthread_mutex_unlock**は、所有者からロックを待っている最高優先度スレッドへ直接ロックを渡します。

PTHREAD_MUTEX_UNLOCK_PARALLEL_NP
: ロックがアンロックされ、ロックを待っている所有者がいる場合、最も重要なスレッドが起こされます。起こされたスレッドは初めてロックを取得しようとする場合と同じようにロックを競います。

PTHREAD_MUTEX_UNLOCK_AUTO_NP
: 起こされるスレッドのPOSIXスケジューリング・ポリシーに基づく上記2つのポリシーの間で選択します。もしスレッドがSCHED_OTHERである場合、パラレル・ポリシーを使用し、そうでない場合はシリアル・ポリシーを使用します。

pthread_mutex_setconsistency_np

このサービスは与えられたミューテックスの整合性状態を設定します。

```
int pthread_mutex_setconsistency_np(pthread_mutex_t *mutex, int state)
```

state は以下のいずれかとなります：:

PTHREAD_MUTEX_ROBUST_CONSISTENT_NP
: 矛盾するミューテックスの矛盾をなくします。アプリケーションは、ミューテックスが矛盾しているとマークされる原因となった問題を解決できた場合のみこれを実行する必要があります。

PTHREAD_MUTEX_ROBUST_NOTRECOVERABLE_NP
回復不可能として矛盾するミューテックスをマークします。アプリケーションは、矛盾するとマークされるミューテックスの原因となった問題を解決できない場合にこれを実行する必要があります。

ミューテックスは当初は矛盾する状態である必要があり、さもなければこのサービスはエラーを返します。ミューテックスの所有者のみこの整合性状態を変更することが可能です。

pthread_mutex_setunlock_np

このサービスはこのミューテックスのアンロック・ポリシーを設定します。

概要

```
int pthread_mutex_setunlock_np(pthread_mutex_t *mutex, int policy)
```

policy は、PTHREAD_MUTEX_UNLOCK_SERIAL_NP, PTHREAD_MUTEX_UNLOCK_PARALLEL_NP, PTHREAD_MUTEX_UNLOCK_AUTO_NPのいずれかとなります。上述の定義は「pthread_mutex_getunlock_np」セクションを参照してください。

pthread_mutexattr_getfast_np

このサービスは*attr* の属性一式で初期化されたミューテックスが高速モードもしくは低速モードで実行するのかどうかを返します。

概要

```
int pthread_mutexattr_getfast_np(const pthread_mutexattr_t *attr,
int *mode)
```

**mode* に次の値が設定されて返されます：

PTHREAD_MUTEX_FASTPATH_NP
attr と共に初期化されるミューテックスは高速モードで実行されます。このモードでは、非競合ロックおよびアンロックはカーネルには入りません。

PTHREAD_MUTEX_SLOWPATH_NP
attr と共に初期化されるミューテックスは低速モードで実行されます。このモードでは、**pthread_mutex_lock**および**pthread_mutex_unlock**毎にカーネルに入ります。

pthread_mutexattr_getprotocol

このサービスはこの属性一式で初期化されるミューテックスのためのプロトコルを返します。

概要

```
int pthread_mutexattr_getprotocol(pthread_mutexattr_t *attr, int *protocol)
```

利用可能なプロトコル：

PTHREAD_PRIO_NONE
スレッドのスケジューリング・ポリシーはこのミューテックスの動作に影響を受けません。

PTHREAD_PRIO_INHERIT
スレッドのスケジューリング・ポリシーは優先度継承のルールに従い変更されます：スレッドがミューテックスの所有者である限り、それは直接的もしくは間接的にミューテックスを取得するために待機している最高優先度スレッドの優先度を継承します。

pthread_mutexattr_getrobust_np

このサービスはこの属性一式で初期化されるミューテックスのためのロウバスト・レベルを返します。

概要

```
int pthread_mutexattr_getrobust_np(const pthread_mutexattr_t *attr,
int *robustness)
```

利用可能なレベル：

PTHREAD_MUTEX_ROBUST_NP
この属性オブジェクトで初期化されるミューテックスはロウバストになります。

PTHREAD_MUTEX_STALLED_NP
この属性オブジェクトで初期化されるミューテックスはロウバストにはなりません。

ロウバスト・ミューテックスはこの所有者が死んで矛盾状態へ移行した時に検出するものです。矛盾状態の定義については「pthread_mutex_consistent_np」を参照してください。

非ロウバスト・ミューテックスはこの所有者が死んで無期限(これは、シグナルに割り込まれるまで、もしくは何か他のスレッドが死んだプロセスに代わりミューテックスをアンロックするまで)でロックされたままの場合に検出しません。

pthread_mutexattr_getunlock_np

このサービスはこの属性一式で初期化されるミューテックスのためのアンロック・ポリシーを返します。

概要

```
int pthread_mutexattr_getunlock_np(const phtread_mutexattr_t *attr, int *mode)
```

利用可能なポリシーは、PTHREAD_MUTEX_UNLOCK_SERIAL_NP, PTHREAD_MUTEX_UNLOCK_PARALLEL_NP, PTHREAD_MUTEX_UNLOCK_AUTO_NPとなります。これらの定義については「pthread_mutex_getunlock_np」セクションを参照してください。

pthread_mutexattr_setfast_np

このサービスはこの属性一式で作成されるミューテックスのための操作モードを設定します。

概要

```
int pthread_mutexattr_setfast_np(pthread_mutexattr_t *attr, int mode)
```

*mode*は、PTHREAD_MUTEX_FASTPATH_NPもしくはPTHREAD_MUTEX_SLOWPATH_NPとなります。これらの定義については「pthread_mutexattr_getfast_np」セクションを参照してください。

pthread_mutexattr_setprotocol

このサービスはこのミューテックス属性一式から作成されるどのミューテックスのプロトコルでも設定します。

概要

```
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, int protocol)
```

protocol は、PTHREAD_PRIO_NONEもしくはPTHREAD_PRIO_INHERITになります。これらの定義については「pthread_mutexattr_getprotocol」を参照してください。

pthread_mutexattr_setrobust_np

このサービスはこのミューテックス属性オブジェクトで作成されるミューテックスのためのロウバスト・レベルを設定します。

概要

```
int pthread_mutexattr_setrobust_np(pthread_mutexattr_t *attr, int robustness)
```

robustness は、PTHREAD_MUTEX_ROBUST_NPもしくはPTHREAD_MUTEX_STALLED_NPになります。これらの定義については「pthread_mutexattr_getrobust_np」を参照してください。

pthread_mutexattr_setunlock_np

このサービスはこのミューテックス属性オブジェクトで作成されるミューテックスのためのアンロック・モードを設定します。

```
int pthread_mutexattr_setunlock_np(pthread_mutexattr_t *attr, int mode)
```

mode は、PTHREAD_MUTEX_UNLOCK_SERIAL_NP, PTHREAD_MUTEX_UNLOCK_PARALLEL_NP, PTHREAD_MUTEX_UNLOCK_AUTO_NPのいずれかになります。これらの定義については「pthread_mutex_getunlock_np」を参照してください。

POSIXミューテックス・プログラムのコンパイル

上述の優先度継承、ロウバスト・ミューテックスを使用するプログラムは標準的な**cc(1)**, **gcc(1), g++(1)**ツールでコンパイルします。

RedHawkの以前のバージョンには、**ccur-gcc**もしくは**ccur-g++**でアプリケーションをコンパイルおよびリンクすることでアクセスされるこれらのミューテックスための拡張部分を提供する代替えの**glibc**が含まれていたことに注意してください。この機能は現在標準**glibc**に含まれており、代替え**glibc**や**ccur-*** コンパイル・スクリプトは既に利用できません。

標準**glibc**追加は代替え**glibc**を通して提供された拡張部分と完全なバイナリ互換です。RedHawkの以前のバージョン上で**ccur-gcc**および**ccur-g++**でコンパイルされた既存のバイナリは、現在のRedHawkのバージョン上で変更なく動き続けます。優先度継承、ロウバスト・ミューテックスを使用する既存のプログラムは標準ツールでソースの変更を必要とせずコンパイルが可能です。しかしながら、**ccur-*** を明記するMakefileは標準ツールを使用するために変更する必要があることに注意してください。その代わり、シンボリック・リンクは名前を**ccur-gcc, ccur-g++**から**gcc, g++**へ指すためにそれぞれ**/usr/bin**に作成することが可能です。

System Vセマフォ

概要

System Vセマフォはプロセスがセマフォ値の交換を介して同期することが可能なプロセス間通信(IPC)メカニズムです。多くのアプリケーションが1つ以上のセマフォの使用を必要としているため、オペレーティング・システムはセマフォの集合もしくは配列を初期化するための機能を持っています。セマフォの集合は1つ以上、最大SEMMSL(**<linux/sem.h>**内に定義)の制限値までのセマフォを収納することが可能です。セマフォのセットは**semget(2)**システムコールを使用することで作成されます。

単純なセマフォだけが必要とされる時、カウンティング・セマフォはより効果的です。
(「POSIXカウンティング・セマフォ」セクションを参照してください)

semgetシステムコールを実行しているプロセスは所有者/作成者になり、いくつのセマフォが集合の中にあるのかを割り出し、自分自身を含む全てのプロセスに対して最初の操作パーミッションを設定します。このプロセスはその後集合の所有権を放棄することが可能、さもなければ**semctl(2)**システムコールを使って操作権限を変更することが可能です。作成されたプロセスは機能が存在する限り常に作成者のままです。操作パーミッションを持つ他のプロセスは、他の制御機能を実行するために**semctl**を使用することが可能です。

セマフォの所有者がパーミッションを許可する場合、どのプロセスでもセマフォを操作することが可能です。集合内の各セマフォを**semop(2)**システムコールによりインクリメントおよびデクリメントすることが可能です(後述の「semopシステムコール」セクションを参照してください)。

セマフォをインクリメントするには、望む大きさの整数値を**semop**システムコールへ渡します。セマフォをデクリメントするには、望む大きさのマイナス(-) 値を渡します。

オペレーティング・システムは、確実にその時点で設定されるセマフォが操作可能なのは1つのプロセスだけとします。同時リクエストは任意の方法で順番に実行されます。

プロセスは値の大きなセマフォの1つをデクリメントすることによりセマフォ値を特定の値よりも大きくするためにテストすることが可能です。もしプロセスが成功する場合、セマフォ値は特定値よりも大きくなります。さもなければセマフォ値はそうなりません。それをしている間、プロセスはセマフォ値が実行を許可(他のプロセスがセマフォをインクリメント)するまでその実行を停止(IPC_NOWAITフラグ未設定)することが可能で、さもなければセマフォ機能は削除されます。

実行を停止する機能は「*ブロッキング・セマフォ操作*」と呼ばれます。この機能もまたゼロと等しいセマフォをテスト(読み取り専用パーミッションが必要)しているプロセスを利用可能で、これは**semop**システムコールへゼロの値を渡すことで実現されます。

一方、プロセスが成功せずその実行を停止するリクエストが無い場合、これは「*非ブロッキング・セマフォ操作*」と呼ばれます。この場合、プロセスは-1を返し、外部変数`errno`にその結果が設定されます。

ブロッキング・セマフォ操作は、プロセスが異なるタイミングでセマフォの値を介して同期することが可能です。IPC機能は許可されたプロセスにより削除されるまで、もしくはシステムが再初期化されるまでオペレーティング・システムの中に留まることを思い出してください。

セマフォの集合が作成された時、集合内の最初のセマフォはセマフォ番号がゼロです。集合内の最後のセマフォ番号は集合の総数よりも1小さい数が設定されます。

1つのシステムコールは、セマフォの集合において一連のこれらのブロッキング/非ブロッキング操作を実行するために使用することが可能です。一連の操作を実行する時、ブロッキング/非ブロッキング操作は集合の一部または全てのセマフォに適用することが可能です。また、操作はセマフォの数のどんな順番でも適用することが可能です。しかし、それらが全て正常に処理されるまでは操作されません。例えば、もし10個のセマフォの集合の6個の処理のうち最初の3個が正常終了し、4つ目の操作でブロックされた場合、6個の操作全てがブロック無しで実行できるようになるまで、集合に対して変更を行うことはありません。操作全てが成功およびセマフォが変更のどちらか一方、もしくは1つ以上の(非ブロック)操作が失敗では、何も変更されません。つまり、操作はアトミックに実行されます。

単一のセマフォもしくはセマフォの集合のための非ブロック操作のどのような失敗も、操作が少しも実行されずに即座に戻る原因となることを思い出してください。これが発生した時、プロセスから-1が返され、外部変数`errno`にその結果が設定されます。

システムコールはプロセスが利用可能なこれらのセマフォ機能を構成します。呼び出し元プロセスシステムコールへ引数を渡し、システムコールはその機能を実行して成功もしくは失敗のどちらか一方となります。もしシステムコールが成功する場合、その機能が実行され適切な情報を返します。そうではない場合、プロセスから-1が返され、外部変数`errno`にその結果が設定されます。

System Vセマフォの利用

セマフォが使用する(実行するまたは制御される)前に一意に識別されるデータ構造体およびセマフォの集合(配列)は作成される必要があります。ユニークな識別子はセマフォ集合識別子(*semid*)と呼ばれ、これは特定のデータ構造体やセマフォの集合を識別するため、もしくは参照するために使用されます。この識別子はシステム内のどのプロセスでもアクセス可能で、通常のアクセス制限事項の対象となります。

セマフォ集合は配列に所定の数の構造体を含みます(集合中のセマフォにつき1つの構造体)。セマフォ集合内のセマフォの数(*nsems*)はユーザーが選択可能です。

semop(2)システムコールで使用される`sembuf`構造体を図5-1に示します。

図5-1 sembuf構造体の定義

```
struct sembuf {
    unsigned short int sem_num; /* semaphore number */
    short int sem_op; /* semaphore operation */
    short int sem_flg; /* operation flag */
};
```

`sembuf` 構造体は**<sys/sem.h>**ヘッダ・ファイル内に定義されています。

semctl(2)サービスコールで使用されるsemid_ds構造体を図5-2に示します。

図5-2 semid_ds構造体の定義

```
struct semid_ds {
    struct ipc_perm sem_perm; /* operation permission struct */
    __time_t sem_otime; /* last semop() time */
    unsigned long int __unused1;
    __time_t sem_ctime; /* last time changed by semctl() */
    unsigned long int __unused2;
    unsigned long int sem_nsems; /* number of semaphores in set */
    unsigned long int __unused3;
    unsigned long int __unused4;
};
```

semid_dsデータ構造体は**<bits/sem.h>**にありますが、ユーザー・アプリケーションは**<bits/sem.h>**ヘッダ・ファイルを内部的に含む**<sys/sem.h>**ヘッダ・ファイルを含める必要があります。

この構造体のメンバーsem_permはipc_perm型であることに注意してください。このデータ構造体は全てのIPC機能(**<bits/ipc.h>**ヘッダ・ファイル)と同じですが、ユーザー・アプリケーションは**<bits/ipc.h>**ヘッダ・ファイルを内部的に含む**<sys/ipc.h>**ファイルを含める必要があります。ipc_permデータ構造体の詳細は3章の「System Vメッセージ」セクション内に記述されています。

semget(2)システムコールは2つの仕事のうち1つを実行します：

- 新しいセマフォ集合識別子を作成し、それ用に対応するデータ構造体とセマフォ集合を作成します
- 既に関連付けられたデータ構造体とセマフォ集合を持つ既存のセマフォ集合識別子を見つけます

実行されるタスクは**semget**システムコールへ渡す*key* 引数の値により決まります。もし*key* が既存の*semid* で使用されておらずIPC_CREATフラグが設定されていない場合、新しい*semid* はシステム・チューニング・パラメータを超えない条件で関連付けられたデータ構造体と作成されたセマフォの集合と共に返されます。

key の値をゼロに指定するためにプライベート・キー(IPC_PRIVATE) として知られる条件もあります。このキーが指定される時、新しい識別子はシステム・チューニング・パラメータを超えない限り、常に関連付けられたデータ構造体と作成されたセマフォの集合と共に返されます。**ipcs(8)**コマンドはsemid用の*key* フィールドを全てゼロとして表示します。

セマフォ集合が作成される時、**semget**を呼び出すプロセスは所有者/作成者になり、関連付けられるデータ構造体はそれに応じて初期化されます。所有権は変更される可能性がありますが、作成されるプロセスは常に作成者のままでいることを思い出してください(「semctlシステムコール」 セクションを参照してください)。セマフォ集合の作成者はこの機能のために最初の操作パーミッションもまた決定します。

もし指定されたキーに対するセマフォ集合識別子が存在する場合、既存の識別子の値が返されます。もし既存のセマフォ集合識別子が返されることを望まないのであれば、制御コマンド(IPC_EXCL)をシステムコールへ渡す*semflg* 引数の中に指定(設定)することが可能です。実際の集合の数よりも大きなセマフォの数(*nsems*)を値として渡された場合はシステムコールは失敗します。もし集合にセマフォがいくつあるのか分からない場合は、*nsems* に対し0を指定してください(詳細な情報については「semgetシステムコール」を参照してください)。

一旦、一意に識別されるセマフォの集合とデータ構造体が作成される、もしくは既存のものが見つかると**semop(2)**および**semctl(2)**を使用することが可能になります。

セマフォの操作はインクリメント、デクリメント、ゼロにするための試験から構成されます。**semop**システムコールはこれらの操作を実行するために使用されます(**semop**システムコールの詳細については「semopシステムコール」を参照してください)。

semctlシステムコールは以下の方法によりセマフォ機能の制御を許可します：

- セマフォの値を返す(GETVAL)
- セマフォの値を設定する(SETVAL)
- セマフォ集合に関する操作を実行する最後のプロセスのPIDを返す(GETPID)
- 現在の値よりもセマフォ値を大きくなるのを待っているプロセスの数を返す(GETNCNT)
- セマフォ値がゼロになるのを待っているプロセスの数を返す(GETZCNT)
- 集合の中の全てのセマフォ値を取得しユーザー・メモリ内の配列の中に収納します(GETALL)
- ユーザー・メモリ内の配列からセマフォ集合内の全てのセマフォ値を設定します(SETALL)
- セマフォ集合に関連付けられたデータ構造体を取得します(IPC_STAT)
- セマフォ集合のために操作パーミッションを変更します(IPC_SET)
- セマフォ集合に関連付けられたデータ構造体とセマフォ集合と共にオペレーティング・システムから特定のセマフォ集合識別子を削除します(IPC_RMID)

semctlシステムコールの詳細は「semctlシステムコール」セクションを参照してください。

semgetシステムコール

semget(2)は新しいセマフォ集合を作成もしくは既存のセマフォ集合を特定します。

本セクションシでは**semget**システムコールの使用方法について説明します。より詳細な情報については、**semget(2)**のmanページを参照してください。このシステムコールの使用を例示するプログラムは、**README.semget.txt**内に提供された多数のコメントと共に**/usr/share/doc/ccur/examples/semget.c**で見つけることが可能です。

概要

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

    int semget (key_t key, int nsems, int semflg);
```

上記の全てのインクルードファイルは、オペレーティング・システムの**/usr/include**サブディレクトリにあります。

key_t はヘッダーファイル**<bits/sys/types.h>**の中で整数型にするためにtypedefによって定義されています(このヘッダーファイルは**<sys/types.h>**内部に含まれています)。正常終了した場合にこのシステムコールから返される整数は、*key* の値に対応するセマフォ集合識別子(*semid*)です。*semid* は本章の「System Vセマフォの利用」セクション内で説明されています。

セマフォ集合およびデータ構造体に対応する新しい*semid* は以下の条件に1つでも該当する場合に作成されます。

- *key* が IPC_PRIVATE
- セマフォ集合およびデータ構造体に対応する*semid* が存在しない*key* 、かつ*semflg*と IPC_CREAT の論理積がゼロではない

semflg 値の組み合わせ：

- 制御コマンド (フラグ)
- 操作パーミッション

制御コマンドはあらかじめ定義された定数です。以下の制御コマンドは**semget**システムコールに適用され、**<sys/ipc.h>**ヘッダーファイル内部に含まれる**<bits/ipc.h>**ヘッダーファイル内に定義されています。

IPC_CREAT	新しいセグメントをセマフォ集合するために使用されます。もし使用されない場合、**semget**は*key* に対応するセマフォ集合の検出し、アクセス許可の確認します。
IPC_EXCL	IPC_CREATと一緒の使用は、指定された*key* に対応するセマフォ集合識別子が既に存在している場合、このシステムコールはエラーを引き起こします。これは新しい(ユニークな)識別子を受け取らなかった時に受け取ったと考えてしまうことからプロセスを守るために必要です。

パーミッション操作はユーザー、グループ、その他のために読み取り/書き込み属性を定義します。
表5-1は有効な操作パーミッションコードの(8進数で示す)数値を示します。

表5-1 セマフォ操作パーミッション・コード

操作パーミッション	8進数値
Read by User	00400
Alter by User	00200
Read by Group	00040
Alter by Group	00020
Read by Others	00004
Alter by Others	00002

特有の値は、必要とする操作パーミッションのために8進数値を追加もしくはビット単位の論理和によって生成されます。これが、もし「Read by User」と「Read/Write by Others」を要求された場合、コードの値は00406 (00400＋00006)となります。

semflg 値は、フラグ名称と8進数の操作パーミッション値を一緒に使用して簡単に設定することが可能です。
使用例：

```
semid = semget (key, nsems, (IPC_CREAT | 0400));
semid = semget (key, nsems, (IPC_CREAT | IPC_EXCL | 0400));
```

以下の値は**<linux/sem.h>**の中で定義されています。これらの値を超えると常に失敗の原因となります。

SHMMNI	いつでも利用可能なユニークなセマフォ集合(*semids*)の最大数
SEMMSL	各セマフォ集合内のセマフォの最大数
SEMMNS	システム全体の全セマフォ集合内のセマフォの最大数

セマフォ制限値のリストは以下のオプションの使用により**ipcs(8)**コマンドで取得することが可能です。詳細はmanページを参照してください。

```
ipcs -s -l
```

特定の関連するデータ構造体の初期化および特定のエラー条件については**semget(2)**のmanページを参照してください。

semctlシステムコール

semctl(2)はセマフォ集合の制御操作を実行するために使用されます。

本セクションでは**semctl** システムコールを説明します。さらに詳細な情報は**semctl(2)**のmanページを参照してください。この呼び出しの使用を説明しているプログラムは、**README.semctl.txt**内に提供された多くのコメントと共に**/usr/share/doc/ccur/examples/semctl.c**で見つけることが可能です。

概要

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl (int semid, int semnum, int cmd, int arg);

union semun
{
    int val;
    struct semid_ds *buf;
    ushort *array;
} arg;
```

上記の全てのインクルードファイルは、オペレーティング・システムの**/usr/include**サブディレクトリにあります。

semid 変数は**semget**システムコールを使って作成された有効な負ではない整数値でなければなりません。

semnum 引数はその数でセマフォを選択するために使用されます。これは集合の(アトミックに実行される)操作の順番に関連します。セマフォの集合が作成される時、最初のセマフォは数が0、最後のセマフォは集合の総数よりも1小さい数が設定されます。

cmd 引数は以下の値のいずれかとなります。

GETVAL	セマフォ集合内の単一のセマフォ値を返します
SETVAL	セマフォ集合内の単一のセマフォ値を設定します
GETPID	セマフォ集合内のセマフォの操作を最後に実行したプロセスのPIDを返します
GETNCNT	現在値よりも大きくなるために特定のセマフォの値を待っているプロセスの数を返します
GETZCNT	ゼロになるために特定のセマフォの値を待っているプロセスの数を返します
GETALL	セマフォ集合内の全てのセマフォの値を返します
SETALL	セマフォ集合内の全てのセマフォの値を設定します

IPC_STAT　指定された*semid* に関連するデータ構造体に含まれるステータス情報を返し、*arg*.`buf` で指し示されたデータ構造体の中に格納します

IPC_SET　指定されたセマフォ集合(*semid*) に対して有効なユーザー/グループIDと操作パーミッションを設定します

IPC_RMID　指定されたセマフォ集合とそれに関連するデータ構造体と共に削除します

NOTE

semctl(2)サービスはIPC_INFO, SEM_STAT, SEM_INFOコマンドもサポートします。しかし、これらのコマンドは**ipcs(8)**ユーティリティで使用するためだけに意図されているので、これらのコマンドについての説明はありません。

IPC_SETまたはIPC_RMID制御コマンドを実行するため、プロセスは以下の条件を1つ以上満たしていなければなりません。

- 有効なOWNERのユーザーIDを所有
- 有効なCREATORのユーザーIDを所有
- スーパーユーザー
- CAP_SYS_ADMINケーパビリティを所有

セマフォ集合は、**-s** *semid* (セマフォ集合識別子)または**-S** *semkey* (対応するセマフォ集合のキー)オプション指定による**ipcrm(8)**コマンドの利用で削除される可能性もあることに注意してください。このコマンドを使用するため、プロセスはIPC_RMID 制御コマンドの実行に必要となるのと同じ権限を持っている必要があります。このコマンドの使用に関して更なる情報は**ipcrm(8)**のmanページを参照してください。

残りの制御コマンドは必要に応じて読み取りもしくは書き込みパーミッションのいずれかが必要になります。

arg 引数は制御コマンドが実行するために適切な共用体をシステムコールに渡して使用されます。制御コマンドの一部に関しては、*arg* 引数は必要とされずに単に無視されます。

- *arg*.`val`に必須： SETVAL
- *arg*.`buf`に必須： IPC_STAT, IPC_SET
- *arg*.`array`に必須： GETALL, SETALL
- *arg* は無視： GETVAL, GETPID, GETNCNT, GETZCNT, IPC_RMID

semopシステムコール

semop(2)は選択されたセマフォ集合のメンバの操作を実行するために使用されます。

本セクションでは**semop**システムコールを説明します。さらに詳細な情報は**semop(2)**のmanページを参照してください。この呼び出しの使用を説明しているプログラムは、**README.semop.txt**内に提供された多くのコメントと共に**/usr/share/doc/ccur/examples/semop.c**で見つけることが可能です。

概要

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop (int semid, struct sembuf *sops, unsigned nsops);
```

上記の全てのインクルードファイルは、オペレーティング・システムの**/usr/include**サブディレクトリにあります。

semopシステムコールは正常終了でゼロ、そうでない場合は-1の整数値を返します。

semid 引数は有効な正の整数値である必要があります。または、それは事前に**semget(2)**システムコールから返されている必要があります。

sops 引数は各セマフォを変更するために以下を含むユーザー・メモリ領域内の構造体の配列を指し示します：

- セマフォの番号 (*sem_num*)
- 実行する操作 (*sem_op*)
- 制御フラグ (*sem_flg*)

sops* 宣言は配列名称(配列の最初の要素のアドレス)もしくは使用可能な配列へのポインタを意味します。`sembuf` は配列内の構造体メンバのテンプレートして使用されるデータ構造体のタグ名称で、それは<sys/sem.h>**ヘッダ・ファイルにあります。

nsops 引数は配列の長さ(配列内の構造体の数)を指定します。この配列の最大サイズはSEMOPMシステム・チューニング・パラメータによって決定されます。従って、SEMOPM操作の上限は各**semop**システムコールに対して実行されることが可能です。

セマフォ番号(*sem_num*) は操作が実行される集合内の特定のセマフォを確定します。

実行される操作は以下によって決定されます：

- *sem_op* が正の場合、セマフォ値は*sem_op* の値によりインクリメントされます
- *sem_op* が負の場合、セマフォ値は*sem_op* の絶対値によりデクリメントされます
- *sem_op* がゼロの場合、セマフォ値はゼロと等しくなるまでテストされます

以下の操作コマンド(フラグ)を使用することが可能：

IPC_NOWAIT	配列内のどのような操作でも設定することが可能です。もしIPC_NOWAITが設定されているどのような操作もうまく実行できない場合、セマフォの値を少しも変更することなくシステムコールは失敗して戻ります。セマフォの現在の値よりもデクリメントしようとする時、そうではなくセマフォをゼロと等しくするためにテストをする時にシステムコールは失敗します。
SEM_UNDO	プロセスが終了する時に自動的にプロセスのセマフォの変更を元に戻すことをシステムに指示し、これはプロセスがデッドロック問題を回避することを可能にします。この機能を実装するため、システム内のプロセス毎のエントリを含むテーブルをシステムは維持します。各エントリはプロセスに使用される各セマフォのためのアンドゥ構造体の集合を指し示します。システムは最終的な変更を記録します。

条件同期

以下のセクションでは、協同プロセスを操作するために使用することが可能な**postwait(2), server_block**/**server_wake(2)**の各システムコールを説明します。

postwaitシステムコール

postwait(2)ファンクションは協同グループのスレッド間で使用する高速で効果的なスリープ/ウェイクアップ/タイマーのメカニズムです。スレッドは同じプロセスのメンバーである必要はありません。

概要

```
#include <sys/time.h>
#include <sys/rescntl.h>
#include <sys/pw.h>

int pw_getukid(ukid_t *ukid);
int pw_wait(struct timespec *t, struct resched_var *r);
int pw_post(ukid_t ukid, struct resched_var *r);
int pw_postv(int count, ukid_t targets[], int errors[], struct
resched_var *r );
int pw_getvmax(void);
```

gcc [*options*] *file* **-lccur_rt** ...

スリープ状態にするため、スレッドは**pw_wait()**を呼び出します。スレッドは次の時に起き上がります：

- タイマーが終了する
- スレッドが、**pw_wait**中スレッドの*ukid* による**pw_post()**または**pw_postv()**の呼び出しで他のスレッドよりポストされる
- 呼び出しが割り込まれる

postwait(2)サービスを使用しているスレッドは*ukid* によって識別されます。スレッドは*ukid* を取得するために**pw_getukid()**を呼び出す必要があります。*ukid* は呼び出し元のユニークなグローバル・スレッドIDへマッピングします。この値はこのスレッドへポストする可能性のある他の共同スレッドと共有することが可能です。

スレッド毎に、**postwait(2)**は多くても1つの未消費ポストを覚えています。未消費ポストを持っているスレッドへポストしても効果はありません。

再スケジューリング変数のポインタ引数を持つ全ての**postwait(2)**サービスにおいて、もしそのポインタがNULLではない場合、関連する再スケジューリング変数のロック・カウントはデクリメントされます。

pw_wait()はポストを消費するために使用されます。これは任意のタイムアウト値および任意の再スケジューリング変数と一緒に呼び出します。これは、ポストを消費した場合は1の値、もしくはポストを消費するための待機がタイムアウトした場合は0の値を返します。

もしタイムアウト値に指定された時間が0より大きい場合、スレッドはポストの消費を待つため多くてもその時間分スリープします。もしポストとの接触なしにこの時間が終了する場合は0が返されます。もし呼び出しが割り込まれた場合はEINTRが返され、タイムアウト値は残り時間を反映するために更新されます。もしこのインターバル中にポストされた、もしくは以前の未消費ポストに接触した場合、ポストは消費され1が返されます。

もしタイムアウト値が0の場合、**pw_wait()**は即座に戻ります。これは、以前の未消費ポストが消費された場合は1を返し、もしくは消費可能なポストが存在しない場合はEAGAINを返します。

もしタイムアウト値のへのポインタがNULLである場合、動作はスレッドが決してタイムアウトしないこと以外は同じです。もし割り込まれた場合、EINTRが返されますが指定されていないタイムアウト値は更新されません。

pw_post()　　*ukid* で指定されたスレッドへポストを送信します。もしそのスレッドがポストを待っている場合、スレッドは起き上がりポストを消費します。もしそのスレッドがポストを待っていなかった場合、次回そのスレッドはポストを待とうとするために未消費ポストは記憶され、それは保存されたポストを消費して警告なしで返します。多くても1つの未消費ポストがスレッド毎に記憶されます。

pw_postv()　　一度で複数のスレッドへポストするために使用することが可能です。全てのポストが完了するまで誰もポストしているスレッドにプリエンプトすることが許可されないという点でこれらのポストはアトミックとなります。ターゲットスレッドの*ukid* は、*targets* 配列の中に格納されている必要があります。それぞれのターゲットのエラーは*errors* 配列の中に返されます。*targets* と*errors* 配列で使用されるエントリの数は、*count* 引数を通して渡す必要があります。

pw_postv()　　全て成功した場合は0を返し、いくつかエラーがある場合は最後のターゲットで発生したエラーのエラー値を返します。

pw_getvmax()　ポストすることが可能なターゲットの最大数を返します。

pw_postv()　　この値はカーネル・チューニング・パラメータPW_VMAXにより決定されます。

発生する可能性があるエラーの種類のリストについては**postwait(2)**のmanページを参照してください。

serverシステムコール

一連のシステムコールは、PowerMAXオペレーティング・システムと互換性のあるインターフェースを使うサーバとして動作するプロセスを操作することが可能です。これらのシステムコールを以下で簡単に説明します：

server_block **server_block**から最後に戻った後にウェイクアップ・リクエストが発生しなかった場合のみ呼び出し元プロセスをブロックします。もしウェイクアップが発生した場合、**server_block**は即座に戻ります。

server_wake1 **server_block**システムコールでブロックされた場合にサーバを起こし、もし指定されたサーバがこの呼び出しでブロックされない場合、ウェイクアップ・リクエストはサーバの次の**server_block**の呼び出しに適用します。

server_wakevec プロセスのベクトルが1つのプロセスよりも指定される可能性があることを除いては**server_wake1**と同じ目的で扱います。

CAUTION

これらのシステムコールはシングル・スレッドのプロセスでのみ使用する必要があります。多重スレッドのグローバル・プロセスIDはスレッドが現在スケジュールされているプロセス次第で変わります。従って、これらのインターフェースを多重スレッドで使用する時、間違ったスレッドが起こされるもしくはブロックされる可能性があります。

server_block

server_blockから最後に戻った後にウェイクアップ・リクエストが発生しなかった場合のみ、**server_block**は呼び出し元プロセスをブロックします。

概要

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/pw.h>

int server_block(options, r, timeout)
int options;
struct resched_var *r;
struct timeval *timeout;
```

gcc [*options*] *file* **-lccur_rt ...**

引数は以下のように定義されます：

options この引数の値はゼロである必要があります。

r 呼び出し元プロセスの再スケジューリング変数へのポインタ。この引数は任意で、この値をNULLにすることが可能です。

timeout 呼び出し元プロセスをブロックする最大時間を含む`timeval`構造体へのポインタ。この引数は任意でこの値をNULLにすることが可能です。もしこの値がNULLの場合、タイムアウトはありません。

もし呼び出し元プロセスが保留中のウェイクアップ・リクエストを持っている場合、**server_block**システムコールは即座に戻り、さもなければ呼び出し元プロセスが次のウェイクアップ・リクエストを受信する時に返ります。戻り値0は呼び出しが成功したことを示します。戻り値-1はエラーが発生したことを示し、errnoはエラーを示すために設定されます。戻るときに呼び出し元プロセスはブロックする原因になった条件を再テストする必要がありますが、プロセスが早期にシグナルで起こされることもあるので条件が変わったことを保証しないことに注意してください。

server_wake1

server_wake1は**server_block**の呼び出しでブロックされているサーバを起こすために呼び出されます。

概要

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/pw.h>

int server_wake1(server, r)
global_lwpid_t server;
struct resched_var *r;
```

gcc [*options*] *file* **-lccur_rt ...**

引数は以下のように定義されます：

server 起こされるサーバ・プロセスのグローバル・プロセスID

r 呼び出し元プロセスの再スケジューリング変数へのポインタ。この引数は任意で、この値をNULLにすることが可能です。

server_wake1呼び出しで使用するために、呼び出し元プロセスの実在するもしくは有効なユーザーIDは、*server* で指定されたプロセスの実在するもしくは(**exec**で)保存されたユーザーIDと一致しなければならないことに注意することが重要です。

もし**server_block**呼び出しで指定されたサーバがブロックされている場合、**server_wake1**はそれを起こします。もしこの呼び出しでサーバがブロックされていない場合、ウェイクアップ・リクエストはサーバの次の**server_block**呼び出しまで持っています。**server_wake1**もやはり*r* に指定された再スケジューリング変数に関連付けられた再スケジューリング・ロックの数をデクリメントします。

戻り値0は呼び出しが成功したことを示します。戻り値-1はエラーが発生したことを示し、errnoはエラーを示すために設定されます。

server_wakevec

server_wakevecシステムコールは**server_block**の呼び出しでブロックされたサーバのグループを起こすために呼び出されます。

概要

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/pw.h>

int server_wakevec(servers, nservers, r)
global_lwpid_t *servers;
int nservers;
struct resched_var *r;
```

gcc [*options*] *file* **-lccur_rt ...**

引数は以下のように定義されます：

servers	起こされるサーバ・プロセスのグローバル・プロセスIDの配列へのポインタ
nservers	配列のエレメント数を指定する整数値
r	呼び出し元プロセスの再スケジューリング変数へのポインタ。この引数は任意で、この値をNULLにすることが可能です。

server_wakevec呼び出しで使用するために、呼び出し元プロセスの実在するもしくは有効なユーザーIDは、*servers* で指定されたプロセスの実在するもしくは(**exec**で)保存されたユーザーIDと一致しなければならないことに注意することが重要です。

もし**server_block**呼び出しで指定されたサーバがブロックされている場合、**server_wakevec**はそれらを起こします。もしこの呼び出しでサーバがブロックされていない場合、ウェイクアップ・リクエストはサーバの次の**server_block**呼び出しまで適用します。**server_wakevec**もやはり*r* に指定された再スケジューリング変数に関連付けられた再スケジューリング・ロックの数をデクリメントします。

戻り値0は呼び出しが成功したことを示します。戻り値-1はエラーが発生したことを示し、`errno`はエラーを示すために設定されます。

これらの呼び出しの使用に関する追加の情報については、**server_block(2)**のmanページを参照してください。

条件同期ツールの適用

再スケジューリング変数、スピンロック、サーバ・システムコールは、共有メモリ領域内のメールボックスの使用を通して生産者プロセスと消費者プロセスのデータ交換を可能にする機能を設計するために使用することが可能です。消費者が空のメールボックスを見つけた時、それは新しいデータが到着するまでブロックします。生産者がメールボックスの中に新しいデータを挿入した後、それは待機中の消費者を起こします。消費者がそれを処理するよりも早く、生産者がデータを生成した時、類似の状況が発生します。生産者が満杯のメールボックスを見つけた時、それはデータが削除されるまでプロックします。消費者がデータを削除した後、それは待機中の生産者を起こします。

メールボックスは以下のように表すことが可能です：

```
struct mailbox {
    struct spin_mutex mx;    /* serializes access to mailbox */
    queue_of consumers;      /* waiting consumers */
    queue_of data;           /* the data, type varies */
};
```

mx フィールドはメールボックスへ順番にアクセスするために使用し、*consumers*フィールドはデータを待っているプロセスを識別し、*data*フィールドは生産者から消費者へ渡されるデータを保持します。*queue_of* 型は通常2つのオペレータ(リストの最後尾に項目をリンクする**enqueue**とリストの先頭に項目をリンクするd**equeue**)をサポートするリンクされたリストを定義します。

spin_acquireと**spin_release**の機能の使用して、消費者がメールボックスからデータを取り出すことが可能になる関数は以下のように定義することが可能です：

```
void
consume (box, data)
    struct mailbox *box;
    any_t *data;
{
    spin_acquire (&box->mx, &rv);
    while (box->data == empty) {
        enqueue (box->consumers, rv.rv_glwpid);
        spin_unlock (&box->mx);
        server_block (0, &rv, 0);
        spin_acquire (&box->mx, &rv);
    }
    *data = dequeue (box->data);
    spin_release (&box->mx, &rv);
}
```

この関数では、消費者プロセスはデータのチェックおよび削除の前にメールボックスをロックすることに注意してください。もしこれが空のメールボックス見つけた場合、生産者がデータを挿入するのを許可するためにメールボックスをアンロックし、データの到着を待つために**server_block**を呼び出します。消費者が起こされた時に再度メールボックスをロックし、データをチェックする必要があります(消費者が早期にシグナルによって起こされる可能性があり、メールボックスがデータを収容している保証がない)。

同様に生産者がメールボックスにデータを収納することを可能にする関数は以下のように定義することが可能です：

```
void
produce (box, data)
    struct mailbox *box;
    any_t data;
{
    spin_acquire (&box->mx, &rv);
    enqueue (box->data, data);
    if (box->consumer == empty)
        spin_release (&box->mx, &rv);
    else {
        global_lwpid_t id = dequeue (box->consumers);
        spin_unlock (&box->mx);
        server_wake1 (id, &rv);
    }
}
```

この関数では、生産者プロセスは新しいデータを挿入する前にメールボックスが空になるのを待ちます。生産者は消費者が待機している時のみデータの到着を通知、これはメールボックスをアンロックした後にそうすることに注意して下さい。起き上がった消費者はデータのチェックおよび削除のためにメールボックスをロックする可能性があるため、生産者は最初にメールボックスをアンロックする必要があります。**server_wake1**の呼び出しの前にメールボックスをアンロックすることもやはり相互排除を短時間保持することを確実にします。不必要なコンテキスト・スイッチを回避するため、再スケジューリングは消費者が起こされるまで無効にします。

6
プログラム可能なクロックおよびタイマー

本章ではタイミングのために使用可能ないくつかの機能の概要を提供します。POSIXクロックおよびタイマー・インターフェースはIEEE規格1003.1b-1993に準拠しています。クロック・インターフェースは、タイムスタンプまたはコード・セグメント長の時間計測などの目的のために使用することが可能な高分解能クロックを提供します。タイマー・インターフェースは将来シグナルを受信する手段もしくは非同期にプロセスを起こす手段を提供します。更に非常に短い時間プロセスをスリープ状態にするために利用可能で、スリープ時間の測定に使用できるクロックを指定できる高分解能システムコールを提供します。追加のクロックとタイマーはRCIM PCIカードにより提供されます。

クロックおよびタイマーの理解

リアルタイム・アプリケーションはアプリケーションまたはシステムイベントをスケジュールするために厳格なタイミングの制約内でデータを操作できる必要があります。高分解能のクロックとタイマーは、アプリケーションが高分解能クロックに基づく相対または絶対時間を使用する事やワンショットまたは定期的にイベントをスケジュールすることが可能です。アプリケーションは各プロセスのために複数のタイマーを作成することが可能です。

いくつかのタイミング機能はiHawkシステム上で利用可能です。これらはPOSIXクロックとタイマーも非割り込みクロックやリアルタイム・クロック＆インタラプト・モジュール(RCIM) PCIカードにより提供されるリアルタイム・クロック・タイマーも含みます。これらのクロックとタイマーおよびそれらのインターフェースは以下のセクションで説明しています。

システム・クロックとタイマーに関する情報は7章を参照してください。

RCIMクロックおよびタイマー

リアルタイム・クロック＆インタラプト・モジュール(RCIM)は2つの非割り込みクロックを提供します。これらのクロックはRCIMがチェーン接続されている時に他のRCIMと同期させることが可能です。2つのRCIMクロックは以下のとおり：

tick clock
: 一般的な400nsのクロック信号のティックを1づつインクリメントする64 bit非割り込みクロック。このクロックは共通のタイムスタンプを提供するチェーン接続されたRCIM全体でゼロにリセットおよび同期することが可能です。

  ティック・クロックはマスターでもスレーブでもどのシステムでもプログラムのアドレス空間に**/dev/rcim/sclk** デバイス・ファイルをマッピングしている時にダイレクト・リードを使用して読み取ることが可能です。

POSIX
: POSIX1003.1フォーマットにコード化された64 bit非割り込みカウンター。上位32 bitは秒を収容し、下位32 bitはナノ秒を収容します。このクロックは一般的な400nsのクロック信号のティックを400づつインクリメントされます。主に高分解能ローカル・クロックとして使用されます。

RCIM POSIXクロックは、同じユーティリティとデバイス・ファイルが使われているという点ではティック・クロックと類似する方法でアクセスされます。POSIXクロックは任意の時間をロードすることが可能ですが、ロードした値はチェーン接続されたRCIMの他のクロックとは同期されません。ホストに接続されているRCIMのPOSIXクロックだけは更新されます。

RCIMは最大8個のリアルタイム・クロック(RTC)タイマーも供給します。これらの各カウンタは特別なデバイス・ファイルを使ってアクセス可能で各々は殆どのタイミングまたは周波数を制御する機能のために使用することが可能です。それらはクロック・カウント値を組み合わせる事で様々なタイミング間隔を供与しそれぞれ異なる分解能にてプログラム可能です。これは所定の周波数(例えば100Hz)でプロセスを実行する、もしくコード・セグメントのタイミングには理想的な状態となります。ホスト・システム上で割り込みを生成することが出来ることに加えて、RTCの出力が対応するホストに対して配信するため、またはRCIMの外部出力割り込み線の1つに接続された外部機器へ配信するため、他のRCIMボードに対して分配することが可能となります。RTCタイマーは**open(2)**, **close(2)**, **ioctl(2)**の各システムコールにより制御されます。

RCIMクロックおよびタイマーに関する全ての情報については*Real-Time Clock and Interrupt Module (RCIM) User's Guide* を参照してください。

POSIXクロックおよびタイマー

POSIXクロックは時間の測定および表示のために高分解能メカニズムを提供します。

POSIXクロックには2種類のタイマー(ワン・ショットと周期)があります。これらは最初の満了時間と繰り返し間隔に関して定義されます。これは絶対時刻(例：午前8:30)もしくは現在時刻からの相対時間(例：30秒後)にすることが可能です。繰り返し間隔はタイマー満了から次までに経過する時間量を指示します。タイミング用に使用されるクロックは、タイマーが作成された時に指定されます。

ワン・ショット・タイマーは絶対または相対初期満了時間とゼロの繰り返し間隔のいずれも実装されています。これは(初期満了時間の)たった1回で終了し、その後実装が解除されます。

周期タイマーは絶対または相対初期満了時間とゼロよりも大きな繰り返し間隔のいずれも実装されます。繰り返し間隔は、常に最後のタイマー満了時点との相対です。最初の満了時間が来た時、タイマーは繰り返し間隔の値をリロードし、カウントを継続します。タイマーは初期満了時間をゼロへ設定することにより実装を解除することが可能です。

ローカルタイマーはPOSIXタイマー満了スケジューリングの割り込みソースとして使用されます。ローカル・タイマーに関する情報については7章を参照してください。

NOTE

高分解能クロックおよびタイマーへのアクセスは、**libccur_rt**および**librt**内のシステムコールにより提供されますが、**libccur_rt**ルーチンは軽視されることになります。常に‘ccur_rt’の前に‘rt’をリンクして**librt**内のルーチンを使用することを推奨します。
例：

gcc [*options*] *file* **-lrt -lccur_rt** ...

POSIX時間構造体の理解

POSIXルーチンに関連するクロックおよびタイマーは時間指定のために2つの構造体(`timespec`構造体と`itimerspec`構造体)を使用します。これらの構造体は**<time.h>**ファイルの中で定義されます。

`timespec`構造体は秒とナノ秒で単一時間値を指定します。クロックの時間設定もしくは時間/クロックの分解能を取得するためにルーチン(これらのルーチンに関する情報は「POSIX clockルーチンの利用」を参照してください)を呼び出す時に`timespec` 構造体へのポインタを指定します。構造体は以下のように定義されます：

```
struct timespec {
    time_t tv_sec;
    long tv_nsec;
};
```

構造体内のフィールドは以下で説明します：

`tv_sec`	時間値内の秒数を指定します。 specifies the number of seconds in the time value
`tv_nsec`	時間値内の追加のナノ秒数を指定します。このフィールドの値は、ゼロから999,999,999の範囲内である必要があります。

`itimerspec`構造体はタイマー用に最初の満了時間と繰り返し間隔を指定します。タイマーが満了する時間の設定もしくはタイマーの満了時間に関する情報の取得のためにルーチン(これらのルーチンに関する情報は「POSIX clockルーチンの利用」を参照してください)を呼び出す時に`itimerspec`構造体へのポインタを指定します。構造体は以下のように定義されます：

```
struct itimerspec {
    struct timespec it_interval;
    struct timespec it_value;
};
```

構造体内のフィールドは以下で説明します：

`it_interval`	タイマーの繰り返し間隔を指定します
`it_value`	タイマーの最初の満了時間を指定します

POSIX clockルーチンの利用

クロックに関連する様々な機能を実行することが可能なPOSIXルーチンを以下で簡単に説明します。

clock_settime	指定したクロックの時間を設定します。
clock_gettime	指定したクロックから時間を取得します。
clock_getres	指定したクロックのナノ秒単位の分解能を取得します。

これらのルーチンの各々の使用手順は以降のセクションで説明します。

clock_settimeルーチンの利用

clock_settime(2)システムコールはシステムtime-of-dayクロック、CLOCK_REALTIMEの時間を設定することが可能です。呼び出し元プロセスはルートもしくはCAP_SYS_NICEケーパビリティを所有している必要があります。定義上、CLOCK_MONOTONICクロックは設定することができません。

もしシステム起動後にCLOCK_REALTIMEを設定した場合、以下の時間は正確ではない可能性があることに注意する必要があります：

- ファイル・システムの作成および変更時間
- アカウンティングおよび監査記録内の時間
- カーネル・タイマー・キュー・エントリの満了時間

システム・クロックの設定はキューイングされたPOSIXタイマーに影響を及ぼしません。

概要

```
#include <time.h>

int clock_settime(clockid_t which_clock, const struct timespec *setting);
```

引数は以下のように定義されます：

which_clock	時間が設定されるクロックの識別子。CLOCK_REALTIMEだけが設定することが可能です。
setting	*which_clock* へ設定する時間を指定する構造体へのポインタ。*which_clock* がCLOCK_REALTIMEの時、time-of-dayクロックは新しい値が設定されます。クロック分解能の整数倍ではない時間値は切り捨てられます。

戻り値0は指定したクロックの設定に成功したことを示します。戻り値-1はエラーが発生したことを示し、`errno`はエラーを示すために設定されます。発生する可能性があるエラーの種類のリストについては**clock_settime(2)**のmanページを参照してください。

clock_gettimeルーチンの利用

clock_gettime(2)システムコールは指定したクロックから時間を取得することが可能です。この呼び出しは常に利用可能な最高のクロック(通常は1マイクロ秒より上)の分解能を返します。

概要

```
#include <time.h>

int clock_gettime(clockid_t which_clock, struct timespec *setting);
```

引数は以下のように定義されます：

which_clock	時間を取得するクロックの識別子。*which_clock* の値はCLOCK_REALTIMEまたはCLOCK_MONOTONICにすることが可能です。
setting	*which_clock* の時間が返される構造体へのポインタ。

戻り値0は**clock_gettime**の呼び出しが成功したことを示します。戻り値-1はエラーが発生したことを示し、errnoはエラーを示すために設定されます。発生する可能性があるエラーの種類のリストについては**clock_gettime(2)**のmanページを参照してください。

clock_getresルーチンの利用

clock_getres(2)システムコールは指定したクロックのナノ秒単位の分解能を取得することが可能です。分解能は、**clock_settime(2)**で設定したタイミング満了を丸めた精度に決定し、その精度は同じクロックを使用する**clock_nanosleep(2)**と**nanosleep(2)**の呼び出しで使用されます。

クロックの分解能はシステム依存でありユーザーが設定することはできません。

概要

```
#include <time.h>

int clock_getres(clockid_t which_clock, struct timespec *resolution);
```

引数は以下のように定義されます：

which_clock	分解能を取得するクロックの識別子。*which_clock* の値はCLOCK_REALTIMEまたはCLOCK_MONOTONICにすることが可能です。
resolution	*which_clock* の分解能が返される構造体へのポインタ。

戻り値0は**clock_getres**の呼び出しが成功したことを示します。戻り値**-1**はエラーが発生したことを示し、`errno`はエラーを示すために設定されます。発生する可能性があるエラーの種類のリストについては**clock_getres(2)**のmanページを参照してください。

POSIX timerルーチンの利用

プロセスはタイマーを作成、削除、設定、問い合わせすることが可能でタイマーが満了した時に通知を受け取ることが可能です。

タイマーに関連した様々な機能を実行可能なPOSIXシステムコールを以下で簡単に説明します。

timer_create	指定したクロックを使用するタイマーを作成
timer_delete	指定したタイマーを削除
timer_settime	満了時間の設定で指定したタイマーを実装または解除
timer_gettime	指定したタイマーの繰り返し間隔とタイマー満了までの残り時間を取得
timer_getoverrun	指定した周期タイマーのオーバーラン総数を取得
nanosleep	指定した時間だけ実行を一時停止
clock_nanosleep	指定したクロックに基づき高分解能一時停止を提供

これらの各システムコールの使用手順は以降のセクションで説明します。

timer_createルーチンの利用

timer_create(2)システムコールは、呼び出し元プロセスがタイミング・ソースとして指定されたクロックを使用するタイマーを作成することが可能です。

それが作成される時、タイマーは解除されます。プロセスが**timer_settime(2)**システムコールを呼び出した時に実装されます(このシステムコールの説明は「timer_settimeルーチンの利用」を参照してください)。

以下に注意することが重要です：

- プロセスが**fork** システムコールを呼び出す時、作成されたタイマーは子プロセスには継承しません。
- プロセスが**exec** システムコールを呼び出す時、作成されたタイマーは解除および削除されます。

同じスレッド・グループ内のLinuxスレッドはタイマーを共有することが可能です。**timer_create**を呼び出したスレッドはシグナル全てを受信しますが、同じスレッド・グループ内の他のスレッドは**timer_settime(2)**の呼び出しを通してタイマーを操作することが可能です。

概要

```
#include <time.h>
#include <signal.h>

int timer_create(clockid_t which_clock, struct sigevent *timer_event_spec, timer_t
created_timer_id);
```

引数は以下のように定義されます：

which_clock タイマーに使用されるクロックの識別子。*which_clock* の値はCLOCK_REALTIMEである必要があります。

timer_event_spec NULLポインタ定数または呼び出し元プロセスにタイマー満了を非同期で通知する方法を指定する構造体へのポインタ：

NULL タイマー満了時にSIGALRMがプロセスへ送信されます

sigev_notify=SIGEV_SIGNAL
sigev_signo に指定されたシグナルはタイマー満了時にプロセスへ送信されます。

sigev_notify=SIGEV_THREAD
指定した*sigev_notify* 機能はタイマー満了時に*sigev_value* を引数として新しいスレッドの中から呼ばれます。現在、**-lccur_rt**ではサポートされていないため、**-lrt**を最初にリンクして使用します。

sigev_notify=SIGEV_THREAD_ID
sigev_notify_thread_id の番号にはタイマー満了時に*sigev_signo*シグナルを受信するスレッドのID(`pthread_t`)を収納する必要があります。

sigev_notify=SIGEV_NONE
タイマー満了時に通知は配信されません。

NOTE

タイマー満了を意味するシグナルは、シグナルを処理するシステムコールを指定しない限りプロセスを終了させる原因となる可能性があります。特定のシグナルへの既定のアクションを決定するために**signal(2)**のmanページを参照してください。

created_timer_id タイマーIDが格納されている場所へのポインタ。この識別子は他のPOSIXタイマーのシステムコールで必要とされ、システムコールでタイマーが削除されるまで呼び出し元プロセスの中では一意です。

戻り値0は**timer_create**の呼び出しが成功したことを示します。戻り値-1はエラーが発生したことを示し、`errno`はエラーを示すために設定されます。発生する可能性があるエラーの種類のリストについては**timer_create(2)**のmanページを参照してください。

timer_deleteルーチンの利用

timer_delete(2)システムコールは呼び出し元プロセスが指定されたタイマーを削除することが可能です。もし選択されたタイマーが既に開始されている場合、これは無効になりタイマーに割り付けられているシグナルもしくはアクションは配信または実行されません。タイマー満了からシグナルが保留中であっても削除されません。

概要

```
#include <time.h>

int timer_delete(timer_t timer_id);
```

引数は以下のように定義されます：

timer_id 削除するタイマーの識別子。この識別子は前の**timer_create(2)**呼び出しから来ています(このシステムコールの説明は「timer_createルーチンの利用」を参照してください)。

戻り値0は指定したタイマーの削除に成功したことを示します。戻り値-1はエラーが発生したことを示し、`errno`はエラーを示すために設定されます。発生する可能性があるエラーの種類のリストについては**timer_delete(2)**のmanページを参照してください。

timer_settimeルーチンの利用

timer_settime(2)システムコールは、タイマーが満了する時間を設定することで呼び出し元プロセスが指定されたタイマーを実装することが可能です。満了する時間は絶対値または相対値で定義します。呼び出し元プロセスは、実装されたタイマーに対して次のタイマー満了までに(1)タイマーの解除、または(2)時間のリセットをするためにこのシステムコールを使用することが可能です。

概要

```
#include <time.h>

int timer_settime(timer_t timer_id, int flags, const struct itimerspec *new_setting,
const struct itimerspec *old_setting);
```

引数は以下のように定義されます：

timer_id 設定するタイマーの識別子。この識別子は前の**timer_create(2)**呼び出しから来ています(このシステムコールの説明は「timer_createルーチンの利用」を参照してください)。

flags 以下のいずれかを指定する整数値：

TIMER_ABSTIME 選択されたタイマーは絶対満了時間を実装します。タイマーは、タイマーに関連付けられたクロックが*it_value* で指定された値に到達する時に満了となります。もしこの時間が既に過ぎている場合、**timer_settime**は成功し、タイマー満了通知が行われます。

	0	選択されたタイマーは相対満了時間を実装します。タイマーは、タイマーに関連付けられたクロックが*it_value* で指定された値に到達する時に満了となります。
new_setting	繰り返し間隔とタイマーの初期満了時間を格納する構造体へのポインタ。 もしワン・ショット・タイマーを望む場合はゼロの繰り返し間隔(*it_interval*)を指定します。この場合、初期満了時間になった時、一旦タイマーが満了となり解除されます。 もし周期的なタイマーを望む場合はゼロではない繰り返し間隔(*it_interval*)を指定します。この場合、初期満了時間になった時、タイマーは繰り返し間隔の値をリロードしてカウントを続けます。 いずれにせよ、初期満了時間として絶対値(例：午後3:00)または現在時刻からの相対値(例：30秒後)を設定することが可能です。初期満了時間に絶対値を設定するには*flags* 引数にTIMER_ABSTIMEビットを設定する必要があります。指定されたタイマーが前に満了となったことが原因で既に保留中のどのようなシグナルもやはりプロセスへ配信されます。 タイマーを解除するために初期満了時間をゼロに設定します。指定されたタイマーが前に満了となったことが原因で既に保留中のどのようなシグナルもやはりプロセスへ配信されます。	
old_setting	NULLポインタ定数または前の繰り返し間隔とタイマーの初期満了時間を返す構造体へのポインタ。もしタイマーが解除されていた場合、初期満了時間の値はゼロとなります。*old_setting* のメンバーはタイマーの分解能に左右され、その時点で呼び出す**timer_gettime(2)**より返される値と同じになります。	

戻り値0は指定したタイマーの設定に成功したことを示します。戻り値-1はエラーが発生したことを示し、`errno`はエラーを示すために設定されます。発生する可能性があるエラーの種類のリストについては**timer_settime(2)**のmanページを参照してください。

timer_gettimeルーチンの利用

timer_gettime(2)システムコールは呼び出し元プロセスが指定したタイマーの繰り返し間隔とタイマーが満了になるまでの残り時間量を取得することが可能です。

概要

```
#include <time.h>

int timer_gettime(timer_t timer_id, struct itimerspec *setting);
```

引数は以下のように定義されます：

timer_id 繰り返し時間と残り時間をリクエストするタイマーの識別子。この識別子は前の**timer_create(2)**呼び出しから来ています(このシステムコールの説明は「timer_createルーチンの利用」を参照してください)。

setting 繰り返し間隔とタイマーの残り時間量を返す構造体へのポインタ。残り時間量は現在時間との相対です。もしタイマーが解除されている場合、値はゼロになります。

戻り値0は**timer_gettime**の呼び出しに成功したことを示します。戻り値-1はエラーが発生したことを示し、`errno`はエラーを示すために設定されます。発生する可能性があるエラーの種類のリストについては**timer_gettime(2)**のmanページを参照してください。

timer_getoverrunルーチンの利用

timer_getoverrun(2)システムコールは呼び出し元プロセスが特定の周期タイマーのオーバーラン回数を取得することが可能です。タイマーはシステムがアプリケーションへシグナルを配信するよりも速く満了となる可能性があります。もしシグナルが他のシグナルのキューイングではなく前回のタイマー満了から保留中である場合、満了を見逃した総数は保留のシグナルと一緒に保持されます。これはオーバーランの総数となります。

シグナルがアプリケーションにブロックされたため、またはタイマーがオーバーコミットしたためにタイマーがオーバーランとなる可能性があります。

シグナルは常にタイマー満了通知SIGEV_SIGNALを使うタイマー付きプロセスをキューイングまたは保留することを前提とします。もしシグナルがキューイングもしくは保留している間にこのタイマーが満了となる場合、タイマーのオーバーランが発生し、追加のシグナルは送信されません。

NOTE

タイマー満了シグナル・ハンドラーからこのシステムコールを呼び出す必要があります。もし外側でこのシステムコールを呼び出す場合、返されるオーバーラン回数は最後に取得したタイマー満了シグナルに関しては有効ではありません。

概要

```
#include <time.h>
int timer_getoverrun(timer_t timer_id);
```

引数は以下のように定義されます：

timer_id オーバーラン回数を取得したい周期タイマーの識別子。この識別子は前の**timer_create(2)**呼び出しから来ています(このシステムコールの説明は「timer_createルーチンの利用」を参照してください)。

もし呼び出しが成功した場合、**timer_getoverrun**は指定されたタイマーのオーバーラン回数を返します。この回数は**<limits.h>**ファイル内のDELAYTIMER_MAXを超えることはできません。戻り値-1はエラーが発生したことを示し、`errno` はエラーを示すために設定されます。発生する可能性があるエラーの種類のリストについては**timer_getoverrun(2)**のmanページを参照してください。

POSIX sleepルーチンの利用

nanosleep(2)と**clock_nanosleep(2)**のPOSIXシステムコールは、呼び出し元プロセスまたはスレッドを(1)指定された時間が経過するまで、または(2)シグナルを受信し関連する処理がシグナル・ハンドリング・システムコールを実行するもしくはプロセスが終了するまで停止させる高分解能スリープのメカニズムを提供します。

clock_nanosleep(2)システムコールは指定されたクロックによる高分解能スリープを提供します。これは現在動作中スレッドの実行を*rqtp* により指定された時間が経過するもしくはスレッドがシグナルを受信するまで一時停止します。

これらのシステムコールの利用はどのシグナルの動作にも影響を与えません。

nanosleepルーチンの利用

概要

```
#include <time.h>

int nanosleep(const struct timespec *req, struct timespec *rem);
```

引数は以下のように定義されます：

req — プロセスをスリープする時間の長さを含む`timespec`構造体へのポインタ。*req* の値はスリープの分解能の整数倍へ切り上げるため、またはシステムによる他の動作スケジュールのために一時停止時間はリクエストされたよりも長くなる可能性があります。シグナルに割り込まれるケースを除き、一時停止時間はCLOCK_REALTIMEで測定される*req* によって指定される時間よりも短くはなりません。ブロック要求に関しては1 μ 秒の分解能を得られます。

rem — NULLポインタ定数または**nanosleep**がシグナルに割り込まれた場合のスリープ間隔の残り時間が返される`timespec`構造体へのポインタ。もし*rem* がNULLかつ**nanosleep**がシグナルに割り込まれた場合、残り時間は返されません。

戻り値0は要求した時間が経過したことを示します。戻り値-1はエラーが発生したことを示し、`errno`はエラーを示すために設定されます。発生する可能性があるエラーの種類のリストについては**nanosleep(2)**のmanページを参照してください。

clock_nanosleepルーチンの利用

概要

```
#include <time.h>

int clock_nanosleep(clockid_t which_clock, int flags, const struct timespec *rqtp,
struct timespec *rmtp);
```

引数は以下のように定義されます：

which_clock 使用するクロックの識別子。*which_clock* の値はCLOCK_REALTIMEまたはCLOCK_MONOTONICとなります。

flags 以下のいずれかを指定する整数値：

TIMER_ABSTIME *rqtp* で指定された時間は*which_clock* で指定されたクロック値に関する絶対値であると解釈します。

0 *rqtp* で指定された時間は現在時刻の相対値であると解釈します。

rqtp プロセスをスリープする時間の長さを含む`timespec` 構造体へのポインタ。もしTIMER_ABSTIMEフラグが指定され、*rqtp* で指定された時間が指定したクロックの現在時刻以下である(またはクロックの値がその時間へ変更される)場合、この機能は即座に戻ります。更にスリープする時間は**clock_nanosleep(2)**を呼び出した後のクロックのどのような変更にも影響を受けます。つまり、設定または実際の通過時間またはこれらの組み合わせを通して、現在の時間が要求した時間以上の時にクロックがその時間に達したかどうかを問わず呼び出しが完了します。

指定された時間値がクロック分解能の整数倍へ切り上げられる、またはスケジューリングや他のシステムの動作のためにスリープする時間は要求よりも長くなる可能性があります。シグナルによる割り込みのケースを除いて、一時停止時間は決して要求よりも小さくはなりません。

rmtp

TIMER_ABSTIMEが指定されていない場合、*rmtp* で示される`timespec`構造体は間隔の残り時間量を収納するために更新されます(すなわち、要求時間 - 実際にスリープした時間)。もし*rmtp* がNULLの場合、残り時間は設定されません。*rmtp* の値は絶対時間値のケースでは設定されません。

成功した場合、**clock_nanosleep**は少なくても指定した時間が過ぎた後に0の値を返します。失敗した場合、**clock_nanosleep**は-1の値を返し、`errno`はエラーを示すために設定されます。発生する可能性があるエラーの種類のリストについては**clock_nanosleep(2)**のmanページを参照してください。

7
システム・クロックおよびタイマー

本章ではシステム機能上のシステム時間計測、ローカル・タイマー、ローカル・タイマー無効時の影響について説明します。

システム時間計測

標準Linuxのシステム時間計測は、タイマー・カウントからナノ秒へ変換するためにタイマーとキャリブレーションの値を読み取るルーチンにて構成される独立したアーキテクチャーのドライバを含む“clocksource” メカニズムを使用します。

RedHawkでは、TSCベースのクロックが殆どの時間計測の要求を満たすために使用されます。カーネル・チューニング・パラメータREQUIRE_TSCおよびREQUIRE_RELIABLE_TSC(カーネル構成GUI上の「Processor Type & Features」項目でアクセス可能)は、TSCが構成されていないカーネルの信頼性は損害を与えることで知られている電源管理の側面を保証するためにデフォルトでプレビルト・カーネルの中で有効になっています。

更にTSCはクロックの安定性を向上させるために2番目のクロックソースに統制されます。RCIMがシステム内に存在する時、RCIMは2番目のクロックソースとして使用されます。そうでなければ、HPETまたはPMタイマーが使用されます。

/sys/devices/system/clocksource/clocksource0/current_clocksourceファイルを読み取ると現在の2番目のクロックソースが表示されます。**echo(1)**を使ってこのファイルへ他のクロックソース名称を書き込むと割り当てが変更されます。

ブート・コマンドライン・オプションは、適切なTSC同期のためにBIOSをチェックしTSCが正しく同期しない場合はブートの最後で再同期(tsc_sync=auto [デフォルト])、強制的に再同期(tsc_sync=force)、BIOSをチェックし同期していない場合は実行できるクロックソースとしてTSCを正確に無効(tsc_sync=check)にすることが可能です。ホットプラグCPUはオペレーティング・システムにより再同期させる機会を持っていないことに注意してください。それらのためにTSC同期のチェックだけは利用可能です。

これらの時間計測機能に関して更に理解するために***/kernel-source*/Documentation/hrtimers**内のテキスト・ファイルを参照してください。

ローカル・タイマー

Concurrent社のihawkシステムでは、各CPUがそのCPUへの周期割り込みのソースとして使用されるローカル(プライベート)・タイマーを持っています。1つのCPUだけがローカル・タイマー割り込みを同時に処理するために既定値ではそれらの割り込みは1秒につき100回、正しいテンポでずらして発生させます、

ローカル・タイマー割り込みルーチンは次のローカル・タイミング機能(以降のセクションで詳細に説明します)を実行します。

- **top(1)**および他のユーティリティを使ってCPU使用率の統計データを収集します
- 周期的にタイム・クォンタムを消費するためにCPU上で実行中のプロセスを起こします
- タイム・クォンタムを使い果たした時に実行中のプロセスをCPUから解放し他の実行中のプロセスを優先させます
- 周期的にCPU間で実行可能なプロセスの負荷バランスを保ちます
- プロセスとシステム・プロファイリングを実行します
- この機能が利用可能なプロセスのためのシステムtime-of-dayクロックおよび実行時間のクォータ制限を実装します
- POSIXタイマーのための割り込みソースを提供します
- リード・コピー・アップデート(RCU)処理中に構造体のデータを解放するために各CPUの正状態をポーリングします
- システムtime-of-dayクロックとブート時からのティック回数を更新します
- システム・タイマー・リストのイベント停止を送り出します。これにはウォッチドッグ・タイマー・ドライバや**alarm(2)**のようなプロセス・タイマー機能を含みます

ローカル・タイマーのシールディングは、ローカルCPUへのアフィニティを持つプロセスによって要求されたスケジューリング・イベントへのローカル・タイマーの使用を制限します。ローカル・タイマー・シールディングは非シールドCPUへ重要ではない仕事を移動するプロセス・シールディングと連動します。これは、「リアルタイム性能」章の中で説明したように割り込み応答の最悪のケースとCPU上のプログラム実行のデターミニズムの両方を改善します。しかし、ローカル・タイマーを無効にすることはRedHawk Linuxにより標準的に提供されるいくつかの機能に関して影響を及ぼします。これらの影響は以下で説明します。

機能

ローカル・タイマーは以降のセクションの中で説明する機能を実行します。ローカル・タイマーを無効にする影響は、いくつかの機能のために実行可能な代案についても説明します。

CPUアカウンティング

プロセス毎のCPU利用率は**top(1)**や**ps(1)**のようなユーティリティにより報告されます。これらのユーティリティは**times(2)**, **wait4(2)**, **sigaction(2)**, **acct(2)**のようなシステム・サービスから利用率の統計値を集めます。

標準的な非RedHawk Linuxカーネルにおいて、プロセスのCPU利用を決定するためにこれらのサービスはローカル・タイマーに依存します。一方、RedHawkカーネルはこれを実現するためにローカル・タイマーの代わりに高分解能プロセス・アカウンティング機能を使用します。高分解能プロセス・アカウンティングはローカル・タイマーが無効であっても機能し続けます。

高分解能プロセス・アカウンティングは、カーネル構成GUI上の「General Setup」項目のHRACCTカーネル・チューニング・パラメータを介して全てのプレビルトRedHawk カーネルにて有効です。この機能に関するすべての情報は**hracct(3)**および**hracct(7)**のmanページを参照してください。

プロセス実行時間および制限

ローカル・タイマーはSCHED_OTHERおよびSCHED_RRスケジューリング・ポリシーでスケジュールされたプロセスのクォンタムを満了するために使用されます。これは同じスケジューリング・ポリシーのプロセスがラウンドロビン方式でCPUを共有することを可能にします。もしローカル・タイマーがCPU上で無効である場合、CPU上のプロセスはもはやそれらのクォンタムを満了することはありません。これは、このCPU上で実行中のプロセスはブロックするまで、または高優先度プロセスが実行可能となるまで実行されることを意味します。つまり、ローカル・タイマー割り込みが無効であるCPU上では、SCHED_RRスケジューリング・ポリシーにスケジュールされたプロセスはまるでSCHED_FIFOスケジューリング・ポリシーにスケジュールされたように動作します。ローカル・タイマーが有効のままであるCPU上にスケジュールされたプロセスは影響を受けないことを注意してください。プロセス・スケジューリング・ポリシーに関する詳細な情報については4章の「プロセス・スケジューリング」を参照してください。

setrlimit(2)および**getrlimit(2)**システムコールは、プロセスが消費可能なCPU時間に関する制限をプロセスが設定および取得することを可能にします。この時間が満了となった時、プロセスにSIGXCPUシグナルが送信されます。CPU時間の蓄積はローカル・タイマー割り込みルーチンの中で行われます。従って、もしCPU上のローカル・タイマーが無効である場合、CPU上のプロセスが実行する時間は計上されません。もしこれがプロセスを実行する唯一のCPUである場合、SIGXCPUシグナルを受信することは決してありません。

インターバル・タイマーのデクリメント

setitimer(2)および**getitimer(2)**システムコールはプロセスが個々に“仮想タイマー”のセットアップ、タイマーの値の取得を可能にします。仮想タイマーはプロセスが実行中の時だけデクリメントされます。仮想タイマーには、ユーザー・レベルでプロセスが実行している時だけデクリメントするものとユーザー・レベルとカーネル・レベルのどちらでもプロセスが実行している時にデクリメントするものの2種類が存在します。仮想タイマーが満了する時、シグナルがプロセスへ送信されます。仮想タイマーのデクリメントはローカル・タイマー・ルーチンで行われます。従って、ローカル・タイマーがCPU上で無効である時、使用時間が仮想タイマーからデクリメントされることはありません。もしこれがプロセスを実行している唯一のCPUである場合、その仮想タイマーは決して満了となりません。

システム・プロファイリング

ローカル・タイマーはシステム・プロファイリングを操作します。プロファイラーが記録するサンプルはローカル・タイマー割り込みの発生により始動します。もしローカル・タイマーが任意のCPU上で無効である場合、**gprof(1)**コマンドと**profil(2)**システム・サービスはそのCPU上で動作するプロセスに対して正しく機能しません。

CPU負荷バランシング

ローカル・タイマー割り込みルーチンは、このCPU上で実行可能なプロセスの数がシステム内の他のCPU上で実行可能なプロセスよりも極めて少なくないことを確認するために周期的にロード・バランサーを呼びます。このような場合、ロード・バランサーは全てのCPU間の負荷のバランスをとるために他のCPUからプロセスを移動します。ローカル・タイマー割り込みが無効になっているCPUで、実行するプロセスがCPUにない時にロード・バランサーは呼ばれます。シールドCPU上でバックグラウンド・プロセスが実行することは通常望ましいことではないため、この機能の損失はシールドCPUにおいて通常は問題ではありません。

CPU再スケジューリング

resched_cntl(2)システムコールのRESCHED_SET_LIMIT機能は、再スケジューリング変数がロックされた状態を維持可能な時間の上限を設定することが可能です。制限時間を超えたときにSIGABRTシグナルがプロセスへ送信されます。この機能はアプリケーション開発中に問題をデバッグするために提供されます。再スケジューリング変数がロックされたプロセスがローカル・タイマーが無効のCPU上で動作する時、制限時間はデクリメントされず、その結果プロセスが指定された制限時間をオーバーランした時にシグナルは送信されない可能性があります。

POSIXタイマー

ローカル・タイマーはPOSIXタイマーのためのタイミング・ソースを提供します。もしCPUがローカル・タイマー割り込みからシールドされた場合、そのCPU上のプロセスがPOSIXタイマーまたは**nanosleep(2)**機能が動作中の場合にローカル・タイマー割り込みはシールドCPU上で発生し続けます。もしプロセスがシールドCPU上で実行することが許可されていない場合、このタイマーはプロセスが動作可能なCPUへ移動されます。

RCU処理

カーネルのリード・コピー・アップデート(RCU)・コードは、伝統的にデータ構造体を解放するために各CPU上で静止状態をポーリングするためにローカル・タイマーに頼っています。CPUがローカル・タイマー割り込みからシールドされている時、そのCPUは必要とするRCU処理を実行することができません。同期メカニズムは任意のポイントでRCU処理を起動し、RCU処理へのローカル・タイマーの関与を除いてタイマー駆動型ポーリングを待つことなく完了します。RCU_ALTERNATIVEパラメータが全てのプレビルト・カーネルでデフォルトのSHIELDパラメータと関連して設定された時にこの同期が発生します。RCU_ALTERNATIVEがカーネルに設定されていない時、RCUコードはローカル・タイマーを使用します。

その他

上述の機能に加えて、ローカル・タイマーが無効である時、標準Linuxのコマンドやユーティリティが提供する一部の機能はCPU上で正しく機能しない可能性があります。これらは以下を含みます：

bash(1)
sh(1)
strace(1)

これらのコマンドやユーティリティの詳細な情報については、対応するmanページを参照してください

ローカル・タイマーの禁止

ローカル・タイマーをシールドすることにより、ローカル・タイマーはどのようなCPUの組み合わせに対しても無効となります。シールディングは**shield(1)**コマンドを介して、または**/proc/shield/ltmrs**への16進数値を割り当てることにより行われます。この16進数値はCPUのビット・マスクで、各々のビットのポジションが1つのCPUを特定し、そのビットの値はそのCPUのローカル・タイマーが無効(**=1**)なのか有効(**=0**)なのかを特定します。詳細な情報については2章の「リアルタイム性能」と**shield(1)**のmanページを参照してください。

8
ファイル・システムとディスクI/O

本章では**xfs**ジャーナリング・ファイル・システムおよびRedHawk Linuxオペレーティング・システム上でのダイレクト・ディスクI/Oの実行手順について説明します。

ジャーナリング・ファイル・システム

従来のファイル・システムは障害の後にファイル・システムの大きさ次第で完了までに多くの時間を必要とする特殊なファイル・システム・チェックを実行する必要があります。ジャーナリング・ファイル・システムは「*ジャーナル*」と呼ばれる特殊なログ・ファイルを保存することによりデータ完全性を確保する障害回復可能なファイル・システムです。ファイルが更新された時、ファイルのメタデータは本来のディスク・ブロックを更新する前にディスク上のジャーナルへ書き込まれます。もしジャーナル・エントリーがコミットされる前にシステム・クラッシュが発生した場合、オリジナル・データはまだディスク上にあり、新しく変更したものだけが失われます。もしディスク更新中にクラッシュが発生した場合、ジャーナル・エントリは発生したと考えられることを示します。再起動時にジャーナル・エントリは再生されて中断された更新は完了します。これはファイル・システム・チェックの複雑さを大幅にカットし、回復時間を削減します。

SGIからのXFSジャーナリング・ファイル・システムのサポートは、RedHawk Linuxではデフォルトで有効となっています。XFSはマルチスレッド化され、100万テラバイト程の大きさのファイルが取り扱い可能な64bitファイル・システムです。大容量ファイルおよび大容量ファイル・システムに加えて、XFSがサポート可能な拡張属性、可変ブロックサイズは、容量をベースにして性能と拡張性の両方を補助するためにBtree(ディレクトリ、大きさ、空き容量)を広範囲に使用します。ユーザー割り当てよびグループ割り当ての両方がサポートされます。

ジャーナリングの構造とアルゴリズムは、ジャーナリングのパフォーマンスへの影響を最小限にして迅速にデータ・トランザクションの読み書きを記録します。XFSはほぼRAW I/O性能を提供することが可能です。

拡張属性はファイルに関連付けられた名前と値のペアとなります。属性は普通のファイル、ディレクトリ、シンボリック・リンク、デバイス・ノード、他のiノードの型全てに付随させることが可能です。属性値は最大64KBの任意のバイナリ・データを含めることが可能です。通常のファイルのアクセス権により保護されている全てのユーザーが利用可能なユーザー名前空間、および特権のあるユーザーだけがアクセス可能なシステム名前空間の2つの属性の名前空間が利用可能です。システム名前空間はアクセス制御リスト(ACLs：Access Control Lists)や階層ストレージ管理(HSM：Hierarchical Storage Manage)ファイルの移動状況のような保護されたファイル・システムのメタデータに使用することが可能です。

NFSバージョン3は、そのプロトコルをサポートする他のシステムへ64bitファイル・システムにエクスポートするために使用することが可能です。NFS V2システムはプロトコルにより強いられる32bitの制限があります。

ローカルおよびリモートのSCSIテープまたはファイルへのXFSファイル・システムのバックアップとリストアは、**xfsdump**と**xfsrestore**の使用で行えます。拡張属性と割り当て情報のダンプがサポートされています。

データ管理API(DMAPI/XDSM) は、ディスクへのRAWアクセスやファイル・システム構造の知識の必要なしに高性能ダンプ・プログラムと同様に階層ストレージに管理ソフトウェアの実行を可能にします。

ツールのフルセットはXFSを提供します。XFSファイル・システムのための多くの文書は以下で見つけることが可能です：

http://oss.sgi.com/projects/xfs/

XFSファイル・システムの作成

XFSファイル・システムを作成するため、以下が必須となります：

- XFSファイル・システムを作成するパーティションを確認します。これは新しいディスク、パーティションで区切られていない既存のディスク、既存のパーティションの上書きで可能です。新しいパーティションを作成する場合は**fdisk(1)**のmanページを参照してください。
- パーティション上にXFSファイル・システムを作成するために**mkfs.xfs(8)**を使用します。もしターゲット・ディスクのパーティションが現在ファイル・システムでフォーマットされている場合、**–f** (強制)オプションを使用してください。

  mkfs.xfs [-f] /dev/*devfile*

  devfile はファイル・システムを作成したいパーティションの場所(例：**sdb3**)。これはパーティション上の現在のあらゆるデータを破壊しますので注意してください。

XFSファイル・システムのマウント

XFSファイル・システムをマウントするために**mount(8)**コマンドを使用します：

mount -t xfs /dev/*devfile /mountpoint*

XFSファイル・システムをマウントする時に利用可能なオプションは**mount(8)**のmanページを参照してください。

XFSはジャーナリング・ファイル・システムであるため、ファイル・システムをマウントする前に未完了のトランザクションのためにファイル・システムトランザクション・ログをチェックし、最新のファイル・システムにします。

データ管理API (DMAPI)

DMAPI(Data Management API)はカーネルと階層ストレージ管理システム(HSM)との間のファイル管理要求を渡すためのXFSファイル・システム内のメカニズムです。

DMAPIを構築するため、リビルドの一部としてカーネル構成GUI上の「File Systems」項目のXFS_DMAPIシステム・パラメータを設定します。

DMAPI構築に関する詳細な情報は、以下を参照してください：

http://oss.sgi.com/projects/xfs/dmapi.html

ダイレクト・ディスクI/O

普通は、ファイルの読み書きはファイル・システム・キャッシュ・バッファを通り抜けます。データベース・プログラムのようないくつかのアプリケーションはそれら自身がキャッシングすることが必要となる可能性があります。ダイレクトI/Oはデータのカーネルのバッファリングを回避するバッファがないI/O方式です。ダイレクトI/Oは、ファイル・システムがディスクとユーザー提供のバッファとの間で直接データを転送します。

RedHawk Linux はその仮想アドレス空間へディスクからの直接読み取り、ディスクへの直接書き込みの両方がユーザー・プロセスで有効で、中間オペレーティング・システムのバッファリングを回避し、ディスクI/O速度を向上します。ダイレクト・ディスクI/Oは転送データのコピーを排除することによりシステムのオーバ・ヘッドもまた減らします。

ダイレクトI/O用にディスク・ファイルを設定するために**open(2)**または**fcntl(2)**システムコールを使用します。以下の手順のいずれかを使用します：

- ディスク・ファイルの名称パスを指定、*arg* 引数の中にO_DIRECTビットを設定してプログラムからの**open**システムコールを呼び出します。
- 開いているファイルに対して開いているファイル記述子を指定、F_SETFLコマンを指定、*arg* 引数の中にO_DIRECTビットを設定して**fcntl**システムコールを呼び出します。

ダイレクト・ディスクI/O転送は以下の要求の全てを満足する必要があります：

- ユーザー・バッファは_PC_REC_XFER_ALIGN **pathconf(2)**変数の整数倍のバイト・バウンダリに整列されている必要があります。
- 現在のファイル・ポインタの設定が次のI/O操作を開始するファイル内のオフセットに位置します。この設定は_PC_REC_XFER_ALIGN **pathconf(2)**変数が返す値の整数倍である必要があります。
- I/O操作で転送されるバイト数は_PC_REC_XFER_ALIGN **pathconf(2)**変数が返す値の整数倍である必要がります。

ダイレクトI/Oをサポートしていないファイル・システム上のファイルに対してダイレクトI/Oを有効にするとエラーを返します。ファイル・システム固有の**soft**オプションでマウントしたファイル・システム内のファイルをダイレクト・ディスクI/Oを有効にしようとするとエラーを引き起こします。**soft**オプションはファイル・システムがアンマウントする直前までキャッシュから物理ディスクへデータを書き込む必要がないことを指定します。

推奨はしませんが、両方のモードの性能を犠牲にしてダイレクト・モードとキャッシュ(ノンダイレクト)・モードの両方で同時にファイルを開くことが可能です。

ダイレクトI/Oの使用する場合、システム障害後にファイルが復旧可能であることを保証しません。そうするためにはPOSIX同期I/Oフラグを設定する必要があります。

プロセスが**mmap(2)**システムコールでファイルの一部を現在マッピングしている場合はダイレクト・モードでファイルを開くことはできません。同様に呼び出しで使われているファイル記述子がダイレクト・モードで開かれている場合、**mmap**の呼び出しは失敗します。

ダイレクトI/Oがより良いI/Oスループットをタスクに提供するかどうかは、アプリケーションに依存します：

- 全てのダイレクトI/O要求はどうきしているため、アプリケーションによるI/Oと処理は重複できません。
- オペレーティング・システムはダイレクトI/Oをキャッシングできないため、read-ahead(先読み)またはwritebehind(分散書き込み)のアルゴリズムのスループットは向上しません。

しかしながら、他のデータのコピーがなくデータが直接ユーザー・メモリからデバイスへ移動するため、ダイレクトI/Oはシステム全体のオーバーヘッドを減らします。システム・オーバーヘッドの削減は、同じプロセッサー・ボード上の内蔵型SCSIディスク・コントローラとローカル・メモリ間のダイレクト・ディスクI/Oを行うときに特に顕著です。

9
メモリ・マッピング

本章ではプロセスが他のプロセスのアドレス空間の内容をアクセスするためにRedHawk Linuxが提供する方法について説明します。

ターゲット・プロセスのアドレス空間へのマッピングの確立

各実行中のプロセスにおいて、**/proc**ファイル・システムはプロセスのアドレス空間を表すファイルを提供します。このファイルの名称は**/proc/***pid***/mem**で、*pid* はアドレス空間が表されているプロセスのIDを意味します。プロセスは**open(2)**で**/proc/***pid***/mem**ファイルを開き、他のプロセスのアドレス空間の内容を読むためおよび変更するために**read(2)**および**write(2)**システムコールを使うことが可能です。

libccur_rtライブラリに備わっている**usermap(3)**ライブラリ・ルーチンは、簡単なCPUの読み書きを利用して現在実行中のプログラムの場所を効率的に監視および修正する方法をアプリケーションに提供します。

このルーチンのための基本的なカーネル・サポートは、プロセスが自分自身のアドレス空間に他のプロセスのアドレス空間の一部のマッピングを許可する**/proc**ファイル・システムの**mmap(2)**システム・サービスコールです。従って、他の実行中のプログラムの監視と修正は、**/proc**ファイル・システムの**read(2)**および**write(2)**呼び出しによるオーバーヘッドを負うことなく、アプリケーション自身のアドレス空間内での簡単なCPUの読み書きになります。

以降のセクションでこれらのインターフェースの説明およびアプリケーション内で**mmap(2)**または**usermap(3)**を使うかどうかを決定する時に考慮すべき事項を紹介します。

mmap(2)の利用

プロセスは**/proc/***pid***/mem**ファイルのアドレス空間の一部をマッピングするために**mmap(2)**を使用することが可能であり、このようにして他のプロセスのアドレス空間の内容を直接アクセスします。**/proc/***pid***/mem**ファイルへのマッピングを確立したプロセスを以下モニタリング・プロセスと呼びます。アドレス空間をマッピングされたプロセスをターゲット・プロセスと呼びます。

/proc/*pid***/mem**ファイルへのマッピングを確立するため、以下の条件を満足する必要があります：

- ファイルは少なくても読み取り権限で開かれている必要があります。もしターゲット・プロセスのアドレス空間を修正するつもりならば、ファイルは書き込み権限で開かれている必要があります。
- マッピングを確立するための**mmap**の呼び出しに関して、*flags* 引数はMAP_SHAREDを指定する必要があり、それ故にターゲット・プロセスのアドレス空間の読み書きはターゲット・プロセスとモニタリング・プロセスとの間で共有されます。
- ターゲットのマッピングはHUGETLB領域内ではない実際のメモリ・ページとする必要があります。現在の実装ではHUGETLB領域へのマッピング作成はサポートしていません。

モニタリング・プロセスで生じる**mmap**マッピングは、現在のレンジ内[*offset, offset + length*) にマッピングされたターゲット・プロセスの物理メモリになることに注意することが重要です。結果、**mmap**呼び出しがされた後にターゲットのマッピングが変更された場合、ターゲット・プロセスのアドレス空間へのモニタリング・プロセスのマッピングは無効となる可能性があります。このような状況では、モニタリング・プロセスは物理ページ下へのマッピングを保持しますが、マッピングはターゲット・プロセスとはもはや共有されていません。何故ならモニタリング・プロセスはマッピングが有効ではないことを検知できないため、モニタリング・プロセスとターゲット・プロセス間の関係を制御するためのアプリケーションを準備する必要があります(表記[*start, end*]は、*start* から*end* への区間(*start* を含み*end* を含まない)を意味します)。

ターゲット・プロセスのアドレス空間へのモニタリング・プロセスのマッピングが無効になる状況は以下のとおり：

- ターゲット・プロセスが終了。
- ターゲット・プロセスが**munmap(2)**または**mremap(2)**のどちらかでレンジ内[*offset, offset + length*) のページをアンマップ。
- ターゲット・プロセスが**mmap(2)**で異なるオブジェクトへレンジ内[*offset, offset + length*) のページにマッピング。
- ターゲット・プロセスが**fork(2)**を呼び出し、子プロセスがする前にレンジ内[*offset, offset + length*) のアンロック済み、プライベート、書き込み可能なページへ書き込む。このケースでは、ターゲット・プロセスはページのプライベート・コピーを受け入れ、そのマッピングと書き込み操作はコピーされたページへリダイレクトされる。モニタリング・プロセスはオリジナル・ページへのマッピングを保持。
- ターゲット・プロセスが**fork(2)**を呼び出してから、子プロセスと共有し続けているレンジ内[*offset, offset + length*) のプライベート、書き込み可能な(copy-on-writeにマークされた)ページをメモリにロック。このケースでは、ロック操作を実行したプロセスは(ページに最初の書き込みを実行したかのように)ページのプライベート・コピーを受け入れる。もしこれがページをロックするターゲット(親)・プロセスの場合、モニタリング・プロセスのマッピングはもはや有効ではない。
- ターゲット・プロセスが子プロセスと共有し続けているレンジ内[*offset, offset + length*) のロック済み、プライベート、読み取り専用の(copy-on-writeにマークされた)ページの書き込み権限を有効にするために**mprotect(2)**を呼び出す。このケースでは、ターゲット・プロセスはページのコピーを受け取る。モニタリング・プロセスはオリジナルのメモリ・オブジェクトへのマッピングを保持。

もしアプリケーションがモニタリング・プロセスのアドレス空間のマッピングの対象になることを要求されている場合、以下を推奨します：

- ターゲット・プロセスのアドレス空間がモニタリング・プロセスにマッピングされる前にターゲット・プロセスにてメモリ・ロック操作を実行
- **fork(2)**を呼び出す前に親プロセスやモニタリング・プロセスによるマッピングが保持される必要のあるあらゆるページをメモリにロック

もしアプリケーションがアドレス空間のマッピングの対象になることを要求されていない場合、**fork**を呼び出した後までメモリ内のページのロックを延期することも可能です。

詳細な情報については**mmap(2)**のmanページを参照してください。

usermap(3)の利用

/procファイル・システムの**mmap(2)**システムサービスコールのサポートに加え、RedHawk Linux はモニタリング・プロセスの仮想アドレス空間の中へターゲット・プロセスのアドレス空間の一部をマッピングするための代替え方法として**usermap(3)**ライブラリ・ルーチンも提供します。このルーチンは**libccur_rt**ライブラリの中に備わっています。

usermapライブラリ・ルーチンはターゲット・アドレス空間のマッピングを作成するための**/proc mmap**システムサービスコール・インターフェースを基本に内部的に使用する一方、**usermap**は以下の特別な機能を提供します：

- 呼び出し元プロセスは仮想アドレスとターゲット・プロセスのアドレス空間内の当該仮想空間の長さを指定する必要があります。**usermap**ルーチンは、**mmap**の呼び出しの前にこの要求の変換内容を整列した開始アドレスのページとページ・サイズの倍数の長さに処理します。
- **usermap**ルーチンは複数のターゲット・プロセスのデータ項目をマッピングするために使用されることを目的としており、従ってこれは重複する**mmap**マッピングの作成を回避するために書かれました。**usermap**は既存の全てのマッピングに関する**mmap**情報を内部的に保持し、要求されたデータ項目のマッピングが既に存在するマッピングのレンジ内に収まる時、重複する新しいマッピングを作成する代わりにこの既存のマッピングを再利用します。
- **mmap**を呼び出す時、既に開かれているファイル記述子を提供する必要があります。適切なタイミングでターゲット・プロセスのファイル記述子を開くおよび閉じることは義務となります。

  usermapを使用する時、呼び出し元プロセスはターゲット・プロセスのプロセスID (`pid_t`)を指定する必要があります。**usermap**ルーチンは**/proc/***pid***/mem**ファイルを正確に開く処理をします。同じターゲット・プロセスIDに対して更なる**usermap(3)**の呼び出しは、この**/proc**ファイル記述子を再度開く必要がないため、このファイル記述子は開いた状態にしておきます。

  ファイル記述子を開いたままにしておくことは全ての場合において適切ではない可能性が或ことに注意してください。しかしながら、明示的にファイル記述子を閉じて"*len*"パラメータの値が0でルーチンを呼び出すことにより**usermap**が使用している内部マッピング情報をフラッシュすることが可能です。呼び出し元プロセスが**usermap**に組み込まれている最適化機能を続いて利用する可能性があるため、モニタリング・プロセスは全てのターゲット・マッピングが作成された後にのみこのclose-and-flush 機能を使うことを推奨します。詳細な情報については**usermap(3)**のmanページを参照してください。

usermapライブラリ・ルーチンもまた同じ**/proc/***pid***/mem mmap(2)**システムコール・サポートを基に内部的に使用するため、もはや有効ではないモニタリング・プロセスのマッピングに関して「mmap(2)の利用」で説明した同じ制限を**usermap**マッピングにも適用されることに注意してください。

usermap(3)ルーチンの使用に関する詳細な情報については**usermap(3)**のmanページを参照してください。

検討事項

前述した**usermap**機能に加えて、アプリケーションの中で**usermap(3)**ライブラリ・ルーチンもしくは**mmap(2)**システム・サービスコールを使用するのかどうかを決定する時に以下の残りのポイントもまた検討することを推奨します：

- **/proc/***pid***/mem**ファイルへのマッピングを確立するために使用する機能はConcurrent RedHawk Linuxの拡張ですが、**mmap(2)**システムコールは標準的なSystem Vです。**usermap(3)**ルーチンは完全にConcurrent RedHawk Linuxの拡張です。
- **mmap(2)**はモニタリング・プロセス内のページ保護とマッピングの位置の直接制御を提供します。**usermap(3)**はそうではありません。

カーネル構成パラメータ

/procファイル・システム**mmap(2)**コールの動作に直接影響を与えるConcurrent RedHawk Linuxカーネル構成パラメータが2つ存在します。**usermap(3)**もまた**/proc**ファイル・システム**mmap(2)**サポートを使用するため、**usermap(3)**はこれらの構成パラメータに同様に影響を受けます。

カーネル構成パラメータは、カーネル構成GUI上の「Pseudo File Systems」項目でアクセス可能です：

PROCMEM_MMAP　もしこのカーネル構成パラメータが有効である場合、**/proc**ファイル・システム**mmap(2)**サポートがカーネルに組み込まれます。

もしこのカーネル構成パラメータが無効である場合、**/proc**ファイル・システム**mmap(2)**サポートはカーネルに組み込まれません。このケースで、**usermap(3)**と**/proc mmap(2)**の呼び出しは`errno`の値がENODEVで返されます。

このカーネル構成パラメータは、全てのConcurrent RedHawk Linuxのカーネル構成ファイルにおいて既定値で有効になっています。

PROCMEM_ANYONE　もしこのカーネル構成パラメータが有効である場合、モニタリング・プロセスが読み取りまたは読み書きによる**open(2)**が成功するどの**/proc/***pid***/mem**ファイルも**/proc mmap(2)**または**usermap(3)**の呼び出しのためにターゲット・プロセスとして使用される可能性があります。

もしカーネル構成パラメータが無効である場合、モニタリング・プロセスにより現在ptraceを実行されているターゲット・プロセスで**/proc mmap(2)**または**usermap(3)**を使用することが可能です。更にptraceを実行されたターゲット・プロセスは**/proc mmap(2)**システム・サービスコールが行われた時点で停止した状態である必要があります(他のプロセスへのptrace実行に関する詳細な情報については**ptrace(2)**のmanページを参照してください)。

このカーネル構成パラメータは、全てのConcurrent RedHawk Linuxのカーネル構成ファイルにおいて既定値で有効になっています。

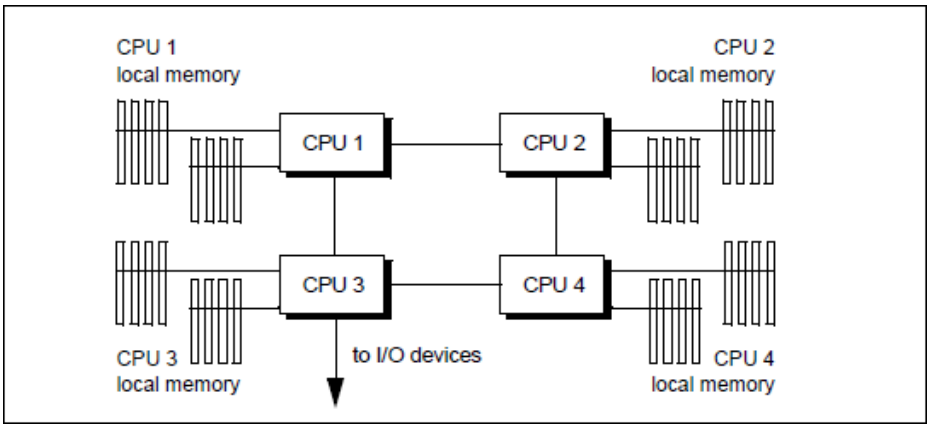
10
Non-Uniform Memory Access (NUMA)

AMD Opteronシステムおよび(Nehalem, Sandy Bridge, Ivy Bridge等を含む)最新のIntelシステムで利用可能なNUMAサポートは、プログラムのページに割り当てられることになるメモリ場所に影響を及ぼす可能性があります。

概要

不均等メモリアクセス(NUMA: Non-Uniform Memory Access)を持つシステムにおいて、他よりも一部のメモリ領域へのアクセスに時間がかかります。AMD Opteron (または最新のIntel)マルチプロセッサ・システムはNUMAアーキテクチャです。これは各CPUチップがそれ自身のメモリ・リソースと一体となっているからです。CPUとそれに対応するメモリはユニークな物理バス上に置かれています。CPUはそのローカル・メモリ・バス上にあるメモリ領域へは速くアクセスすることが可能ですが、他のCPUはローカルではないCPUのメモリへアクセスするために1つ以上の余計な物理バス接続を横断する必要があります。CPUーバス間の関係を図10-1に示します。

図10-1 NUMAシステム上のCPU/Busの関係



AMD Opteron (または最新のIntel)システム上のメモリへアクセスする時間は、プログラムが実行されるCPUとプログラムのページが割り当てられるメモリ領域に依存することになることを意味します。

NUMAノードは、1つのメモリ領域とNUMAノードのメモリ領域として同じ物理バス上に存在する全てのCPUとすることを定義します。システムのブート中にカーネルはNUMAメモリーCPUレイアウトを決定し、CPUとNUMAノードの関連を定義する仕組みを作成します。現在のNUMAシステム上では、メモリ領域に存在する物理バスは1つのCPUにのみ直接接続されています。

最適な性能を得るため、プログラムに利用されているメモリ・ページのローカルCPU上でプログラムは実行される必要があります。本章内で説明されるNUMAインターフェースは、プログラムがプログラムのページが割り当てられる場所からノードを指定すること、および、リアルタイム・アプリケーション用にリモート・メモリへのアクセス量を減らすためにユーザー・ページをシールドされたノードのメモリへ(から)移動するためNUMAノードをシールドすることが可能です。

プロセスのCPUアフィニティを設定するためのメカニズムと組み合わせた時、これらのインターフェースはプログラムが極めてデターミニスティックなメモリ・アクセス時間を獲得することを可能にします。

NUMAサポートはAMD Opteronおよび最新のIntelプロセッサのiHawkシステム上だけで利用可能です。ローカルに少しのメモリも備えていない一部のCPUのためにNUMAシステムを構成することが可能です。メモリを持たないCPUのような状況では、メモリ・リソースなしのNUMAノード(32bitモード)を割り当てる、もしくはメモリ付きNUMAノード(64bitモード)に人工的に割り当てられます。どちらのケースでも、全てのCPUからのメモリ・アクセスはリモート・メモリ・アクセスになります。これはローカル・メモリなしのCPU上で実行中のプロセスのメモリ性能だけでなくリモート・アクセス要求が発生しているNUMAノード上で実行中のこれらのプロセスに影響を与えます。

これはデターミニスティックなプログラム実行のための最適な構成ではありません。構成の詳細については本章で後述する「構成」セクションを参照してください。メモリ性能の最適化やデターミニスティックなメモリ・アクセス時間を得るための方法に関する詳細な情報については「性能ガイドライン」セクションを参照してください。デターミニスティックなメモリ・アクセスはデターミニスティックなプログラム実行時間を得るためにも重要であることに注意してください。

メモリ・ポリシー

NUMAサポートはメモリ・ポリシーの概念を実装しています。これらのメモリ・ポリシーはユーザー単位タスクを基準にしてタスク全体に適用されます。任意のタスク内の仮想アドレス空間の範囲もまたそれら自身にそれらのページに対しタスク全体メモリ・ポリシーを優先する個別のメモリ・ポリシーを所有する可能性があります。タスク全体および仮想アドレス空間の両方のメモリ・ポリシーはfork/clone操作中の子タスクに継承されます。

NUMAメモリ・ポリシーは以下のとおり：

MPOL_DEFAULT　これは、メモリが利用可能である場合、メモリ・ページはローカル・メモリから現在のCPUへ割り当てられたところがデフォルトになります。これはタスクまたはこの子タスクが特定のメモリ・ポリシーを割り当てなかった時に使用されるポリシーです。タスク全体メモリ・ポリシーとして、もしくは異なるタスク全体メモリ・ポリシーが設定されているタスク内の仮想メモリ空間のために明示的にMPOL_DEFAULTポリシーを設定することが可能です。

MPOL_BIND　これは、このメモリ・ポリシーが設定される時点でノードマスクに指定されたノードのみにメモリ割り当てを制限する厳格なポリシーです。ページは指定されたノードからのみ割り当てられ、ページ割り当てはバインドしたノードマスクではないほかのノードでメモリが利用可能であっても失敗する可能性があります。この種のページ割り当て失敗が発生する時、プロセスおよび同じアドレス空間を共有するこの子プロセス全てとスレッド全てはカーネルのSIGKILLシグナルにより終了します。このポリシーはどのノードからページが割り当てられるかに関しては他のメモリ・ポリシーよりもより多くの確実性を提供します。

留意すべきは、プロセスのローカル・メモリになるのために将来のメモリ割り当て全てを保証する唯一の方法は、シングルCPUまたは同じNUMAノード内に存在する全てのCPUセットへCPUアフィニティとMPOL_BINDポリシーの両方を設定することです。

MPOL_PREFERRED　このポリシーは割り当てのために優先される(単一の)ノードを設定します。カーネルは最初にこのノードからページを割り当てようとし、優先されるノードがメモリ不足の時は他のノードを使用します。

MPOL_INTERLEAVE　このポリシーはノードマスクに指定されたノードへの割り当てを(ラウンド・ロビン方式で)交互に行います。これは遅延の代わりに処理能力を最適化します。効果的にするためには、メモリ領域を相当大きくする必要があります。

ユーザー空間ページ割り当てに加えて、カーネル・メモリ割り当て要求の多くもまた現在実行中タスクのタスク全体メモリ・ポリシーにより決定されます。しかし、全てのカーネル・ページ割り当てが現在のタスクのメモリ・ポリシーに制御されているわけではありません。例えば、DMA目的のためにメモリを割り当てる殆どのデバイス・ドライバは、デバイスのI/Oに存在するノード、もしくはそのI/Oバスに最も近いノードからメモリを代わりに割り当てます。

既に行われたページ割り当てはタスクのメモリ・ポリシーの変更に影響されません。例えば、2つのCPUを搭載したシステムにおいてCPUとノードが1対1に対応しているものと仮定します：

> タスクがCPUアフィニティが0x1かつメモリ・ポリシーがMPOL_DEFAULTでCPU0上でしばらくの間実行している状況で、その後、そのCPUアフィニティが0x2、メモリ・ポリシーがノードマスク値が0x2のMPOL_BINDへ変更すると、大抵は一旦そのタスクがCPU1上で実行を開始したらタスクに対して非ローカルにあるそのアドレス空間内のページとなります。

以下のセクションでNUMA管理のための利用可能なシステム・サービス、ライブラリ機能、ユーティリティについて説明します。

NUMAユーザー・インターフェース

shield(1)コマンドはNUMAノード・メモリ・シールディングの制御および問合せに使用することが可能です。**run(1)**コマンドは実行時にタスクのメモリ・ポリシーを固定するもしくは変更するため、指定したプロセスもしくはスレッドの各NUMAノード内ページのユーザー・ページ数を表示するために使用することが可能です。**shmconfig(1)**は共有メモリ領域のために使用することが可能です。

ライブラリ機能、システム・サービス、他のユーティリティやファイルもまたNUMA制御に利用可能です。

このサポートの詳細は以降のセクションで提供します。

メモリ・シールドされたノード

shield(1)コマンドはメモリ・シールドされたNUMAノードを作成するために使用することが可能です。

NUMAノードのメモリがシールドされた時、シールドされたノード上で実行するために割り付けられていないアプリケーションに属するユーザー・ページはシールドされたノードのメモリの外へ移動されるためにリモート・メモリ・アクセス量は減ります。同様にシールドされたノードに割り付けられたアプリケーションに属するユーザー・ページはシールドされたノードのメモリ内へ移動されます。NUMAノードが最初にメモリ・シールドされた時、タスクのCPUアフィニティが修正される度に、現在1つ以上のメモリ・シールドされたNUMAノードが構成されたシステムでこの「*ページ移動*」は自動的に実行されます。メモリ・シールディングに関する詳細な情報は、**memory_shielding(7)**のmanページを参照してください。

shieldのための以下のオプションは、メモリ・シールディング・サポートの有効、無効、問合せをするために使用されます：

--mem=*MEMSHIELD*, **-m** *MEMSHIELD*

MEMSHIELD メモリ・シールディング・サポートの無効、有効、問合せをするためにそれぞれ**0**, **1, q**のいずれかを指定します。

オプションなしもしくは**-c**オプション付きのマルチ・ノードNUMAシステムで使用される**shield**は、メモリ・シールドされているCPUを表示します。**cpu(1)**コマンドもまたメモリ・シールドされたCPUを表示します。

NUMAノードをメモリ・シールドするためには2つの条件があります：

- メモリ・シールディング・サポートを**shield -m1**コマンドにより有効にする必要があります。
- 同じNUMAノード上に存在する全てのCPUは**shield -p**もしくは**shield -a**のどちらかによりプロセス・シールドする必要があります。**run(1)**コマンドの**-M***c* オプションはシステムの各NUMAノード上のCPUを見るために使用することが可能です。

これらの2つのステップは一緒にもしくは個別に**shield**を呼び出して使用することが可能です。詳細は**shield(1)**のmanページを参照してください。

最高のパフォーマンスのために以下の手順に従うことを推奨します：

- 最初にメモリ・シールドされたNUMAノードを作成し、その後、
- そのノード上でリアルタイム・アプリケーションを実行

以下の例は4CPU-Dual Coreシステム上での正しい手順を示します：

```
shield -m1 -p 2,3
run -b 2,3 rt-app &
```

共有された(システム・ライブラリ・テキストや読み取り専用データ・ページのような)読み取り専用ページがマッピングされ多くのタスクからアクセスすることが可能であるため、これらのページがリクエスト中のCPUもしくはプロセスが存在するノードとは異なるノード内に存在する場合、これらのページはローカルNUMAノードのメモリに複製(同じアイデンティティを保持している間はそれらの内容をコピー)されます。これはなお一層システムのリモート・メモリ・アクセス回数を減らします。

numapgs(1)と**/proc/***pid***/numa_maps**は現在プロセスの複製されたページを(存在するときに)見るために使用することが可能です。

このサポートを常にアクティブとするため、または手動で起動するためにカーネルへ組み込むことが可能です。詳細はページ10-19の「構成」を参照してください。ページ複製の実行方法に関する更に詳細は、**page_replication(7)**のmanページを参照してください。

メモリ・シールドとプリアロケート・グラフィック・ページ

プリアロケート・グラフィック・ページ・サポートの概要については、付録Fの「グラフィクス割り込み」セクションを参照してください。

NVIDIAグラフィックカード付きNUMAシステム上では、シールディング構成のシステム・メモリの一部としてプリアロケート・グラフィック・ページを特定のNUMAノードに任意に設定することが可能です。ノードがメモリ・シールドされた時にプリアロケート・グラフィック・ページは非メモリ・シールド・ノードに自動的に再割り当てされないことに注意してください。

プリアロケート・グラフィック・ページは、4GB境界以下のアドレスに置かれたメモリを持つ全てのNUMAノード間でインターリーブ方式により最初に割り当てられます。XやXorgのようなグラフィック・アプリケーションはそれらのアドレス空間内にグラフィック・ページをマッピング、従って、通常はシステム内の様々なNUMAノード間に広がったグラフィックのマッピングを持ちます。これらのマッピングはグラフィック・アプリケーションが実行しているときにI/Oのためにロック・ダウンされるため、これらのページはカーネルのメモリ・シールド・サポートにアンマップまたはフリーされることはなく、従ってこれらのマッピングのどのような自動によるページ移動でも保護されます。

メモリ・シールドNUMAノード構成の一部として特定のNUMAノード・セットへプリアロケート・グラフィック・ページを任意にセットするため、以下の手順を取ることができます：

1. 全てのグラフィック機能(X/Xorg)を停止。システムをX機能のない最も大きなinitの状態3(ランレベル3)へ移行します。
2. コマンドにより1つ以上のNUMAノードをメモリ・シールド。
3. **graphics-memory**ユーティリティを使って全てのグラフィック・ページを解放します。詳細については**graphics-memory(1)**のmanページを参照してください。
4. グラフィック・ページを割り当てるNUMAノードに属する少なくても1つのCPUを含むインターリーブ・メモリ・ポリシーでシェル(**bash**, **ksh**, 等)を作成します。
5. **graphics-memory**ユーティリティを使って希望するノードにグラフィック・ページを再割り当てします。
6. init 5(ランレベル5)へ戻る、もしくは希望するX機能を再起動します。

例

以下の例は、4つのNUMAノード、16CPU-Quad Coreのシステムの最初のノードにメモリ・シールド・ノードを作成します。この例のグラフィック・ページは2つの非メモリ・シールドNUMAノード間(ノード1と2)に広がっています。この例では、ノード3のメモリは4GB境界よりも上に位置しており、従って、ノード3の中にプリアロケート・グラフィック・ページは存在しないことに注意してください。

1. 全てのグラフィック機能を停止します。全てのXアプリケーション, X, Xorgを終了します。例えば、ルートでログインでしてinit 3(ランレベル3)へ移行します。

   ```
   # init 3
   ```

2. 最初のノードをメモリ・シールドし、構成を確認します：

   ```
   # /usr/bin/shield -m1 -a 0-3 -c
   ```

```
CPUID     irqs    ltmrs    procs     mem
----------------------------------------
    0      yes      yes      yes      yes
    1      yes      yes      yes      yes
    2      yes      yes      yes      yes
    3      yes      yes      yes      yes
    4       no       no       no       no
    5       no       no       no       no
    6       no       no       no       no
    7       no       no       no       no
    8       no       no       no       no
    9       no       no       no       no
   10       no       no       no       no
   11       no       no       no       no
   12       no       no       no       no
   13       no       no       no       no
   14       no       no       no       no
   15       no       no       no       no
```

3. プリアロケート・グラフィック・ページを解放します：

```
# graphics-memory -U
Pre-allocated graphics memory:          0 pages
Total allocated graphics memory:        0 pages
Graphics memory in use:                 0 pages
Maximum graphics memory used:           0 pages

            Node 0  Node 1  Node 2  Node 3
 Preal:          0       0       0       0
 Total:          0       0       0       0
 InUse:          0       0       0       0
   Max:          0       0       0       0

 Remote NUMA node page usage statics:
  Free:          0       0       0       0
 Alloc:          0       0       0       0
```

4. ノード1およびノード2にページを割り当てますす：

```
# graphics-memory –n 1 –p 5120 –n 2 –p 5120
Pre-allocated graphics memory:      10240 pages
Total allocated graphics memory:    10240 pages
Graphics memory in use:                 0 pages
Maximum graphics memory used:           0 pages

            Node 0  Node 1  Node 2  Node 3
 Preal:          0    5120    5120       0
 Total:          0    5120    5120       0
 InUse:          0       0       0       0
   Max:          0       0       0       0

 Remote NUMA node page usage statics:
  Free:          0       0       0       0
 Alloc:          0       0       0       0
```

5. nvidiaサービスを再開し、Xを再起動またはinit 5へ戻ります：

```
# service nvidia start
# init 5
```

RedHawkカーネルは、PREALLOC_GRAPHICS_PAGESに静的にコンパイルされた値を無効にし、起動時にプリアロケートされるグラフィック・ページのバッファ・プールに割り当てられるページ数を定義するために`pregraph_pgs=`*`numpages`*ブート・パラメータを使って起動することが可能です。

あるいは、RedHawkカーネルは、プリアロケート・グラフィック・ページのサポートを完全に無効にするために`no_pregraph_pgs`ブート・パラメータを使い起動することが可能です。`no_pregraph_pgs`は`pregraph_pgs`より優先することを注意してください。

run(1)を利用したNUMAサポート(プロセス用)

run(1)の“mempolicy” オプションは、関連する情報を表示するだけでなく、実行しようとするプロセスにタスク全体NUMAメモリ・ポリシーを規定するために使用することが可能です。

概要：

run [*OPTIONS*] *COMMAND* [*ARGS*]

“mempolicy” は利用可能な*OPTIONS* の1つで以下の書式があります：

--mempolicy=*MEMPOLICY_SPECIFIER*
-M *MEMPOLICY_SPECIFIER*

runで実行された既存のプロセスまたはスレッドを特定する*PROCESS/THREAD_SPECIFIER* は、生成されようとするプロセスにだけ影響を与えるmempolicyオプションを使用することはできません。

MEMPOLICY_SPECIFIER は以下の1つのみを含みます。各々はその最初のユニークな文字に省略することが可能です。*list* はカンマ区切りリストまたはCPUの範囲です(例：“0,2-4,6”)。“active” または “boot” は全てのアクティブなプロセッサまたはブート・プロセッサをそれぞれ指定するために使用することが可能です。オプションのティルダ[~]はリストの否定ですが、“active”では否定を使用できません。

NUMAが有効なシステムにおいては、**bias(-b)**および**mempolicy**の**bind**と**interleave**オプションへの*list*の書式は代わりとなる省略表現も受け付けます。このNUMAの省略表現書式が使用される場合、**bias(-b)**および**mempolicy**のリストはどちらも各値もしくは範囲値の前に先導する“n”, “C”, “c”の文字を持つ必要があり、この表記法は以下の意味を含みます：

n[**nodeid**]
以下の先導する”n”の値はNUMAのノードIDまたはノードIDの範囲となり、この表記は「指定したNUMAノード内の全てのCPU」を意味します。例：

```
run –M n=n0,n2-3 …
```

C[**cpu**]
以下の先導する”C”の値はCPU IDまたはCPU IDの範囲となり、この表記は「指定したCPUに加え同じNUMAノード内にある全てのCPU」を意味します。例：

```
run –M i=C2,C4-5,n2 …
```

c[**pu**]
以下の先導する"c"の値はCPU IDまたはCPU IDの範囲となります。例：

```
run –mempolicy bind=c0-1,n3 …
```

実行できる*MEMPOLICY_SPECIFIER：*

[**~**]*list*

b[**ind**]=*list*
ローカルから*list* 内のCPUのメモリを使いMPOL_BINDメモリ・ポリシーを使って指定されたプログラムを実行します。

b[**ind**]
ローカルから**--bias**オプションで指定されたCPUのメモリを使いCPUMPOL_BINDメモリ・ポリシーを使って指定されたプログラムを実行します。**--bias**オプションは実行する予定およびこの選択肢を指定する必要のあるプログラムのCPUを定義します。

i[**nterleave**]=[**~**]*list*
ローカルから*list* 内のCPUのメモリを使いMPOL_INTERLEAVEメモリ・ポリシーを使って指定されたプログラムを実行します。

p[**referred**]=*cpu*
ローカルから単一の指定されたCPUの使用を選び、MPOL_PREFERREDメモリ・ポリシーを使って指定されたプログラムを実行します。

p[**referred**]
選択されたメモリは、('local' 割り当てポリシーで)割り当てを開始するCPUを含むノード上に置かれ、MPOL_PREFERREDタスク全体NUMAメモリ・ポリシーにて指定されたプログラムを実行します。

d[**efault**]
MPOL_DEFAULTメモリ・ポリシーを使って指定されたプログラムを実行します。これは既定のメモリ・ポリシーです。

n[**odes**]
各ノード上のトータル・メモリと現在の空きメモリに加えて各NUMAノードに含まれるCPUを表示します。**run**のこの呼び出しで指定される他のオプションやプログラムはありません。

v[**iew**]
現在のプロセスのメモリ・ポリシー設定を表示します。**run**のこの呼び出しで指定される他のオプションやプログラムはありません。

システムに1つ以上のローカル・メモリなしCPUを含む時、これらのCPUはシステム初期化中にラウンドロビン方式でノードに割り当てられます。ノードへ割り当てられますが、これらは実際はローカル・メモリを所有しておらず、常に(所有する割り当てられたノードへのメモリ・アクセスを含む)非ローカル・メモリ・アクセスが行われます。このタイプの構成下では、**v**[**iew**]の出力はローカル・メモリを含まない各NUMAノード上のCPUを表示する追加の"NoMemCpus" 列を含みます。NUMA対応カーネルを使用する時は各CPUにメモリ・モジュールが組み込まれた構成になっているハードウェアを推奨します。

マルチ・ノード・システム上で**--mappings**/**-m**オプション付きで**run**を指定すると *PROCESS/THREAD_SPECIFIER* 引数により指定されたプロセスまたはスレッドのこの各NUMAノードのユーザー・マッピング・ページ数を表示します。このオプションは実行時に'command' パラメータを使用することができません。

runのほかのオプションについては、**run(1)**のmanページまたは4章の「runコマンド」セクションを参照してください。

もし**numactl(8)**がシステム上で利用可能である場合、NUMAメモリ・ポリシーを設定するために使用することが可能です。

shmconfig(1)を利用したNUMAサポート(共有メモリ領域用)

NUMAポリシーは“mempolicy” オプションにより**shmconfig(1)**を使用して新しい共有メモリ領域を割り当てるまたは既存の共有メモリ領域を変更することが可能です。

概要：

/usr/bin/shmconfig -M *MEMPOLICY* [**-s** *SIZE*] [**-g** *GROUP*] [**-m** *MODE*] [**-u** *USER*] [**-o** *offset*] [**-S**] [**-T**] {*key* | **-t** *FNAME*}

“mempolicy” オプションは以下の書式があります：

--mempolicy=*MEMPOLICY*
-M *MEMPOLICY*

MEMPOLICY は以下の1つのみを含みます。各々はその最初のユニークな文字に省略することが可能です。*LIST* はカンマ区切りリストまたはCPUの範囲です(例：“0,2-4,6”)。“active” または “boot” は全てのアクティブなプロセッサまたはブート・プロセッサをそれぞれ指定するために使用することが可能です。オプションのティルダ[~]はリストの否定ですが、“active”では否定を使用できません。

各ノードに含まれているCPU、各ノードのトータルおよび利用可能な空きメモリを見るために**run -M nodes**を使用します。

[**~**]*LIST*

b[**ind**]=*LIST*
: ローカルから*LIST* 内のCPUのメモリを使い指定された領域をMPOL_BINDメモリ・ポリシーに設定します。

i[**nterleave**]=[**~**]*LIST*
: ローカルから*LIST* 内のCPUのメモリを使い指定された領域をMPOL_INTERLEAVEメモリ・ポリシーに設定します。

p[**referred**]=*CPU*
: ローカルから単一の指定されたCPUの使用を選び、指定された領域をMPOL_PREFERREDメモリ・ポリシーに設定します。

p[**referred**]
: 選択されたメモリは、(’local’ 割り当てポリシーで)割り当てを開始するCPUを含むノード上に置かれ、指定された領域をMPOL_PREFERRED NUMAメモリ・ポリシーに設定します。

d[**efault**]
: 指定された領域をMPOL_DEFAULTメモリ・ポリシーを設定します。これは既定値です。

v[**iew**]
: 指定した領域の現在のメモリ・ポリシー設定を表示します。

mempolicyオプションで使用可能な追加のオプションは以下となります：

--size=*SIZE*
-s *SIZE*
領域のサイズをバイトで指定します。

--offset *OFFSET*
-o *OFFSET*
既存の領域の先頭からのオフセットをバイトで指定します。この値はページサイズの倍数へ切り上げられます。もし**-s**オプションも指定された場合、offset+size の合計値は領域の合計サイズ以下である必要があります。

--user=*USER*
-u *USER*
共有メモリ領域の所有者のログイン名を指定します。

--group=*GROUP*
-g *GROUP*
領域へのグループ・アクセスが適用可能なグループの名称を指定します。

--mode=*MODE*
-m *MODE*
共有メモリ領域へのアクセスを管理するパーミッションのセットを指定します。パーミッションを指定するために8進数を使用する必要があります(既定値は0644)。

--strict
-S
領域の範囲内のページが指定された現在適用されているメモリ・ポリシーと一致しない場合はエラーを出力します。

--touch
-T
範囲内の各ページへ接触(読み取り)させ、早期にメモリ・ポリシーを適用します。既定値では、アプリケーションがこれらの領域およびページ内(割り当てたページ)の傷害へアクセスする時にポリシーが適用されます。

key 引数は共有メモリ領域のユーザ選択識別子を意味します。この識別子は整数または既存のファイルを参照する標準的なパス名のどちらも可能です。パス名が提供される時、`ftok(key,0)`は**shmget(2)**呼び出しのkeyパラメータとして使用されます。

--tmpfs=*FNAME* / **-t** *FNAME* はkeyの代わりにtmpfsファイルシステムのファイル名を指定するために使用することが可能です。**-u, -g, -m**オプションはこの領域のファイル属性を設定または変更するために使用することが可能です。

shmconfigの他のオプションについては、manページまたは3章内の「shmconfigコマンド」セクションを参照してください。

もし**numactl(8)**がシステム上で利用可能である場合、それもまたNUMAメモリ・ポリシーを設定するために使用することが可能です。

システムコール

以下のシステム・サービス・コールが利用可能です。**numaif.h**ヘッダー・ファイルはこれらいずれの呼び出しを行うときもインクルードする必要があることに注意してください。

set_mempolicy(2)	現在のプロセスにタスク全体メモリ・ポリシーを設定します
get_mempolicy(2)	現在のプロセスまたはメモリ・アドレスのメモリ・ポリシーを取得します
mbind(2)	共有メモリを含むアドレス空間の特定範囲にポリシーを設定します
move_pages(2)	プロセスのページ・セットを異なるNUMAノードへ移動します

ライブラリ機能

/usr/lib64/libnuma.soライブラリは、NUMA対応の単純なプログラミング・インターフェースを提供します。これはNUMAメモリ・ポリシーやノードをサポートするルーチンの様々な種類、および基礎となるNUMAシステム・サービス・コールを使用するための代わりのインターフェースを含みます。詳細については**numa(3)**のmanページを参照してください。

情報提供ファイルおよびユーティリティ

以降のセクションでは、NUMAノードに関連する情報を表示するために使用可能なファイルやユーティリティについて説明します。

ノード統計値

NUMAがカーネル内で有効である時、各ノードは**/sys/devices/system/node/node#** サブディレクトリ内のファイル情報セットを所有します(# はノード番号、例：0, 1, 2...)。このサブディレクトリ内のいくつかのファイルを以下に記載します。

cpumap	このノード内のCPUの16進数ビットマップを表示します。例： > **cat /sys/devices/system/node/node3/cpumap** 08
cpulist	このノード内のCPUのリストを表示します。例： > **cat cpulist** 4-7
numastat	ノードのhit/miss統計値を表示します。表示されるフィールドの説明については次のセクションを参照してください。
meminfo	ノードの様々なメモリ統計値を表示します(空き、使用済み、ハイ、ロー、全メモリの合計を含みます)。
distance	ローカル・ノードから各ノードのメモリの距離を表示します。“10”の値はメモリがローカルであることを示し、“20”の値はメモリが、例えば離れた1つのハイパーチャネル接続を示します。
cpu#	ノードに関連付けられているCPUデバイス・ファイルです。例：

```
$ ls -l /sys/devices/system/node/node3/cpu3
lrwxrwxrwx 1 root root 0 jan 21 03:01 cpu3
->../../../../devices/system/cpu/cpu3
```

マッピングされたページのノードID

指定したプロセスまたはスレッドに現在マッピングされている各ページのNUMAノードIDにより**numapgs(1)**は場所を表示します。**-a**オプションを指定しない限り、物理メモリ・ページにマッピングされている場所だけを出力します。

構文：

numapgs [*OPTIONS*]

OPTIONS は以下のとおり：

--pid=*pid*, **-p** *pid*
プロセスIDまたはスレッドIDのアドレス空間が表示されます。

--start=*saddr*, **-s** *saddr*
表示されるマッピングの範囲を制限するため、この*saddr* 16進の仮想アドレス値未満にマッピングされているノードIDは表示されません。もし**--end**が指定されていない場合、*saddr* からアドレス空間の最後までの全てのノードIDエントリが表示されます。

--end=*eaddr*, **-e** *eaddr*
表示されるマッピングの範囲を制限するため、この*eaddr* 16進の仮想アドレス値以上にマッピングされているノードIDは表示されません。もし**--start**が指定されていない場合、アドレス空間の先頭から*eaddr*-1までの全てのノードIDエントリが表示されます。

--all, **-a**
物理メモリへの有効なマッピングを含んでいるこれらの場所だけでなく、プロセスのアドレス内の全仮想アドレス・ページの場所を表示します。出力のピリオド(.)はマッピングされていない場所または(I/O空間マッピングのような)メモリ・オブジェクトへのマッピングを表します。このオプションは指定した範囲内の全てのページの場所を表示するために**--start**または**--end**と一緒に使用することが可能です。

--version, **-v**
numapgsの現在のバージョンを表示して終了します。

--help, **-h**
利用可能なオプションを表示して終了します。

各出力ラインは最大8個の10進数のノードID値を含みます。

もし(**mlock(2)**または**mlockall(2)**を通して)現在ロックされている場合、"L" がNUMAノードID値の右側に表示されます。もしページが現在複製されている場合(「メモリ・シールドされたノード」を参照してください)、"R" がNUMAノードID値の右側に表示されます。

以下は、各ノードID値の隣のLが示すとおり**mlockall(2)**を使い全てのページがロックされたプロセスの**numapgs**出力のサンプルの抜粋です。複製されたページはそれらのノードID値の隣にRで表されます。

```
3a9b000000-3a9b12b000 r-xp /lib64/tls/libc-2.3.4.so
   3a9b000000: 0L 0L 0L 0L 0L 0L 0L 0L
   3a9b008000: 0L 0L 0L 0L 0L 0L 0L 0LR
   3a9b010000: 0L 0L 0L 0L 0L 0L 0L 0L
```

pagemap(1)ユーティリティは指定されたプロセスのアドレス空間に現在マッピングされている各ページのNUMAノードIDも表示します。更にマッピングされている各ページに関連する様々なページ・フラグも表示します。例：

```
# pagemap –p $$ -s 0x400000 –e 0x404000
00400000-0055b000 default r-xp /bin/ksh93
   0x400000: pfn: 0x37dcfb node: 1 mapcnt: 3 flags: ref uptd lru act map dsk
   0x402000: pfn: 0x39ece8 node: 0 mapcnt: 3 flags: ref uptd lru act map dsk
   0x403000: pfn: 0x36c52e node: 2 mapcnt: 3 flags: ref uptd lru act map dsk
```

いくつかのページ・フラグはユーザーが適切な特権を持っている場合にのみ表示される事に注意してください。詳細については**pagemap(1)**のmanページを参照してください。

numastatを利用したNUMA成功/失敗統計値

numastatは全てのノードの**/sys/devices/system/node/node#/numastat**ファイルから情報を結合するスクリプトです。

```
$ numastat
                 node 3     node 2     node 1     node 0
numa_hit          43674      64884      79038      81643
numa_miss             0          0          0          0
numa_foreign          0          0          0          0
interleave_hit     7840       5885       4975       7015
local_node        37923      59861      75202      76404
other_node         5751       5023       3836       5239
```

`numa_hit`	このノードで行われたメモリ割り当てが成功した数
`numa_miss`	このノードで行うことが出来ず、代わりに他のノードへ割り当てたメモリ割り当ての数
`numa_foreign`	他のノードでメモリ割り当てに失敗し、代わりにこのノードから割り当てられた割り当ての数
`interleave_hit`	このノードで行われたインターリーブ・メモリ割り当てが成功した数
`local_node`	ローカル・ノードから行われたメモリ割り当ての数
`other_node`	非ローカル・ノードへ行ったメモリ割り当ての数

kdbサポート

以下の**kdb**コマンドがNUMAをサポートするために追加または修正されました。この追加のサポートはカーネルがNUMAサポート有効として構成された時にだけ現れることに注意してください。

task	mempolicyとil_nextタスク構造体フィールドを加えて出力します

pgdat [*node_id*] 指定したノードのゾーンリスト、または*node_id*が指定されない場合はゾーン0をデコードします

カーネル・テキスト・ページの複製

RedHawk Linuxカーネルは、各NUMAノード内のカーネル・テキストおよび読み取り専用データ・ページの複製をサポートしています。この複製は、カーネル命令および読み取り専用データがローカルから各ノードへアクセスすることによってシステム全体のノード間のメモリ・トラフィックを減らすことに役立ちます。

プレビルトRedHawk Linuxカーネルを使っている場合、常駐カーネル・テキストおよび動的にロードされたカーネル・モジュール・テキストの両方は自動的に複数ノードのNUMAシステムに複製されます。

このサポートを制御する2つのカーネル構成オプションが存在します：カーネル・テキストの複製を有効および無効にするCONFIG_KTEXT_REPLICATION、およびカーネル常駐テキスト複製は有効にしたままの状態でカーネル・モジュール・テキストの複製を無効にするだけに使用することが可能なCONFIG_KMOD_REPLICATION。ブート時にカーネル・テキスト複製サポートを無効にするために2つのgrubオプションが追加されました。詳細についてはページ10-19「構成」を参照してください。

ktrinfo(1)ユーティリティは、カーネル・テキストの複製情報や統計データを表示するために利用可能です。

オプションなしで**ktrinfo**が呼び出された場合、システム内の全てのNUMAノードの複製情報を表示します。-nオプションは単一ノードに出力を制限するために使用することが可能です。ノード1の情報を表示する例を以下に示します：

```
> ktrinfo -n1
Node 1 Text Translations
virtual_address                    physical_address size
ffffffff81000000-ffffffff811fffff  0000000236a00000-0000000236bfffff 2048 kB
ffffffff81200000-ffffffff813fffff  0000000236400000-0000000236ffffff 2048 kB
ffffffff81400000-ffffffff81469fff  0000000236600000-0000000236669fff  424 kB
Node 1 Read-only Data Translations
virtual_address                    physical_address size
ffffffff81600000-ffffffff817fffff  0000000236000000-0000000236ffffff 2048 kB
ffffffff81800000-ffffffff81852fff  0000000236200000-0000000236252fff  332 kB
Node 1 statistics
total_repli_pages          17676 kB
repli_resident_pages        6900 kB
repli_module_pages         10740 kB
repli_module_pgtbls           36 kB
page_alloc_failures            0
pagetbl_alloc_failures         0
```

上記の出力は、ノード1に対するカーネル・テキストとカーネル読み取り専用データの両方の常駐カーネル仮想アドレスから物理アドレスへの変換に続いて、ノード1で常駐およびモジュールの複製に使用されているメモリの量を示してします。最後の2つの割り当て失敗ページ・カウンターは、カーネル・モジュールがロードされた時にページがノード1へ正常に割り当てできなかった場合にゼロ以外の値となります。

ktrinfo(1)ユーティリティは、-mオプションによりモジュール単位で各ノードに複製されたモジュール空間の量を表示するために使用することも可能です。-mオプション出力の一部を以下に示します：

```
> ktrinfo -m
Module                  Text_RO_sz      Node0      Node1      Node2      Node3
sr_mod                       16384      16384      16384      16384          -
cdrom                        28672      28672      28672      28672          -
sd_mod                       32768      32768      32768          -      32768
crc_t10dif                    8192       8192       8192          -       8192
crc32c_intel                  8192       8192       8192       8192          -
aesni_intel                  36864      36864      36864      36864          -
cryptd                        8192       8192       8192       8192          -
aes_x86_64                   12288      12288      12288      12288          -
aes_generic                  32768      32768      32768      32768          -
ahci                         20480      20480      20480      20480          -
igb                         102400     102400     102400          -     102400
dca                           8192       8192       8192          -       8192
megaraid_sas                 49152      49152          -      49152      49152
button                        8192       8192       8192          -       8192
dm_mirror                    16384      16384          -      16384      16384
dm_region_hash                8192       8192          -       8192       8192
dm_log                        8192       8192          -       8192       8192
dm_mod                       49152      49152      49152      49152          -
```

2番目の列(Text_RO_sz)は、各モジュール内のテキストと読み取り専用データの合計サイズをKバイトで記述しています。node#行は現在ロードされた各モジュールの各ノードに実際に割り当てられたKバイト量を示します。ノードのKバイトのサイズがText_RO_szの値未満である場合、これはモジュールがロードされた時のノードにおいてページが正常に割り当てられなかったことを示します。

長音記号"-"は、カーネル・モジュールがシステムのほかのノードに複製される前に初めはそのノードに動的にロードされたことを示すことに注意して下さい。

カーネル・モジュール・ページの割り当て

カーネル・モジュールは通常何の問題もなく全てのNUMAノードにロードされ複製されます。しかし、ノード内のメモリの空き容量が小さい場合、複製されるモジュール・テキストと読み取り専用データを設定するためのそのノードへのページ割り当ては失敗する可能性があります。これはシステム起動中に発生する可能性がある一方、システムが起動し、様々なシステムのアクティビティが既にアクティブになった後にカーネル・モジュールがロードされ、利用可能な空きメモリが少なくなった後に発生する可能性がそれ以上にあります。

1つ以上のページがページ複製でノードに割り当てることが出来ない場合、モジュールが最初にロードされたノード上のページは小さなメモリ・ノード上の仮想アドレス転送のために使用されます。

カーネル・モジュール・ページ割り当てに失敗した場合、メッセージがコンソール及び**/var/log/messages**ファイルに出力されます。例：

```
Kernel text replication: nvidia module page alloc
failure(s) in node 1: no?space
```

上述のメッセージはノード1上の**nvidia.ko**のカーネル・モジュール・テキストおよび読み取り専用データの複製に必要な全てのページが利用可能なメモリの不足により達成できなかったことを示します。

ノード1上でのカーネル・モジュール複製中に失敗したページおよびページ・テーブル割り当ての数を以下のコマンドで表示することが可能です：

```
# ktrinfo -n1 | grep failures
page_alloc_failures           68
pagetbl_alloc_failures         4
```

ノード1上に複製された空間の実際の総量に対する**nvidia.ko**モジュールのテキストおよび読み取り専用データ(Text_RO_sz)の合計は以下のコマンドで表示することが可能ですが、表示される値はKバイトです：

```
# ktrinfo -m | grep Module ; ktrinfo -m | grep nvidia
Module             Text_RO_sz     Node0     Node1
nvidia                8581120         -    331776
```

最後に以下のコマンドは各NUMAノード上で利用可能および空きメモリの量を速やかに表示するために使用することが可能です：

```
# run -Mc
Node    MemSize      MemFree      Cpulist
    0   4094 MB      3709 MB      0-3
    1   4096 MB        55 MB      4-7
```

NUMAバランシング

標準Linuxで自動的にサポートするNUMAバランシングはRedHawk Linuxのプレビルト・カーネル内に含まれていますが、デフォルトで有効化されていません。本オプション機能はgrubオプションで立ち上げ時に、または**sysctl(8)**パラメータを介して起動後に動的に動的に有効化することが可能です。

アプリケーションは、通常そのタスクが実行中のNUMAノードのローカルにあるメモリへアクセスしている時に最も機能します。NUMAバランシングはアプリケーションのデータをそれを参照するタスクの近くのメモリへ移動します。これはアクセスするメモリの近くのCPUで実行するようにタスクのスケジューリングも変更する可能性があります。これはNUMAバランシングが有効である場合にカーネルが全て自動的に行います。

NUMAバランシングはマルチNUMAノードのシステムにおいてフェア・スケジューリング・クラスで実行中のタスクにのみ影響する事に注意して下さい。

NUMAバランシング・ページの移動とタスク・スケジューリングの変更を行うための判断は、専用に作成されるNUMAバランシング・ページ・フォルトの利用を経て長時間に渡り収集される統計値に基づいています。有効から無効(違反)への様々なタスクのユーザー・アドレス空間変換の周期的な変更はフェア・スケジューリング・クラスのティック・タイマーから駆動されます。

NUMAバランシング・ページ・フォルトの処理は：

- 統計値がバランシングの決定を行うために集められます。
- 違反ページ変換は修復され、恐らく異なるNUMAノードへ移動されます。
- タスクは異なるCPUセット上で実行されるようスケジュールされる可能性があります。

NUMAバランシングの有効化

ブート時にNUMAバランシングを有効にするgrubオプションは次のとおり：

```
numa_balancing=enable
```

あるいは、**/etc/sysctl.conf**ファイルに以下の行を追加する事でNUMAバランシングをシステム起動中に有効にします：

```
kernel.numa_balancing=1
```

最後は、手動で**sysctl**に対応する**/proc**ファイルへ書き込みことでONとOFFを切り替える事も可能です。例えば：

```
# /bin/echo 1 > /proc/sys/kernel/numa_balancing
# /bin/echo 0 > /proc/sys/kernel/numa_balancing
```

シールディングの相互作用

NUMAバランシングが有効でプロセスまたはローカル・タイマーがシールドされたCPUがない場合、NUMAバランシングはシステムの全てのCPUおよびNUMAノードの中で標準的なLinuxの方法で機能します。

ところが、1つ以上のCPUでプロセスまたはローカル・タイマーがシールドされている場合、NUMAバランシングの挙動は現在のリアルタイム・シールディング構成に作用して向上するように変更します。これらの変更はRedHawk Linuxカーネル特有のものとなります：

- NUMAバランシングは、プロセスがシールドされたCPU上で実行している間はフェア・スケジューリング・タスクのユーザー・アドレス空間では行われません。これはそうではなく発生するランダムなページ・フォルトは除外します。

- NUMAバランシング・ページ・フォルトはローカル・タイマー・シールドされたCPU上で実行しているフェア・スケジューリング・タスクでは発生しません。これはローカル・タイマーをシールドしているCPUは人為的なNUMAバランシング・ページ・フォルト変換を定期的に生成するために使用されるフェア・スケジューラーのティック・タイマーを無効にするという事実が原因です。

シールディングの制限

シールドされたCPUのプロセスにおけるNUMAバランシング障害の制限に関する主な注意事項はマルチスレッド化されたアプリケーションに関係します。

CPUアフィニティを非シールドCPUおよびシールドCPUに設定したいくつかのスレッドで同じユーザー・アドレス空間を持つ複数のスレッドが存在する場合、NUMAバランシング処理は非シールドCPU上で実行している一連のスレッドにおいてそのアドレス空間内で発生します。

結果として、シールドCPUのプロセスが実行しているタスクは、それらのアドレス空間内の他の非シールドCPUに設定したNUMAバランシング・ページ・フォルトの対象となる可能性があります。

従って、NUMAバランシングを有効化した環境のシールドCPU内のいくつかのCPUにマルチ・スレッド・アプリケーションをスケジュールした時、シールドCPUのプロセスのNUMAバランシング・ページ・フォルトを回避したい場合は、シールドCPUのマルチ・スレッド・プロセスに属する全てのスレッドを完全にONまたはOFFにすることが最善です。

性能ガイドライン

アプリケーションを特定NUMAノードへ割り付けおよびバインドするCPUシールディングを通して、ページの割り当てをNUMAシステム上で最も効率的な方法で行うことが可能です。タスクや共有メモリ領域を扱うためのガイドラインを以下に示します。

タスク全体の**NUMA mempolicy**

MPOL_BINDポリシーはタイム・クリティカル・アプリケーションにとって通常もっとも有用なポリシーです。ページ割り当てのためにデターミニスティックにノードを指定することを許可する唯一のポリシーです。もしメモリ割り当てが指定するノードまたは指定するノード一式から行えない場合、プログラムはSIGKILLシグナルにより終了します。

MPOL_BINDメモリ・ポリシーでCPUシールドとCPU割り付けを組み合わせることにより、シールドCPUが作成され、シールドCPUのNUMAノードからだけ割り当てられるアプリケーションのページのあるシールドCPU上でそのアプリケーションが実行されます。書き込みデータ・ページ上のコピーは一旦書き込まれるとローカルになりますが、書き込みデータ・ページ上の既存の共有テキスト・ページのコピーはローカルではない可能性があることに注意してください。

run(1)コマンドはMPOL_BINDメモリ・ポリシーによりシールドされたCPU上でアプリケーションを起動するために使用することが可能です。あるいは、アプリケーションのアドレス空間内に既に存在するページがNUMAメモリ・ポリシーのその後の変更により影響されないために、**mpadvise(3)**や**set_mempolicy(2)**またはNUMAライブラリ機能の呼び出しで実行を開始した後直ぐにそのCPUアフィニティとNUMAメモリ・ポリシーをアプリケーションに設定することが可能です。

以下の例は、**run(1)**コマンドのバイアスとメモリ・ポリシー・オプションを使い、CPU2にあるNUMAノードだけからメモリを割り当てるMPOL_BINDメモリ・ポリシーのシールドCPU上でアプリケーションを起動する方法を示します。

```
$ shield -a 2
$ run -b 2 -M b my-app
```

シールドCPUおよび**shield(1)**コマンドに関する詳細な情報は、2章と**shield(1)**のmanページを参照してください。

共有メモリ領域

MPOL_BINDメモリ・ポリシーを共有メモリ領域のために使用することも通常は推奨します。共有領域のNUMAメモリ・ポリシーは**mbind(2)**システム・サービスコールまたは**shmconfig(1)**ユーティリティにより指定することが可能です。

共有メモリ領域が複数のCPUから参照されることになる場合、メモリ・アクセス性能を最大にするために共有メモリ領域の異なる部分に異なるMPOL_BINDメモリ・ポリシー属性を指定することが可能です。

例として、主に共有メモリ領域の下半分へ書き込む“low”アプリケーションと主に共有メモリ領域の上半分へ書き込む“high”アプリケーションがあると見なします。

1. ‘123’の値をキーとする共有メモリ領域を作成します。ページ割り当てのためにCPU2のNUMAノードでMPOL_BINDメモリ・ポリシーを使う領域の下半分、ページ割り当てのためにCPU3のNUMAノードでMPOL_BINDを使う上半分を変更します。

   ```
   $ shmconfig -s 0x2000 123
   $ shmconfig -s 0x1000 -M b=2 123
   $ shmconfig -o 0x1000 -M b=3 123
   ```

2. CPU2とCPU3の両方をシールドします。

   ```
   $ shield -a 1,2
   ```

3. メモリ割り当てのためにCPU2のNUMAノードを使うMPOL_BINDメモリ・ポリシーのCPU2上で“low”アプリケーションを実行開始、メモリ割り当てのためにCPU3のNUMAノードを使うMPOL_BINDメモリ・ポリシーのCPU3上で“high”アプリケーションを実行開始します。

   ```
   $ run -b 2 -M b low
   $ run -b 3 -M b high
   ```

構成

AMD Opteronと最新のIntelプロセッサーだけがNUMAアーキテクチャを持っています。以下のカーネル・パラメータはNUMAノード上の処理に影響を及ぼします。これら全てのパラメータは64bitのRedHawkプレビルト・カーネルでデフォルトで有効になっています。

NUMAとACPI_NUMA, X86_64_ACPI_NUMAとAMD_NUMA

これらのカーネル・パラメータはNUMAカーネル・サポートのために有効である必要があります。これらはカーネル構成GUIの「Processor Type and Features」項目でアクセス可能であり、全てのプレビルトRedHawkカーネルでデフォルトで有効となっています。

`numa=off` はブート時にNUMAシステム上でNUMAカーネル・サポートを無効にするために指定することが可能なブート・オプションです。これはノードに全CPUが属する単一ノードのシステムを作成します。NUMAサポートが組み込まれていないカーネルとは異なり、この場合にはノードなしのフラット・メモリ・システムであり、NUMAユーザー・インターフェースが呼ばれた時エラーを返します。

AMD Opteronまたは最新のIntelシステム上でNUMAが有効なカーネルを使用する時、以下のハードウェアを推奨します：

- システムの各CPUにメモリ・モジュールが組み込まれていることを大いに推奨します。さもなければ、ローカル・メモリ・モージュールのないCPUはメモリ・アクセスの度に他のメモリ・モジュールへ離れてアクセスすることになり、従ってシステム性能が低下します。

- いくつかのBIOSがサポートするメモリ・モジュール・インターリーブ・ハードウェア・サポートはBIOSで無効にする必要があります。もし無効ではない場合、NUMAが有効なカーネルのNUMAサポートは無効となり、結果、システムの全てのCPUを含む単一NUMAノードとなります。

PAGE_REPLICATION

有効にすると、ページキャッシュ複製サポートはカーネルへ纏められます。PAGE_REPLICATION_DYNAMICの設定次第で、ページキャッシュ複製はシステム起動時からシステム上で常にアクティブとなる、または手動で起動するまで非アクティブとなります。

PAGE_REPLICATION_DYNAMIC

PAGE_REPLICATIONと一緒に有効にすると、ページキャッシュ複製はシステム起動時はアクティブではありませんが、**/proc/sys/vm/page_replication_enabled**へ「1」を書き込むことにより、または1つ以上のメモリ・シールドされたNUMAノードを作成するために**shield(1)**を使うことにより、手動で起動することが可能です。

これを無効にしPAGE_REPLICATIONを有効にすると、ページキャッシュ複製はシステム起動時からシステム上で常にアクティブとなります。

MEMSHIELD_ZONELIST_ORDER

有効にすると、NUMAノードの領域リストはNUMAノードがメモリ・シールドされた時に再指示されます。これらの領域リストはローカル・ノードのメモリ・リソースが低下した時に利用可能なメモリのノード検索順序を制御します。領域リストは、ローカル・ノードがメモリ割り当て要求に満足できない時、シールドされたノードのメモリの使用に頼る前に他の非シールド・ノードからのメモリを使用するように再指示されます。システムにシールドされたメモリもない場合、オリジナルの領域リストの支持は自動的に元に戻されます。

CONFIG_KTEXT_REPLICATION

64bit NUMAカーネルが有効である場合、カーネル常駐部分のカーネル・テキストおよび読み取り専用データ・ページの複製は有効です。このパラメータは全てのRedHawk Linuxプレビルト64bitカーネルで有効です。
このパラメータが有効である場合、grubオプション"no_ktext_repli"をブート時にこのサポートを無効にするために使用することが可能です。

CONFIG_KMOD_REPLICATION

CONFIG_KTEXT_REPLICATIONが有効である場合、このパラメータは更にモジュールのロード時にカーネル・モジュール・テキストおよび読み取り専用データの複製を有効にすることが可能です。このパラメータは全てのRedHawk Linuxプレビルト64bitカーネルで有効です。

このパラメータが有効である場合、grubオプション"no_kmod_repli"をブート時にモジュール複製サポートを無効にするために使用することが可能です。

CONFIG_NUMA_BALANCING
: 有効にすると、このパラメータはカーネルに自動NUMAバランシング・サポートを編成します。本パラメータは全てのRedHawk Linuxプレビルト・カーネルで有効となっています。

CONFIG_NUMA_BALANCING_DEFAULT_ENABLED
: 有効にした場合、NUMAバランシングはブート時に自動的に有効となります。このパラメータはRedHawk Linuxプレビルト・カーネルでは有効となっていません。

11
カーネルの構成および構築

本章ではRedHawk Linuxカーネルの構成および構築方法に関する情報を説明します。

序文

RedHawkカーネルは**/boot**ディレクトリ内に置かれています。実際のカーネル・ファイル名称はリリース毎に変更されますが、通常は以下の形式となります：

vmlinuz-*kernelversion***-RedHawk-***x.x*[**-***flavor*]

kernelversion	RedHawkカーネルのベースとなる(-rclまたは-pre7のような接尾語を含む可能性のある)Linuxカーネル・ソース・コードの公式なバージョンです。
x.x	RedHawkリリースのバージョン・番号です。
flavor	特有のカーネルにより提供される追加のカーネル機能を特定するオプションのキーワードです。

システムがブートするたびにカーネルはメモリへロードされます。それはシステムの基本的な機能を実行する最も重要なコードの核です。システムが動作している間中、カーネルは物理メモリ内に留まります(ほとんどのユーザー・プログラムのようにスワップされません)。

カーネルの正確な構成は以下次第です：

- システム実行時の動作を定義する多数のチューニング・パラメータ
- オプションのデバイス・ドライバとローダブル・モジュールの数

カーネル構成または再構成は、1つ以上のカーネル変数を再定義し、新しい定義に従い新しいカーネルを作成する処理となります。

一般的には、供給されるカーネルは殆どのシステムに適合するチューニング・パラメータやデバイス・ドライバにより作成されています。しかし、特定のニーズに対しカーネル性能を最適化するためにチューニング・パラメータのいずれかを変更したい場合、カーネルの再構成を選択することが可能です。

チューニング・パラメータの変更またはハードウェア構成を変更した後は、カーネルの再構築、インストール、再起動が必要となります。

ccur-configを利用したカーネルの構成

RedHawk Linux製品にはいくつかのプレビルト・カーネルが含まれています。カーネルは"***-flavor***"接尾語により互いに区別されます。以下のフレイバが定義されています：

generic (no suffix)	4GB以下のジェネリック・カーネル。このカーネルは最も最適化され、最高の総合的な性能を提供しますが、NightStar RTツールを十分に活用するために必要な特定の機能が不足しています。
trace	トレース・カーネル。このカーネルはジェネリック・カーネルの全機能をサポートし、更にNightStar RT性能分析ツールのカーネル・トレース機能のサポートを提供するため殆どのユーザーに推奨します。
debug	デバッグ・カーネル。このカーネルはトレース・カーネルの全ての機能をサポートし、更にカーネル・レベル・デバッグのサポートを提供します。このカーネルはドライバを開発する、またはシステムの問題をデバッグしようとするユーザーに推奨します。

各プレビルト・カーネルは、対応するカーネル構成の詳細全てを保存する構成ファイルを持っています。これらのファイルはカーネル・ソース・ツリーの**configs**ディレクトリの中に置かれています。プレビルト・カーネルのための構成ファイルは以下のように指定されています：

i386アーキテクチャ上 (32-bit)：

ジェネリック・カーネル	**standard**
トレース・カーネル	**trace**
デバッグ・カーネル	**debug**

x86_64アーキテクチャ上 (64-bit)：

ジェネリック・カーネル	**standard**
トレース・カーネル	**trace**
デバッグ・カーネル	**debug**

プレビルト・カーネルの1つと一致するカーネルを構成および構築するため、カーネル・ソース・ツリーの最上層へ**cd**を行い、**ccur-config(8)**ツールを実行する必要があります。

NOTE

ccur-configスクリプトはルートで実行する必要があります。もしカーネル変更が行われる場合、システムはグラフィック・モード(ランレベル5)である、または有効なDISPLAY変数が設定されている必要があります。

以下の例では、RedHawk Linux 6.5.2トレース・カーネルの構成をベースとする新しいカーネルを構築するためにカーネル・ソース・ツリーを構成します。構成ファイルの"**.config**"接尾語は自動的に添えられますので、指定する必要がないことに注意してください。

```
# cd /usr/src/linux-3.10.47RedHawk6.5.2
# ./ccur-config trace
```

ccur-configは、**configs**ディレクトリ内にある適切なカスタム構成ファイルを指定することによりカスタマイズされたカーネルに使用することも可能です。**-k** *name* オプションは新しいフレイバの名前を付けるため、**-s**オプションは**configs**ディレクトリ内に構成ファイルを保存するために使用することが可能です。使用例：

```
# ./ccur-config -s -k test debug
```

は、RedHawk i386 **debug**カーネルがベースとなる**-test**フレイバ接尾語のカーネルを構成し、その結果の構成を**configs/test.config**として保存します。

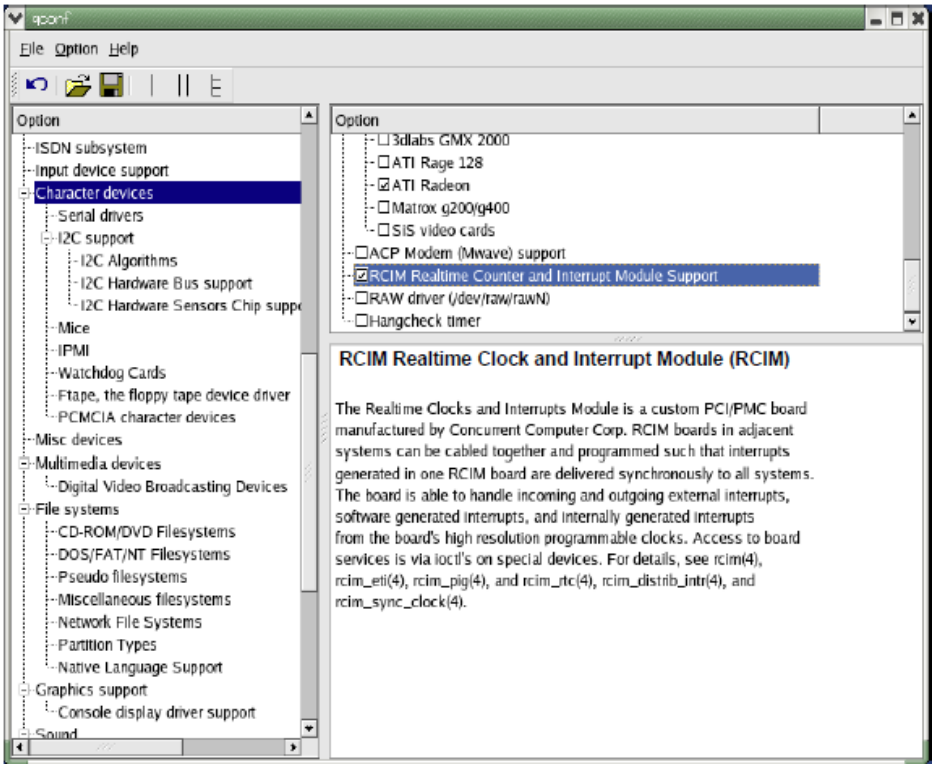
ccur-configの実行中、RedHawk Linuxカーネルの多くの異なる機能がカスタマイズ可能なグラフィカル構成インターフェース(GUI)で表示されます。カーネル構成GUIの例については画面11-1を参照してください。

「File」メニューから「Save」の選択は、変更を保存しプログラムを終了するために選択する必要があります。たとえ構成パラメータを変更していなくても、カーネル構成ファイルを適切に更新するために「Save」をやはり選択する必要があることに注意してください。

グラフィカル構成ウィンドウを経て利用可能な設定と構成オプションの完全なリストはこの文書の範疇を超えていますが、ユニークなRedHawkの機能とリアルタイム性能に関連する多くのチューニング・パラメータは本マニュアルの至る所で明文化され、また、付録Bに一覧表となっています。更に、パラメータが選択された時、そのパラメータに関する情報はGUIの別ウィンドウ内に表示されます。

もしカーネル・パラメータを変更したくない場合、**-n**オプションを**ccur-config**に指定することでGUIは表示されません。

画面11-1 カーネル構成GUI



カーネルの構築

どのカーネル構成が使用されているかに関わらず、トップレベルの**Makefile**内に定義される時、カーネルは接頭語 "vmlinuz" の後に現在のカーネルのバージョンの文字列、引き続き接尾語 "-custom" が加えられた名前が付けられます。例：

vmlinuz-3.10.47-rt50-RedHawk-6.5.2-custom

最終的な接尾語は**-k** *name* オプションを**ccur-config**へ指定することにより変更することが可能です。これはトップレベルの**Makefile**内でREDHAWKFLAVOR変数として*name* を定義し、再び**-k**オプションまたは**Makefile**の編集により変更されるまで有効のままとなります。同じカーネル・ソース・ツリーから複数のカーネルを構築する時、既存のカーネルを誤って上書きすることを回避するために接尾語を変更することが重要となります。

NOTES

Concurrentより提供されるプレビルト・カーネルは、Concurrentが使用するために予約されている接尾語を持っています。従って、次の接尾語を設定してはいけません：(文字列なし)、"-trace"、 "-debug"。

カーネル用のドライバ・モジュールを構築する必要がある場合は、**ccur-config -c**オプションを使用してください。(本章で後述する「ドライバ・モジュールの構築」セクションを参照してください)

一旦カーネル構成が完了すると、カーネルは適切な**make(1)**コマンドの発行により構築することが可能となります。トップレベルの**Makefile**にはいくつものターゲットが存在しますが、以下は特に重要です：

make bzImage スタンドアロン・カーネルを構築します。

make modules カーネル構成内で指定されたカーネル・モジュールを構築します。

make modules_install
現在構成されたカーネルに関連するモジュールのディレクトリへモジュールをインストールします。このディレクトリの名称は、トップレベルの**Makefile**で定義されるカーネルのバージョンの文字列から生成されます。例えば、REDHAWKFLAVORが "-custom" として定義される場合、結果として生じるモジュールのディレクトリは "**/lib/modules/***kernelversion***-RedHawk-***x.x.x***-custom**" となります。

make install **/boot**ディレクトリに関連する**System.map**ファイルと一緒にカーネルをインストールします。

NOTE

新しいカーネルを完全に構築しインストールするためには、**Makefile**のターゲットの全てを上記の順序で発行される必要があります。

カーネルの構成および構築セッションの実例については、図11-1を参照してください。

図11-1 カーネルの構成および構築セッションの実例

```
# cd /usr/src/linux-3.10.47RedHawk6.5.2
# ./ccur-config -k test debug
Configuring version: 3.10.47-rt50-RedHawk-6.5.2-test
Cleaning source tree...
Starting graphical configuration tool...
[ 要望に応じてカーネルパラメータを設定 ]

Configuration complete.

# make bzImage
# make modules
# make modules_install
# make install
[ 新しいカーネルを参照するよう/boot/grub2/grub.cfgを編集して再起動 ]
```

ドライバ・モジュールの構築

既存のConcurrent提供のカーネルもしくはカスタム・カーネルのいずれかを使用するためにしばしばドライバ・モジュールを構築することが必要となります。

カーネル用にドライバ・モジュールを構築するため、以下の条件を満足する必要があります：

- 要求するカーネルは現在実行中のカーネルである必要があります。
- カーネル・ソースのディレクトリは、**ccur-config**を通して現在実行中のカーネル用に適切に構成されている必要があります。

もしカスタム・カーネルが「カーネルの構築」セクション内で解説された手順を使用して構築された場合、カーネル・ソース・ディレクトリが既に適切に構成され**ccur_config**を実行している必要がないことに注意してください。

ccur_configへの**-c**オプションは、カーネル・ソースのディレクトリが適切に構成されていることを保証するために使用することが可能です。このオプションは実行中のカーネルを自動的に検知し、実行中のカーネルに一致するソース・ツリーを構成します。ドライバ・モジュールは実行中のカーネルでその後に使用するために適切にコンパイルすることが可能です。

NOTE

> **ccur_config**への**-c**オプションは、ドライバ・モジュールを構築するためにカーネル・ソース・ツリーを構成することだけを目的とし、新しいカーネルを構築する時に使用すべきではありません。

ccur_configへの**-n**オプションは、構成パラメータを変更する必要のない時に指定することも可能です。**-n**付きで構成GUIは表示されず、構成のカスタマイズは実行されません。

プレビルトRedHawkカーネルへの動的なロード・モジュールを構築する例については、次のセクションを参照してください。

プレビルト**RedHawk**カーネルで動的にロード可能なモジュールの構築例

RedHawkシステムへの機能追加は、システムに追加のハードウェア・コントローラを設置することで達成されます。静的カーネル・ドライバの要求がない限り、新しいハードウェア・デバイス用にサポートを追加するためにカスタム・カーネルを作る必要はありません。

以下の例では、RedHawkシステムへComtrol RocketPortシリアル・カード・サポートを追加します。Comtrol RocketPortのドライバのソースはRedHawkカーネル・ソース・ツリーに含まれています。

RedHawk **trace**カーネルがこの例で実行中のカーネルとなります。

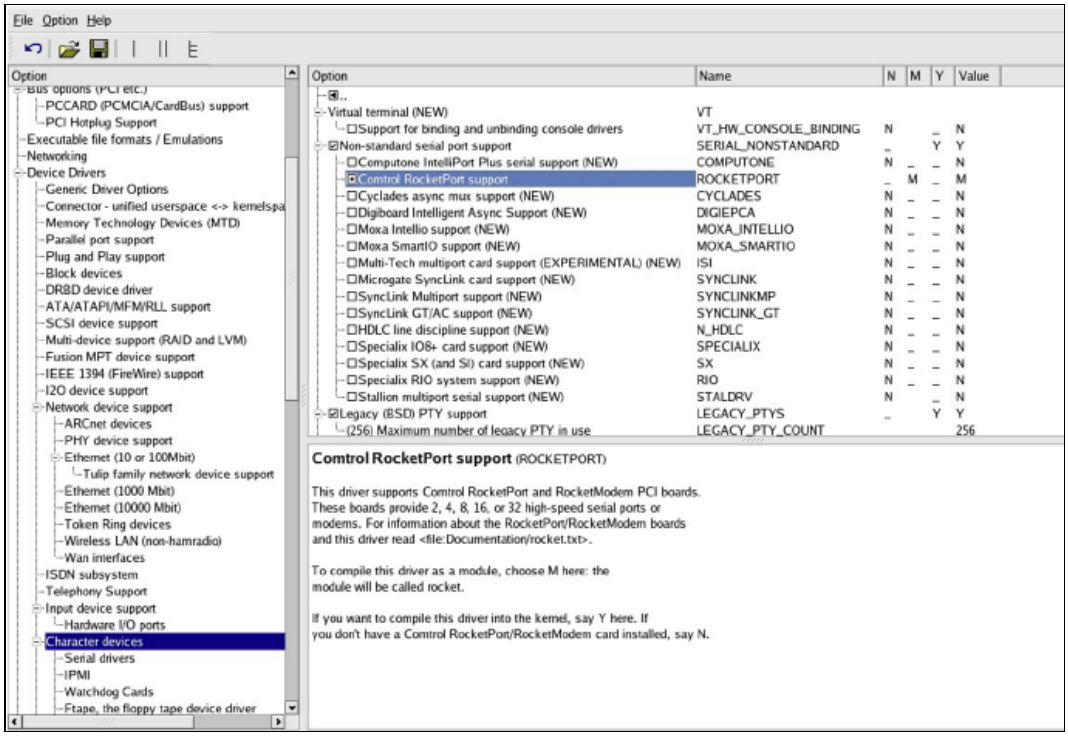
1. カーネル・ソース・ツリーを構成するために**ccur-config**を実行します。*kernelname* は実行中カーネルの‘uname -r’による出力であることに注意してください：

   # cd /lib/modules/*kernelname***/build**
   # ./ccur-config –c

2. GUIウィンドウ内で、「Device Drivers ->Character Devices->Non-standard serial port support-> Comtrol RocketPort support」の値を “M” (モジュール)に設定します。 GUIの図例については画面11-2を参照してください。(Show Name, Show Range, Show Dataオプションを選択して表示しています)

3. 構成を保存し、GUIを終了します。

画面11-2 カーネル構成GUIでシリアル・カード・サポートの追加



4. 新しいカーネル・モジュールを構築するために**make**を実行します：

   # make REDHAWKFLAVOR=-trace modules

5. makeが終了すると出力にRocketPortドライバが見つかります。出力例：

   ```
   LD [M] drivers/char/rocket.ko
   ```

   そして以下のようにコピーします：

   # mkdir /lib/modules/*kernelname***/kernel/extras**
   # cp drivers/char/rocket.ko /lib/modules/*kernelname***/kernel/extras/**

6. モジュールをロードするため、**modprobe(8)**を使って依存ファイルを設定します。

   # depmod

7. **/lib/modules/***kernelname***/build/Documentation/rocket.txt**ファイルはComtrol RocketPortカードに関する構成要件を含んでいます。デバイス・エントリが作成され、適切なコマンド(**MAKEDEV(8)**および**modprobe(8)**)でカーネル初期化時に実行される**/etc/rc.modules**ファイルへ挿入することによってドライバは自動的にロードされます。

   a. **/etc/modprobe.conf**へ以下のエイリアスを挿入します：

      ```
      alias char-major-46 rocket
      ```

   b. もし**/etc/rc.modules**ファイルがシステム上に存在しない場合は作成する必要があります。それは機能させるために0x755のファイル・パーミッションを持つ必要があります。そのファイルには以下を含めてください：

      ```
      #!/bin/bash
      /sbin/MAKEDEV ttyR
      modprobe rocket
      ```

カーネル・ソース・ツリー内に存在しないドライバを追加する例については、RedHawkシステム上の**/usr/share/doc/ccur/examples/driver**を参照してください。

追加情報

Linuxカーネルの構成と構築の理解および謎解きに役立つ情報を提供することが利用できるリソースが多く存在します。有効な最初の方法はインストールされたRedHawkカーネル・ソース・ツリーのトップレベルにある**README**ファイルを読むことです。更に、オンライン・ガイドがLinux Documentation ProjectやLinuxtopiaを含むいくつかのLinux資料サイトで利用可能です：

Linux Kernel in a Nutshell
http://www.linuxtopia.org/online_books/linux_kernel/kernel_configuration/index.html

12
カーネル・デバッギング

本章はRedHawk Linuxが提供するカーネル・デバッギングやクラッシュ・ダンプ解析のためのツールについて説明します。

概要

カーネルの対話型デバッグの機能が標準Linuxの**kgdb**/**kdb**カーネル・デバッガを介してRedHawk Linuxデバッグ・カーネルで提供され、追加のConcurrent拡張機能も含まれます。

標準的なCentOSの**kexec-tools**ユーティリティは、システム・クラッシュ・ダンプの生成をサポートするkexecベースのvmcore kdumpを提供します。Concurrentが拡張した**crash(8)**ユーティリティはvmcoreクラッシュ・ファイルおよび動作中のシステムを解析するために提供されます。

VMcore生成イベント

kdumpサポートが構成され有効化された場合、vmcoreクラッシュ・ファイルは以下の理由のいずれかで生成されます：

- カーネル・パニック。
- **sysctl(1)**のkdump発生イベントの1つに遭遇した。後述の「Sysctl(1) Kdumpオプション」項を参照して下さい。
- **/proc/sys/kernel/sysrq**が設定され、Alt-Sysrq-cがキーボードから入力された。
- **/proc/sys/kernel/sysrq**が設定され、以下のコマンドが実行された。

  ```
  # echo c > /proc/sysrqtrigger
  ```

- **kdump**コマンドが**kgdb**/**kdb**カーネル・デバッガ・プロンプトから実行された。

vmlinuxネームリスト・ファイルの保存

vmcoreのkdumpファイルが生成されシステムがオリジナル・カーネルでリブートした後、vmcoreクラッシュ・ファイルを解析するために**vmlinux**ネームリスト・ファイルがcrashで必要になるため、**vmlinux**ファイルへのソフト・リンクの生成またはそれをvmcoreファイルがあるディレクトリへのコピーのどちらかを推奨します。

例：

```
# cd /var/crash/127.0.0.1-2014.12.11-10:28:57
```

そして、ネームリスト・ファイルをこのディレクトリへコピー：

```
# cp /var/vmlinux/vmlinux-`uname -r` vmlinux
```

もしくは、将来これを削除する予定がないのであれば、ネームリスト・ファイルへのソフト・リンクを作成：

```
# ln -s /var/vmlinux/vmlinux-`uname -r` vmlinux
```

VMcore Kdump構成

デフォルトで、パニックまたは他のクラッシュ状況に遭遇したカーネルのために`crashkernel` GRUBオプションが指定された場合は、RedHawk Linuxのkdumpカーネルはサブディレクトリ**/var/crash**以下にvmcoreクラッシュ・ファイルを生成するように構成されています。`crashkernel` GRUBオプションの推奨値は`crashkernel=128M@64M`となります。

以下のコマンドのいずれかまたは両方を実行する事で現在起動したカーネルのkdump構成の状態を確認する事が可能です：

```
# systemctl -l status kdump.service
# kdumpctl status
```

既定のvmcore kdump構成が殆どのシステムおよび状況で十分である一方、2つのシステム構成ファイルおよびシステム管理者がデフォルト設定の変更が可能な**systemd** `kdump.service`が存在します：

- **/etc/sysconfig/kdump**ファイル

  本構成ファイルはkexec kdumpカーネル設定の構成を制御する様々な変数を含んでいます。

  KDUMP_KERNELVER変数は、kexecを実行するvmcore kdumpファイルを使用するカーネルを表示するために使用することが可能です。デフォルトでKDUMP_KERNELVER変数の文字列は**ccur-kernel-kdump** RPMのインストール中にRedHawk Linuxのkdumpカーネルに設定されます。この変数の変更は通常は必要ではないはずですが、カスタム・ビルトkdumpカーネル等の他のkdumpカーネルを試したい場合は本変数を変更する事が可能です。

  KDUMP_COMMANDLINE_APPEND変数は、kdumpカーネルをkexec実行時に使用する追加のGRUBオプションを含みます。通常これらのオプションは殆どのシステムに適用されるはずですが、kdumpカーネルの実行で問題がある場合には本変数の修正やGRUBオプションの追加または削除する事も可能です。

- **/etc/kdump.conf**ファイル

  本構成ファイルはkdumpカーネルを使用するための様々なkexec実行後の命令を含みます。これらの命令はkdumpサービスにより生成されたinitramfsファイル内に格納されます。**kdump.conf(5)**のmanページは本構成ファイルに関する更なる情報を含んでいます。

ローカルのルート・ファイルシステム内にvmcore kdumpファイルを生成するためのシステムを構成する場合、本ファイル内に有用な2つの命令オプションが存在します：

- `path /var/crash`

  vmcoreファイルは「path」ディレクトリ下のサブディレクトリに生成されます。デフォルトで、たとえこの行がコメント・アウトされていても/var/crashディレクトリは使用されます。通常はこの値を変更する必要はありません。しかしながら、指定した「path」の値はルート「/」ファイルシステム内にあるディレクトリとする必要があります。

- `core_collector makedumpfile -l --message-level 1 -d 31`

  この行は実際のvmcore kdumpファイルを生成するために使用されるコマンドを定義します。デフォルトで、**makedumpfile**(1)ユーティリティがlzo圧縮(`-l`)でダンプ・レベル31(`-d 31`)のvmcoreファイルを生成するために使用されます。

  makedumpfile(1)のmanページは`core_collector`呼出し命令行に指定可能な様々なオプションを記述しています。

  ダンプ・レベルはvmcoreファイルに含まれないページの形式を制御するので、ファイルを生成するために必要となるサイズと時間を削減します。ダンプ・レベル値31はvmcoreファイルが生成される時に保存する最も大きいページの形式を飛ばします。代わって、ダンプ・レベル値0はvmcoreファイルに全ての物理ページを保存します。ユーザー空間のページを調査する必要があるケースでは、**crash(1)**解析中にこれらのページを見るためにダンプ・レベル0が使用される必要があります。

  Makedumpfileは生成されるvmcoreファイルのサイズを減らすために設計された3つの圧縮形式(lzo(`-l`), snappy(`-p`), zlib(`-c`))をサポートします。あるいは、vmcoreファイルはELF形式(-E)で生成する事が可能です。しかしながら、ELF形式が使用される場合、圧縮は不可能です。RedHawk Linux 7.0のcrashユーティリティは、ファイルを最初に解凍する必要なく圧縮されたvmcoreファイルを直接読む機能がありことに注意して下さい。

- **systemctl**の利用

  vmcore kdumpファイルの生成を完全に停止したい場合、次のコマンドを実行する事が可能です：

  ```
  # systemctl disable kdump.service
  # systemctl stop kdump.service
  ```

  後にkdumpサービスを再度有効にしたい場合は次のコマンドを実行して下さい：

  ```
  # systemctl enable kdump.service
  # systemctl start kdump.service
  ```

kdump構成の更新

/etc/sysconfig/kdumpまたは**/etc/kdump.conf**構成設定のどちらかを修正する場合、次に起こりうるkdumpイベントに対して新しい設定を再構成させるためにkdumpサービスの再起動が必要となります。

kdump構成を再設定するためにシステムを再起動するまたは以下のコマンドを実行するのどちらかが可能です。作成した新しい構成に問題があるかどうかをすぐに判断できるように一般的には手動でkdumpサービスを再起動することと推奨します。

2つの方法のいずれかでkdump構成を変更する事が可能です：

```
# touch /etc/kdump.conf
# systemctl restart kdump.service
```

または：

```
# touch /etc/kdump.conf
# kdumpctl restart
```

構成の問題に遭遇した場合、各々の方法は若干異なる出力およびエラー情報を提供します。以下の2つのコマンドは現在のkdump構成に関するステータス情報を取得するために使用することが可能です：

```
# systemctl -l status kdump.service
# kdumpctl status
```

NOTE

RedHawk Linuxカーネルが動作した時にsystemctlおよびkdumpctlコマンドにより出力される以下のメッセージは無視して構いません。

```
cat: /sys/kernel/security/securelevel: No such file or
directory
```

scp VMcore生成の構成

ローカルのルート・ファイルシステム上のvmcore kdumpファイルを生成する代わりにリモート・システムにvmcore kdumpファイルを作成するセキュア・コピー(**scp**)を利用する事が可能です。

NOTE

CentOSのkexec-toolsユーティリティは、NFSルート・ベース・ファイルシステムで起動したシステムでは現在この機能をサポートしません

以下の方法はリモートでscp kdumpを行うためのシステムを構成します：

- リモートkdumpサーバー・システムで**sshd**サービスを有効とし、sshとscpを介したrootのログインを許可して下さい。詳細については**sshd(8)**および**sshd_config(8)**のmanページを参照して下さい。
- **/etc/kdump.conf**ファイルを編集し、次の項目をファイルの下部に追記して下さい：
  1. ssh の行をコメントアウトし次の値に変更して下さい：

```
ssh root@ip_address
```

`ip_address`はサーバー・システムのIPアドレスにする必要があります。例：

```
ssh root@192.168.1.10
```

NOTE

kdumpサーバー・システムのシンボル名称ではなく実際のIPアドレスを使用する必要があります。

2. core_collector行を編集し`-F`オプションをmakedumpfileオプションに追加して下さい。`-F`オプションは`scp` vmcoreの生成用にkexec-toolsユーティリティが必要としています。`-F`オプションを使用した場合、生成されたvmcoreの名称はkdumpサーバー・システム上で**vmcore.flat**となります。RedHawk Linux 7.0 crashユーティリティはこのvmcoreフォーマットを読んで解析する事が可能です。

   例：

   ```
   core_collector makedumpfile -F -l –messagelevel 1 -d 31
   ```

   NOTE

   kdumpサーバー・システムをRedHawk Linux 7.0ベースのシステムにすることを推奨します。RedHawk Linux 7.0のcrashユーティリティだけが圧縮されたvmcoreフォーマットを読んで解析する機能があります。kdumpサーバー・システムとしてRedHawk Linuxの古いバージョンを使う必要がある場合、上記`-l` lzo圧縮オプションに対して`-E` ELF非圧縮のmakedumpfileオプションに置き換えて下さい。

- リモート・サービス・システムへの**ssh**/**scp**アクセスで促されるパスワードが不要のrootユーザーを設定して下さい：

  ```
  # kdumpctl propagate
  ```

  上記コマンド実行後、パスワードのプロンプト無しでリモート・サーバー・システムにログインできることを手動で確認して下さい。例：

  ```
  # ssh -i /root/.ssh/kdump_id_rsa root@kdumpserver
  ```

- vmcore kdump構成を再設定するためにkdumpサービスを再起動して下さい：

  ```
  # kdumpctl restart
  ```

  または：

  ```
  # systemctl restart kdump.service
  ```

vmcoreイメージの保存に成功後、ローカルのvmlinuxファイル(`/var/vmlinux/vmlinux-` `uname -r`)をリモート・システムのvmcoreイメージが存在するディレクトリにコピーすることは良策である事に留意して下さい。

NFS VMcore生成の構成

ローカルのルート・ファイルシステムにvmcore kdumpさいるを作成する代わりに、リモートkdumoサーバー・システム上にあるNFSマウントされたファイルシステムにvmcore kdumpファイルを作成する事が可能です。

NOTE

CentOSのkexec-toolsユーティリティは、NFSルート・ベース・ファイルシステムで起動したシステムでは現在この機能をサポートしません

以下の方法はリモートでNFS kdumpを行うためのシステムを構成します：

- リモートNFSサーバーにお手持ちのシステムをNFSマウントしてリモート・ファイルシステムを利用可能として下さい：
  - リモート・システムの**/etc/exports**へのエントリを追加してください。
  - 次のコマンドを実行して下さい：

    ```
    # systemctl restart nfs
    ```

  - 新しいファイルシステムがマウントされているリモートNFSで利用可能であることを確認するために**/usr/sbin/exportfs**を実行して下さい。

  詳細については**exports(5)**および**exportfs(8)**のmanページを参照して下さい。

- ローカルの**/etc/fstab**ファイルへエントリを追加し、vmcoreイメージを保存したいリモート・ファイルシステムをNFSマウントするローカルのマウント・ディレクトリを作成して下さい。

  /etc/fstabのエントリはシンボル名称ではなくサーバー・システムの実際のIPアドレスを使用する必要があります。例：

  ```
  192.168.1.3:/kdumps /server/kdumps nfs
  rw,rsize=8192,wsize=8192,timeo=10,retrans=5 0 0
  ```

- これが正しく機能していることを確認するためにリモートNFSファイルシステムを手動でマウントして下さい。例：

  ```
  # mount /server/kdumps
  # /bin/df
  ```

- ファイルを編集しサーバー・システムのIPアドレスとディレクトリを含めるためにファイルの最後部付近の`nfs`行を変更して下さい。例：

  ```
  nfs 192.168.1.3:/kdumps
  ```

/etc/kdump.confの`path`命令変数は、使用するリモート・システムのNFSマウントされる位置の下のディレクトリを特定することに注意して下さい。例えば、`path`値のデフォルト**/var/crash**が前述の例を使用する場合、crash vmcoreファイルはサーバー・システムの**/kdumps/var/crash**ディレクトリ内に置かれます。

- NFSマウントされたファイルシステムを介してvmcoreファイルの保存を再構成するためにkdumpサービスを再開して下さい：

```
# kdumpctl restart
```

または：

```
# systemctl restart kdump.service
```

vmcoreイメージの保存に成功後、ローカルのvmlinuxファイル(`/var/vmlinux/vmlinux-` `uname -r`)をリモート・システムのvmcoreイメージが存在するディレクトリにコピーすることは良策である事に留意して下さい。

Sysctl(1) Kdumpオプション

sysctl.conf(5)の構成ファイル**/etc/sysctl.conf**を介して構成することが可能なkdump vmcoreクラッシュ処理に直接関連している様々な構成可能なカーネルパラメータがあります。

/etc/sysctl.confファイル内のエントリを追加または変更する場合、システムを再起動または次のコマンドを入力する必要があります：

```
# systemctl restart systemd-sysctl
```

kdumpの挙動を変更するために以下のパラメータを**/etc/sysctl.conf**ファイルに任意に追加する事が可能です：

`kernel.kexec_boot_cpu = 1`

`kdump`カーネルを起動するためにブートCPU(CPU 0)の使用を強要します。一部のシステムはCPU0でしかkdumpカーネルが正常に起動しません。

`kernel.panic_on_oops = 1`

カーネルのoopsイベントが発生した場合にkdumpを誘発します。

`vm.panic_on_oom = 1`

メモリ不足の状況が発生した場合にkdumpを誘発します。

`kernel.unknown_nmi_panic = 1`

不明なNMIが発生またはNMIボタンが押された場合にkdumpを誘発します。詳細については後述の「NMIウォッチドッグ」を参照して下さい。

`kernel.panic_on_io_nmi = 1`

カーネルがIOエラーに起因するNMIを受信した場合にkdumpを誘発します。

`kernel.panic_on_unrecovered_nmi = 1`

カーネルが訂正不可能なパリティ/ECCメモリ・エラーのような既知の復旧不可能なNMI割込みに遭遇した場合にkdumpを誘発します。

crashを利用したダンプの解析

RedHawk Linuxの**crash**ユーティリティは**/usr/ccur/bin/crash**に配置されています。vmcoreまたはRedHawk Linuxカーネルを使用している動作中のシステムを解析する場合はcrashのこのバージョンを使用することを推奨します。

crashはダンプ・ファイル上または動作中のシステム上で実行することが可能です。**crash**コマンドは、特定のカーネル・サブシステムに及ぶ調査を行う様々なコマンドと一緒に全プロセス、ソース・コード逆アセンブル、フォーマット済みカーネル構造と変数の表示、仮想メモリ・データ、リンク先リストのダンプ等のカーネル・スタック・バック・トレースのような共通カーネル・コア分析ツールで構成されます。関連する**gdb**コマンドは入力することも可能ですが、実行するために組み込まれた**gdb**クラッシュ・モジュールへは順番に渡されます。

ダンプ・ファイルの解析

vmcoreダンプ・ファイル上で**crash**を実行するには、少なくても2つの引数が必要となります：

- カーネル*namelist* と呼ばれるカーネル・オブジェクト・ファイル名称。
- ダンプファイルは**vmcore**を指名します。

カーネル・パニックの場合、以下で示すように**crash**を呼び出します。引数は任意の順序で提供することが可能です。例：

```
# cd /var/crash/127.0.0.1-2014.12.11-10:28:57
# ls
vmcore vmcore-dmesg-incomplete.txt
# ln -s /var/vmlinux/vmlinux-`uname -r` vmlinux
# /usr/ccur/bin/crash vmlinux vmcore

      KERNEL: vmlinux
    DUMPFILE: vmcore [PARTIAL DUMP]
        CPUS: 8
        DATE: Thu Dec 11 10:28:47 2014
      UPTIME: 00:06:32
LOAD AVERAGE: 1.32, 0.91, 0.44
       TASKS: 175
    NODENAME: ihawk
     RELEASE: 3.16.7-RedHawk-7.0-trace
     VERSION: #1 SMP PREEMPT Sun Nov 30 20:30:19 EST 2014
     MACHINE: x86_64 (2199 Mhz)
      MEMORY: 8 GB
       PANIC: "Oops: 0002 [#1] PREEMPT SMP " (check log for details)
         PID: 14527
     COMMAND: "echo"
        TASK: ffff8800ce6c5730 [THREAD_INFO: ffff8800ca4f8000]
         CPU: 1
       STATE: TASK_RUNNING (PANIC)

crash>
```

DUMPFILE行のPARTIAL DUMPは、生成されたvmcoreファイルからページの特定の形式をフィルタで除くために**makedumpfile** **-d**オプションが使用されたことを示している事に注意して下さい。詳細については前述の「VMcore Kdump構成」の2ページ目を参照して下さい。

実行中システムの解析

実行中のシステム上で**crash**を実行するため、引数なしで指定します。**crash**は**vmlinux**ファイルを検索し、メモリ・イメージとして**/dev/mem**を開きます：

```
# /usr/ccur/bin/crash

      KERNEL: /boot/vmlinux-3.16.7-RedHawk-7.0-trace
    DUMPFILE: /dev/mem
        CPUS: 8
        DATE: Thu Dec 11 10:37:04 2014
      UPTIME: 00:07:24
LOAD AVERAGE: 1.08, 0.89, 0.46
       TASKS: 170
    NODENAME: ihawk
     RELEASE: 3.16.7-RedHawk-7.0-trace
     VERSION: #1 SMP PREEMPT Sun Nov 30 20:30:19 EST 2014
     MACHINE: x86_64 (2200 Mhz)
      MEMORY: 8 GB
         PID: 4550
     COMMAND: "crash"
        TASK: ffff8800caaf26c0 [THREAD_INFO: ffff8800cde38000]
         CPU: 2
       STATE: TASK_RUNNING (ACTIVE)

crash>
```

ヘルプの入手

crashのオンライン・ヘルプは以下の動作を通して利用可能です：

- crashコマンドのリストを表示するには「`crash>`」プロンプトで**help**または**?**を指定して下さい。その後、特定のコマンドに関するヘルプ情報を見るために**help** *command*を指定して下さい。
- 利用可能なオプション全てを一覧表示するヘルプ画面を表示するには**/usr/ccur/bin/crash -h**を指定して下さい。
- *cmd*で指定されたコマンドに関するヘルプ・ページを見るにはシステム・プロンプトで**/usr/ccur/bin/crash -h** *cmd*を指定して下さい。
- RedHawk Linuxの**crash(1)**のmanページを見るには**man -M /usr/ccur/man crash**を実行して下さい。

カーネル・デバッガ

標準的なLinuxカーネルデバッガ**kgdb/kdb**は、プレビルトRedHawk "debug" カーネルで有効となっています。

kgdb/kdb

kdbデバッガは、プログラムがカーネル・メモリを対話的に調査、カーネル・スタックのトレースバックを観察、カーネル機能の逆アセンブル、カーネル・コードにブレークポイントの設定、レジスタの内容を調査および修正することが可能です。**kdb**は一時的ユーザーやシステム管理者よりもむしろカーネル開発者がカーネルの問題を突き止めて修正する手助けを目的とします。

kdbはRedHawkデバッグ・カーネルに組み込まれていますが、デフォルトでは有効になっていません。ブート時に自動的に**kdb**を有効にするには、以下のカーネルコマンド行オプションの1つを**/boot/grub2/grub.cfg**内のデバッグ・カーネル行へ追加してそのカーネルを再起動してください：

```
kgdboc=ttyS0,115200          /dev/ttyS0がコンソールである場合
kgdboc=kbd                   PC画面がコンソールである場合
```

kdbは複数の方法で意図的に誘発させる事が可能です：

```
echo g > /proc/sysrq-trigger          シェル・プロンプトからシステム要求メカニズムのコマンド・シーケンスgキーを経由
```

kdbはそのトリガー・サービスのためのシステム要求メカニズム(*sysrq*)を活用します。例えば、お手持ちのシステムにCOM1が備えられ、COM1がそのコンソールである状態のターミナル・サーバーにtelnetで接続されていると仮定すると、システム上で以下のように入力する事でこのキー・シーケンスは**kdb**を起動します：

```
Ctrl-]
send break
g
```

これは、**Ctrl-]**キーの組み合せがtelnetをコマンド・モードにして、そこへ`send break`コマンドが*break*条件をtelnetに送信させます。その後、通常のASCII文字`g`を送信します。*break*の組み合わせに続く`g`(数秒以内)はCOM1を持つシステムを**kdb**に突入させます。

もしPC画面がお手持ちのシステムのコンソールである場合、**Break**キーの入力(続けて`g`)は取り付けられているシステムを**kdb**に突入させます。

一旦**kdb**に入ると、有用ないくつかの単純なコマンドを見つける事が可能となります：

```
pc              全てのタスクの概要を1行で表示します
btc             全てのCPUのスタック・トレースバックを表示します
bta             全てのタスクのスタック・トレースバックを表示します
btp 1           TIDが1のタスクのスタック・トレースバックを表示します
help            全てのkdbコマンドを表示します
bp pw_wait      pw_waitのアドレスにブレークポイントを仕掛けます
bc 0            ブレークポイント0を消去します
g               kdbを終了します
```

NMI割り込み

RedHawk Linuxでは、発生する各NMI (Non-maskable Interrupt)は*既知*または*不明*のいづれかです。NMIを引き起こす機能を持つマザーボード上の各デバイスのステータス・ビットを検索している時、NMIハンドラがNMIを引き起こしているステータスを伴うデバイスを見つけた場合はNMIは*既知*となります。NMIハンドラがそのようなデバイス見つける事ができない場合は、そのNMIは*不明*であると宣言します。

NMIウォッチドッグはそのようなステータスを可視化しないため、その割り込みは不明となります。これはシステム上に存在する可能性のある如何なるNMIボタンについても同様となります。

従って、お手持ちのシステムがNMIボタンを持っておりそれを使用したい場合、NMIウォッチドッグを停止する必要があり、そして不明NMIが発生した時にそのシステムをパニックにさせる構成ではないことを確認する必要があります。これは**/boot/grub2/grub.cfg**にあるカーネル・コマンド行に`nmi_watchdog=nopanic`オプションを追加する事で保証することが可能となります。

カーネル・コマンド行に`nmi_dump`オプションもまた存在する場合、不明NMIはカーネル・クラッシュ・ダンプに取り込まれます。これはその後にシステムの自動再起動を行います。

NMIウォッチドッグ

RedHawkはデバッグ・カーネルを除く全てのカーネルでNMIウォッチドッグが無効となっています。NMIウォッチドッグが通常無効となっているのは、それが有効であると全てのCPU上で約10秒毎に1回のNMI割り込みを生成してしまうためで、これらの割り込みはRedHawkの保証する割り込みシールド機能を妨害する事になります。

従って、デバッグ・カーネルは、システムがハングするまたはアプリケーションがハングする問題のデバッグを可能とするためにRedHawkのリアルタイム割り込みシールド機能の例外を許可しています。このケースでのNMIウォッチドッグは(割り込みがCPU上で長時間ブロックされる)*ハード・ロックアップ*と(実行待ちアプリケーションのいるCPU上でコンテキスト・スイッチが長時間発生しない)*ソフト・ロックアップ*の両方を検出するために使用されます。

NMIウォッチドッグは、`nmi_watchdog=0`または`nmi_watchdog=panic`カーネル・コマンド行オプションを介してデバッグ・カーネルにおいても無効にすることが可能です(**/boot/grub2/grub.cfg**を編集しどちらかのオプションをデバッグ・カーネルのコマンド行に追加して下さい)。`nmi_watchdog=0`オプションはNMIウォッチドッグを停止して発生する不明NMIを無視させますが、一方`nmi_watchdog=panic`オプションは発生する如何なる不明NMIでもシステムをパニックにします。

13
Pluggable Authentication Modules (PAM)

本章では、アプリケーションがユーザー認証要求を使用する可能性のある機能のライブラリを通して達成される安全と適切な認証スキームを提供するPAM機能について説明します。

序文

PAM(Pluggable Authentication Modules)は、認証プログラムの再コンパイルを必要とせずにシステム管理者が認証ポリシーの設定を許可する手段です。PAMを用いて、構成ファイルの編集によりプログラムへモジュールを接続させる方法を制御します。

殆どのユーザーはこの構成ファイルに触れる必要はありません。認証を必要とするプログラムを**rpm(8)**を使ってインストールする時、通常のパスワード認証をする必要のある変更を自動的に行います。しかしながら、構成をカスタマイズしたい場合には、構成ファイルを理解する必要があります。

PAMモジュール

PAM標準で定義される4種類のモジュールが存在します：

auth	実際の認証、場合によりパスワードの要求およびチェック、グループ・メンバーシップのような「証明書(Credential)」の設定を提供します。
account	認証が許可されていることを確認するためにチェックします(アカウントが有効期限切れではない、この時点でユーザーがログインを許可されている、その他)。
password	パスワードを設定するために使用します。
session	一旦、ユーザーがそのアカウントを使用することの許可が認証されると、場合によってはユーザーのホーム・ディレクトリのマウントもしくはメールボックスを利用可能にするために使用します。

複数のモジュールが使用されるため、これらのモジュールは積み重ねることが可能です。例えば、*rlogin* は通常少なくても2つの認証方法(*rhosts* 認証が成功した場合は接続を許可するのに十分であり、もしそれが失敗した場合はその後通常のパスワード認証が行われます)を使用します。

新しいモジュールはいつでも追加することが可能で、PAM認識アプリケーションはその後にそれを利用させることが可能です。

サービス

PAMを使用する各々のプログラムはそれ自身の「サービス」名称を定義します。**login**プログラムはサービス・タイプを*login* と定義し、**ftpd**はサービス・タイプを*ftp* と定義します。一般的にサービス・タイプはサービスへアクセスするために利用されるプログラムの名称で、(もしそれが異なる場合は)サービスを提供するためにプログラムは利用されません。

ロール・ベース・アクセス制御

RedHawk Linuxのロール・ベース・アクセス制御はPAMを使って実行されます。ロール・ベース・アクセス制御スキームでは、**capability.conf(5)**ファイル内に一連のロール(役割)を設定します。ロールは有効なLinuxケーパビリティ一式として定義されます。現在の全ての有効なLinuxケーパビリティは**/usr/include/linux/capability.h**カーネル・ヘッダ・ファイルの中、または`_cap_names[]`文字配列の使用により見ることが可能です。これらは付録Cで更に詳しく説明されています。

一旦ロールを定義するとそれはその次のロールのケーパビリティの1つとして使用されるという点では、ロールは積み木のように作用します。このように新たに定義されたロールは、以前に定義されたロールのケーパビリティを継承します。この機能の例は後述されています。詳細な情報については**capability.conf(5)**のmanページを参照して下さい。

一旦役割を定義したら、それは**capability.conf(5)**ファイル内のユーザーまたはグループへ割り当てられます。ユーザーは、現在のシステムのログインで有効なユーザーと一致する標準Linuxユーザーのログイン名です。グループは、現在のシステムで定義されている有効なグループと一致する標準Linuxグループ名称です。

/etc/pam.dにあるファイルは、ユーザーがシステムへログインするために使用することが可能なサービスに対応します。これらのファイルは**pam_capability**セッション行(**pam_capability**セッション行をサービス・ファイルへ追加する例は「例」セクションで後述)を含めて変更される必要があります。例：**/etc/pam.d/login** (または**/etc/pam.d/remote**)ファイルは、telnetを介してのログインをカバーするのに良い候補です。もしユーザーが変更されていないサービスを使いシステムへログインする場合、特別なケーパビリティの割り当ては行われません。

NOTE: もしケーパビリティが使用される場合、**/etc/pam.d/su**ファイルは、**su -l nobody** *daemon*のような呼び出しが**nobody**ユーザーに対してケーパビリティの一覧だけを *daemon* へ与え、呼び出しユーザーからは余分なケーパビリティは与えられないことを確実に行うセキュリティ措置として修正される必要があります。

以下のオプションは、**/etc/pam.d**のファイルに**pam_capability**セッション行が供給する時に指定することが可能です：

conf=*conf_file* 構成ファイルの場所を指定します。もしこのオプションが指定されない場合、既定の場所は**/etc/security/capability.conf**となります。

debug **syslog**を介してデバッグ情報をロギングします。デバッグ情報は**syslog** *authpriv*クラスに記録されます。通常、このログ情報は**/var/log/secure**ファイルに集められます。

例

以下の例は、**/etc/pam.d/login**へセッション行の追加を説明しています。**/etc/pam.d/remote**が存在する場合はそれの変更も行います。

NOTE: i386システム上のPAMファイルへのパスは**/lib/security**です。
x86_64 システム上のパスは**/lib64/security**です。

1. **telnet(1)**を介してシステムにログインするユーザーへ割り当るために**/etc/security/capability.conf**ファイルに定義されたロールを許可するためには以下の行を**/etc/pam.d/login**へ追加します：

   session required /lib/security/pam_capability.so

2. **ssh(1)**を介してシステムにログインするユーザーへ割り当るために**/etc/security/capability.conf**ファイルに定義されたロールを許可するためには以下の行を**/etc/pam.d/sshd**へ追加します：

   session required /lib/security/pam_capability.so

3. **su(1)**を介してユーザーが入れ替わるために**/etc/security/capability.conf**ファイルに定義されたロールを許可するため、そして入れ替わったユーザーが**su(1)**の呼び出しから不適切なケーパビリティを継承しないことを確実にするため、以下の行を**/etc/pam.d/su**へ追加します：

   session required /lib/security/pam_capability.so

4. **ssh**ユーザーが**/etc/security**に置かれているものとは異なる**capability.conf**ファイルからロール定義を取得するためには以下の行を**/etc/pam.d/sshd**へ追加します：

   session required /lib/security/pam_capability.so ¥
   conf=/root/ssh-capability.conf

   このようにして**/root/ssh-capability.conf**ファイルに定義されるロールを**ssh**を介してログインするユーザーに適用します。

ケーパビリティの定義

capability.confファイルはユーザーやグループへ定義および割り当てが可能なロールに関する情報を提供します。このファイルはロール、ユーザー、グループの3種類のエントリがあります。

Roles ロールは有効なLinuxケーパビリティの定義一式です。現在有効な全Linuxケーパビリティー式は**/usr/include/linux/capability.h**カーネル・ヘッダ・ファイルの中、または**cap_from_text(3)**のmanページで説明されている`_cap_names[]`文字配列を使って見る事が可能です。このケーパビリティは付録Cの中でも詳細に説明されています。尚、次のケーパビリティのキーワードは予め定義されています：

`all`	(`cap_setcap`を除く)全ケーパビリティ
`cap_fs_mask`	ファイル・システムに関する全ケーパビリティ
`none`	如何なるケーパビリティもなし

名前が示すとおり、様々なシステムのユーザーおよびグループが実行する必要のある職務に準じて、異なるロールが定義されることが求められます。

capability.confファイル内のロール・エントリの書式は以下となります：

role *rolename capability_list*

ケーパビリティ・リストのエントリは以前定義されたロールを参照することが可能です。例えば、ファイル内で*basic* と呼ばれるロールを定義して、それに続くロールのケーパビリティ・リストにケーパビリティの1つとしてそのロールを追加することが可能です。ケーパビリティ・リストは、空白またはカンマで区切ったユーザーの継承設定をONにするケーパビリティのリストであることに注意してください。

Users

ユーザーは、現在のシステムのログインが有効なユーザーに一致する標準Linuxユーザーのログイン名です。(**getpwnam(3)**による確認で)現在のシステムで有効なユーザーと一致しないユーザー・エントリは無視されます。

capability.confファイル内のユーザー・エントリの書式は以下となります：

user *username rolename*

特殊なユーザー名‘*’は、リストアップされたどのユーザとも一致しないユーザー、またはリストアップされたグループのメンバーシップを所有していないユーザーに対して既定のロールを割り当てるために使用することが可能です。

user * *default_rolename*

Groups

グループは、現在のシステムで定義される有効なグループに一致する標準Linuxのグループ名です。(**getgrnam(3)**による確認で)現在のシステムで有効なグループと一致しないグループ・エントリは無視されます。

capability.confファイル内のグループ・エントリの書式は以下となります：

group *groupname rolename*

例

1. 次の例は、ルートとほぼ等しい管理用のロール(`admin`)を設定します：

```
role       admin            all
```

2. 次の例は、継承可能なケーパビリティセットにsys_bootとsys_timeを追加するデスクトップ・ユーザー用のロールを設定します：

```
role       desktopuser      cap_sys_boot ¥
                            cap_sys_time
```

3. 次の例は、前もって作成されたデスクトップ・ユーザー・ロールを使い、パワー・ユーザー用のロールを設定します：

```
role       poweruser         desktopuser¥
                             cap_sys_ptrace¥
                             cap_sys_nice¥
                             cap_net_admin
```

4. ユーザーへdesktopuserロールを割り当てるため、**capability.conf**ファイルのUSERSセクションに以下を入力します：

```
user       joe               desktopuser
```

5. グループへpoweruserロールを割り当てるため、**capability.conf**ファイルのGROUPSセクションに以下を入力します：

```
group      hackers           poweruser
```

実装詳細

以下の項目は、完全なPAM機能の実現のための要件に対応します：

- **pam_capability**は、実行中のカーネルが**exec()**システムコール中に継承するケーパビリティを修正することを要求します。
  このモジュールに同梱しているカーネル・パッチが適用されたカーネルは、カーネル構成GUI(本書の「カーネルの構成および構築」章を参照して下さい)の「General Setup」項目にあるINHERIT_CAPS_ACROSS_EXEC構成オプションを使いケーパビリティ継承を有効にすることが可能です。全てのRedHawk Linuxカーネルはデフォルトでこのオプションが有効となっています。

14
デバイス・ドライバ

本章は、RedHawk Linux下のユーザー・レベルおよびカーネル・レベルのデバイス・ドライバに関する問題点に対応します。これはリアルタイム性能の問題に加えてデバイス・ドライバの書き方を容易にする追加機能に関する情報も含まれています。Linuxベースのデバイス・ドライバを記述する方法の予備知識を前提とします。ユーザー空間I/O(UIO)ドライバも説明します。

PCI-to-VMEブリッジ・デバイスのRedHawkサポートに関する情報は15章の「PCI-to-VMEサポート」で見ることが可能です。

デバイス・ドライバの種類の理解

RedHawk Linuxの下ではユーザー・レベル・デバイス・ドライバを簡単に書く事が可能です。ユーザー・レベル・ドライバはデバイス・レジスタを読み書きする、つまりプログラムI/O操作を開始するため、I/O空間にアクセスすることが可能です。カーネル・ドライバ・スケルトンの支援により、ユーザー・レベル・ドライバは割り込みの受信と同時に処理を開始することも可能になります。これは割り込みルーチンに付随するユーザー・レベル・ドライバの中でシグナル・ハンドラを許可する機能をサポートすることにより得られます。割り込みのハンドリングやユーザー・レベル・プロセスへシグナルを送信するためのサンプルのカーネル・ドライバ・テンプレートの場所については、本章で後述する「カーネル・スケルトン・ドライバ」セクションを参照して下さい。

Linuxの下でDMA I/O操作をするユーザー・レベル・ドライバを書くことは現実的ではありません。ユーザー・レベルからDMA操作を禁止するいくつかの問題(例えば、ユーザー空間バッファの物理アドレスを決定する方法が現在のところサポートされていない)が存在します。カーネル・レベル・デバイス・ドライバはI/O操作にDMAを利用するデバイスのために使用する必要があります。

ユーザー空間I/O(UIO)は、複数のI/Oボードに対してユーザー・レベル・デバイス・ドライバを記述するために使用することが可能です。UIOは、(ユーザー空間に記述されるドライバの主要部分で)ユーザー空間アプリケーションに使用される一般的なツールやライブラリを利用する小規模なデバイス単位のカーネル・モジュールが必要です。14-14ページの「ユーザー空間I/Oドライバ(UIO)」を参照して下さい。

ユーザー・レベル・デバイス・ドライバの開発

後述のセクションで、ユーザー・レベル・デバイス・ドライバの記述に影響を及ぼすRedHawk Linuxオペレーティング・システムの詳細について説明します。

PCIリソースへのアクセス

ブート処理中、PCIバス上のデバイスは自動的に構成され、割り込みが割り当てられて、デバイス・レジスタがメモリ・マップドI/O操作を通してアクセス可能なメモリ領域にレジスタがマッピングされます。これらのメモリ領域はベース・アドレス・レジスタ(BAR: Base Address Register)として知られています。デバイスは最大6個のBARを持つことが可能です。BARの内容はデバイスによって異なります。この情報についてはデバイスのマニュアルを参考にしてください。

RedHawk Linuxは、PCIデバイスのレジスタをマッピングするために必要となるコードを単純化する**/proc/bus**にあるPCIリソース・ファイル・システムをサポートします。このファイル・システムは、プログラムのアドレス空間へマッピング可能なメモリ領域を表すBARファイルを提供し、デバイスに関わる物理アドレスを知る必要なしにデバイスへのアクセスを提供します。PCI BARファイル・システムは、デバイスのPCI構成空間の読み書きに使用可能な *config-space* ファイルもまた提供します。*config-space* ファイルの先頭64バイトはPCIの仕様により定義されています。
残りの192バイトはデバイス・ベンダー固有となります。

各PCIハードウェア・デバイスはベンダーIDとデバイスIDが関連付けられています。これらは時間経過またはシステム間で変化しない固定値です。ブート時にPCIデバイスの動的構成のため、一旦システムがブートするとドメイン、バス、スロット、機能番号は固定されたままですが、各システムの同じPCIバス・スロットに差し込まれているように見えるボードでも基礎をなすハードウェアに応じてシステム間で異なる可能性があります。**/proc/bus/pci**とBARファイル・システム内のパスは、カーネルによって割り当てられたドメイン、バス、スロット、機能番号から生成され、ホスト・システムの物理ハードウェア・レイアウトに影響を受けます。例えば、物理的に異なるスロットにボードを差し込む、システムへデバイスを追加する、またはシステムBIOSの変更のような変更は、特定のデバイスに割り当てられたバスおよび/またはスロット番号を変更することが可能となります。

後述するPCI BARスキャン・インターフェースは、特定デバイスに関連するBARファイルを見つけるための方法を提供します。ドライバはBARファイルへのアクセスを得るために適切なデバイスのスロット・アドレスを突き止める必要があるため、これらのインターフェースがなければ、これらBARファイル・パスのハードウェア依存性質はユーザー・レベル・デバイス・ドライバのプログラミングの仕事を若干不便にします。

BARファイル・システム用ライブラリ・インターフェース、固定ベンダーIDとデバイスIDの値を使用して、PCIデバイスに関連する現在の他の値を獲得することが可能です。これらはデバイスへのBARファイル・ディレクトリのパスの他にそのディレクトリ内の各BARファイルに関する情報も含みます。これは各デバイスに関連するベンダーID、デバイスID、クラスID、サブクラスID、(割り当てられていれば)IRQ番号、ドメイン、バス、スロット、機能番号を返します。

このサポートは、カーネル構成GUIの「Bus options」項目にあるPROC_PCI_BARMAPカーネル・パラメータを通して全てのRedHawkプレビルト・カーネルでデフォルトで有効となっています。

PCI BARインターフェース

次のセクションでPCI BARインターフェースを説明します。

ライブラリのスキャン機能は反復します。もしシステムが求めるデバイス・タイプのインスタンスを複数持っている場合、これらのライブラリ機能は複数回呼び出される必要があります。ある関数はシステム内の一致するデバイス全ての数を返します。その他の関数は検索基準に一致するデバイスに関する情報を反復的に返します。デバイス情報は**/usr/include/pcibar.h**で定義される`bar_context`型に返されます。この構造体は**bar_scan_open**の呼び出しで作成されます。複数スキャンは、各々が持っているユニークな`bar_context`を同時にアクティブにすることが可能です。

インターフェースを以下に簡単に説明します：

bar_scan_open	PCIデバイスの新しいスキャンを開始します
bar_scan_next	次に一致するPCIデバイスを取得します
bar_device_count	アクティブ・スキャンに残る一致するデバイスの数を返します

bar_scan_rewind	スキャンを再開します
bar_scan_close	アクティブ・スキャンを閉じて関連するメモリを解放します
free_pci_device	見つけたデバイスに関する割り当てられたメモリ全てを解放します
bar_mmap	適切なページに整列したBARファイルを**mmap**します
bar_munmap	**bar_mmap**でマッピングしたBARファイルを**munmap**します

これらのインターフェースを使用するため、アプリケーションに**libccur_rt**ライブラリをリンクする必要があることに注意してください。

gcc [*options*] *file* **-lccur_rt** ...

これらの機能の使用を解説する例は、**/usr/share/doc/ccur/examples/pci_barscan.c**に提供されます。

bar_scan_open(3)

この機能は、PCIデバイスの検索のために初期コンテキストを作成するために使用します。返された`bar_context`は、反復子(イテレータ)インターフェースの状況データを指定する**/usr/include/pcibar.h**に定義された不透明なポインタ型です。この値はその後の**bar_scan_next**, **bar_device_count**, **bar_scan_rewind, bar_scan_close**の呼び出しに提供される必要があります。

概要

```
#include <linux/pci_ids.h>
#include <pcibar.h>
bar_context bar_scan_open(int vendor_id, int device_id);
```

引数は以下のように定義されます：

vendor_id **/usr/include/linux/pci_ids.h**に定義されたベンダーIDの値または特殊な値ANY_VENDOR。ANY_VENDORはホストシステム上の全てのデバイスに対する全ての*vendor_id* の値と適合します。

device_id **/usr/include/linux/pci_ids.h**に定義されたデバイスIDの値または特殊な値ANY_DEVICE。ANY_DEVICEはホストシステム上の全てのデバイスに対する全ての*device_id* の値と適合します。

エラー状態についてはmanページを参照して下さい。

bar_scan_next(3)

この機能は、検出した次に一致するPCIデバイスの`pci_device`構造体オブジェクトへのポインターを返します。

概要

```
#include <linux/pci_ids.h>
#include <pcibar.h>
struct pci_device * bar_scan_next(bar_context ctx);
```

引数は以下のように定義されます：

ctx **bar_scan_open**より返されるアクティブな`bar_context`。

これ以上利用可能なデバイスが一致しない時、この機能はNIL_PCI_DEVICEを返し、*errno* にゼロを設定します。エラー状態についてはmanページを参照して下さい。

bar_device_count(3)

この機能は、アクティブ・スキャンの中に残っている未処理デバイスの数を返します。**bar_scan_open**または**bar_scan_rewind**の呼び出しの直後に呼び出した時、これは指定した *vendor_id* と*device_id* に対して一致するデバイスの総計となります。この値は**bar_scan_next**の呼び出しごとに1ずつへ減少します。

概要

```
#include <linux/pci_ids.h>
#include <pcibar.h>
int bar_device_count(bar_context ctx);
```

引数は以下のように定義されます：

ctx **bar_scan_open**より返されるアクティブな`bar_context`。

成功すると、この機能はその後の**bar_scan_next**の呼び出しによって返される報告されないデバイスの数を負ではない数で返します。エラー状態についてはmanページを参照して下さい。

bar_scan_rewind(3)

この機能は、最初の**bar_scan_open**の呼び出し後に直ぐの状況へ指定された`bar_context`をリセットします。

概要

```
#include <linux/pci_ids.h>
#include <pcibar.h>
void bar_scan_rewind(bar_context ctx);
```

引数は以下のように定義されます：

ctx **bar_scan_open**より返されるアクティブな`bar_context`。もし値がNIL_BAR_CONTEXTまたは有効な`bar_context`オブジェクトを指定しない場合、この呼び出しは効果がありません。

bar_scan_close(3)

この機能は、指定した`bar_context`に関連する割り当てられた全てのメモリを解放します。NIL_BAR_CONTEXTの値は`bar_context`オブジェクトに割り当てられ、この呼び出しの後はもう使用することはできません。

概要

```
#include <linux/pci_ids.h>
#include <pcibar.h>
void bar_scan_close(bar_context ctx);
```

引数は以下のように定義されます：

ctx **bar_scan_open**より返されるアクティブな`bar_context`。

free_pci_device(3)

この機能は、指定した`pci_device`構造体オブジェクトに関連する割り当てられた全てのメモリを解放します。

概要

```
#include <linux/pci_ids.h>
#include <pcibar.h>
void free_pci_device(struct pci_device * dev);
```

引数は以下のように定義されます：

dev **bar_scan_next**.から獲得した有効な`pci_device`構造体

bar_mmap(3)

この機能は、指定したBARファイルをメモリへマッピングするために使用することが可能です。これはmmapされた領域の先頭ではなくmmapされたBARデータの開始位置に小さなBARファイルを整列する**mmap(2)**のラッパーです。この機能を使いマッピングされたファイルをアンマップするためには**bar_munmap(3)**を使用してください。

概要

```
#include <linux/pci_ids.h>
#include <pcibar.h>
void * bar_mmap(char * barfilepath, void * start, size_t length, int prot, int flags,
int fd, off_t offset);
```

引数は以下のように定義されます：

barfilepath **mmap**するBARファイルのパス

他のパラメータの説明については**mmap(2)**を参照して下さい。

bar_munmap(3)

この機能は、**bar_mmap(3)**を使いマッピングされたファイルをアンマップするために使用する必要があります。

概要

```
#include <linux/pci_ids.h>
#include <pcibar.h>
int bar_munmap(void * start, size_t length);
```

パラメータの説明については**munmap(2)**を参照して下さい。

カーネル・スケルトン・ドライバ

デバイス・ドライバで処理される必要のある割り込みをデバイスが出すとき、Linuxではユーザー・レベル・ルーチンを割り込みに結合する方法がないため、完全にユーザー・レベルでデバイス・ドライバを構築することは出来ません。しかしながら、ユーザー・レベル・ドライバを実行中のユーザー・レベル・アプリケーションへデバイスの割り込みとシグナルの発行を処理する簡易カーネル・デバイス・ドライバを構築することは可能です。シグナルは実行プログラムへ非同期で配信されるため、およびシグナルはコードがクリティカル・セクション中はブロックすることが可能であるため、シグナルはユーザー・レベル割り込みのように振舞います。

後述のスケルトン・カーネルレベル・ドライバの例は、デバイス割り込みの発生とシグナルをトリガにする割り込みサービス・ルーチン用のコードへシグナルを結合する方法を示します。このスケルトン・ドライバの関する全てのコードは、RedHawkがインストールされたシステムの**/usr/share/doc/ccur/examples/driver**ディレクトリで見つけることが可能です。割り込み処理とユーザー・レベル・プロセスへのシグナル送信を行う簡易カーネルレベル・ドライバを記述するためのテンプレートとしてサンプル・ドライバ(**sample_mod**)を使用することが可能です。

サンプル・ドライバの機能の理解

サンプル・ドライバは、割り込みを生成するハードウェア・デバイスとしてリアルタイム・クロック(rtc)0を使用します。rtc0は、ConcurrentのReal-Time Clock and Interrupt Module (RCIM)上のリアルタイム・クロックの1つです。このクロックは、所定の分解能で0までカウントダウンし、その後初めからやり直します。カウントが0に到達する度に割り込みが生成されます。リアルタイム・クロック0用の設定の一部は、ドライバがデバイス・レジスタにアクセスする可能性があるため、それらのレジスタがメモリ空間へマッピングされるモジュールの「初期化」ルーチン内で実行されます。モジュールの「初期化」ルーチンとして示すコードの最後の部分は、割り込みベクタに割り込みルーチンを結合するコードです。

```
*****************************************************************************
int sample_mod_init_module(void)
{
...
    // find rcim board (look for RCIM II, RCIM I, and finally RCIM I old rev)
    dev = pci_find_device(PCI_VENDOR_ID_CONCURRENT, PCI_DEVICE_ID_RCIM_II,dev);
    if (dev == NULL) { //try another id
        dev = pci_find_device(PCI_VENDOR_ID_CONCURRENT_OLD, PCI_DEVICE_ID_RCIM, dev);
    }
    if (dev == NULL) { //try another id
        dev = pci_find_device(PCI_VENDOR_ID_CONCURRENT_OLD, PCI_DEVICE_ID_RCIM_OLD, dev);
    }
    if (dev == NULL) { //no rcim board, just clean up and exit
        unregister_chrdev(major_num,"sample_mod");
        return -ENODEV;
    }
...
    if ((bd_regs = ioremap_nocache(plx_mem_base, plx_mem_size)) == NULL)
        return -ENOMEM;
...
    if ((bd_rcim_regs = ioremap_nocache(rcim_mem_base, rcim_mem_size)) == NULL)
        return -ENOMEM;
...
    sample_mod_irq = dev->irq;
    res = request_irq(sample_mod_irq, rcim_intr, SA_SHIRQ, "sample_mod", &rtc_info);
```

rtc0デバイスの完全な初期化は、モジュールの“open” メソッドで実行されます。この例では、デバイスは割り込みがデバイスにより生成されるように自動的に設定されます。デバイスがオープンされる時、rtc0に関連する割り込みは有効となり、そのデバイスは1μ秒の分解能により10000から0へカウントするようにプログラムされ、クロックのカウントを開始します。カウントが0に達する時に割り込みを生成します。

```
*****************************************************************************
int rcim_rtc_open(struct inode *inode, struct file *filep)
{
    u_int32_t val;
    if (rtc_info.nopens > 0) {
        printk(KERN_ERR “You can only open the device once.¥n”);
        return -ENXIO;
    }
    rtc_info.nopens++;
    if (!rtc_info.flag)
        return -ENXIO;

    writel(0, &bd_rcim_regs->request);
    writel(ALL_INT_MASK, &bd_rcim_regs->clear);
    writel(RCIM_REG_RTC0, &bd_rcim_regs->arm);
    writel(RCIM_REG_RTC0, &bd_rcim_regs->enable);
    writel(RTC_TESTVAL, &bd_rcim_regs->rtc0_timer);//rtc data reg
    val = RCIM_RTC_1MICRO | RCIM_RTC_START|RCIM_RTC_REPEAT;
    writel(val, &bd_rcim_regs->rtc0_control);
    return 0;
}
*****************************************************************************
```

ユーザー・レベル・ドライバは、カーネルレベル・ドライバが割り込みを受信した時に送信されるべきシグナルを指定する必要があります。ユーザー・レベル・ドライバは、カーネルレベル・ドライバの**ioctl**メソッドにより処理される**ioctl()**呼び出しを行います。ユーザー・レベル・ドライバがこの**ioctl()**機能を呼び出すと、ユーザー・レベル・プロセスが指定したシグナルのためのシグナル・ハンドラを既に構成したカーネルレベル・ドライバに知らせ、ユーザー・レベル・ドライバは直ぐにシグナルを受信できるようになります。

呼び出し元のユーザー空間プロセスは、モジュールから受信したいシグナルの番号を指定します。ドライバは“current” 構造体を使用することにより要求されたシグナル番号に関連するプロセスIDを記憶します。「シグナルID/プロセスID」のペアは、モジュールの`rtc_info`構造体の中に格納され、その後、後述する“notification” メカニズムにより使用されます。

```
*****************************************************************************
int rcim_rtc_ioctl(struct inode *inode, struct file *filep, unsigned int cmd,
unsigned long arg)
{
    if (!rtc_info.flag)
        return (-ENXIO);

    switch (cmd)
    {
    // Attach signal to the specified rtc interrupt
    case RCIM_ATTACH_SIGNAL:
        rtc_info.signal_num = (int)arg;
        rtc_info.signal_pid = current->tgid;
        break;
    default:
        return (-EINVAL);
    }
    return (0);
}
*****************************************************************************
```

実際の通知はモジュールの割り込みハンドラ内で実施されます。割り込みをrtc0から受信した時、この割り込みサービス・ルーチンはそれを要求したプロセスへシグナルを送信するかどうかを判断します。もし`rtc_info`構造体内に「シグナルID/プロセスID」のペアが登録されている場合、指定されたシグナルは`kill_proc()`関数を使い対応するプロセスへ送信されます。

```
*********************************************************************************
int rcim_intr(int irq, void *dev_id, struct pt_regs *regs)
{
    u_int32_t isr;
    isr = readl(&bd_rcim_regs->request);
    writel(0, &bd_rcim_regs->request);
    writel(ALL_INT_MASK, &bd_rcim_regs->clear);

/* Use isr to determine whether the interrupt was generated by rtc 0 only if
“rcim” module is not built into the kernel. If “rcim” is active, its
interrupt handler would have cleared “request” register by the time we
get here. */

    // if (isr & RCIM_REG_RTC0) {
    // Send signal to user process if requested
        if (rtc_info.signal_num && rtc_info.signal_pid &&
        (kill_proc(rtc_info.signal_pid, rtc_info.signal_num, 1) == -ESRCH))
        {
            rtc_info.signal_pid = 0;
        }
    // }
    return IRQ_HANDLED;
}
*********************************************************************************
```

デバイスがクローズされた時、rtc0はシャット・ダウンされます。カウント値は0へリセットされ、クロックは停止されます。さらに割り込みを受信した場合にシグナルがこれ以上送信されないように割り込み/シグナルの結合はクリアされます。

```
*********************************************************************************
int rcim_rtc_close(struct inode *inode,struct file *filep)
{
    if (!rtc_info.flag)
        return (-ENXIO);
    rtc_info.nopens--;
    if(rtc_info.nopens == 0) {
        writel(~RCIM_RTC_START, &bd_rcim_regs->rtc0_control);
        writel(0, &bd_rcim_regs->rtc0_timer);
        rtc_info.signal_num = 0;
        rtc_info.signal_pid = 0;
    }
    return 0;
}
*********************************************************************************
```

ドライバのテスト

サンプル・カーネル・モジュールをテストする最良の方法は、RCIMドライバなしのカーネルを構築し、サンプル・ドライバをロードすることです。しかしながら、このモジュールはカーネルに既に組み込まれたRCIMドライバの有無に関わらず動くように設計されています。

RCIMカーネル・モジュールとサンプル・カーネル・モジュールは同じ割り込みラインを共有します。割り込みが発生した時、RCIMの割り込みハンドラが最初に起動し、RCIM上のハードウェア割り込みレジスタはクリアされます。その後、サンプル・モジュールの割り込みハンドラが呼び出されます。

もし両方のモジュールがロードされた場合、もう1つのハンドラはクリアされた割り込みレジスタを見つけ、もし「割り込みソース」のチェックが実行されるとハンドラは割り込みがrtc0とは異なるデバイスから来たと思い込みます。RCIMとサンプル・モジュールの両方がロードされる時にこの障害を克服するため、サンプル・モジュールの割り込みハンドラの以下の行をコメントアウトしました：

```
// if (isr & RCIM_REG_RTC0) { .
```

次のコードは、RCIMスケルトン・ドライバの割り込み発生でいつでもこのルーチンが呼び出されるようにどのような方法でユーザー・レベル・ドライバをルーチンと結合するかをデモする簡易ユーザー・レベル・プログラムです。このルーチン“interrupt_handler” は、RCIMのrtc0の割り込み発生時に呼び出されるルーチンです。このプログラムはプログラムが実行されている端末で「Ctrl-C」の入力することにより終了します。このサンプル・コードは **/usr/share/doc/ccur/examples/driver/usersample**でも入手できることに注意してください。

サンプル・モジュールをロードして正常にユーザー・サンプル・プログラムを実行するには、RCIMドライバを使用する全てのアプリケーションを停止する必要があります。

以下が**usersample**プログラムです。

```
#include <stdio.h>
#include <fcntl.h>
#include <signal.h>
#include <errno.h>
#include "sample_mod.h"

static const char *devname = "/dev/sample_mod";
static int nr_interrupts = 0;
static int quit = 0;

void interrupt_handler (int signum)
{
    nr_interrupts++;
    if ((nr_interrupts % 100) == 0) {
    printf (".");
    fflush(stdout);
}

if ((nr_interrupts % 1000) == 0)
    printf (" %d interrupts¥n", nr_interrupts);
}

void ctrl_c_handler (int signum)
{
    quit++;
}

int main()
{
    int fd;
    struct sigaction intr_sig = { .sa_handler = interrupt_handler };
    struct sigaction ctrl_c_sig = { .sa_handler = ctrl_c_handler };

    sigaction (SIGUSR1, &intr_sig, NULL);
    sigaction (SIGINT, &ctrl_c_sig, NULL);

    if ((fd = open (devname, O_RDWR)) == -1 ) {
        perror ("open");
        exit(1);
    }

    if (ioctl (fd, RCIM_ATTACH_SIGNAL, SIGUSR1) == -1) {
        perror ("ioctl");
        exit(1);
    }

    printf ("waiting for signals...¥n");
    while (! quit)
        pause();

    printf ("¥nhandled %d interrupts¥n", nr_interrupts);
    close(fd);
    exit(0);
}
```

カーネル・レベル・ドライバの開発

後に続くセクションで、カーネル・レベル・ドライバの記述とテストに影響するRedHawk Linuxオペレーティング・システムの詳細について説明します。

ドライバ・モジュールの構築

既存のRedHawkカーネルまたはカスタム・カーネルのどちらかで使用するドライバ・モジュールの構築に関する説明は、11章の「カーネルの構成および構築」で提供されます。

カーネルの仮想アドレス空間

カーネル・サポート・ルーチン**vmalloc()**と**ioremap()**の動的マッピングが引き当てたカーネル仮想アドレス空間の量が、デバイスの要求に対応するには十分ではない時にいくつかのケースが存在します。32bitカーネルののデフォルト値128MBは、**ioremap**されることになるとても大きなオンボード・メモリを持つI/Oボードを除く全てのシステムに対しては十分です。一例は、128MBメモリが搭載されるiHawkシステムにインストールされたVMICのリフレクティブ・メモリ・ボードです。

128MBの予約カーネル仮想アドレス空間が十分ではない時、この値はブート時に指定されるカーネル・ブート・パラメータ(vmalloc=)を使うことにより増やすことが可能となります。このオプションに関する詳細な情報は、付録-H「ブート・コマンド・ライン・パラメータ」を参照して下さい。

リアルタイム性能の問題

カーネルレベル・デバイス・ドライバはカーネル・モードで実行され、カーネル自身の拡張です。従ってデバイス・ドライバは、リアルタイム性能に影響を与える可能性のあるカーネル・コードと同様にシステムのリアルタイム性能に影響を及ぼす能力を持っています。後に続くセクションで、デバイス・ドライバとリアルタイムに関連するいくつかの問題のハイレベルな概要を提供します。

Linuxで利用可能な多くのオープン・ソース・デバイス・ドライバが存在する一方、それらのドライバは、特にリアルタイム・システムに対する適合性に関しては広範囲にわたる品質が存在することに注意する必要があります。

割り込みルーチン

割り込みルーチンは高優先度タスクを実行するためにプリエンプトできないため、割り込みルーチンの継続時間はリアルタイム・システムではとても重要です。非常に長い割り込みルーチンは割り込みが割り当てられているCPU上で実行しているプロセス・ディスパッチ・レイテンシーに直接影響を与えます。用語「*プロセス・ディスパッチ・レイテンシー*」は、割り込みにより示される外部イベントの発生から、外部イベントを待っているプロセスがユーザー・モードで最初の命令を実行するまでの経過時間を意味します。割り込みがプロセス・ディスパッチ・レイテンシーに影響を与える方法の詳細については、「リアルタイム性能」章を参照して下さい。

もしリアルタイム製品環境でデバイス・ドライバを使用している場合は、割り込みレベルで実行される仕事量を最小限に抑える必要があります。RedHawk Linuxは、割り込みレベルで実行される必要のない処理を遅らせるためのいくつかの異なるメカニズムをサポートしています。これらのメカニズムは、プログラム・レベルでカーネル・デーモンのコンテキストで実行される処理をトリガーすることを割り込みルーチンに認めます。これらのカーネル・デーモンの優先度は変更可能であるため、延期される割り込み処理よりも高い優先度レベルで高優先度リアルタイム・プロセスを実行することが可能です。これはリアルタイム・プロセスが、通常割り込みレベルで実行される可能性のあるいくつかの活動よりも高い優先度になることを許可します。このメカニズムを使用することで、リアルタイム・タスクの実行は延期された割り込み動作により遅延することはありません。割り込みの遅延に関する詳細については「割り込み機能の遅延(ボトム・ハーフ)」セクションを参照して下さい。

通常、デバイスの割り込みルーチンは、以下のタイプのタスクを実行するためにデバイスと相互作用することが可能です。

- 割り込みに応答
- その後ユーザーへ転送するためデバイスから受信したデータを保存
- 前の操作の完了を待っているデバイス操作を開始

デバイスの割り込みルーチンは以下のタイプのタスクを実行してはいけません。

- ある内部バッファから他へデータをコピー
- デバイスへバッファを割り当てまたは補充
- デバイスに使用されているほかのリソースを補充

これらのタイプのタスクは、遅延割り込みメカニズムの1つを介してプログラム・レベルで実行する必要があります。例えば、デバイスのためのバッファをプログラム・レベルで割り当てて、ドライバへの内部フリー・リスト上に保持されるようにデバイス・ドライバを設計することが可能です。プロセスが読み書き操作を実行する時、利用可能なバッファの数が入ってくる割り込みトラフィックに対して十分であるかどうかを判断するためにドライバはフリー・リストをチェックします。割り込みルーチンは、実行時間の面ではとても高くつくカーネル・バッファ・割り当てルーチンの呼び出しをこのようにして回避することが可能です。デバイスがリソースを使い果たして割り込みレベルでこれを通知するだけである場合、新しいリソースは割り込みレベルではなく遅延割り込みルーチンの一部として割り当てられる必要があります。

割り込み機能の遅延(ボトム・ハーフ)

Linuxは機能の実行を遅らせることが可能ないくつかのメソッドをサポートしています。直接機能を呼び出す代わり、その後に機能が呼び出されるように「トリガ」が設定されます。ボトム・ハーフと呼ばれるこれらのメカニズムは、割り込みレベルで行われた処理を遅延するためにLinux下の割り込みルーチンによって使用されます。割り込みレベルからこの処理を削除することにより、システムは上述されているようにより良い割り込み応答時間を実現することが可能となります。

割り込みを遅延するためにソフトIRQ、タスクレット、ワーク・キューの3つの選択が存在します。タスクレットはソフトIRQ上に構築されており、従ってそれらの動作はよく似ています。ワーク・キューは異なった動作でカーネル・スレッド上に構築されます。ボトム・ハーフを使用する上での判断は重要です。表14-1は、以降のセクションで詳細に説明されているタイプの要約です。

表14-1 ボトム・ハーフのタイプ

ボトム・ハーフ・タイプ	コンテキスト	シリアル化
ソフトIRQ	割り込み	なし
タスクレット	割り込み	同じタスクレットに対して
ワーク・キュー	プロセス	なし(プロセス・コンテキストとしてスケジュール)

Softirqs and Tasklets

割り込み処理を遅延するための2つのメカニズムは、遅延されるコードが再入可能である必要があるかどうかについては異なる要件があります。これらの遅延機能のタイプはソフトIRQとタスクレットです。ソフトIRQの単一インスタンスは同時に複数のCPU上で実行可能であるため、「*ソフトIRQ*」は完全に再入可能である必要があります。「*タスクレット*」はソフトIRQの特殊なタイプとして実装されます。この違いは特定のタスクレット機能は常にそれ自身に対してシリアライズ(順番に並べられる)されるということです。言い換えると、2つのCPUは同時に同じタスクレットを決して実行しません。タスクレット内のコードはそれ自身に対して再入可能である必要がないため、この特性はデバイス・ドライバにおいてよりシンプルなコーディング・スタイルを可能にします。

標準Linuxにおいて、ソフトIRQとタスクレットは通常、割り込みからプログラム・レベルへの割り込みハンドラ移行の直後に割り込みコンテキストから実行されます。時折、標準Linuxはカーネル・デーモンにソフトIRQとタスクレットを譲ります。両方のメソッドは割り込みを有効にして実行することをソフトIRQとタスクレットに許可しますが、これらは通常割り込みコンテキストから実行されるため、ソフトIRQとタスクレットはsleepできません。

RedHawkは、ソフトIRQとタスクレットがカーネル・デーモンのコンテキスト内で実行されることを保証するオプション(デフォルトでON)により機能強化されました。これらのカーネル・デーモンの優先度とスケジューリングのポリシーはカーネル構成パラメータを介して設定することが可能です。これは、高優先度リアルタイム・タスクが遅延された割り込み機能の動作をプリエンプトすることが可能になるように構成することをシステムに許可します。

ソフトIRQとタスクレットは**ksoftirqd**デーモンにより両方実行されます。これは論理CPU毎に1つの**ksoftirqd**デーモンが存在します。ソフトIRQまたはタスクレットはこの実行をトリガーにしたCPU上で実行されます。従って、もしハード割り込みが特定のCPUへのアフィニティ・セットを持っている場合、対応するソフトIRQまたはタスクレットはそのCPU上でも実行されます。**ksoftirqd**の優先度は、カーネル構成GUIの「General Setup」項目にあるSOFTIRQ_PRIとSOFTIRQ_PREEMPT_BLOCKカーネル・チューニング・パラメータにより決定されます。SOFTIRQ_PRIに正の数が設定される時、その数は**ksoftirqd**が実行される優先度になります。既定値ではこのチューニング・パラメータはゼロに設定され、SOFTIRQ_PREEMPT_BLOCKの設定はデーモンの優先度に影響します。「Y」に設定すると、最高リアルタイム優先度よりも1つ小さい優先度でSCHED_FIFOスケジューリング・ポリシーとして**ksoftirqd**デーモンが実行されます。「N」に設定すると、**ksoftirqd**デーモンは優先度ゼロで実行されます。

Work Queues

「*ワーク・キュー*」はもう1つの遅延実行メカニズムです。ソフトIRQとタスクレットとは異なり、標準Linuxは常にカーネル・デーモンのプロセス・コンテキスト内でワーク・キューが処理される結果、ワーク・キュー内のコードはsleepが許可されています。

ワーク・キューを処理するカーネル・デーモンはワーカー・スレッドと呼ばれます。ワーカー・スレッドは常に単一CPUへバインドされた各スレッドとCPU毎に一組のスレッドとして作成されます。ワーク・キュー上の仕事はCPU毎に保持され、そのCPU上のワーカー・スレッドとして処理されます。

カーネルはデフォルトでドライバを使用する可能性のあるワーク・キューを提供します。デフォルトでワーク・キューを処理するワーカー・スレッドは**events/***cpu* と呼ばれ、*cpu* はスレッドがバインドされているCPUです。

任意にドライバはプライベート・ワーク・キューとワーカー・スレッドを作成する可能性があります。これはキューイングされた仕事がプロセッサ負荷が高いまたはパフォーマンスが重要である場合、ドライバに有利となります。これはデフォルト・ワーカー・スレッドの負荷も軽減し、デフォルト・ワーク・キューの他の仕事がなくなるのを防ぎます。

ワーカー・スレッドは、ワーク・キュー上に仕事がセットされた時にCPU上で実行します。従って、ハード割り込みが特定CPUへのアフィニティ・セットを持ち、割り込みハンドラが仕事をキューイングした場合、対応するワーカー・スレッドもそのCPU上で実行されます。通常のワーカー・スレッドはナイス値0で作成され、高優先度ワーカー・スレッドはナイス値-20で作成されますが、その優先度は**run(1)**コマンドで変更することが可能です。

Understanding Priorities

リアルタイム・プロセスが遅延割り込みデーモンよりも高い優先度で実行することが可能なシステムを構成する時、それらのリアルタイム・プロセスがデーモンより提供されるサービスに依存するかどうかを理解することが重要です。もし高優先度リアルタイム・タスクが遅延割り込みデーモンよりも高いレベルでCPUにバインドされた場合、CPU実行時間を受信しないようにデーモンを空にすることが可能です。もしリアルタイム・プロセスも遅延割り込みデーモンに依存する場合、デッドロックが生じる可能性があります。

マルチスレッディングの問題

RedHawk Linuxは単独のシステムで複数CPUをサポートするために構築されています。これは、全てのカーネル・コードとデバイス・ドライバがそれらのデータ構造体を1つ以上のCPUで同時に変更されることから保護するために記述されている必要があることを意味します。データ構造体への全ての変更がシリアル化されるようにマルチ・スレッド・デバイス・ドライバの処理はそれらのデータへのアクセスの保護を必要とします。一般的にLinuxではこれらの種類のデータ構造体アクセスを保護するためにスピンロックを使用することにより実現されます。

スピンロックをロックすることは、プリエンプションおよび/または割り込みが無効になる原因となります。どちらのケースでも、これらの機能が無効であるCPU上で実行中のプロセスにとってプロセス・ディスパッチ・レイテンシーの最悪のケースは、どれくらいそれが無効であるかによって直接影響を受けます。それは、プリエンプションおよび/または割り込みが無効である時間に影響するスピンロックが保持される時間を最小化するためにデバイス・ドライバを記述する時に重要となります。スピンロックをロックすることは暗黙のうちにプリエンプションまたは割り込みが(スピンロック・インターフェースの使用に応じて)無効になる原因となることを覚えてください。このトピックに関する詳細については「リアルタイム性能」章を参照して下さい。

ユーザー空間I/Oドライバ(UIO)

UIOはユーザー・レベル・ドライバを記述するために標準化されたメソッドです。これはやはり小さなドライバ単位のカーネル・モジュールを必要としますが、ドライバの主要部分は使い慣れたツールやライブラリを使用してユーザー空間で記述します。

UIOを使用すると、標準的なPCIカードの取り込みや任意の目的のために簡単なユーザー空間ドライバを作ることが可能となります。これらのドライバは実装やテストが容易でありカーネルのバージョン変更から分離されています。そのドライバのバグはカーネルをクラッシュすることはなく、ドライバのアップデートはカーネルの再コンパイルなしに行うことが可能です。

現在、UIOドライバはキャラクタ・デバイス・ドライバだけに使用することが可能でユーザー空間からDMA操作を提供するために使用することは出来ません。

小さなドライバ単位のカーネル・モジュールは次が必要となります：

- ボードのデバイスIDとベンダーIDが一致
- 低レベルでの初期化を実行
- 割り込みの応答

一旦所有ハードウェア用に動作するカーネル・モジュールを所有してしまえば、ユーザー･アプリケーションを記述するために通常使用されるツールやライブラリを使用してユーザー空間ドライバを記述することが可能となります。**lsuio(1)**ツールはUIOデバイスとその属性をリストアップするために使用することが可能です。

各UIOデバイスはデバイスファイル**/dev/uio0**, **/dev/uio1**などを介してアクセスします。変数の読み書きをするために使用されるドライバの属性は、**/sys/class/uio/uio***X*ディレクトリの下にあります。メモリ領域は**mmap(1)**を介してサクセスされます。

UIOデバイス・ドライバを記述するための完全な説明書は
/usr/share/doc/ccur/examples/driver/uio/uio-howto.pdfで見ることが可能です。

ConcurrentのRCIMボードとPMC-16AIOボード両方のためのUIOカーネルとユーザー・ドライバの例は、**/usr/share/doc/ccur/examples/driver/uio**で提供されています。両方ともドライバがどのような機能を実行するかを説明するコメントを含みます。

RedHawkは、カーネル構成GUIの「Userspace I/O」項目にあるUIOカーネル・チューニング・パラメータを通してプレビルトカーネルの中でデフォルトでUIOサポートが有効となっています。

性能の解析

Concurrentが提供するグラフィカル解析ツールのNightTrace RTは、アプリケーションやカーネル内の重要なイベントに関する情報をグラフィカルに表示することが可能で、そしてアプリケーションの動作でパターンや例外を特定するために使用することが可能です。変化する状況下でコードを対話的に分析するための能力は、デバイス・ドライバのリアルタイム性能をチューニングするために非常に有益です。

ユーザー・レベル・コードのトレース・ポイントの提供、トレース・データのキャプチャー、結果表示の処理は「*NightTrace RT User's Guide* (文書番号：0890398) 」の中で全て説明されています。ユーザー/カーネルのトレース・イベントは、解析するために記録および表示することが可能です。

カーネル・トレースは、トレース・カーネルおよびデバッグ・カーネルの中に含まれている事前に定義されたカーネル・トレース・イベントを利用します。ユーザー定義イベントは事前に定義されたCUSTOMトレース・イベントを使用して記録する、または動的に作成することが可能です。全ては解析のためにNightTrace RTにより表示されます。カーネルのトレース・イベントに関する詳細については付録Dを参照して下さい。

15
PCI-to-VMEサポート

本章では、RedHawk Linuxがサポートする PCI-to-VMEバス・ブリッジについて説明します。

概要

PCI-to-VMEバス・アダプターは、PCIベースのiHawkシステムとVMEバス・システムを接続するために使用することが可能です。これは、あたかもiHawkのPCIバックプレーンに直接装着されたかのように全VMEメモリ空間への透過的なアクセス、VMEカードへの割り込みレベルでの制御や応答を可能にします。

RedHawk Linuxは、SBS Technologies社のPCI-to-VMEバス・アダプター Model 618-3と620-3のサポートを含みます。このアダプターを使用することで、メモリが2つのシステム間で共有されます。メモリ・マッピングとダイレクト・メモリ・アクセス(Direct Memory Access : DMA)の2つのメソッドが利用されています。メモリ・マッピングはどちらのシステムからの双方向ランダム・アクセス・バス・マスタリングをサポートします。これはVMEバスRAM、デュアルポート・メモリ、VMEバスI/Oへのプログラム I/Oアクセスを可能にします。各システム上のバス・マスターは、それぞれのアドレス空間内のウィンドウから他のシステムのメモリにアクセスすることが可能です。マッピング・レジスタは、PCIデバイスが最大32MBのVMEバスのアドレス空間へのアクセス、VMEバス・デバイスが最大16MBのPCI空間へのアクセスを可能にします。

コントローラ・モードDMAとスレーブ・モード・DMAの2つのDMA技術がサポートされています。コントローラ・モードDMAは、あるシステムのメモリから直接他のシステムのメモリへの高速データ転送を提供します。データ転送はどちらのプロセッサでも最大35MB/秒および最大16MB/転送の速度により両方向で開始することが可能です。

自身のDMAコントローラを持つVMEバス・デバイスは、コントローラ・モードDMAの代わりにスレーブ・モードDMAを使用することが可能です。これはVMEバス・デバイスが15MB/秒を越えるデータ転送速度で直接PCIメモリへデータ転送することを可能にします。

アダプターは、PCIアダプター・カード、VMEバス・アダプター・カード、光ファイバー・ケーブルの3つのパーツで構成されます。

PCIアダプター・カードはブート時に自分自身で構成します。A32メモリとI/Oアクセスに対応および生成し、D32, D16, D8のデータ幅をサポートします。

VMEバス・アダプター・カードはジャンパーを介して構成されます。VMEバス・アダプター･カードはA32, A24, A16アクセスに対応および生成し、D32, D16, D8のデータ幅をサポートします。

アダプターをサポートするソフトウェアは、RedHawk Linux下で実行および最適化のために改良されたSBS Linuxモデル 1003 PCIアダプター・サポート・ソフトウェアVer.2.2を含みます。このソフトウェアはデュアル・ポートおよび/またはアプリケーションからリモート・メモリ空間をアクセスすることが可能なデバイス・ドライバ、アダプターおよびシステム構成と共にアプリケーション・プログラマーに役立つプログラム例を含みます。

文書

本章ではRedHawk下で本サポートを構成および使用するために必要な情報を提供します。

本章の範囲を超える情報については、RedHawk Linuxの文書に含まれている以下の文書を参照して下さい：

- *SBS Technologies Model 618-3, 618-9U & 620-3 Adapters Hardware Manual* (**sbs_hardware.pdf**)
- *SBS Technologies 946 Solaris, 965 IRIX 6.5, 983 Windows NT/2000, 993 VxWorks & 1003 Linux Support Software Manual* (**sbs_software.pdf**)

ハードウェアのインストール

アダプターは、PCIアダプター・カード、VMEバス・アダプター・カード、光ファイバー・ケーブルの3つのパーツで構成されます。それらをインストールするための手順を以下のとおりです。

通常、ハードウェアの取り付けと構成はConcurrent社により行われます。この情報は、PCI-to-VMEブリッジが製造後のシステムへ追加されるような状況のために提供されます。

開梱

輸送箱から装置を開梱するとき、内容明細書を参照し全てのアイテムがあることを確認して下さい。梱包材料は装置の保管および再出荷のために残しておいて下さい。

NOTE

もし輸送箱が受領時に破損している場合、開梱および装置の検品中は運送業者が立ち会うよう要請して下さい。

システムにカードを取り付けようとする前に以下を読んでください：

CAUTION

静電気の放電が回路に損害を与える可能性があるため、集積回路面に触れることは避けて下さい。

プリント基板を取り付けおよび取り外す時は静電気防止のリスト・ストラップと導電フォーム・パッドを使用することを強く推奨します。

アダプター・カードの設定

PCIアダプター・カード上に設定するためのジャンパーはありません。

VMEアダプター・カードのジャンパーの設定は、VMEアダプター・カードを取り付ける前、またはVMEアダプターカードのジャンパーにより制御されているVMEバス属性の現在の設定を変更する必要になった時に行う必要があります。

VMEバス・アダプター・カードの構成に関する情報については、「SBSテクノロジー・ハードウェア・マニュアル」の10章を参照して下さい。以下の追加情報は役に立つかもしれません：

- このVMEアダプター・カードがスロット1でシステム・コントローラとして、または他のVME
  スロットで非システム・コントローラとして使用されているのかどうかに基づき、システムのジャンパーは適切に設定される必要があります。

-
  VMEバス上のデバイスにVMEスレーブ・ウィンドウを通してiHawkシステムのメモリへアクセスさせるbt_bind()バッファ・サポートもしくはローカル・メモリ・デバイス・サポート(BT_DEV_LM)を使用するために、リモートREM-RAM HIおよびLOジャンパーはVMEバス上のVMEバス・ベース・アドレスとVMEスレーブ・ウィンドウ出力の範囲を知らせるために設定する必要があります。

  ベース・アドレスは16MB境界に置く必要があり、そしてこの領域のサイズはSBSハードウェア(例えば、0xC0000000から0xC1000000の範囲のA32アドレスを設定するためジャンパーを以下の設定に構成する必要があります)でサポートされている領域の総量を利用するために一般的には16MB(を超えないサイズ)に設定される必要があります。

  A32アドレス範囲を設定するため、REM-RAMの下部のジャンパーは次のように設定する必要があります：

  A32ジャンパー：IN
  A24ジャンパー：OUT

  開始アドレス0xC0000000を指定するため、LOアドレスREMRAMジャンパーの列は次のように設定する必要があります：

  31, 30ジャンパー：OUT
  他全てのLOジャンパー：IN (16-29)

  終了アドレス0xC1000000を指定するため、HIアドレスREMRAMジャンパーの列は次のように設定する必要があります：

  31, 30, 24ジャンパー：OUT
  他全てのHIジャンパー：IN (29-25, 23-16)

PCIアダプター・カードのインストール

お手持ちのiHawkシステムにPCIアダプターを取り付けるために以下の手順を使用して下さい：

1. iHawkシステムがパワー・ダウンされていることを確認して下さい。
2. バス・マスターをサポートする筐体内の空いているPCIカード・スロットを確認します。
3. 筐体背面のケーブル出口を覆う金属板を取り外します。
4. コネクタにPCIアダプター・カードを差し込みます。
5. アダプター・カードを取り付けねじで所定の位置に固定します。
6. カバーを元の位置に戻します。

VMEバス・アダプター・カードのインストール

NOTE

VMEバス・バックプレーンはデイジー・チェーン、バス・グラント、未使用カード一周辺の割り込みACK信号を接続するためのジャンパーを持っています。これらのジャンパーはアダプター・カードが差し込まれるスロットから取り外されていることを確認して下さい。

1. VMEバス筐体がパワー・ダウンされていることを確認して下さい。
2. VMEバス・アダプター・カードがシステム・コントローラかどうかを決定します。もしVMEバス・アダプター・カードがシステム・コントローラの場合、スロット1へ差し込む必要があります。

   もしアダプター・カードがシステム・コントローラではない場合、そのアダプター用にVMEバス・カード・ケージで未使用の6Uスロットを決めて下さい。

3. 選択したスロットのコネクタへカードを差し込みます。

アダプター・ケーブルの接続

NOTE

光ファイバー・ケーブルの端はきれいな状態にしておいて下さい。塵や埃のような小さな汚染物質を取り除くためにアルコール・ベースの光ファイバ・ワイプを使用して下さい。

光ファイバー・ケーブルはガラスで作られていますので、半径2インチ以下のループに潰すまたは曲げた場合はそれらは破損する可能性があります。

1. iHawkコンピュータ・システムとVMEバス筐体がパワーOFFであることを確認して下さい。
2. 光ファイバー・トランシーバのゴム製ブーツ、同様に光ファイバー・ケーブルのそれも取り外します。ケーブルが使用されていない時はそれらのブーツを確実に元に戻して下さい。

3. 光ファイバー・ケーブルの一端をPCIアダプター・カードのトランシーバへ接続します。
4. 光ファイバー・ケーブルのもう片方をVMEバス・アダプター・カードのトランシーバへ接続します。
5. PCIとVMEバス・システムの両方の電源を入れて下さい。
6. 両アダプター・カードのREADYのLEDが点灯していることを確認して下さい。アダプターを操作するためにON である必要があります。

ソフトウェアのインストール

ソフトウェアはRedHawk Linuxと一緒に納品されるオプションのプロダクトCDに収納されています。これは**install-sbsvme**インストール・スクリプトを使いインストールします。

ソフトウェアをインストールするため、以下の手順を実行します：

1. RedHawkバージョン2.1以降が動作しているiHawkシステム上に、ルートでログインし、シングル・ユーザー・モードへシステムをダウンして下さい：

   a. デスクトップ上で右クリックし、「New Terminal」を選択します。
   b. システム・プロンプトで「**init 1**」と入力します。

2. “RedHawk Linux PCI-to-VME Bridge Software Library” というラベルのディスクを見つけ、CD-ROMドライブへ挿入します。

3. CDROMデバイスをマウントするため、以下のコマンドを実行します：

   NOTE: 以下の例では**/media/cdrom**が使用されています。お手持ちのシステムに取り付けられたドライブの型式に応じて、実際のマウント・ポイントは異なる可能性があります。正しいマウント・ポイントについては**/etc/fstab**を確認して下さい。

   ```
   mount /media/cdrom
   ```

4. インストールするため、以下のコマンドを実行して下さい：

   ```
   cd /media/cdrom
   ./install-sbsvme
   ```

   インストール・スクリプトが完了するまで画面上の指示に従ってください。

5. インストールが完了したら、以下のコマンドを実行して下さい：

   ```
   cd /
   umount /media/cdrom
   eject
   ```

6. CD-ROMドライブからディスクを取り出し、保管して下さい。シングル・ユーザー・モードを抜けます(Ctrl-D)。

構成

後述のセクションでRedHawk Linux下のモジュール構成およびシステム初期化時に確立される可能性のある他の属性について説明します。

btpモジュール

事前に定義されたRedHawkカーネルは、デフォルトのモジュールとして構成されたSBSテクノロジーのPCI-to-VMEバス・ブリッジを持っています。もし望むのであれば、カーネル構成GUI上の「Device Drivers -> SBS VMEbus-to-PCI Support」項目においてSBSVMEオプションを通してこれを無効にすることが可能です。このモジュールは"btp"と呼ばれています。

デバイス・ファイルおよびモジュール・パラメータ仕様

/dev/btp*デバイス・ファイルは、**/etc/init.d/sbsvme**を介して初期化時に作成されます。これらのファイルの属性は、**/etc/sysconfig/sbsvme**の中で定義されています。更に、以下のモジュール・パラメータの仕様はこのファイルで作ることが可能です。既定値ではパラメータはありません。

`btp_major=`*num*
: メジャー・デバイス番号(*num*)を指定します。デフォルトは、カーネルが選択することが可能な0(ゼロ)となります。もしゼロ以外のデバイス番号を提供する場合、それは既に使用中であってはいけません。**/proc/devices**ファイルは、どのデバイスが現在使用されているかを判断するために調べることが可能です。

`icbr_q_size=`*size*
: 割り込みキューに割り当てられるICBRエントリの数(*size*)を指定します。一旦設定すると、この値はbtpドライバのアンロードおよびリロードなしに変更することは出来ません。既定値は割り込みキュー空間から1KBです。

`lm_size=`*size1, size2, ...*
: システムに存在する各SBS PCI-to-VMEコントローラ(unit)に対しローカル・メモリ(BT_DEV_LM)・サイズの配列をバイトで指定します。もしこの値に0(ゼロ)が設定された場合、ローカル・メモリはそれを指定したユニットだけ無効にされます。既定値はローカル・メモリから64KBで最大値が4MBとなります。詳細については本章の「ローカル・メモリ」セクションを参照して下さい。

`trace=`*flag_bits*
: デバイス・ドライバのトレース・レベルを指定します。これはどのトレース・メッセージをbtpドライバが表示するかを制御するために使用されます。使用可能なビットは**/usr/include/btp/btngpci.h**にあるBT_TRC_*xxx*の値です。トレースは性能に影響を及ぼすため、この機能はbtpドライバの問題をデバッグするためだけに使用すべきです。既定値はトレース・メッセージなしの0(ゼロ)です。

以下はbtpモジュール・パラメータ仕様の例です：

```
BTP_MOD_PARAMS='bt_major=200 trace=0xff lm_size=0'
BTP_MOD_PARAMS='icbr_q_size=0x1000 lm_size=0x8000,0x4000'
```

VMEバス・マッピング

PCI-to-VMEバス・マッピングの自動的な作成および削除のサポートは**/etc/init.d/sbsvme**初期化スクリプトに含まれています。マッピングが**/etc/sysconfig/sbsvme-mappings**に定義されている場合、“/etc/init.d/sbsvme start”の処理中に作成され、“stop”の処理中に削除されます。

/etc/sysconfig/sbsvme-mappingsファイルはVMEバス・マッピング作成のためのヘルプ情報とコマンド出力テンプレートを含みます。必要であれば、テンプレートの例はカスタマイズされたVMEバス・マッピングを作成するために使用することが可能です。**sbsvme-mappings**ファイル内のコメントおよび本章で後述する「/procファイル・システム・インターフェース」セクションで説明されている**/proc/driver/btp/***unit***/vme-mappings**ファイルに書き込まれている値により、マッピングは作成されます。

システム初期化中にPCI-to-VMEバス・マッピングを作成するために**sbsvme-mappings**ファイルを使用することで、VMEバス空間へバインドするグローバル・ビジブル共有メモリ領域を作成する**shmconfig(1)**を呼び出すために**/etc/rc.d/rc.local**スクリプトへ追加の行をセットすることが可能です。これを説明するサンプル・スクリプトが提供されています。詳細については「アプリケーション例」セクションを参照して下さい。

ユーザー・インターフェース

標準サポートソフトウェアへのいくつかの修正がRedHawk Linux用に行われました。インストールの変更に加え、以下が追加されました。

- 複数の様々なサイズのバッファのバインドをサポート。複数のユーザー・レベル・ドライバを持つシステムで、この機能は各ドライバが複数のデバイス間で共通のバインド・バッファを共有する代わりにそれぞれのバインド・バッファを割り当てることを可能にします。この機能は複数の大きなバインド・バッファ(ハードウェアでサポートされているVMEバス・スレーブ・ウィンドウ空間から合計16MBの領域)を割り当てることにより利用できることも意味します。詳細については「バインド・バッファの実装」セクションを参照して下さい。プログラム例は、VMEバス空間へ複数のバッファの割り当ておよびバインドする手順が追加されています(「アプリケーション例」セクションを参照して下さい)。

- 特定のプロセスと結びついていないVMEバス空間マッピングの作成と削除、および共有メモリのバインドを許可するためにそのマッピングのPCIバス・アドレス開始位置の取得をサポート。これは次の2つのいずれかで達成することが可能です：

  - bt_hw_map_vme/bt_hw_unmap_vmeライブラリ関数の使用
  - to the **/proc/driver/btp**ファイル・システムへの書き込み

  詳細については「VMEバス空間へのマッピングおよびバインド」セクションを参照して下さい。プログラム例は、両方の方法を使いVMEバス・マッピングの作成、表示、削除の手順を示しています(「アプリケーション例」セクションを参照して下さい)。

API機能

表15-1は**libbtp**ライブラリに含まれているAPI関数を記載しています。修正されたもしくは追加された関数は後に続くセクションで言及および説明されています。残りの関数はRedHawk Linuxの文書に含まれているSBSテクノロジー・ソフトウェアのマニュアルに記載されています。

表15-1 PCI-to-VMEライブラリ関数

関数	概要
bt_str2dev	文字列から論理デバイスへ変換
bt_gen_name	デバイス名を生成
bp_open	アクセス用に論理デバイスをオープン
bt_close	論理デバイスをクローズ
bt_chkerr	ユニット上のエラーをチェック
bt_clrerr	ユニット上のエラーをクリア
bt_perror	エラー・メッセージをstderrに出力
bt_strerror	エラー・メッセージの文字列を作成
bt_init Initialize	ユニットの初期化
bt_read	論理デバイスからデータの読み取り
bt_write	論理デバイスへデータの書き込み
bt_get_info	デバイスの構成設定を取得(**以下のNote 1を参照**)
bt_set_info	デバイスの構成設定を設定(**以下のNote 1を参照**)
bt_icbr_install	割り込みコール・バック・ルーチンをインストール
bt_icbr_remove	割り込みコール・バック・ルーチンを削除
bt_lock	ユニットのロック
bt_unlock	以前ロックしたユニットをアンロック
bt_mmap	論理デバイスへメモリ・マッピングしたポインタを作成
bt_unmmap	メモリ・マッピングした場所をアンマップ
bt_dev2str	論理デバイス・タイプを文字列へ変換
bt_ctrl	ドライバI/O制御関数を直接呼出し
bt_bind	アプリケーション提供バッファをバインド(**以下のNote 1を参照**)
bt_unbind	バインドしたバッファをアンバインド(**以下のNote 1を参照**)
bt_reg2str	レジスタを文字列へ変換
bt_cas	アトミック処理の比較とスワップ
(次ページに続きます)	

Note:

1. 複数の様々なサイズのバッファはこれらの関数を通してサポートされています：「バインド・バッファの実装」セクションを参照して下さい。
2. このPCI-to-VME のマッピング/バインドのサポートはユニークです：本章の「VMEバス空間へのマッピングおよびバインド」セクションを参照して下さい。

表15-1 PCI-to-VMEライブラリ関数(続き)

関数	概要
bt_tas	アトミック処理のテストおよび設定
bt_get_io	アダプターのCSRレジスタの読み取り
bt_put_io	アダプターのCSRレジスタの書き込み
bt_or_io	レジスタへ1回書き込み
bt_reset	リモートでVMEバスをリセット
bt_send_irq	離れたVMEバスに割り込みを送信
bt_status	デバイスのステータスを返す
bt_hw_map_vme	PCI-to-VMEバス・マッピングの作成(**以下のNote 2を参照**)
bt_hw_unmap_vme	PCI-to-VMEバス・マッピングを削除(**以下のNote 2を参照**)

Note:

1. 複数の様々なサイズのバッファはこれらの関数を通してサポートされています：「バインド・バッファの実装」セクションを参照して下さい。
2. このPCI-to-VME のマッピング/バインドのサポートはユニークです：本章の「VMEバス空間へのマッピングおよびバインド」セクションを参照して下さい。

バインド・バッファの実装

RedHawk sbsvmeのバインド・バッファのサポートは、VMEバス空間に同時に複数、サイズが異なるカーネルのバインド・バッファを割り当てるため、bt_mmap()およびbt_bind()を許可します。このセクションでは、SBSテクノロジー・ソフトウェア・マニュアルのバインド・バッファに関する資料とはサポートがどのように異なるかを含め、このバインド・バッファのサポートに関する情報を提供します。

SBSの資料とRedHawkバインド・バッファの実装との間で唯一ユーザー・インターフェースが異なるのは、後述されているbt_set_info() BT_INFO_KFREE_BUF呼び出しにおける‘value’ パラメータの使い方であることに注意して下さい。他のユーザー・インターフェース全てはSBSテクノロジー・ソフトウェア・マニュアルで示すのと同じとなります。

bt_get_info BT_INFO_KMALLOC_BUF

概要

```
bt_error_t bt_get_info(bt_desc_t btd, BT_INFO_KMALLOC_BUF, bt_devdata_t
*value_p)
```

複数のbt_get_info() BT_INFO_KMALLOC_BUFコマンドの呼び出しは、それぞれが返すバッファのアドレス、value_pパラメータの位置に格納されている複数のカーネル・バッファを割り当てることが可能となり、VMEバスへそのバッファをマッピングおよびバインドするためにその後bt_mmap()やbt_bind()の呼び出しを使用することが可能になります。

BT_INFO_KMALLOC_BUF呼び出しは、最後に成功したbt_set_info() BT_INFO_KMALLOC_SIZ呼び出しで設定した最後の値と等しいサイズのカーネル・バインド・バッファを割り当てます。(もしBT_INFO_KMALLOC_BUF呼び出しがされた時にそのような呼び出しがされなかった場合、64KBのデフォルト・サイズが使用されます。)

最大BT_KMALLOC_NBUFS (16)のカーネル・バッファは、BT_INFO_KMALLOC_BUFコマンドの呼び出しにより同時に割り当てることが可能です。もしこれらが既に16個のバインド・バッファを割り当てられていた場合、このBT_INFO_KMALLOC_BUF呼び出しは失敗してBT_EINVALのエラー値を返します。

もしbt_set_info() BT_INFO_KMALLOC_SIZ呼び出しがバインド・バッファのサイズをゼロへ設定するために使用される場合、新しいバインド・バッファのサイズがbt_set_info() BT_INFO_KMALLOC_SIZ呼び出しを介して非ゼロの値に設定されるまで、その後に続くBT_INFO_KMALLOC_BUF呼び出し全てはBT_EINVALのエラー値と共に返されることに注意して下さい。

もしカーネルが新しいカーネル・バインド・バッファ用に十分な空間を割り当てることが出来ない場合、このBT_INFO_KMALLOC_BUF呼び出しは失敗し、BT_EINVALのエラー値を返します。

bt_set_info BT_INFO_KMALLOC_SIZ

概要

```
bt_error_t bt_set_info(bt_desc_t btd, BT_INFO_KMALLOC_SIZ, bt_devdata_t
value)
```

bt_set_info() BT_INFO_KMALLOC_SIZコマンドが新しいバインド・バッファのサイズを設定するために使用される場合、そのコマンドは将来のbt_get_info() BT_INFO_KMALLOC_BUFコマンドの呼び出しに影響を及ぼすだけです。異なるバインド・バッファのサイズで既に割り当てられたどのカーネル・バインド・バッファも新しいBT_INFO_KMALLOC_SIZにより影響を受けることはありません。

このようにして、異なるサイズのカーネル・バインド・バッファは1回以上のbt_get_info() BT_INFO_KMALLOC_BUF呼び出しを行った後、異なるBT_INFO_KMALLOC_SIZ 'value'パラメータを使用することによって割り当てることが可能となります。

2のべき乗の'value' パラメータでバインド・バッファのサイズを使用することを推奨しますが、必須ではありません。カーネル・バインド・バッファ割り当ては2のべき乗に切り上げるため、2のべき乗の'value' パラメータ値の指定および使用は割り当てられたカーネル・バインド・バッファの使用されていない領域を排除します。カーネル・バインド・バッファのサイズの初期既定値は64KBです。

通常、bt_get_info() BT_INFO_KMALLOC_BUF呼び出しで割り当てに成功することが可能なカーネル・バインド・バッファの最大サイズは4MBです。しかしながら、システムの物理メモリ量およびシステムメモリの使用状況に依存しますので、4MBのカーネル・バインド・バッファを正常に割り当てることが常に可能ではない場合があります。この場合、複数のより小さなサイズのバインド・バッファを割り当てること、あるいは、システム・メモリを他に使用してメモリ・リソースを使い果たす前に4MBのカーネル・バインド・バッファを割り当てることが可能です。

bt_set_info BT_INFO_KFREE_BUF

概要

```
bt_error_t bt_set_info(bt_desc_t btd, BT_INFO_KFREE_BUF, bt_devdata_t value)
```

bt_set_info() BT_INFO_KFREE_BUFコマンドのインターフェースは、SBSテクノロジー・マニュアルに記述されていることとRedHawkの下ではわずかに異なります。

具体的には、’value’ パラメータはSBSの実装では使用されませんが、RedHawkの実装では以下の方法でそのパラメータを使用します：

’value’ パラメータがゼロの場合：

この呼び出しは、現在ユーザー空間からbt_mmap()されていない全ての カーネル・バインド・バッファをアンバインドおよび解放します。

もしアンバインドよび解放数r事が可能なバインド・バッファが見つからない場合、この呼び出しは失敗し、呼び出し元へBT_EINVALが返されます。

’value’ パラメータがゼロではない場合：

この呼び出しは特定のカーネル・バインド・バッファを1つだけアンバインドおよび解放するためのものです。この場合、呼び出し元の’value’ パラメータは、以前のbt_get_info() BT_INFO_KMALLOC_BUF呼び出しで’value_p’ パラメータに返されたカーネル・バッファのアドレスと同じである必要があります。

もしこの呼び出しの’value’ パラメータに指定したバッファのアドレスが有効なカーネル・バインド・バッファと一致しない場合、この呼び出しは失敗してBT_EINVALのエラー値を返します。

もしこの呼び出しの’value’ パラメータが有効なカーネル・バインド・バッファと一致していても現在そのバッファがユーザー空間からbt_mmap()されている場合、この呼び出しは失敗してBT_EFAILの値が返されます。この場合、この呼び出しが成功する前にそのバッファをまずbt_unmmap()する必要があります。

バインド・バッファの追加情報

以降のセクションではバインド・バッファのサポートがRedHawkの下で影響を及ぼす更なる領域について説明します。

bigphysareaパッチ

SBSテクノロジー・ソフトウェア・マニュアルに明記されているbigphysareaパッチは、RedHawk sbsvme btpデバイス・ドライバでサポートされていないもしくは必要とされていません。複数の大きなバインド・バッファを使用することによって、VMEバスからiHawkのメモリへアクセスするためにVMEバス・スレーブ・ウィンドウ空間の16MB全てをサポートすることが可能です。

btpモジュールのアンロード

sbsvme ’btp’ カーネル・モジュールは、現在プロセスのアドレス空間にbt_mmap()されたどのカーネル・バインド・バッファも存在する間はアンロードすることが出来ません。カーネル・ドライバ・モジュールがアンロードされる前にプロセスはまずbt_unmmap()呼び出しにてカーネル・バインド・バッファへのマッピングを削除する必要があります。

現在ユーザー空間からbt_mmap()されたバインド・バッファが存在しない場合、btpカーネル・モジュールは“/etc/init.d/sbsvme stop” コマンドにてアンロードすることが可能で、現在割り当てられているどのカーネル・バインド・バッファも(現在バインドされている場合は)ハードウェアVMEバス・スレーブ・ウィンドウ空間から暗黙のうちにアンロードされ、将来のカーネル・メモリ割り当てのために解放されます。

bt_bind rem_addr_pパラメータ

bt_bind()呼び出しの'rem_addr_p' パラメータは呼び出し元がカーネル・バインド・バッファをバインドしたい遠隔のVMEバス・スレーブ・ウィンドウ内のオフセットを指定します。この値はオフセットであり、絶対的なVMEバスの物理アドレスではないことに注意して下さい。このオフセット値は、SBS VMEアダプター・カード上にあるREM-RAM LOのジャンパー設定により定義されたVMEバスアドレスの基点からとなります。

実際の'rem_addr_p' オフセット値を指定、もしくは'rem_addr_p' パラメターにBT_BIND_NO_CARE値を使用してbtpドライバに適切なバインド・アドレス位置を見つけさせることのどちらでも可能です。この値が使われる場合、bt_bind()呼び出しから正常に戻った時の'rem_addr_p' メモリ位置はカーネルbtpドライバがバインド・バッファにバインドしたオフセット値を含みます。

例として、もしREM-RAM LOのジャンパー設定が0xC0000000の値に設定されオフセット値が0x10000の場合、VMEバスからアクセス可能なこのバッファの実際のバインド・アドレスは0xC0010000となるでしょう。

ローカル・メモリ

カーネル・バインド・バッファのサポートとは別に、btpドライバもまたローカル・メモリのコンセプトをサポートします。バインド・バッファ機能のために通常使用されるBT_DEV_A32, BT_DEV_A24, 他のVMEバス・デバイス・タイプの変わりにBT_DEV_LMデバイス・タイプの使用を通じてこの機能が利用可能となります。

ローカル・メモリ・バッファは、btpドライバがロードされた時にVMEバス・スレーブ・ウィンドウ領域へ割り当たられバインドされたローカルiHawkメモリから構成されます。このメモリの割り当てとバインドはbtpドライバがロードされている限り実施されたままとなります。もしbtpドライバが"/etc/init.d/sbsvme stop" コマンドによりアンロードされた場合、このローカル・メモリ・バッファはVMEバス空間からアンロードされ、他のカーネルで使用するために解放されます。

ローカル・メモリ・バッファは、VMEアダプター・カード上のREM-RAM LOジャンパー設定にて定義されたとおりに常にVMEバス・スレーブ・ウィンドウの底辺領域にバインドします。例えば、もしローカル・メモリのサイズが64KB、REM-RAM LOジャンパー設定が0xC0000000の値へ設定された場合、ローカル・メモリ・バッファは物理VMEバス・アドレスの0xC0000000から0xC0000FFFまでのVMEバスへバインドされます。

ローカル・メモリ・バッファは常にVMEバス・リモート・スレーブ・ウィンドウの底辺領域を占有するため、カーネル・バインド・バッファはローカル・メモリ・サポートが有効の時はいつでもこの領域へバインドされるとは限らないことに注意して下さい。既定値で、ローカル・メモリ・サポートは、(REM-RAM LOジャンパー設定が16MBをカバーする範囲に設定されていると仮定して)バインド・バッファ用に16 MB - 64 KB のVMEバス・スレーブ・ウィンドウ空間を残して、64KBのローカル・メモリ・バッファ・サイズで有効となっています。

ローカル・メモリ・バッファのサイズは、**/etc/sysconfig/sbsvme**構成ファイル(本章の「構成」セクションを参照して下さい)内の'lm_size' パラメータを変更することにより増やすことが可能です。サポートされる'lm_size' の値の最大は4MBであることに注意して下さい。もしより大きな値が指定された場合、btpドライバのバッファ割り当ては成功せず、ローカル・メモリ機能はbtpドライバのロード時に無効となります。

ローカル・メモリ・サポートは、'lm_size' btpモジュール・パラメータをゼロへ設定することにより無効にすることが可能です。ゼロへ設定した場合、btpドライバはローカル・メモリ・バッファは割り当てず、VMEバス・スレーブ・ウィンドウ領域全体はカーネル・バインド・バッファを使用するために解放されます。

ローカル・メモリ・サポートは、バインド・バッファ・サポートととてもよく似ています：

- ローカル・メモリとバインド・バッファの両方が、スレーブ・ウィンドウ領域を通してVMEバスからアクセスが可能です。
- ローカル・メモリとバインド・バッファの両方のバッファ領域は、bt_read(), bt_write(), bt_mmap()関数を使用する時に適切なデバイス・タイプを指定することによってアクセスすることが可能となります。

ローカル・メモリとバインド・バッファの各サポートでの主な違いは：

- 1つのローカル・メモリ・バッファ領域だけが存在する可能性があります。このバッファはbtpドライバのロード時に設定され、ドライバがアンロードされるまで割り当ておよびバインドされたままとなります。

  対照的に複数の異なるサイズのバインド・バッファは動的に割り当ておよびバインド、動的にアンバインドおよび解放することが可能です。

- ローカル・メモリ・バッファは常にVMEバス・スレーブ・ウィンドウ領域の底辺を占有します。

  対照的にバインド・バッファのためにユーザーがVMEバス空間へバインドさせる各バインド・バッファの位置/オフセットのどちらも指定すること、またはカーネルに動的に使用する次の空いている位置/オフセットを見つけさせることが可能です。

VMEバス空間へのマッピングおよびバインド

RedHawkは特定のプロセスとは関係なく、マッピングを作成したプロセスが終了した後もそのまま残るVMEバス空間マッピングを作成する方法を提供します。このマッピングは単独で作成および削除することが、bt_hw_map_vmeとbt_hw_unmap_vmeライブラリ機能を通して、または**/proc**ファイル・システム・インターフェースへ書き込むことでどちらも可能となります。

I/O空間の領域へこのセグメントをバインドするために**shmbind(2)**または**shmconfig(1)**を使い、アクティブVMEバス空間マッピングに対応する一意のPCIバス開始アドレスを取得および使用することが可能となります。

この機能は以下のセクションで説明されています。

bt_hw_map_vme

この関数は新しいPCI-to-VMEバス・マッピングを作成します。

概要

```
bt_error_t bt_hw_map_vme(bt_desc_t btd, void **phys_addr_p, bt_devaddr_t
vme_addr, size_t map_len, bt_swap_t swapping)
```

引数

`btd`	成功したbt_open()関数呼び出しから返されたデバイス記述子。
`phys_addr_p`	このマッピングのためのローカルPCIバス開始/ベース・アドレスが返されるユーザー空間の位置。

vme_addr	開始/ベース・ターゲットVMEバスの物理アドレス。このアドレスは4KBの境界線上に揃えられている必要があります。
map_len	作成されるハードウェア・マッピングのサイズ。この値は4KBの倍数に切り上げれられます。
swapping	ハードウェア・マッピングに使用するバイト・スワッピング方式。**/usr/include/btp/btngpci.h**ヘッダ・ファイルに含まれているBT_SWAP_*xxx* 定義を使用することが可能です。

戻り値

成功した場合、BT_SUCCESSの値が返されます。phys_addr_p位置に返されたPCIバスのアドレスは、リモートVMEバスアドレスのこの範囲へアクセスするために使用可能な共有メモリ領域を作成するため**shmbind(2)**または**shmconfig(1)**を使用することが可能です。

失敗した場合、失敗の原因を示す適切なbt_error_tの値が返されます：

BT_EDESC	無効なbtd記述子が指定された。記述子はデバイス・タイプBT_DEV_A32, BT_DEV_A24, BT_DEV_A16のbt_open()呼び出しから返された記述子である必要があります。
BT_EINVAL	無効なvme_addr, map_len, phys_addr_p, スワッピング・パラメータが指定された。
BT_ENXIO	sbsvmeハードウェアがオンラインではない、または正しく接続されていない。
BT_ENOMEM	sbsvmeハードウェア・マッピング・レジスタが必要とする数を割り当てることが出来なかった。
BT_ENOMEM	このマッピングの追跡に使用されるカーネルデータ構造体用のメモリを割り当てることができなかった。

bt_hw_unmap_vme

この関数はbt_hw_map_vme関数で既に作成されたまたは**/proc/driver/btp/***unit***/vmemappings**ファイルへの書き込みによるPCI-to-VMEバス・マッピングを削除します。

概要

```
bt_error_t bt_hw_unmap_vme(bt_desc_t btd, void *phys_addr)
```

パラメータ

btd	成功したbt_open()関数呼び出しから返されたデバイス記述子。
phys_addr	削除するVMEバス・マッピングのPCIバス開始アドレス。

戻り値

成功した場合、BT_SUCCESSの値が返されます。

失敗した場合、失敗の原因を示す適切な`bt_error_t`の値が返されます：

BT_EDESC	無効な`btd`記述子が指定された。記述子はデバイス・タイプBT_DEV_A32, BT_DEV_A24, BT_DEV_A16のbt_open()呼び出しから返された記述子である必要があります。
BT_ENOT_FOUND	`phys_addr`パラメータで指定されたマッピングが存在しない。

/procファイル・システム・インターフェース

sbsvme btpカーネル・モジュールがロードされた時、以下の**/proc**ファイルが作成されます：

/proc/driver/btp/*unit*/vme-mappings

unit はsbsvme PCIブリッジ・カードのユニット番号です。最初のカードはユニット番号が0となります。複数のブリッジを持つシステム上では、2番目のカードはユニット番号1となります。

既存のPCI-to-VMEバス・マッピングはそのファイルの読み取りにより見ることが可能です。マッピングはそのファイルへの書き込みにより作成および削除が可能となります。これらのテクニックは以下で説明します。

VMEバス・マッピングの表示

cat(1)を使った**vme-mappings**ファイルの読み取りは、現在確立された全てのVMEバス・マッピングを表示します。以下の出力は2つのPCI-to-VMEバス・マッピングを示します：

```
$ cat /proc/driver/btp/0/vme-mappings

pci=0xf8019000 vme=0x00008000 size=0x0001000 space=A16 admod=0x2d swap=5
pci=0xf8011000 vme=0x00fe0000 size=0x0008000 space=A24 admod=0x39 swap=0
```

`pci=`	マッピングが開始されるローカル`PCI`バス・アドレスを示します
`vme=`	開始VMEバス・アドレスを示します
`size=`	マッピングのサイズ/長さを示します
`space=`	`VME`バス・アドレス空間のマッピングのタイプを示します
`admod=`	**/usr/include/btp/btdef.h**に定義されるBT_AMOD_*xxx* で記述されたVMEバス・アドレス・モディファイアを示します
`swap=`	**/usr/include/btp/btngpci.h**.に定義されるBT_SWAP_*xxx* で記述されたビット・スワップ方式を示します

VMEバス・マッピングの作成

VMEバス空間へのマッピングは**vme-mappings**ファイルへの書き込みにより作成することが可能です。このファイルへ書き込むためにはCAP_SYS_ADMIN権限を持っている必要があることに注意して下さい。マッピングを作成するためには、以下の3つのパラメータをここで定めた順番で指定する必要があります：

`vme=`	マッピングするためにページに揃えられた開始`VME`バス・アドレスを指定します (例： 0xfffff000)

`size=` マッピングのサイズ(ページの倍数であること)を指定します。(例：0x1000) sbsvmeハードウェアはマッピングが合計で32MBのVMEバス空間に制限されていることに注意して下さい。

`space=` VMEバス・アドレス空間のマッピングのタイプ(A32, A24, A16)を指定します。

以下のオプション・パラメータは、上述の必須パラメータに続いて任意の順番で与えることも可能です：

`admod=` **/usr/include/btp/btdef.h**に定義されるBT_AMOD_*xxx* で記述されたVMEバス・アドレス・モディファイアを指定します。もし指定しない場合、以下のデフォルト値が使用されます：

BT_AMOD_32 0x0d
BT_AMOD_24 0x3d
BT_AMOD_16 0x2d

`swap=`
/usr/include/btp/btngpci.hに定義されるBT_SWAP_*xxx* で記述されるビット・スワッピング方式を指定します。もし指定しない場合、デフォルト値のBT_SWAP_DEFAULTが使用されます。

以下の例は、**vmemappings**ファイルへの書き込みによる2つのVMEバス・マッピングの作成を示します。

```
$ echo "vme=0xe1000000 size=0x10000 space=A32" > /proc/driver/btp/0/vme-mappings
$ echo "vme=0xc0000000 size=0x1000 space=A32 swap=7 admod=0x9" >
/proc/driver/btp/0/vme-mappings
```

sbsvme btpカーネルドライバが"/etc/init.d/sbsvme stop" (「VMEbus Mappings」を参照して下さい)にてアンロードされる時、現在の全てのVMEバス・マッピングはドライバがアンロードされる前に削除されることに注意して下さい。もしマッピングが存在し、"modprobe -r btp"がドライバをアンロードするために使われた場合、アンロードは全てのVMEバス・マッピングが削除されるまで失敗します。

VMEバス・マッピングの削除

VMEバス空間へのマッピングは、**vme-mappings**ファイルにマッピングのローカルPCIバス位置を書き込むことにより策することが可能です。このファイルへ書き込むためにはCAP_SYS_ADMIN権限を持っている必要があることに注意して下さい。PCIバスの位置はbt_hw_map_vme()および**vme-mappings**ファイルの**cat**により返されます。
例：

```
$ cat /proc/driver/btp/0/vme-mappings
pci=0xf8019000 vme=0x00008000 size=0x0001000 space=A16 admod=0x2d swap=5
pci=0xf8011000 vme=0x00fe0000 size=0x0008000 space=A24 admod=0x39 swap=0

$ echo "pci=0xf8019000" > /proc/driver/btp/0/vme-mappings

$ cat /proc/driver/btp/0/vme-mappings
pci=0xf8011000 vme=0x00fe0000 size=0x0008000 space=A24 admod=0x39 swap=0
```

アプリケーション例

プログラム例は、sbsvme btpデバイス・ドライバの機能の実演を提供しその利用を促進します。それらは**/usr/share/doc/ccur/examples/sbsvme**で見つけることが可能です。そのプログラムは次のために便利なツールです：

- デバッギング
- バイナリ・データのアップロードおよびダウンロード
- プログラム化した割り込みの受信をよび集計
- ハードウェアのテスト
- VMEバス・マッピングの作成および共有メモリ領域へのバインド

表15-2はプログラム例を記載しています。アスタリスク(*)はRedHawk Linuxに加えられたプログラムを示し、続くセクションで説明されています。他のプログラムはSBSテクノロジー・ソフトウェア・マニュアルで説明されています。

表15-2 PCI-to-VMEプログラム例

名称	概要	使用される関数
bt_bind	リモートVMEバスへローカル・バッファをバインドし、ユーザー入力を待って、バインドしたバッファの先頭256byteを出力します。	bt_bind() bt_unbind()
bt_bind_mult *	リモートVMEバスへ複数のローカル・バッファをバインドする方法を示します。任意でユーザー入力待機の前にローカル・バッファに値を書き込みます。ユーザー入力発生後、各ローカル・バッファの各ページの先頭16byteを出力します。	bt_bind() bt_unbind()
bt_bind_multsz *	複数の異なるサイズのバインド・バッファを作成する方法を示します。	bt_bind() bt_unbind()
bt_cat	'cat' プログラムに似ています。リモートVMEバスからの読み取りを標準出力(stdout)へ、または標準入力(stdin)からリモートVMEバスへのデータ書き込みを可能にします。	bt_read() bt_write()
bt_datachk	特定のパターンを使いデバイスからの読み書きおよびデータまたはステータスのエラーが発生していないことを検証します。	bt_read() bt_write()
bt_dumpmem	リモートVMEバスのデータ256byteを読み取り標準出力へ出力します。	n/a
bt_getinfo	全ドライバのパラメータを取得しそれらの値を標準出力へ表示するスクリプト。	
bt_hwmap *	VMEバス・マッピングを作成します。	bt_hw_map_vme()
bt_hwunmap *	VMEバス・マッピングを削除します。	bt_hw_unmap_vme()
bt_icbr	任意の割り込みタイプの登録および割り込みを受信します。	bt_icbr_install() bt_icbr_remove()
bt_info	ドライバのパラメータの取得または設定を行います。	bt_get_info() bt_set_info()
bt_readmem	リモートVMEバスのデータ256byteの読み取って標準出力へ表示します。	bt_read()
bt_reset	リモートVMEバスをリセットします。	bt_reset()

(次ページへ続く)

表15-2 PCI-to-VMEプログラム例 (続き)

名称	概要	使用される関数
bt_revs	ドライバのバージョンおよびハードウェアのファームウェア・バージョン情報を標準出力へ出力します。	bt_open()
bt_sendi	リモート・バスへ割り込みを送信します。	bt_send_irq()
readdma *	CPUに代わりカーネル・ドライバで使用されるDMAハードウェアがデータをコピーすることになるこのプログラムがより大きいなデータを読み取ることを除いては、readmemと同じです。	bt_read()
shmat *	アタッチするために共有メモリ・キー・パラメータを利用し、共有メモリ領域から読み取ります。shmconfig-scriptプログラムで使用されます。	shmconfig(1) shmat(2)
shmbind *	PCI-to-VMEバス・マッピングにマップされた共有メモリ領域を作成しアタッチしてそれを読み書きします。	shmget(2) shmbind(2) shmat(2)
shmconfig-script *	**/proc**ファイル・システムを介してPCI-to-VMEバス・マッピングを作成し、VMEバス領域へバインドする共有メモリ領域を作成するスクリプトです。	shmconfig(1)
vme-mappings *	**/proc**ファイル・システムを介してPCI-to-VMEバス・マッピングを作成、表示、削除する方法を示すスクリプトです。	n/a
writemem *	リモートVMEバスへ256byteのデータを書き込み、リモートVMEバスから256byteのデータを読み戻して、そのデータを標準出力へ出力します。	bt_read() bt_write()
writedma *	CPUがデータをコピーする代わりにカーネル・ドライバでDMAハードウェアが使用されることになり、このプログラムがより大きいなデータを書き込むことを除いては、writememと同じです。この例はリモートVMEバスへデータを書き込むだけで、リモートVMEバスからのデータ読み戻しはしません。	bt_write()

bt_bind_mult

bt_bind_multサンプル・アプリケーションは、複数の同じサイズのバッファをリモート・バスへバインドするためにbt_bind()関数を使用します。これはユーザー入力を待機し、バインドされた各バッファの各ページの最初の4ワードを出力します。任意で待機前にバッファへデータの書き込みも行います

使用方法：bt_bind_mult -[natulws]

オプション	機能
-n <nbufs>	割り当ておよびバインドするバッファの数。既定値は2。
-a <vmeaddr>	バッファをバインドするVMEアドレス。既定値はBT_BIND_NO_CARE。
-t <logdev>	論理デバイス(BT_DEV_MEM, BT_DEV_IO, BT_DEV_DEFAULT等)。 既定値はBT_DEV_DEFAULT。
-u <unit>	オープンするユニット番号。既定値は0。
-l <len>	バインドするバッファの長さ。既定値は1ページ(0x1000)。
-w <value>	最初にバッファの各ページの先頭4ワードにこの値を書き込みます。
-s <swapbits>	bt_bind()を呼び出すためにスワップ・ビット値を設定します。シンボリック名は認識されないことに注意して下さい。

bt_bind_multsz

bt_bind_multszサンプル・アプリケーションは、様々なサイズの複数のバッファをリモート・バスへバインドするためにbt_bind()関数を使用します。これはユーザー入力を待機し、バインドされた各バッファの各ページの最初の4ワードを出力します。任意で待機前にバッファへデータの書き込みも行います。

使用方法：bt_bind_multsz -[atuws]

オプション	機能
-a <vmeaddr>	バッファをバインドするVMEアドレス。既定値はBT_BIND_NO_CARE。
-t <logdev>	論理デバイス(BT_DEV_MEM, BT_DEV_IO, BT_DEV_DEFAULT等)。 既定値はBT_DEV_DEFAULT。
-u <unit>	オープンするユニット番号。既定値は0。
-w <value>	最初にバッファの各ページの先頭4ワードにこの値を書き込みます。
-s <swapbits>	bt_bind()を呼び出すためにスワップ・ビット値を設定します。シンボリック名は認識されないことに注意して下さい。

bt_hwmap

bt_hwmapサンプル・アプリケーションは、VMEバス空間の領域へハードウェア・マッピングを作成するためにbt_hw_map_vme関数を使用します。

使用方法：bt_hwmap -a[ltus]

オプション	機能
-a <addr>	VMEバスの物理アドレス。この引数は必須です。
-l <len>	PCIバス上にマッピングするVMEバス領域の長さ。既定値は1ページ(0x1000)。
-t <logdev>	アクセスする論理デバイス (BT_DEV_A32, BT_DEV_A24, BT_DEV_A16, BT_DEV_IO, BT_DEV_RR)。既定値はBT_DEV_A32。
-u <unit>	オープンするユニット番号。既定値は0。
-s <swapbits>	bt_bind()を呼び出すためにスワップ・ビット値を設定します。シンボリック名は認識されないことに注意して下さい。 既定値はBT_SWAP_DEFAULT。

bt_hwunmap

bt_hwmapサンプル・アプリケーションは、VMEバス空間の領域からハードウェア・マッピングを削除するためにbt_hw_unmap_vme関数を使用します。

使用方法：bt_hwunmap -p[tu]

オプション	機能
-p <pciaddr>	削除するマッピングのローカルPCIバスの物理アドレス。この引数は必須です。
-t <logdev>	論理デバイス(BT_DEV_A32, BT_DEV_A24, BT_DEV_A16, BT_DEV_IO, BT_DEV_RR) 。既定値はBT_DEV_A32。
-u <unit>	オープンするユニット番号。既定値は0。

readdma

CPUに代わりカーネル・ドライバで使用されるDMAハードウェアがデータをコピーすることになるこのプログラムがより大きいなデータを読み取ることを除いては、このサンプル・プログラムはbt_readmemと同じです。

使用方法：readdma -[atulo]

オプション	機能
-a <addr>	データ転送を開始するアドレス。デフォルト値＝0x00000000
-t <logdev>	アクセスする論理デバイス。既定値はBT_DEV_A32。
-u <unit>	オープンするユニット番号。既定値は0。
-l <length>	読み取るバイト数。ページサイズへ切り捨てます。既定値は0x1000。
-o <outlen>	各ページ境界線の先頭に出力するバイト数。既定値は16byte。この値は409以下であること。

shmat

このサンプル・プログラムはshmconfig-scriptスクリプトにより呼び出されます。これは共有メモリの'key' 値を利用してアタッチし、VMEバス空間にバインドされた共有メモリ領域から読み出します。

使用方法：shmat -k shmkey -s size [-o outlen]

オプション	機能
-k <shmkey>	10進数、または'0x' か'0X' で始まる16進数の共有メモリのキー値。
-s <size>	共有メモリ領域のサイズ(byte)。
-o <outlen>	標準出力へ出力する各共有メモリ・ページの先頭からのバイト数(16進数)。既定値は32byte。

shmbind

このプログラム例は、PCI-to-VMEバス・マッピングへ共有メモリ領域をアタッチするために **shmget(2)**, **shmbind(2), shmat(2)**を使用します。共有メモリにアタッチされた領域を使いVMEバス空間の読み書きが可能となります。PCI-to-VMEハードウェア・マッピングは既に作成されている必要があります。

使用方法：shmbind -p pci_addr -s size [-r | -w value] [-o len]

オプション	機能
-p <pci_addr>	VMEマッピングが置かれているローカルPCIバス・アドレス(16進数)。
-s <size>	作成する共有メモリ領域のサイズ(byte, 16進数)。
-r	共有メモリ領域からの読み取り(既定値)。
-w <value>	指定された値を使い、共有メモリ領域へ書き込み(16進数)
-o <len>	標準出力へ出力する各共有メモリ・ページの先頭からのバイト数(16進数)。既定値は32byte。

shmconfig-script

これはPCI-to-VMEバス・マッピングによる特定のVMEバス領域へバインドされた共有メモリ領域を作成するために**shmconfig(1)**を使用する方法のサンプル・スクリプトです。このスクリプトは共有メモリ領域が作成された後にshmatサンプル・プログラムを呼び出します。

vme-mappings

これは**/proc/driver/btp/***unit***/vme-mappings**ファイルを使いPCI-to-VMEバス・マッピングを作成、調査、削除する方法を示すサンプル・スクリプトです。

writemem

このサンプル・プログラムは、Bit 3論理デバイスのいずれかに書き込むためbt_write() Bit 3 Mirror API関数を使用します。

使用方法：writemem -[atud]

オプション	機能
-a <addr>	データ転送を始めるアドレス。デフォルト値＝0x00000000。
-t <logdev>	アクセスする論理デバイス(BT_DEV_RDP, BT_DEV_A32, 等)。
-u <unit>	オープンするユニット番号。既定値は0。
-d <value>	書き込みを開始するデータの値。規定値は0。

全ての数値はC言語の基数表記法を使用します。

例：　アドレス0x00001000で始まるBT_DEV_RDPから最初の256byteのデータを書き込みます：

```
./writemem -a 0x00001000
```

writedma

このサンプル・プログラムは、CPUがデータをコピーする代わりにカーネル・ドライバでDMAハードウェアが使われることになって、それがより大きいなデータを書き込むことを除いては、writememと同じです。この例はリモートVMEバスへデータを書き込むだけで、リモートVMEバスからのデータ読み戻しはしません。

使用方法：writedma -[atuld]

オプション	機能
-a <addr>	VMEアドレスの先頭。既定値＝0x00000000.
-t <logdev>	アクセスする論理デバイス。規定値はBT_DEV_A32。
-u <unit>	オープンするユニット番号。既定値は0。
-l <length>	書き込むバイト数。ページサイズへ切り下げます。規定値は0x1000。
-d <value>	書き込みを開始するデータの値。規定値は0。

A
メッセージ・キュー・プログラム例

本付録にはPOSIXおよびSystem Vのメッセージ・キュー機能の使用を説明するサンプル・プログラムが含まれています。更なるサンプル・プログラムは**/usr/share/doc/ccur/examples**ディレクトリにオンラインで提供されます。

POSIXメッセージ・キュー例

ここにあるサンプル・プログラムはC言語で記述されています。このプログラムでは、親プロセスがPOSIXメッセージ・キューをオープンして、キューが空から空ではない状態へ遷移した時にリアルタイム・シグナルを介して通知されるように登録しています。親プロセスは子プロセスを生成し、子プロセスが空のキューへメッセージを送信するまで子プロセスを待機します。子プロセスはメッセージを送信し、その記述子をクローズして終了します。

親プロセスはリアルタイム・シグナルを受信し、シグナル・ハンドラ内で`siginfo_t`構造体を通して配信される`sigev_value (si_value)`を取得します。親プロセスは子プロセスのテスト・メッセージを受信する前に`si_code (SI_MESGQ)`の配信もテストします。親プロセスは`si_value(`共用体`)`の配信が事前に登録された`sigev_value`と合っていることを検証します。シグナル・ハンドラは、psignalを使い受信したリアルタイム・シグナル値(SIGRTMAX)も表示します。psignal関数はSIGRTMAXと明示する方法を知らないので、unknown signalと判定し、値を出力して終了します。

このプログラムをビルドするには、以下を指定します：

gcc mq_notify_rtsig.c -Wall -g -l rt -o mq_notify_rtsig

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <unistd.h>
#include <mqueue.h>
#include <stdlib.h>
#include <ctype.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <time.h>
#include <sched.h>
#include <signal.h>
#include <bits/siginfo.h>
#define MSGSIZE 40
#define MAXMSGS 5
#define VAL 1234
```

```
void handlr(int signo, siginfo_t *info, void *ignored);
int val, code;
int main(int argc, char **argv)
{
    struct sigaction act;
    struct sigevent notify;
    struct mq_attr attr;
    sigset_t set;
    char *mqname = "/mq_notify_rtsig";
    char rcv_buf[MSGSIZE];
    mqd_t mqdes1, mqdes2;
    pid_t pid, cpid;
    int status;

    memset(&attr, 0, sizeof( attr));

    attr.mq_maxmsg = MAXMSGS;
    attr.mq_msgsize = MSGSIZE;

    mq_unlink(mqname);

    mqdes1 = mq_open(mqname, O_CREAT|O_RDWR, 0600, &attr);

    sigemptyset(&set);
    act.sa_flags = SA_SIGINFO;
    act.sa_mask = set;
    act.sa_sigaction = handlr;
    sigaction(SIGRTMAX, &act, 0);

    notify.sigev_notify = SIGEV_SIGNAL;
    notify.sigev_signo = SIGRTMAX;
    notify.sigev_value.sival_int = VAL;

    mq_notify(mqdes1, &notify);

    printf("¥nmq_notify_rtsig:¥tTesting notification sigev_value¥n¥n");
    printf("mq_notify_rtsig:¥tsigev_value=%d¥n",¥
    notify.sigev_value.sival_int);

    if( (pid = fork()) < 0) {
        printf("fork: Error¥n");
        printf("mq_notify_rtsig: Test FAILED¥n");
        exit(-1) ;
    }

    if(pid == 0) { /* child */
        cpid = getpid() ;

        mqdes2 = mq_open(mqname, O_CREAT|O_RDWR, 0600, &attr);

        printf("child:¥t¥t¥tsending message to empty queue¥n");

        mq_send(mqdes2, "child-test-message", MSGSIZE, 30);
```

```
        mq_close(mqdes2);
        exit(0);
    }
    else { /* parent */
        waitpid( cpid, &status, 0); /* keep child status from init */

        printf("parent:¥t¥t¥twaiting for notification¥n");

        while(code != SI_MESGQ)
            sleep(1);
        mq_receive(mqdes1, rcv_buf, MSGSIZE, 0);

        printf("parent:¥t¥t¥tqueue transition - received %s¥n",rcv_buf);
    }

    printf("mq_notify_rtsig:¥tsi_code=%d¥n",code);
    printf("mq_notify_rtsig:¥tsi_value=%d¥n",val);

    if(code != -3 || val != VAL) {
        printf("¥nmq_notify_rtsig:¥tTest FAILED¥n¥n");
        return(-1);
    }

    mq_close(mqdes1);
    mq_unlink(mqname);

    printf("¥nmq_notify_rtsig:¥tTest passed¥n¥n");

    return(0);
}

void handlr(int signo, siginfo_t *info, void *ignored)
{
    psignal(signo, "handlr:¥t¥t¥t");

    val = info->si_value.sival_int;
    code = info->si_code;

    return;
}
```

System Vメッセージ・キュー例

ここにあるサンプル・プログラムはC言語で記述されています。このプログラムでは、親プロセスは作業の一部の負荷を取り去るために子プロセスを生成します。親プロセスは自身および子プロセスが使用するためにメッセージ・キューも作成します。

子プロセスがその作業を完了すると、メッセージ・キューを介して親プロセスへ結果を送信し、親プロセスへシグナルを送信します。親プロセスがシグナルを受信すると、メッセージ・キューからメッセージを読み取ります。

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <signal.h>
#include <errno.h>

#define MSGSIZE 40/* maximum message size */
#define MSGTYPE 10/* message type to be sent and received */
/* Use a signal value between SIGRTMIN and SIGRTMAX */
#define SIGRT1(SIGRTMIN+1)

/* The message buffer structure */
struct my_msgbuf {
    long mtype;
    char mtext[MSGSIZE];
};

struct my_msgbuf msg_buffer;

/* The message queue id */
int msqid;

/* SA_SIGINFO signal handler */
void sighandler(int, siginfo_t *, void *);

/* Set after SIGRT1 signal is received */
volatile int done = 0;
pid_t parent_pid;
pid_t child_pid;

main()
{
    int retval;
    sigset_t set;
    struct sigaction sa;

    /* Save off the parent PID for the child process to use. */
    parent_pid = getpid();

    /* Create a private message queue. */
    msqid = msgget(IPC_PRIVATE, IPC_CREAT | 0600);

    if (msqid == -1) {
        perror("msgget");
        exit(-1);
    }
```

```
/* Create a child process. */
child_pid = fork();

if (child_pid == (pid_t)-1) {
    /* The fork(2) call returned an error. */
    perror("fork");

    /* Remove the message queue. */
    (void) msgctl(msqid, IPC_RMID, (struct msqid_ds *)NULL);

    exit(-1);
}

if (child_pid == 0) {
    /* Child process */
    /* Set the message type. */
    msg_buffer.mtype = MSGTYPE;

    /* Perform some work for parent. */
    sleep(1);

    /* ... */

    /* Copy a message into the message buffer structure. */
    strcpy(msg_buffer.mtext, "Results of work");

    /* Send the message to the parent using the message
    * queue that was inherited at fork(2) time.
    */

    retval = msgsnd(msqid, (const void *)&msg_buffer,
    strlen(msg_buffer.mtext) + 1, 0);
    if (retval) {
        perror("msgsnd(child)");
        /* Remove the message queue. */
        (void) msgctl(msqid, IPC_RMID, (struct msqid_ds *)NULL);
        exit(-1);
    }

    /* Send the parent a SIGRT signal. */
    retval = kill(parent_pid, SIGRT1);
    if (retval) {
        perror("kill SIGRT");

        /* Remove the message queue. */
        (void) msgctl(msqid, IPC_RMID, (struct msqid_ds *)NULL);
        exit(-1);
    }

    exit(0);
}

/* Parent */
/* Setup to catch the SIGRT signal. The child process
* will send a SIGRT signal to the parent after sending
* the parent the message.
*/

sigemptyset(&set);
sa.sa_mask = set;
sa.sa_sigaction = sighandler;
```

```
    sa.sa_flags = SA_SIGINFO;
    sigaction(SIGRT1, &sa, NULL);

    /* Do not attempt to receive a message from the child
    * process until the SIGRT signal arrives. Perform parent
    * workload while waiting for results.
    */
    while (!done) {
        /* ... */
    }

    /* Remove the message queue.
    (void) msgctl(msqid, IPC_RMID, (struct msqid_ds *)NULL);
    */

    /* All done.
    */

    exit(0);
}

/*
* This routine reacts to a SIGRT1 user-selected notification
* signal by receiving the child process' message.
*/
void
sighandler(int sig, siginfo_t *sip, void *arg)
{
    int retval;
    struct ucontext *ucp = (struct ucontext *)arg;

    /* Check that the sender of this signal was the child process.
    */
    if (sip->si_pid != child_pid) {
        /* Ignore SIGRT from other processes.
        */
        printf("ERROR: signal received from pid %d¥n", sip->si_pid);

        return;
    }

    /* Read the message that was sent to us.
    */
    retval = msgrcv(msqid, (void*)&msg_buffer,
    MSGSIZE, MSGTYPE, IPC_NOWAIT);

    done++;

    if (retval == -1) {
        perror("mq_receive (parent)");
        return;
    }

    if (msg_buffer.mtype != MSGTYPE) {
        printf("ERROR: unexpected message type %d received.¥n",
        msg_buffer.mtype);
        return;
    }

    printf("message type %d received: %s¥n",
    msg_buffer.mtype, msg_buffer.mtext);
}
```

B
リアルタイム機能のためのカーネル・チューニング

表B-1は、RedHawk Linuxの独自機能およびRedHawkがサポートするカーネル構成設定の一覧です。これらはリアルタイム業務を支援するConcurrentにより開発された機能およびオープン・ソース・パッチから組み込まれた機能を含んでいます。

各機能において、カーネル構成GUIオプションとチューニング・パラメータ名は必要に応じて設定を表示および変更の手助けをするために提供されます。更に、各RedHawk Linuxプレビルト・カーネルの各機能のデフォルト設定が用意されています。カーネルの構成および構築に関する詳細な情報については、11章を参照して下さい。

個々の機能に関する情報は様々な場所で入手できます。表B-1では、以下の参考文献が提供されています：

- 本*RedHawk Linux User's Guide* に含まれている情報が提供されるページ番号
- 他の適切なConcurrentの文書の名称および文書番号

情報が取得可能な他の情報源は次のとおり：

- 情報はパラメータ選択時に表示するカーネル構成GUIの別のヘルプ・ウィンドウで提供されます
- カーネル・ソース・ツリーの**Documentation**ディレクトリ内にあるテキスト・ファイル
- インターネット上のLinuxドキュメンテーション・サイト

表B-1 リアルタイム機能用カーネル・チューニング・パラメータ

機能	カーネル構成 GUIオプション	チューニング・パラメータ名	既定値*/ ﾌﾟﾚﾋﾞﾙﾄｶｰﾈﾙ	ページ/ 参考文献
シールドCPU				
CPUｼｰﾙﾃﾞｨﾝｸﾞ有効	General Setup	SHIELD	Y / all	page 2-1
CPU停止有効		CPU_IDLING	Y / all	page 2-28
RCUプロセシング		RCU_ALTERNATIVE	Y / all	page 7-4
CPU毎にデーモンのロック		DAEMON_CPU_LOCK	Y / all	n/a
再ｽｹｼﾞｭｰﾘﾝｸﾞ変数	General Setup	RESCHED_VAR	Y / all	page 5-3
時間計測				
ﾃｨｯｸﾚｽ･ｼｽﾃﾑ有効	Processor Type and Features	NO_HZ	Y / all	page H-1
ｼｽﾃﾑ起動時のﾃｨｯｸﾚｽ有効/無効		NO_HZ_ENABLED	Y / all	
高分解能ﾌﾟﾛｾｽ･ｱｶｳﾝﾃｨﾝｸﾞ	General Setup	HRACCT	Y / all	page 7-2
TSC信頼性	Processor Type and Features	REQUIRE_TSC	N / all	page 7-2
		REQUIRE_RELIABLE_TSC	Y / all	
ｼｽﾃﾑのｸﾛｯｸｿｰｽとしてRCIM有効	Device Drivers	RCIM_CLOCKSOURCE	Y / all	RCIM User's Guide (0898007)
RCIM PPSサポート		RCIM_PPS	Y / all	RCIM User's Guide (0898007)
PPS APIサポート		PPSAPI	Y / all	
ｼﾘｱﾙ上のPPS API		PPSAPI_SERIAL	Y / all	
PPS用NTPサポート	Processor Type and Features	NTP_PPS	Y / all	
RCIMサポート	Device Drivers	RCIM	Y / all	RCIM User's Guide (0898007)
POSIXﾒｯｾｰｼﾞ･ｷｭｰ	General Setup	POSIX_MQUEUE	Y / all	page 3-2
Post/Waitサポート	General Setup	POST_WAIT	Y / all	page 5-37
exec間でｹｰﾊﾟﾋﾞﾘﾃｨ継承	General Setup	INHERIT_CAPS_ACROSS_EXEC	Y / all	page 13-5
*** Y = 設定, N = 非設定, M =カーネル・モジュールがロードされた時に有効**				

表B-1 リアルタイム機能用カーネル・チューニング・パラメータ (続き)

機能	カーネル構成 GUIオプション	チューニング・パラメータ名	既定値*/ ﾌﾟﾚﾋﾞﾙﾄｶｰﾈﾙ	ページ/ 参考文献
ﾌﾟﾛｾｽ・ｽｹｼﾞｭｰﾘﾝｸﾞ	Processor Type and Features	SCHED_SMT	Y / all	page 2-34
		SCHED_SMT_IDLE	N / all	
RedHawk製品オプション				
Frequency-based Scheduler (FBS)	Frequency-Based Scheduling	FBSCHED	Y / all	FBS User's Guide (0898005)
Performance Monitor (PM)	Frequency-Based Scheduling	FBSCHED_PM	Y / all	FBS User's Guide (0898005)
SNARE Audit	General Setup	AUDIT	N / all	RedHawk-FAQ
SBS VMEbus-to-PCI	Device Drivers	SBSVME	M / all page	15-1
/proc ファイルシステム				
/proc/ccur	Pseudo File Systems	PROC_CCUR_DIR	Y / all	n/a
/proc/*pid*/affinity		PROC_PID_AFFINITY	Y / all	n/a
/proc/*pid*/resmem		PROC_PID_RESMEM	Y / all	n/a
PCI BAR Access	Bus options	PROC_PCI_BARMAP	Y / all	page 14-1
メモリ・マッピング				
プロセス空間の mmap/usermap サポート	Pseudo File Systems	PROCMEM_MMAP	Y / all	page 9-1
Interrupt Processing				
ｿﾌﾄIRQﾃﾞｰﾓﾝの優先度	General Setup	SOFTIRQ_PRI	0 / all	page 14-12
ｿﾌﾄIRQﾌﾟﾘｴﾝﾌﾟｼｮﾝのﾌﾞﾛｯｸ		SOFTIRQ_PREEMPT_BLOCK	Y / all	
RCIM IRQ拡張有効	Device Drivers	RCIM_IRQ_EXTENSIONS	Y / all	RCIM User's Guide (0898007)
shmbind呼び出し有効	Kernel Tracing	SHMBIND	Y / all	page 3-16
プロセッサ間割り込み削減				
ｸﾞﾗﾌｨｯｸ・ﾍﾟｰｼﾞのﾌﾟﾘｱﾛｹｰｼｮﾝ	Device Drivers	PREALLOC_GRAPHICS_PAGES	10240 / all	page G-3
ｿﾌﾄﾛｯｸｱｯﾌﾟの検出	Kernel Hacking	DETECT_SOFTLOCKUP	N / all	page 2-34
*** Y = 設定, N = 非設定, M =カーネル・モジュールがロードされた時に有効**				

表B-1 リアルタイム機能用カーネル・チューニング・パラメータ (続き)

機能	カーネル構成 GUIオプション	チューニング・パラメータ名	既定値*/ ﾌﾟﾚﾋﾞﾙﾄｶｰﾈﾙ	ページ/ 参考文献
ロック中断処理	General Setup	LOCK_BREAK_THROTTLE	Y / all	n/a
		LOCK_BREAK_THROTTLE_LIMIT	30 / all	n/a
XFSファイルシステム				
XFS有効	File Systems	XFS_FS	Y / all	page 8-1
ﾘｱﾙﾀｲﾑ・ｻﾌﾞﾎﾞﾘｭｰﾑ ｻﾎﾟｰﾄ		XFS_RT	Y / all	
ｶｰﾈﾙ･ﾌﾟﾘｴﾝﾌﾟｼｮﾝ	Processor Type and Features	PREEMPT	Y / all	page 1-6
ptrace拡張	General Setup	PTRACE_EXT	Y / all	page 1-6
NUMAサポート	Processor Type and Features	NUMA	Y / all	page 10-1
		AMD_NUMA	Y / all	
		X86_64_ACPI_NUMA	Y / all	
		PAGE_REPLICATION	Y / all	
		PAGE_REPLICATION_DYNAMIC	Y / all	
		MEMSHIELD_ZONELIST_ORDER	Y / all	
		CONFIG_KTEXT_REPLICATION	Y / all	
		CONFIG_KMOD_REPLICATION	Y / all	
	General Setup	CONFIG_NUMA_BALANCING	Y / all	
		CONFIG_NUMA_BALANCING_DEFAULT_ENABLED	N / all	
システム・ダンプ				
kdumpｸﾗｯｼｭ・ﾀﾞﾝﾌﾟ有効	Processor Type and Features	KEXEC	Y / all	page 12-1
ﾃﾞﾊﾞｯｸﾞ・ｼﾝﾎﾞﾙ生成		DEBUG_INFO	Y / all	
ｶｰﾈﾙ・ｸﾗｯｼｭ・ﾀﾞﾝﾌﾟ有効		CRASH_DUMP	Y / kdump only	
*** Y = 設定, N = 非設定, M =カーネル・モジュールがロードされた時に有効**				

表B-1 リアルタイム機能用カーネル・チューニング・パラメータ (続き)

機能	カーネル構成 GUIオプション	チューニング・パラメータ名	既定値*/ ﾌﾟﾚﾋﾞﾙﾄｶｰﾈﾙ	ページ/ 参考文献
カーネル・トレーシング				
ｶｰﾈﾙ･ﾄﾚｰｼﾝｸﾞ有効	Kernel Tracing	XTRACE	Y / trace, debug N / generic	page D-1
ｸﾗｯｼｭ・ﾀﾞﾝﾌﾟからxtraceﾃﾞｰﾀを抽出		XTRACE_CRASH	Y / debug, trace N / generic	n/a
nVIDIAグラフィクスのサポート	Device Drivers	NVIDIA	M / all	Release Notes (0898003)
	Unified Memory CUDA Support	CONFIG_NVIDIA_UVM	M / all	
ﾊｲﾊﾟｰｽﾚｯﾃﾞｨﾝｸﾞ	Processor Type and Features	X86_HT	Y / all	page 2-28
UIOサポート	Userspace I/O	UIO	Y / all	page 14-14
*** Y = 設定, N = 非設定, M =カーネル・モジュールがロードされた時に有効**				

C
ケーパビリティ

本付録では、RedHawk Linuxに含まれるケーパビリティと各ケーパビリティが提供するパーミッションを掲載します。

概要

ケーパビリティは、スーパーユーザーに関連する伝統的な権限が個別に有効および無効にすることが可能な別個のユニットに分けられたLinuxの方式です。無節操なユーザーは、Linuxが提供するセキュリティ・メカニズムを無効にするためのケーパビリティが提供されたパーミッションの一部を使用することが可能であるため、この機能は十分に注意して使用する必要があります。ケーパビリティは**/usr/include/linux/capability.h**で定義されています。

ケーパビリティをLinuxで機能させる方法に関する詳細な情報については、**capabilities(7)**のmanページを参照して下さい。ケーパビリティを利用した認証スキームを提供するPAM機能に関する情報については、13章を参照して下さい。

ケーパビリティ

本セクションでは、RedHawk Linuxの下で定義される各ケーパビリティより提供されるパーミッションについて説明します。標準Linuxからの機能だけでなくRedHawk Linux独自の機能もこの説明に含まれています。

CAP_CHOWN

本ケーパビリティは、有効なユーザーID、グループID、追加グループIDの1つがファイルのUID/GID属性と一致しない時、ユーザーまたはグループのファイル所有権の変更の制限を無効にします。

CAP_DAC_OVERRIDE

不変または追加専用(**chattr(1)**を参照して下さい)としてマークされたファイルによって強要されたファイル・アクセス制限を除き、アクセス制御リスト(Access Control List: ACL)のサポートがファイルシステム用にカーネルに構成されている場合に、このケーパビリティは、どのようなファイルも所有者/グループ/その他ユーザー、読み込み/書き出し/実行のファイルシステム・パーミッション属性とACL制限を標準的に強要される随意アクセス制御(Discretionary Access Control: DAC)を無効にします。(詳細は**acl(5)**を参照して下さい)

読み込みと書き出しのアクセスDAC制限はこのケーパビリティによって常に無効となる可能性があります。DAC制限の実行は、少なくとも1つの所有者/グループ/その他ユーザー、実行ビットが設定されている限りケーパビリティによって無効となる可能性があります。

本ケーパビリティは、**fbsintrpt(3)**および**fbsresume(3)**コマンドを使用している時にパーミッション・アクセス制限もまた無効とします。

CAP_DAC_READ_SEARCH

本ケーパビリティはACLのサポートがファイルシステム用にカーネルに構成されている場合に、このケーパビリティは、どのようなファイルも所有者/グループ/その他ユーザー、読み込み/実行のファイルシステム・パーミッション属性とACL制限を標準的に強要されるDACを無効にします。(詳細は**acl(5)**を参照して下さい)

本ケーパビリティは常にファイルおよびディレクトリの読み取りアクセス、ディレクトリの検索(実行)アクセスを許可します。

本ケーパビリティは、**fbsintrpt(3)**および**fbsresume(3)**コマンドを使用している時にパーミッション・アクセス制限もまた無効とします。

CAP_FOWNER

本ケーパビリティは：

- ファイル所有者IDがユーザーIDと等しくなければならないファイル属性の変更に関する全てのDAC制限を無効にします。
- fbs作成者ユーザーIDおよびユーザーIDが呼び出し元の有効なユーザーIDと一致しない時に**fbsctl(2)**コマンドのFBS_RMIDおよびFBS_SETを許可します。

本ケーパビリティはDAC制限を無視しません。

CAP_FSETID

本ケーパビリティは、ファイルのS_ISGIDビットを設定している場合に有効なグループID(または追加グループのいずれか)がファイルのグループIDと一致しなければならない制限を無視します。

CAP_IPC_LOCK

本ケーパビリティは、**mlock(2)**および**mlockall(2)**システム・サービス・コールを通してメモリのロックを許可します。

これは**shmctl(2)**コマンドのSHM_LOCKおよびSHM_UNLOCKを通して共有メモリ領域のロック、アンロックもまた許可します。

CAP_IPC_OWNER

本ケーパビリティはIPC共有メモリ領域、メッセージ・キュー、セマフォ配列に関連するIPCパーミッション・セットを無効にします。IPCパーミッションはファイルに関連する読み込み/書き出し、所有者、グループ、その他ユーザーのパーミッションと同じフォーマットと意味を持っています。実行のパーミッションは使用されないことに注意して下さい。**ipcs(1)**コマンドは現在のIPCリソースの所有者とパーミッションを見るために使用することが可能です。

CAP_KILL

本ケーパビリティは、シグナルを送信するプロセスの実際のまたは有効なユーザーIDはシグナルを受信するプロセスの実際のまたは有効なユーザーIDと一致する必要があるという制限を無視します。

本ケーパビリティは、ttyの所有者になるため、またはCAP_SYS_TTY_CONFIGケーパビリティを所有するために呼び出し元プロセスに要求するKDSIGACCEPT **ioctl(2)**呼び出しの制限もまた無視します。

CAP_LEASE

本ケーパビリティは、プロセスのユーザーIDがファイル・システムのユーザーID値と一致しない場合であっても、**fcntl(2)**のF_SETLEASEコマンドによりファイルの資産をユーザーに取得させます。

CAP_LINUX_IMMUTABLE

本ケーパビリティは、ファイル属性S_IMMUTABLEとS_APPENDの変更を可能にします。これらのファイル属性に関する詳細な情報については、**chattr(1)**のmanページを参照して下さい。

CAP_MKNOD

本ケーパビリティは、**mknod(1)**/**mknod(2)**の特権の側面を利用することを許可します。これはXFS_IOC_FSSETDM_BY_HANDLE xfsファイルシステム **ioctl(2)**コマンドの使用もまた許可します。

CAP_NET_ADMIN

本ケーパビリティは、以下のネットワーク管理作業を許可します：

- ソケット上のデバッグおよび優先度オプションの設定
- IPファイヤウォール、マスカレード、アカウンティングの管理
- インターフェース構成
- マルチキャスト
- ネットワーク・デバイス・レジスタの読み書き
- トンネリングの追加/削除
- ルーティング・テーブルの変更
- TOS (Type Of Service)の設定
- ATM制御ソケットのアクティベイション

CAP_NET_BIND_SERVICE

本ケーパビリティは、1024未満のTCP/UDPおよびストリーム制御転送プロトコル(Stream Control Transmission Protocol: SCTP)ソケット、32未満のATM VCLへのバインドを許可します。

本ケーパビリティは、RPCクライアント・トランスポート作成時に使用するため予約済みポートももたらします。

CAP_NET_BROADCAST

本ケーパビリティは現在使われていません。

CAP_NET_RAW

本ケーパビリティは、SOCK_RAWおよびSOCK_PACKETソケットの作成、**setsockopt(2)**のSO_BINDTODEVICEソケット・オプションの使用を許可します。

CAP_SETGID

本ケーパビリティは、**setregid(2)**, **setgid(2)**, **setresgid(2)**, **setfsgid(2), setgroups(2)**システム・サービス用に非ルートプロセスのグループID値に設定された制限を無視します。

本ケーパビリティは、現在のプロセスの現在の/有効な/保存されたグループIDと一致しないグループID値を含むソケット・レベル証明書制御メッセージの送信をプロセスに許可します。(更に、証明書制御メッセージのプロセスIDはプロセスのスレッド・グループIDと一致する、またはプロセスはCAP_SYS_ADMINケーパビリティを持っている必要があり、証明書制御メッセージのユーザーIDはプロセスの保存された/有効な/現在のユーザーIDと一致、またはCAP_SETUIDケーパビリティを持っている必要があります。)

CAP_SETPCAP

本ケーパビリティは、プロセスの認められたセットの任意のケーパビリティを任意のプロセスID(PID)へ転送すること、プロセスの認められたセットの任意のケーパビリティを任意のPIDから削除することをプロセスに許可します。

CAP_SETUID

本ケーパビリティは、スーパーユーザーのユーザーIDを含む任意のユーザーIDに現在のユーザーIDを設定することを許可します。本ケーパビリティを許可している時は細心の注意を払う必要があります。

本ケーパビリティは、現在のプロセスの現在の/有効な/保存されたユーザーIDと一致しないユーザーID値を含むソケット・レベル証明書制御メッセージの送信をプロセスに許可します。(更に、証明書制御メッセージのプロセスIDはプロセスのスレッド・グループIDと一致する、またはプロセスはCAP_SYS_ADMINケーパビリティを持っている必要があり、証明書制御メッセージのグループIDはプロセスの保存された/有効な/現在のグループIDと一致、またはCAP_SETGIDケーパビリティを持っている必要があります。)

本ケーパビリティは、ptraceを実行されているプロセスが実行する "setuid/setgid" のユーザーIDまたはグループIDを継承しないプロセスによってptraceを実行されているプロセスの制限を無視します。

CAP_SYS_ADMIN

本ケーパビリティは、以下のシステム管理作業を提供します：

- **bdflush(2)**の使用の許可
- オープンするファイルの制限を無視
- ディスク割当量の調査および構成を許可
- XFSファイルシステム下でユーザー単位またはグループ単位でディスクの使用状況の検査および構成を許可(XFS_QUOTAが有効の場合)
- **umount()**と**mount()**を許可
- **fork(2)**/**clone(2)**呼び出し中にプロセスの名前空間のコピーを許可
- プロセスの有効なユーザーIDと一致するユーザーIDまたは所有者ユーザーIDを持っていないメッセージ・キュー、セマフォ、共有メモリ領域のための**msgctl(2)**, **semctl(2), shmctl(2)** のIPC_SETと IPC_RMIDを許可
- 共有メモリ領域のユーザーIDまたは所有者ユーザーIDがプロセスの有効なユーザーIDと一致しない共有メモリ領域のための **shmctl(2)** SHM_PHYSBINDを許可
- 非ルート・ユーザーがCAP_SYS_RESOURCEケーパビリティを持っていない場合に**fork(2)**/**clone(2)**呼び出ししているプロセス毎にプロセスの最大数に関する制限を無視
- 起こされるプロセスが呼び出し元プロセスの有効なユーザーIDまたはユーザーID値と同じユーザーIDまたは保存ユーザーIDを持っていない場合に**pw_post(2)**, **pw_postv(2)**, **server_wake1(2), server_wakevec(2)**呼び出しにより起こすのを許可
- RCIM_WRITE_EEPROMとRCIM_TESTIRQ **ioctl(2)** RCIMドライバ・コマンドの利用を許可
- システム・ダンプ**ioctl(2)**コマンドの利用、および**sysctl(2)**のkernel.dump.device変数の設定を許可
- シリアル・ポートの構成を許可
- **sethostname(2)**および**setdomainname(2)**呼び出しの許可
- **swapon(8)**および**swapoff(8)**呼び出しの利用を許可

- HP SA 5xxxおよび6xxxコントローラ用ディスク・アレイ・ドライバのRAWボリュームをゼロでのオープンおよびCCISS_SETINTINFOとCCISS_SETNODENAME **ioctl(2)**コマンドを許可
- Mylex DAC960 PCI RAIDコントローラ・ドライバの**ioctl(2)**コマンドを許可
- Compaq SMART2コントローラ・ディスク・アレイ・ドライバのRAWボリュームをゼロでのオープンを許可
- フロッピーをルートのみ**ioctl(2)**コマンド(0x80ビットを設定)の利用、およびFDSETPRMとFDDEFPRMのジオメトリ・コマンドも許可
- 次のブロック・デバイスでの**ioctl(2)**コマンドの利用を許可：BLKPG(パーティションの追加/削除)、BLKRRPART(パーティションの再読み取り)、BLKRASET(ブロック・デバイスに先読みを設定) 、BLKFRASET(ファイルシステムの先読みを設定)、BLKBSZSET(論理ブロック・サイズを設定)、BLKFLSBUF(バッファ・キャッシュをフラッシュ)、BLKROSET(デバイスを読み取り専用に設定)
- ループバック・ファイルシステムで暗号キーの設定を許可
- ネットワーク・ブロック・デバイスで**ioctl(2)**コマンドを許可
- MTRR(Memory Type Range Registers)の変更を許可
- カーネルのAPMが有効の場合に電源管理に**ioctl(2)**コマンドの使用を許可
- 特定のBIOS設定用に一部の**ioctl(2)**コマンドの使用を許可
- VM86_REQUEST_IRQ **vm86(2)**サポートの使用を許可
- CDROMRESET, CDROM_LOCKDOOR, CDROM_DEBUG **ioctl(2)** CDROMコマンドの使用を許可
- sbpcd CDROMドライバのDDIOCSDBG DDIデバッグ **ioctl(2)**を許可
- 読み取り専用DRM(Direct Rendering Manager) **ioctl(2)**コマンド、DRM **mmap(2)** DMAメモリ・コマンドの使用を許可
- Specialix RIOスマート・シリアル・カード・ドライバで読み取り専用**ioctl(2)**コマンドの使用を許可
- I2Cシリアル・バスでVAIO EEProm センサー・チップの先頭16byteの読み取りを許可
- **/proc/ide/ide*n*/config**ファイルへの書き込み、IDEドライブ設定の変更、以下の**ioctl(2)** IDEコマンドを許可：HDIO_DRIVE_TASKFILE (RAWタスクファイルの実行), HDIO_SET_NICE (ナイス・フラグの設定), HDIO_DRIVE_RESET (デバイス・リセットの実行), HDIO_GET_BUSSTATE (ハードウェア・インターフェースのバス状態を取得), HDIO_SET_BUSSTATE (ハードウェア・インターフェースのバス状態を設定)
- SNDRV_CTL_IOCTL_POWERサウンド**ioctl(2)**コマンドの使用を許可
- Live!やSound Blaster 512のような様々なPCIベースのサウンド・カード用の様々なルート専用**ioctl(2)**コマンドの使用を許可
- 試験的なネットーワークSIOCGIFDIVERTとSIOCSIFDIVERTフレーム・ダイバー**ioctl(2)**タコマンドの使用を許可
- 証明書のユーザーIDが現在のプロセスの有効な/保存された/現在のユーザーID値と一致しない場合、SCM_CREDENTIALSソケット・レベル制御メッセージの送信を許可

- mdデバイス(Multiple Devices－RAIDやLVM)の管理を許可
- デジタル・ビデオ放送インターフェースの追加と削除を許可
- CAP_SYS_RAWIOケーパビリティが有効ではない場合、Philips saa7134-based TV card video4linuxデバイス・ドライバ用のVIDIOC_S_FBUF **ioctl(2)**コマンドを許可
- CAP_SYS_RAWIOケーパビリティが有効ではない場合、bttvとZoranビデオ・デバイス・ドライバのVIDIOCSFBUFおよびVIDIOC_S_FBUF **ioct(2)**コマンドの使用を許可
- CAP_SYS_RAWIOケーパビリティが有効ではない場合、planbビデオ・デバイス・ドライバのVIDIOCSFBUF **ioctl(2)**コマンドの使用を許可
- stradis 4:2:2 MPEGデコーダ・ドライバのVIDIOCSFBUF **ioctl(2)**コマンドの使用を許可
- インテリジェント入力/出力(I2O) **ioctl(2)**コマンドの使用を許可
- ISDN CAPIサポート・ドライバの製造会社コマンドを許可
- PCIバス空間を最大245byte(非標準化された領域)の読み取り、**pciconfig_read(2)**および**pciconfig_write(2)**システム・サービス・コールの使用を許可
- ルート専用PCMCIA **ioctl(2)**コマンドの使用を許可
- aacraid Adaptec RAIDドライバのFSACTL_SEND_RAW_SRB **ioctl(2)**コマンドの使用を許可
- QLogic ISP2x00 NVRAMの読み書きを許可
- MegaRAID **ioctl(2)**コマンドへのアクセスを許可
- MTSETDRVBUFFER SCSIデープ・ドライバ**ioctl(2)**コマンドの使用を許可
- カーネルのSCSI_DEBUGが有効である場合(CAP_SYS_RAWIOケーパビリティが必要)、**/proc** SCSIデバッグ・ファイルへの書き込みアクセスを許可
- SCSI_IOCTL_SEND_COMMAND **ioctl(2)**コマンドを介して任意のSCSIコマンドの送信を許可(CAP_SYS_RAWIOケーパビリティが必要)
- SCSIスキャッター・ギャザーSG_SCSI_RESET **ioctl(2)**コマンド、**/proc/sg/allow_dio**および**/proc/sg/def_reserved_size write(2)**の使用を許可(CAP_SYS_ADMINケーパビリティが必要)
- Quicknet Technologies Telephonyカード・ドライバ用のIXJCTL_TESTRAMとIXJCTL_HZ **ioct(2)**コマンドの使用を許可
- 一部のautofsルート専用**ioctl**を許可
- ファイルシステム・オブジェクトの拡張属性の取得および設定を許可(**getfattr(1)**, **setfattr(1)**)
- NCP(NetWare Core Protocol)ファイルシステム用のルート専用コマンドを許可
- 新しいsmbファイルシステム接続の設定を許可

- UDFファイルシステムのUDF_RELOCATE_BLOCKS **ioctl(2)**コマンドを許可(一部のCD-ROMおよびDVDで使用)
- ランダム・デバイスの管理を許可
- ブロック・デバイスへRAWキャラクタ・デバイス(**/dev/raw/raw***n*)のバインディングを許可
- カーネルのsyslogの設定を許可(printk動作)
- カーネルのSBSVMEが有効の場合にPCI-to-VMEバス・マッピングの作成および削除するために**/proc/driver/btp/***unit#***/vmemappings**ファイルへの書き込みを許可
- プリアロケート・グラフィック・メモリ・プールのサイズを修正するために**/proc/driver/graphics-memory**への書き込みを許可

CAP_SYS_BOOT 本ケーパビリティは、**reboot(2)**システム・サービス・コールの使用を許可します。

CAP_SYS_CHROOT 本ケーパビリティは、**chroot(2)**システム・サービス・コールの使用を許可します。

CAP_SYS_MODULE 本ケーパビリティは、**sys_delete_module(2)**, **init_module(2)**, **rmmod(8), insmod(8)**を使いカーネル・モジュールの挿入、削除を許可します。

本ケーパビリティはカーネル・ケーパビリティに結合している設定値cap_bset(**sysctl(2)** kernel.cap-boundパラメータを介してアクセス可能な値)の修正もまた許可します。

CAP_SYS_NICE 本ケーパビリティは、以下を許可します：
- 同じユーザーIDを持つプロセスのスケジューリング優先度を上昇
- 異なるユーザーIDを持つほかのプロセスの優先度を設定
- 同じユーザーIDを持つプロセスに対しSCHED_FIFOおよびSCHED_RRスケジューリング・ポリシーを設定
- 異なるユーザーIDを持つプロセスのスケジューリング・ポリシーを変更
- **sched_setaffinity(2)**または**/proc/***pid***/affinity**ファイルを介して異なるユーザーIDを持つプロセスに対してCPUアフィニティを変更
- **fbsconfigure(3)**の使用を許可

CAP_SYS_PACCT 本ケーパビリティは、**acct(2)**システム・サービス・コールを通してプロセス・アカウンティングの構成を許可します。

CAP_SYS_PTRACE 本ケーパビリティは、プロセスが他のプロセスに**ptrace(2)**を実行することを許可します。CAP_SETUID設定に関係なく、本ケーパビリティは、setuid実行ファイルに**ptrace(2)**を実行することもまた許可します。

CAP_SYS_RAWIO 本ケーパビリティは、以下のRAW I/O動作を許可します：
- **shmctl(2)** SHM_PHYSBINDコマンド
- **resched_cntl(2)** RESCHED_SET_VARIABLEコマンド
- PCIバス空間の**mmap(2)**およびPCI BAR(Base Address Registers)へのアクセス

- **/dev/port**および**/proc/kcore**の**open(2)**
- **ioperm(2)**および**iopl(2)**システム・サービス・コールの利用
- ファイルシステム**ioctl(2)** FIBMAPコマンド
- カーネルのMICROCODEが有効の場合、**/dev/cpu/microcode**ファイルの**open(2)**
- HP SA 5xxx and 6xxxコントローラ**ioctl(2)**コマンド用の次のディスク・アレイ・ドライバ：
  CCISS_PASSTHRU, CCISS_BIG_PASSTHRU, CCISS_DEREGDISK, CCISS_REGNEWD
- Compaq SMART2コントローラ用のディスク・アレイ・ドライバの**open(2)**、およびIDAPASSTHRU **ioctl(2)**コマンド
- IDEコントローラの構成、および次のIDE **ioctl(2)**コマンド：
  HDIO_DRIVE_TASKFILE, HDIO_DRIVE_CMD, HDIO_DRIVE_TASK, HDIO_SCAN_HWIF, HDIO_UNREGISTER_HWIF
- ファイバ・チャネル・ホスト・バス・アダプタ CPQFCTS_SCSI_PASSTHRU **ioctl(2)**コマンド
- カーネルのSCSI_ DEBUGが有効の場合、**/proc** SCSIデバッグ・ファイルへの書き込みアクセス(CAP_SYS_ADMINも必要)
- SCSI_IOCTL_SEND_COMMAND **ioctl(2)**コマンドを介して任意のSCSIコマンドの送信(CAP_SYS_ADMINも必要)
- SCSIスキャッター・ギャザーSG_SCSI_RESET **ioctl(2)**コマンド、**/proc/sg/allow_dio, /proc/sg/def_reserved_size write(2)**の使用(CAP_SYS_ADMINケーパビリティも必要)
- ATMSIGD_CTRL **ioctl(2)**コマンド
- CAP_SYS_ADMINケーパビリティが有効ではない場合、bttvおよびZoranビデオ・デバイス・ドライバのVIDIOCSFBUFおよびVIDIOC_S_FBUF **ioctl(2)**コマンドの使用
- CAP_SYS_ADMINケーパビリティが有効ではない場合、planbビデオ・デバイス・ドライバのVIDIOCSFBUF **ioctl(2)**コマンドの使用
- Baycom EPP無線およびHDLCパケット無線ネットワーク・デバイス・ドライバのHDLCDRVCTL_SETMODEMPARおよびHDLCDRVCTL_CALIBRATE **ioctl(2)**コマンドの使用
- AX.25デバイス・ドライバ用Z8530 based HDLCカードのSIOCSCCCFG, SIOCSCCINI, SIOCSCCSMEM, SIOCSCCCAL **ioctl(2)** コマンド
- AM無線モデム・デバイス・ドライバのSIOCYAMSCFG **ioctl(2)**コマンド
- SRPおよびCOSA同期シリアル・カード・デバイス・ドライバ用のCOSAIOSTRT, COSAIODOWNLD, COSAIORMEM, COSAIOBMSET **ioctl(2)**コマンド
- SiSフレーム・バッファ・デバイス・ドライバ用のFBIO_ALLOCおよびFBIO_FREE **ioctl(2)**コマンド
- CAP_SYS_ADMINが無効である場合、Philips saa7134-based TV card video4linuxデバイス・ドライバ用のVIDIOC_S_FBUF **ioctl(2)** コマンド

CAP_SYS_RESOURCE

本ケーパビリティは、ユーザーに以下を許可します：

- ディスク割り当て制限を無視
- **msgctl(2)** IPC_SETコマンドでIPCメッセージ・キューのサイズ制限を無視
- 非ルート・ユーザーがCAP_SYS_ADMIN ケーパビリティを持っていない場合、**fork(2)**/**clone(2)**呼び出ししているプロセス毎にプロセスの数を無視
- **setrlimit(2)**システム・サービスでユーザーのリソース制限を増加
- リアルタイム・クロック(rtc)の周期的IRQレートを設定、または64Hzを超える周期に周期的IRQ割り込みの有効化
- コンソール・ターミナルのオープン/割り当て数の制限を無視
- コンソール・キーボードのキーマップ数の制限を無視
- ufs/ext2/ext3ファイルシステム上で追加で空間を割り当てる場合、予約済み空間量の制限を無視
  Note: リソースのチェックを無視する場合にext2ファイルシステムはファイル・システムのユーザーIDを受け取り、**setfsuid(2)**の使用を無視することも許可
- ext3ファイルシステムにおいて、データ・ジャーナリング・モードを変更

CAP_SYS_TIME

本ケーパビリティは、以下を許可します：

- **clock_settime(2)**, **stime(2)**, **settimeofday(2), adjtimex(2)**を介して時間の設定または修正
- **/dev/rtc**リアルタイム・クロック・デバイスに対してRTC_SET_TIMEおよびRTC_EPOCH_SET **ioctl(2)**コマンドの使用

CAP_SYS_TTY_CONFIG

本ケーパビリティは、以下を許可します：

- **vhangup(2)**システム・サービスの使用
- 全てのコンソール・ターミナルおよびキーボードの**ioctl(2)**コマンドの使用(ユーザーがコンソール・ターミナルの所有者でない場合のケースを含む)

  ユーザーがコンソール・ターミナルの所有者であっても KDKBDREP, KDSETKEYCODE, VT_LOCKSWITCH, VT_UNLOCKSWITCHコンソール・ターミナルおよびキーボード **ioctl(2)**コマンドは本ケーパビリティを必要とすることに注意して下さい。

D
32bitコードから64bitコードへの移植

本付録ではx86_64アーキテクチャ上の64bit処理へ32bitコードを移行するために必要となる情報について提供します。

序文

RedHawk Linuxのバージョン2.X以降は、64bit AMD OpteronおよびEM64Tプロセッサだけでなく32bit Intel Pentium Xeonプロセッサ上でも実行することが可能です。RedHawk Linuxのx86_64バージョンは、x86_64プロセッサ上でネイティブ・モードで32bitおよび64bitの両方を実行する完全な64bitオペレーティング・システムです。

Opteronプロセッサは、EM64T ISA(Instruction Set Architecture)をサポートする最近のIntelプロセッサ(例：全てのIntel Noconaプロセッサ)と殆ど同じAMD64 ISAを利用しています。AMD64とEM64Tの両方とも真の64bit実行が可能であり、“x86_64” アーキテクチャとして知られています。

x86_64プロセッサの“long” 実行モードは2つのサブモード(“64bit” および “互換”)を持っています。既存の32bitアプリケーションのバイナリは、RedHawk Linux下で互換モードで再コンパイルせずに実行する、もしくはアプリケーションを64bitモードで実行するために再コンパイルすることが可能です。

32bitアプリケーションは、性能が低下する「エミュレーション・モード」ではなくネイティブに実行されます。この理由から、多くのアプリケーションは64bitへ移植する必要はありません。

NOTE

リアルタイム拡張および機能は、64bitオペレーティング・システム(x86_64)の下で動作する32bitアプリケーションでは利用できません。リアルタイム機能を利用するためには、32bitアプリケーションを64bitへ移行する、もしくは代わりに32bitオペレーティング・システムを起動します。

x86_64用に最適化されたソフトウェアは、科学技術計算、データベース・アクセス、シミュレーション、CADツール等のような最も要求の多いアプリケーションに必要とされる大きなアドレス指定可能なメモリや64bitアーキテクチャ上の機能強化を利用することが可能です。もしアプリケーションが64bit処理で得られるより大きな仮想および物理アドレス空間から恩恵を受ける場合、本セクション内の情報は所有されているコードの移行に役に立つでしょう。

既存の32bitアプリケーションの64bitへの移植は、以降のセクションで詳述されている次の範囲を必要とします：

- 32bit用に記述されたソース・コードは、64bitモードで実行するための修正が恐らく必要となります。
- 32bit演算用にコンパイルされたバイナリは、64bitモードで実行する前に64bit用に再コンパイル必要があります。

- ビルド処理(Makefile, プロジェクト・ファイル, 他)は、64bitの実行可能ファイルを構築するために更新およびコンパイル用オプションを調べて移植性を追加する必要がある可能性があります。
- 唯一64bitのデバイス・ドライバだけは、64bitオペレーティング・システムで使用することが可能です。必要とするドライバの64bit版が存在しない場合、デバイス・ドライバを組み込むアプリケーションは正確に動作しない可能性があります。RedHawk Linuxが供給する全てのドライバは64bit互換です。

更に、お手持ちのアプリケーションから最大限性能を得るためのヒントを提供します。

AMD64 Developer Resource Kitは、Opteronプロセッサ用のアプリケーションおよびドライバの移植または開発を行うプログラマのために全てが揃ったリソースです。AMD64 DRKは、資料、ホワイト・ペーパー、詳細なプレゼンテーション、リファレンス・ガイドを含む技術情報を収納しています。本キットは、**www.amd.com** のWEBサイトから入手することが可能です。

手順

体系的に64bitへ移植するためにお手持ちのコードの修正に取り組むため、以下のガイド・ラインに従ってください。ヘッダ/インクルード・ファイル、リソース・ファイル、Makefileを含む全てのソース・ファイルは再調査およびそれに応じた修正をする必要があます。これらの手順に関する詳細は以降のセクションで提供されます。

- AMD64アーキテクチャ固有のコード用に`#if defined __x86_64__` or `__amd64__` を使用
- 組み込み関数またはネイティブ・アセンブリ・サブルーチンを使用するために全てのインライン・アセンブリ・コードを変換
- 必要に応じて既存のアセンブリ・コードの呼び出し規約を修正
- ポインタ演算の使用の再調査および結果の確認
- ポインタ、整数、物理アドレスへの参照の再調査および32bitと64bitアーキテクチャの違いに対応するため可変サイズのデータ型を使用
- 64bit実行可能ファイルをビルドするためにMakefileの調査および移植性をチェックするオプションの追加

コーディング要件

データ型のサイズ

32bitと64bitの移植性の主要な問題は、アドレスのサイズまたは`int`, `long`等のサイズとの関連に関して推定があってはならないということです。

表D-1は、AMD64システム上のRedHawk Linux下での様々なANSIデータ型のサイズを示します。

表D-1 データ型のサイズ

ANSIデータ型	サイズ(Byte)
`char`	1
`short`	2
`int`	4
`long`	8
`long long`	8
`intptr_t, uintptr_t`	8
`float`	4
`double`	8
`long double`	16

様々なデータ型のサイズを取得するために「`sizeof`」演算子を使用することが可能です。(例：もし変数`int x`がある場合、`sizeof(x)`により`x`のサイズを取得することが可能となります)この使用法は構造体もしくは配列に対しても働きます。例えば、`a_struct`という名前の構造体型変数がある場合、どれくらいメモリが必要となるのかを調べるために`sizeof(a_struct)`を使用することが可能です。

long型

`long`型は64bitとなるため、`long`と`int`の値間で直接または暗黙的な割り当てまたは比較をすべて調査する必要があります。有効性を確実にするため`long`と`int`の間の割り当ておよび比較を認めることをコンパイラに任せるすべてのキャストを調査して下さい。`long`のサイズを解決するためにBITS_PER_LONGマクロの値を利用して下さい。

もし`int`と`long`が異なるサイズのままでなければならない場合(例：既存の公開`API`定義のため)、`64bit`項目の値が`32bit`項目の最大値を超えないことを確かめるアサーションを実装し、それが発生した場合に対処するための例外条件を生成して下さい。

ポインタ

ポインタは64bitとなるため、ポインタと`int`の値間で直接または暗黙的な割り当てまたは比較もまたすべて調査する必要があります。ポインタと`int`の間の割り当ておよび比較を認めることをコンパイラに任せるすべてのキャストを削除して下さい。(ポインタのサイズと等しい)可変サイズ型へ型を変更して下さい。表`D-2`は可変サイズのデータ型を示します。

表 D-2 可変サイズのデータ型

ANSIデータ型	定義
intptr_t	ポインタを格納するための符号付き整数型
uintptr_t	ポインタを格納するための符号なし整数型
ptrdiff_t	2つのポインタ値の符号付き差分を格納するための符号付き型
size_t	ポインタが参照可能な最大バイト数を示す符号なしの値
ssize_t	ポインタが参照可能な最大バイト数を示す符号付きの値

配列

32bitコードの下では、intとlongは配列のサイズを格納するために使用することが可能です。64bitの下では、配列は4GBよりも長くすることが可能です。intまたはlongに代わって、移植性のためにsize_tデータ型を使用してください。これは64bitターゲット用に、もしくは32bitで32bitターゲット用にコンパイルした場合に64bit符号付き整数型となります。sizeof()およびstrlen()の両方からの戻り値は、どちらもsize_t型です。

宣言

表D-2で示されるサイズ可変型のいずれかを使用するために64bitへ変更する必要のある変数、パラメータ、関数/メソッドが返す型のどの宣言もまた修正する必要があります。

明示的なデータ・サイズ

明示的にアドレス・データのサイズが必要である場合、表D-3のデータ型を使用してください。本質的にデータのサイズを解決するANSIデータ型は存在せず、これらの型はLinux固有となります。

表D-3 固定精度のデータ型

データ型	定義
int64_t	64-bit符号付き整数
uint64_t	64-bit符号なし整数
int32_t	32-bit符号付き整数
uint32_t	32-bit符号なし整数
int16_t	16-bit符号付き整数
uint16_t	16-bit符号なし整数
int8_t	8-bit符号付き整数
uint8_t	8-bit符号なし整数

定数

定数(特に16進数または2進数の値)は、32bit仕様である確立が高いです。例えば、32bit定数の0x80000000は64bitでは0x0000000080000000になります。それが使用されている方法次第で、結果は好ましくないことになる可能性があります。この問題を回避するために「~」演算子および型接尾語を活用して下さい。(例：0x80000000定数は代わりに~0x7fffffffulとしても良いでしょう)

API

コードは64bitAPIを使用するように変更する必要がある可能性があります。一部のAPIは、明示的な32bitデータ型と競合する64bitとしてコンパイラが解釈することになるデータ型を使用します。

呼び出し規約

呼び出し規約はプロセッサ・レジスタが機能の呼び出し元と呼び出し先で使用する方法を明記します。これは、Cコードおよびインライン・アセンブリ記述を同時に使用するハンド・コーディングされたアセンブリ・コードを移植する場合に適用します。x86_64向けのLinux呼び出し規約は表D-4に記載されています。

表D-4 呼び出し規約

レジスタ	状態	用途
`%rax`	volatile	可変引数が使用されているSSEレジスタの数に関する情報を渡す一時的なレジスタ；最初に戻るレジスタ
`%rbx`	Non-volatile	任意にベース・ポイントとして使用、呼び出し先が保護する必要あり
`%rdi, %rsi, %rdx, %rcx, %r8, %r9`	volatile	整数の引数(1,2,3,4,5,6)を渡すために使用
`%rsp`	Non-volatile	スタック・ポインタ
`$rbp`	Non-volatile	フレーム・ポインタとして使用、呼び出し先が保護する必要あり
`%r10`	volatile	関数の静的チェーン・ポインタを渡すために使用する一時的なレジスタ
`%r11`	volatile	一時的なレジスタ
`%r12-%r15`	Non-volatile	呼び出し先が保護する必要あり
`%xmm0-%xmm1`	volatile	浮動小数点引数を渡すおよび返すために使用
`%xmm2-%xmm7`	volatile	浮動小数点引数を渡すために使用
`%xmm8-%xmm15`	volatile	一時的なレジスタ
`%mmx0-%mmx7`	volatile	一時的なレジスタ
`%st0`	volatile	long double引数を返すために使用する一時的なレジスタ
`%st1-%st7`	volatile	一時的なレジスタ
`%fs`	volatile	システムがスレッド固有のデータ・レジスタとして使用するために予約

条件付コンパイル

32bitと64bit実行用の条件付コードを提供する必要がある場合、表D-5のマクロを使用することが可能です。

表D-5 条件付コンパイル用マクロ

マクロ	定義
`__amd64__`	コンパイラはAMD64用のコードを生成します
`_i386`	コンパイラはx86用のコードを生成します

その他

その他の様々な問題は符号拡張、メモリ割り当てサイズ、桁送り、配列オフセットから生じる可能性があります。整数オーバーフローのセマンティクスに関する条件を構成する全てのコードについては特に注意して下さい。

コンパイル

既存のMakefileは、少しの修正もしくは修正なしでx86_64プロセッサ上でネイティブ64bitの実行ファイルを構築するはずです。

以下の**gcc**スイッチは移植性の問題を見つけるために使用することが可能です。詳細は**gcc(1)**のmanページを参照して下さい。

```
-Werror -Wall -W -Wstrict-prototypes -Wmissing-prototypes
-Wpointer-arith -Wreturn-type -Wcast-qual -Wwrite-strings
-Wswitch -Wshadow -Wcast-align -Wuninitialized -ansi
-pedantic -Wbad-function-cast -Wchar-subscripts -Winline
-Wnested-externs -Wredundant-decl
```

テスト/デバッグ

64bitコードに対して標準的なRedHawk Linuxのテストおよびデバッグ手法に従ってください。

性能問題

本章の情報は、お手持ちの64bitアプリケーションから最高のパフォーマンスを得る方法を説明します。

メモリのアライメントおよび構造体のパディング

アライメントの問題は例外は発生しませんが、性能の衝突を引き起こす可能性があります。アライメントの不整はいくつかのクロック・サイクルを犠牲にして実行時に処理されます。不十分に整列したオペランドの性能の副作用は大きくなる可能性があります。

構造体の中のデータは、結果として空間を無駄にするため非効率となる可能性のある境界線に自然と並べられます。自然な整列とは2byteオブジェクトは2byteの境界線上、4byteのオブジェクトは4byteの境界線上に格納されることを意味します。

例えば、以下の構造体の定義は64bitコードを生成するときに24byteを消費します：

```
typedef struct _s {
   int x;
   int *p;
   int z;
} s, *ps;
```

ポインタ`p`は、`x`メンバーの後に追加するために`4byte`のパディングを引き起こして`8byte`境界線上に整列されます。更に、構造体を`8byte`の境界線に合わせようと穴埋めするために`z`メンバーの後に`4byte`のパディングが追加されます。

最も効果的な構造体のパッキングは、構造体内で最大から最小へメンバーをパッキングすることにより実現されます。以下の宣言はより効果的です。これはたったの16byteで、どのようなパディングも必要としません：

```
typedef struct _s }
   int *p;
   int x;
   int z;
} s;
```

潜在的なパディングのために、構造体内のフィールドの一定のオフセットを見つける最も安全な方法は、**stddef.h**に定義されている`offsetof()`マクロを使用することです。

E
シールドCPU上のカーネル・レベル・デーモン

Linuxカーネルは、システム機能を実行するため多くのカーネル・デーモンを使用します。これらのデーモンの一部はシステムのCPU毎に複製されます。プロセスからのCPUシールディングはこれらの一部の「CPU毎」デーモンを除去しません。

以下のデーモンはプロセスをシールドしたCPU上で深刻なジッターの問題を引き起こす可能性があります。幸い、これらのデーモンは慎重にシステムを構成および使用することにより回避することが可能です。

kmodule *cpu*

これらのデーモンはカーネル・モジュールがアンロードされる度に作成および実行されます。リアルタイム・アプリケーションがシステム上で実行している間はカーネル・モジュールがアンロードされないことを強く推奨します。

migration/*cpu*

これらは特定のCPUからタスクを移動するために責任を負うタスク移動デーモンです。プロセス・シールドしたCPUで動作しているプロセスがそのCPUからの移動を強いられる状況において、これらのデーモンはプロセス・シールドしたCPU上で動作します。以下のいずれかのインターフェースが使用される時、強制的な移動が発生する可能性があります：

/proc/*pid***/affinity**
sched_setaffinity(2)
/proc/shield/procs
cpucntl(2)
delete_module(2)

バックグラウンド・プロセスのジッターが容認される可能性がある場合のみ、シールドCPU上で実行中のアプリケーションはこれらのインターフェースを使用する必要があります。

強制的な移動は、CPU_FREQおよびNUMAのカーネル構成オプションにより有効にすることが可能な様々なカーネル機能によっても行われます。これらのオプションは全てのRedHawk Linuxカーネル構成でデフォルトで無効にされています。

kswapd*node*

これらは、メモリが残り少なくなった時にページを回収するためにスワップ・ページをスワップ・デバイスへ追い出すページ・スワップ・アウト・デーモンです。

NUMA構成オプションが有効でカーネルが構築される時、各々がシングルCPUへ割り付けられたこれらのデーモンのいくつかが存在する可能性があります。CPUがプロセス・シールドされたまたは(**cpu(1)**を使い)ダウンされた時、デーモンはシールドされていないアクティブなCPUへ移動します。CPUがもはやシールドされていないまたはダウンされていない場合、デーモンは元へ戻されます。

NUMAが無効の時、これらは特定のCPUに割り付けられていない1つのシステム全体のデーモンとなるため、**kswapd**はプロセスからシールドされたCPUでは実行されず、非シールドCPU上の問題となります。

NUMAはプレビルトRedHawk x86_64カーネルのみデフォルトで有効になっています。

kapmd これは電源管理要求を処理する拡張型電源管理(APM: Advanced Power Management)デーモンです。これは常にCPU0へ割り付けられます。APMはカーネル・ブート・パラメータ "apm=off" で無効にする、またはAPMカーネル構成オプションを無効にすることで完全に排除することが可能です。APMは全てのRedHawk Linuxカーネル構成でデフォルトで無効となっています。何故ならこのデーモンはCPU毎デーモンではないため、プロセスからシールドされたCPUでは実行されず、その結果、非シールドCPU上でのみ問題となります。

以下のデーモンはプロセス・シールドされたCPU上で実行する可能性があります。しかし、これらはそのCPUへ割り付けられたプロセスまたは割り込みのために必要な機能を実行するため、これらのデーモンはシールドされたCPUへ割り付けられたプロセスまたは割り込みにより開始される処置の結果として作動されるだけであるため、デターミニズムに対する影響という点ではこれらのデーモンは問題は少ないと考えられます。

ksoftirqd/*cpu* これらは特定CPU用にソフトIRQルーチンを実行するソフトIRQデーモンです。デバイス・ドライバ割り込みハンドラが直接またはタスクレットを介して間接的にソフトIRQを使用する場合、これらのデーモンのいずれかがプロセス・シールドされたCPU上で実行されます。ソフトIRQはローカル・タイマー、SCSI、ネットワークの割り込みハンドラにより直接使用されます。タスクレットは多くのデバイス・ドライバにより使用されます。

ksoftirqdの優先度は、カーネル構成GUIの「General Setup」にあるSOFTIRQ_PRIカーネル・チューニング・パラメータにより決定されます。SOFTIRQ_PRIに正の数が設定される時、この数は**ksoftirqd**が実行される優先度です。規定値でこのチューニング・パラメータはゼロが設定され、SOFTIRQ_PREEMPT_BLOCKの設定はデーモンの優先度に影響を及ぼします。Yを設定した時、**ksoftirqd**デーモンは最も高いリアルタイム優先度よりも1小さい優先度でSCHED_FIFOスケジューリング・ポリシーとして実行されます。Nを設定した時、**ksoftirqd**デーモンは優先度ゼロで実行されます。

events/*cpu* これらは特定CPU上のプロセスにより開始される様々なカーネルサービスのために仕事を実行するデフォルトのワーク・キュー・スレッドです。これらは同じCPUへ割り付けられたデバイス・ドライバ割り込みルーチンにより保留された仕事を実行することも可能です。これらのデーモンは-10のナイス値で実行します。

aio/*cpu* これらは特定CPU上のプロセスにより完全な非同期I/O要求が**io_submit(2)**システムコールで起こされるワーク・キュー・スレッドです。これらのデーモンは-10のナイス値で実行します。

reiserfs/*cpu* これらはレイザー・ファイル・システムで使用されるワーク・キュー・スレッドです。これらのデーモンは-10のナイス値で実行します。

xfsdatad/*cpu*
xfslogd/*cpu* これはIRIXジャーナリング・ファイル・システム(XFS)で使用されるワーク・キュー・スレッドです。これらのデーモンは-10のナイス値で実行します。

cio/*cpu*
kblockd/*cpu*
wanpipe_wq/*cpu*

これらは様々なデバイス・ドライバに使用されるワーク・キュー・スレッドです。これらのスレッドは特定CPU上のプロセスによって開始される様々なカーネル・サービスのために仕事を実行します。これらは同じCPUへ割り付けられたデバイス・ドライバ割り込みルーチンにより保留された仕事を実行することも可能です。これらのデーモンは-10のナイス値で実行します。

どのサード・パーティのドライバでも、シールドCPUへ割り付けられたプロセスもしくは割り込みハンドラにより始動されるプライベート・ワーク・キューおよびワーク・キュー・スレッドを作成することが可能であることにも注意して下さい。これらのデーモンは常に*name/cpu* と命名され-10のナイス値で実行します。

F
シールドCPU上のプロセッサ間割り込み

本付録では、シールドCPU上でのプロセッサ間割り込みの影響および最高のパフォーマンスのためにこれらの割り込みの軽減、排除する方法について説明します。

概要

1つ以上のシールドCPUで構成されるRedHawkプラットフォームにおいて、他のCPUの特定の動作はシールドCPUへ割り込みが送信される要因となる可能性があります。これらのプロセッサ間割り込みは、例えば、それぞれのデータ・キャッシュのフラッシュまたはそれぞれのトランスレーション・ルックアサイド・バッファ・キャッシュ(TLB: Translation Look-aside Buffer)のフラッシュのような一部のCPU毎の特定タスクを処理することを他のCPUに強制するための方法として使用されます。

プロセッサ間割り込みは潜在的にシールドCPUに対する顕著なジッターを引き起こす可能性があるため、これらの割り込みが発生する原因となる動作、そしてこれらの割り込みの一部を排除するためにお手持ちのシステムを構成する方法を理解することに役立ちます。

メモリ・タイプ・レンジ・レジスタ(MTRR)割り込み

Intel P6ファミリー・プロセッサー(Pentium Pro, Pentium II以降)上のMTRR(Memory Type Range Register)は、メモリ領域へのプロセッサ・アクセスを制御するために使用することが可能です。これはPCIまたはAGPバス上にビデオ(VGA)カードがある場合に最も役に立ちます。Write-combiningを有効にするとPCI/AGPバス上で破棄する前により大きなデータ転送へ結合するためにバス書き込み転送を許可します。これは画像書き込み動作の性能を2.5倍以上向上することが可能です。

RedHawkカーネルに含まれているNVIDIAデバイス・ドライバは、ページ属性テーブル(PAT: Page Attribute Table)レジスタがシステム上のプロセッサにサポートされている場合、MTTRレジスタの代わりにCPUのPATレジスタを利用します。システムのプロセッサがPATサポートを含まない場合のみNVIDIAドライバはMTRRレジスタの使用へフォールバックします。従って殆どのシステムでは、プロセッサ間割り込みに関連するMTRRについて後述するこの問題は適用されません。

MTRRは有益な性能の恩恵を提供する一方、新しいMTRR領域が構成もしくは削除されるときはいつでも、それに応じて各々のCPUはCPU毎のMTRRレジスタを変更させるためにプロセッサ間割り込みが他のCPU全てに送信されます。この特定の割り込みを処理するために掛かる時間は非常に長くなる可能性があり、システムの全CPUはそれぞれのMTRRレジスタを修正する前に最初に同期/ハンドシェイクする必要があるため、それぞれの割り込みルーチンを終了する前にさらにもう一度ハンドシェイクを行う必要があります。

プロセッサ間割り込みのこの分野は、1割り込みにつき最大3ミリ秒に達してしまうデターミニズムに深刻な影響を及ぼす可能性があります。

システム起動後にXサーバーが最初に開始される時、MTRR領域が構成され、このMTRRプロセッサ間割り込みの1つがシステムの他のCPU全てに送信されます。同様にXサーバーが終了する時、このMTRR領域は削除され、システムのほかのプロセッサ全てはさらに他のMTRR割り込みを受信します。

以下の3つの方法は、シールドCPU上でタイム・クリティカル・アプリケーション実行中にプロセッサ間割り込みに関連するMTRRを排除するために利用することが可能です：

1. MTRRカーネル構成オプションが無効となるようにカーネルを再構成して下さい。カーネル構成GUIを使用する場合、このオプションは「Processor Type and Features」セクションにあり、"MTRR (Memory Type Range Register) support" と呼ばれています。この機能をサポートするカーネルは存在しないため、これはMTRRプロセッサ間割り込みを取り除きます。このオプションはグラフィックI/O動作に対し深刻な性能の不利益となる可能性があることに注意して下さい。

2. シールドCPU上でタイム・クリティカルなアプリケーションを実行する前にXサーバーを開始し、タイム・クリティカルなアプリケーションが完了するまでXサーバーを実行し続けてください。MTRR割り込みは発生し続けますが、タイム・クリティカルな動作中ではありません。

3. プロセッサ間割り込みが発生しないようにMTRR領域を事前に構成することが可能です。Xサーバーが使用するMTRRを事前に構成するため以下の手順を利用して下さい：

   a. システム起動後(Xサーバーを開始する前)、現在のMTRR設定を調査します。ランレベルは1または3のどちらかである必要があります。

      cat /proc/mtrr
      ```
      reg00: base=0x00000000 ( 0MB), size=1024MB: write-back, count=1
      reg01: base=0xe8000000 (3712MB), size= 128MB: write-combining,
      count=1
      ```

   b. Xサーバーが開始された後、MTRRレジスタの設定を再調査します：

      cat /proc/mtrr
      ```
      reg00: base=0x00000000 ( 0MB), size=1024MB: write-back, count=1
      reg01: base=0xe8000000 (3712MB), size= 128MB: write-combining,
      count=2
      reg02: base=0xf0000000 (3840MB), size= 128MB: write-combining,
      count=1
      ```

   c. この例では、新しいXサーバーのエントリは最後のエントリ"reg02" です。もしシステムに複数のグラフィック・カードが存在する、または1つ以上の新しいエントリを表示する場合、これらの追加エントリは更に**rc.local**スクリプトにも適用する必要があります。

   d. XサーバーのMTRRエントリを確保するために**/etc/rc.d/rc.local**スクリプトへ行を追加します。この例では1つのXサーバーエントリだけを確保します：

      ```
      echo "base=0xf0000000 size=0x8000000 type=write-combining" >
      /proc/mtrr
      ```

   e. システムのハードウェア構成が変更されるたびに、Xサーバーの起動および使用で**/etc/rc.d/rc.local**のMTRRエントリが間違っていないことをチェックすることは良いアイデアです：

cat /proc/mtrr

MTRRの出力を調査し、以前のMTRR設定との違いについて確認します。

グラフィクス割り込み

グラフィクス・アプリケーションが実行中は複数のプロセッサ間割り込みが発生します。

NVIDIAドライバといったカーネル・グラフィクス・ドライバは、NVIDIAグラフィクス処理ユニット(GPU: Graphics Processing Unit)のデータの読み書きをするために様々なキャッシュ禁止グラフィクス・メモリ・バッファを割り当ておよび構成します。

グラフィクス実行中にこれらのバッファが追加または削除されるたびに、これらバッファ・キャッシュ・モード移行様式のためのこれらのデータやTLBキャッシュをフラッシュするためにプロセッサ間割り込みがシステムの他の各々のCPUへ送信されます。これらのプロセッサ間割り込みの種類は、1割り込みにつき50～250μ秒に達してしまう相当深刻な影響を持っている可能性があります。キャッシュ禁止カーネル・グラフィクス・バッファの割り当てと解放が発生するのは次のとおり：

- Xサーバーの開始と終了
- グラフィクス・アプリケーションの実行
- Ctrl+Alt+F#キーボード操作による非グラフィクスTTYからグラフィクス画面への切り替え

NVIDIA PCIeおよび/またはPCIグラフィクス・カードによるシステムに関しては、これらのプロセッサ間割り込みの種類は、キャッシュ禁止バッファ・ページのプールがプリアロケート時に排除または低減される可能性があります。グラフィクス・バッファ割り当てが行われた時に、これらの要求を満足するために必要となるページがプリアロケート・ページ空きリストから取得されます。これらのページが既にキャッシュ禁止であるため、これらのページが使用された時に更なるフラッシュ操作をする必要がありません。バッファ割り当てが削除された時、ページはページ空きリストへ戻され、残りのキャッシュ禁止が取り除かれます。要求された時にプリアロケート・ページのプールが空であるならば、ページは動的に割り当てられプロセッサ間割り込みは通常の方法で発行されます。従って、利用可能なページのプールが決して空にならないように十分なページをプリアロケートすることが通常は最善です。

このサポートを有効にするには、PREALLOC_GRAPHICS_PAGESカーネル・パラメータはプール内のプリアロケート・ページの数を意味する正の値である必要があります。10240の値がすべてのプレビルトRedHawk Linuxカーネルに設定されています。あるいは、ブート・パラメータ“pregraph_pgs=*numpages*”はブート時にPREALLOC_GRAPHICS_PAGESに静的にコンパイルされた値を無効にするために使用する事が可能です。

このサポートを無効にするには、プレビルトRedHawk Linuxカーネルを使用しGRUBカーネル・パラメータ “no_pregraph_pgs”を指定、またはカスタム・カーネルを構築しPREALLOC_GRAPHICS_PAGESカーネル・パラメータに対しゼロの値を指定することが可能です。このサポートは、PREALLOC_GRAPHICS_PAGES パラメータの値に関係なくNVIDIA PCI/PCIeグラフィクス・カードが存在しないシステム上では常に無効となっています。

PREALLOC_GRAPHICS_PAGESオプションはカーネル構成GUIの「Device Drivers ->Graphics Support」のサブセクションにあります。

/proc/driver/graphics-memoryファイルは、グラフィクス・アプリケーションが実行している間、実際に使用しているグラフィクス・メモリ・ページの最大量を監視するためにいつでも調査することが可能です。例えば：

```
$ cat /proc/driver/graphics-memory
Pre-allocated graphics memory:                10240 pages
Total allocated graphics memory:              10240 pages
Graphics memory in use:                          42 pages
Maximum graphics memory used:                    42 pages
```

プール内のページ数を増やすまたは減らすためにファイルへ書き込むことが可能です。これはカーネル構成パラメータを変更する前に様々な値でお手持ちのシステムをテストすることを可能にします。以下の例はプール内のプリアロケート・ページ数を5120へ下げます。

```
$ echo 5120 > /proc/driver/graphics-memory
```

ユーザーはこのファイルへ書き込むためにCAP_SYS_ADMINケーパビリティを持っている必要があります。ファイルへ書き込むページの値は"Graphics memory in use" フィールドの現在値以上である必要があることに注意して下さい。もし現在割り当てられたページの数を低くする必要がある場合、Xサーバーを終了します。

非現実的な大きな値の指定はページ割り当ての失敗という結果になり、割り当ては取り消されます。ファイルへの書き込み後、ページ割り当てが成功したことを検証するためにファイルを読み出して下さい。

システムでプリアロケートされたページ数を表示および変更するために**graphics-memory(1)**ユーティリティもまた使用する事が可能です。詳細については**graphics-memory(1)**のmanページを参照してください。

同じシステム上で、NVIDIAドライバがロードまたはアンロードされた時、PATプロセッサ間割り込みは各CPUへ送信されます。生じるジッターを最小限に抑えるため、タイム・クリティカル・アプリケーションの実行中はNVIDIAモジュールのロードまたはアンロードは避けて下さい。タイム・クリティカル・アプリケーションの実行前、または以下のコマンドでシステム起動中にNVIDIAドライバをプリロードして下さい：

```
$ modprobe nvidia
```

NVIDIA CUDA割り込み

NVIDIA CUDAは、CPU上で必要となる時間のほんの一部で多くの複雑な計算の問題を解決するためにNVIDIA GPU内に存在する並列演算エンジンを利用した多目的並列演算アーキテクチャです。

CUDAアプリケーションはNVIDIA GPUのインターフェースにキャッシュ禁止バッファを利用するため、前のセクションで述べた同じプリアロケート・グラフィクス・バッファのサポートもまたCUDAアプリケーションが実行されているシステムのシールドCPU上のジッターを非常に低減することに役立ちます。

CUDAアプリケーションによるプリアロケート・グラフィクス・バッファの利用は、プール内のプリアロケート・バッファが使用されていない限り自動的に生じます。(特別なCUDAアプリケーションのコーディングまたは設定は必要ありません)

ユーザー空間でのTLBフラッシュ割り込み

シールドCPU上で実行するためにバイアスされたプロセスおよび他のCPU上で実行するプロセスのアドレス空間の共有はユーザー空間TLBフラッシュ・プロセッサ間割り込みを受信する可能性があります。共有メモリ領域は利用していますが同じCPU上のプロセスだけでアドレス空間を共有しているプロセスは、どのような共有メモリ動作に起因するプロセッサ間割り込みも気づくことはありません。

Pスレッド・ライブラリを使用するマルチスレッド・アプリケーションとAdaアプリケーションは共有メモリ・アプリケーションの実例です(プログラマーは共有メモリを作成するために明示的に呼び出しを行っていません)。これらのプログラムのタイプでは、Pスレッド・ライブラリとAdaの実行時はユーザーのために共有メモリ領域を作成しています。従って、これらのアプリケーションは、同じスレッド・グループまたは同じAdaプログラムからのスレッドがシステム内の別のCPU上で実行する時にこのタイプのプロセッサ間割り込みの影響を受けやすくなります。

ユーザー・アドレスTLBフラッシュ・プロセッサ間割り込みは、同じアドレス空間を共有している他のプロセスが異なるCPUで実行している時に発生し、そのアドレス空間属性の変更の原因となります。ページ・フォルト、ページ・スワップ、**mprotect()**呼び出し、共有メモリ領域の作成/削除等を引き起こすメモリ参照のような動作は、このタイプのプロセッサ間割り込みの原因になり得るアドレス空間属性変更の実例となります。この類のプロセッサ間割り込みは、1割り込みにつき最大10μ秒に達する小さな影響を与えます。大量のメモリを共有されている場合、影響はもっと深刻となる可能性があります。

これらのタイプのプロセッサ間割り込みを排除するために、シールドCPU上で実行するタイム・クリティカル・プロセスがそのアプリケーションのタイム・クリティカル部分の間中は共有メモリ領域に影響を与える操作を回避するようなアプリケーションをユーザーは利用および記述することを推奨します。これはメモリのページをロック、**mprotect()**を介したメモリ保護を変更しない、新しい共有メモリ領域を作成しない、既存の共有メモリ領域を削除しないことにより達成することが可能です。

G
シリアル・コンソールの設定

本章ではRedHawk Linux下でシリアル・コンソールを構成するために必要な手順を提供します。

USBキーボードが付いているシステム上で**kdb**カーネル・デバッガを使用したい場合はシリアル・コンソールが必要となることに注意して下さい。

1. 以下のカーネル・オプションを含めるためにブート・コマンド行を修正します：

   ```
   console=tty#,baud#
   ```

   tty#はコンソール用に使用するシリアル・ポート、baud#は使用するシリアル通信速度です。通常は殆どが以下のようになります：

   ```
   console=ttyS0,115200
   ```

2. 適当なデータ端末装置をシリアル・ポートに接続し、選択された通信速度で通信するよう構成されている事を確認してください。使用されているデバイスの仕様によっては、ヌル・モデムが必要となる可能性があります。

   安価なLinux PCはデータ端末装置としては優れた選択であることに気付いて下さい。シリアル通信セッションの作成に関する詳細な情報は**minicom(1)**のmanページを参照して下さい。

   Windows PCもまた使用することが可能ですが、この説明は本資料の範疇を超えています。

シリアル・コンソールのもう1つの用途は、ハングする可能性のあるシステムを調査するためにリアルタイム・シェルを構成することです。この手順は問題が発生している全てのアプリケーションがロードされる前に設定されたシリアル・コンソール上で完了している必要があります。

1. ハングする可能性のあるシステムのシリアル・コンソールを設定します。例えば：

   - GRUBブート・コマンド行にこの文字列を追加します：

     ```
     console=ttyS0,115200
     ```

   - **/etc/inittab**ファイルにこの行を追加します：

     ```
     S0:2345:respawn:/sbin/agetty 115200 ttyS0 vt100
     ```

   - **/etc/securetty**ファイルに“ttyS0” エントリを追加します。
   - シリアル・ケーブルを一番低い番号のシリアル・ポートと他の計算機またはラップトップのシリアルポートに接続します。

2. もう一方の計算機がLinuxである場合：
   - シェルを開きます。
   - # **minicom –s**.
   - 「Serial Port Setup」を選択し「Enter」。
   - デバイスを**/dev/ttyS0**へ変更します。
   - 通信速度を115200へ変更します。
   - 「Exit」を選択し「Enter」(「Exit from minicom」ではありません)。

- ログイン・プロンプトからルートとしてログインします。

もう一方の計算機がWindowsの場合：

- ターミナル・アプリケーションを起動します。
- COM1を使用して接続します。Connect using COM 1.
- 通信速度を115200に設定します。
- ログイン・プロンプトからルートとしてログインします。

3. ルート・ログインから、以下で示す**RTConsole.sh**スクリプトを実行します。引数としてどのタスクよりも高いリアルタイム優先度を与えます。例えば：

```
# ./RTConsole.sh 90
```

この手順はデバッグ用に「ハング」している間もアクティブなままであるログイン・シェルおよびシステムへのアクセスと視認性を提供します。幸先良い出足はどのプロセスがシステムを支配しているのかを割り出すために**top(1)**を実行することです。

デバッグが終了したらシステムを再起動する必要があります：

```
# reboot
```

RTConsole.sh

```
#!/bin/bash
if [ $UID -ne 0 ]
then
    echo "Must be root to execute."
    exit
fi

if [ $# -eq 0 ]
then
    echo "Usage: RTConsole <Login shell priority>"
    exit
fi

for i in $(ps -e -o pid,cmd | fgrep /0 | fgrep -v fgrep | awk '{print $1}');
do
    run -s fifo -P $1 -p $i
done
run -s fifo -P $1 -p $PPID
```

H
ブート・コマンド・ライン・パラメータ

表H-1にRedHawkで独自に動作するブート・コマンド・ライン・パラメータを掲載しています。これはLinux下で利用可能な全てのブート・コマンド・ライン・パラメータを含んでいません。この一覧表に関して、カーネル・ソース・ディレクトリ内の**Documentation/kernel-parameters.txt**を参照または**info grub**と入力して下さい。

ブート・パラメータはカーネルに組み込まれている機能を定義します。ブート・コマンドはカーネル起動時に自動的に包含するために**/boot/grub2/grub.cfg**へ追加すること、またはカーネルが起動するときにブート・コマンド・ラインに指定することが可能です。

個々の機能に関する情報は様々な場所で入手可能です。表H-1で以下の参考文献が提供されます。

- 本書*RedHawk Linux User's Guide* で提供される情報が含まれるページ番号
- 他の適切なConcurrentの資料の名称および文書番号

以下を含む取得可能な情報のある他のソース：

- カーネル・ソース・ツリーの**Documentation**ディレクトリ下のファイル
- Internet上のLinuxの資料サイト

表H-1 ブート・コマンド・ライン・パラメータ

パラメータ	オプション	概要	Concurrent 関連資料
crashkernel	=*size*@64M	クラッシュ・ダンプの保存および解析するために破損したカーネルのコア・イメージを含む「クラッシュ」カーネルをロードするためのメモリと非既定の場所を予約。 *size* は予約するメモリのサイズ： 32M, 64M, 128M (既定値) 64Mはオフセット・アドレス	12-1ページ
kgdboc	=*serial_device[,baud]*	kdbのttyインターフェースを有効化。 例： kgdboc=ttyS0,115200	12-1ページ
kgdbwait		カーネルの実行を停止して最も早いタイミングでカーネル・デバッグに進入。	
ekgdboc	=kbd	初期のカーネル・コンソール・デバッグを許可。 earlyprintk=vgaと併せて使用。	
memmap	=*size*<*delimiter*>*address*	予約するメモリ領域を定義。 <*delimiter*>は、システムRAMが'@'、 予約が'$'、ACPIが "#"	2-23ページ
	=exactmap	正確なBIOSマップの使用を指定。	
mm	=*size*<*delimiter*>*address*	memmapの別名(x86_64のみ)。 予約するメモリ領域を定義。	2-23ページ
	=ex	exactmapの別名(x86_64のみ)。 正確なBIOSマップの使用を指定。	
nmi_dump		デバッガへエントリするためにシステムNMIボタンの押下を有効化。 nmi_watchdog=0でない限り、デバッガを終了するとクラッシュ・カーネルをロードしダンプを取得。	12-11ページ

表H-1 ブート・コマンド・ライン・パラメータ (続き)

パラメータ	オプション	概要	Concurrent 関連資料
nmi_watchdog	=0	NMIウォッチドッグ機能を停止。 RedHawkカーネルの既定値。	12-11ページ **/*kernel-source*/ Documentation/ nmi_watchdog.txt**
	=1	各CPUはNMI固有のタイミングで実行。 カーネルのAPICサポートが必要。 現在はこの設定は動作せず「2」へ変更。	
	=2	ブロードキャストを介して全CPUへ送信する割り込みを生成する外部NMIタイマーを使用。 i386デバッグ・カーネルの既定値。	
	=-1	カーネルは1または2を選択(x86_64のみ)。 x86_64デバッグ・カーネルの既定値。	
nohz	=on	動的ティックを有効化(規定値)。	B-1ページ
	=off	動的ティックを無効化。	
no_ktext_repli	n/a	全てのカーネル・テキスト・ページ複製のサポートを無効化。	10-14ページ
no_kmod_repli	n/a	カーネル・モジュール・テキスト・ページ複製のサポートを無効にするが、常駐カーネル・テキスト・ページ複製は有効のまま。	10-14ページ
no_pregraph_pgs	n/a	プロセッサ間割り込みを抑えるために使用されるプリアロケート・グラフィクス・ページ・サポートの全てを無効化。	10-5ページ
numa	=off	カーネル内でカーネル・チューニング・パラメータNUMAが有効となっているx86_64システムのNUMAサポートを無効化。 全てのCPUが属する単一NUMAノードのシステムを作成。 これはカーネルにNUMAサポートが組み込まれていない(ノードなしのフラット・メモリ・システムでNUMAユーザー・インターフェースは呼ばれた時にエラーを返す)のとは異なる。	10-1ページ
pregraph_pgs	=*numpages*	PREALLOC_GRAPHICS_PAGESに静的にコンパイルされた値を無効にしてプリアロケートされたグラフィック・ページのバッファ・プール用にブート時に割り当てるページ数を定義。 no_pregraph_pgsオプションは本パラメータよりも優先する事に注意。	10-5ページ

表H-1 ブート・コマンド・ライン・パラメータ (続き)

パラメータ	オプション	概要	Concurrent 関連資料
rcim	=*rcimoptions*	RCIM用構成オプション(割り込みの特性、関連性、タイミング・ソース、RCIMマスターのホスト名等)を定義。	RCIM User's Guide (0898007)
rhash_entries	=*n*	エントリ数を固定するため、IPルート・キャッシュ・テーブルをサイズ調整(ksoftirqdによる周期的なフラッシュ)。 規定値では、サイズは動的で利用可能なメモリの量に基づく。 本エントリは過度のksoftirqdの実行を減らすために小さなサイズを定義して使用。	2-34ページ
tsc_sync	=auto	BIOSが正しくTSCに同期したかどうかを確認。もしされていない場合はTSCと再同期。 既定値。	7-1ページ
	=check	BIOSが正しくTSCに同期したかどうかを確認。BIOSが失敗した場合、利用可能なクロックソースとしてTSCは無効化。	
	=force	無条件で起動の最後で全てのTSCを再同期。	
vmalloc	=*nn*[KMG]	*nn* のサイズを正確に所有するためにvmalloc領域を強制。これは最小サイズを増やすために使用することが可能(32bitカーネルでは128MB)。また、サイズを小さくするため、および直接マッピングされたカーネルRAMに対しより多くの空間を残すために使用することも可能。 サイズ単位の指定にK, M, G(kilo-byte, mega-byte or giga-byte)を使用することが可能。	14-11ページ

用語解説

本用語解説はRedHawk Linuxで使用される用語を定義します。*斜体* の用語もまたここで定義されています。

アフィニティ

実行が許可されたプロセスまたは割り込みとCPUとの間の関連性。これはアフィニティ・マスクに含まれていないCPU上での実行が禁止されます。もし1つ以上のCPUがマフィニティ・マスクに含まれている場合、カーネルは負荷やそのほか考慮すべき事項に基づいてプロセスや割り込みを自由に移動しますが、アフィニティ・マスク内の他のCPUだけとなります。既定の状態はシステムの全てのCPU上で実行するアフィニティとなっていますが、指定は**mpadvise(3)**, **shield(1)**, **sched_setaffinity(2), /proc**ファイル・システムを通して行うことが可能です。*シールドCPU* と一緒にアフィニティを使用することでアプリケーション・コードのより優れたデターミニズムを提供することが可能となります。

AGP

PC上のメイン・メモリへのアクセスが普通のPCIバスよりも高速になる低コスト3Dグラフィクス・カードを提供するIntelのバス仕様。

非同期セーフ

ライブラリ・ルーチンをシグナル・ハンドラ内から安全に呼び出すことが可能な場合。いくつかの非同期セーフなコードを実行しているスレッドがシグナルに割り込まれた場合に*デッドロック* (*deadlock*)することはありません。これはロックを取得する前にシグナルをブロックすることで達成されます。

アトミック

一連の操作全てが同時に実行され、それらが同時に全て実行することが可能な場合のみ。

認証

セキュリティ目的のためにユーザー名、パスワード、プロセス、コンピュータ・システムの身元の照合。*PAM* はRedHawk Linux上での認証方法を提供します。

ブロッキング・メッセージ操作

メッセージの送信または受信の試みが失敗した場合に実行を停止。

ブロッキング・セマフォ操作

セマフォ値のテスト中に実行を停止。

ブレークポイント

実行が停止されプロセッサの制御がデバッガへ切り替わるプログラム内の位置。

ビジー・ウェイト

ハードウェアでサポートされるテスト&セット操作を使用しているロックを取得する*相互排他*の方法。プロセスが現在ロックされた状態のビジー・ウェイト・ロックを取得しようとする場合、ロックしているプロセスは、現在ロックを保持するプロセスがクリアされテスト&セット操作が成功するまでテスト&セット操作をリトライし続けます。別名*スピン・ロック*。

ケーパビリティ

スーパーユーザーに関連する伝統的な*特権* を単独で有効および無効にすることが可能な別個の単位に分割。現在の全ての有効なLinuxケーパビリティ一式は**/usr/include/linux/capability.h**で入手する事が可能で付録Cに詳述されています。*PAM* を通して、ルートだけが通常認められる特権を必要とするアプリケーションを非ルート・ユーザーが実行する設定にすることが可能です。

条件同期

アプリケーションが定義する条件を満足するまでプロセスの進行を遅らせるためにスリープ/ウェイクアップ/タイマーのメカニズムを利用。RedHawk Linuxでは、**postwait(2)**および**server_block(2)**/**server_wake(2)**システムコールはこの目的のために提供されます。

コンテキスト・スイッチ

マルチタスク・オペレーティング・システムが実行中のあるプロセスを止めて他を実行した時。

クリティカル・セクション

ソフトウェアの正しい動作を保証するために順序正しくかつ中断なしで実行されなければならない一連の命令。

デッドロック

2つ以上のプロセスが両方ともあるリソースを他方が解放するのを待っているためにそれらのプロセスが進行することが出来ない複数の状況すべて。

割り込みハンドリング遅延

割り込みルーチンが割り込みレベルでされたであろう処理を遅延する方法。RedHawk Linuxはカーネル・デーモンのコンテキスト内で実行される*ソフトIRQ*、*タスクレット*、*ワーク・キュー*をサポートします。高優先度*リアルタイム*・タスクが遅延された割り込み機能の動作を*プリエンプト*することが可能となるようにこれらのデーモンの優先度とスケジューリング・ポリシーを構成することが可能です。

デターミニズム

一定の時間内に特定のコード・パス(順に実行される一連の命令)を実行するためのコンピュータ・システムの能力。あるインスタンスから他へコード・パスが変化する実行時間の範囲はシステムのデターミニズムの度合いを表します。デターミニズムはユーザー・アプリケーションのタイム・クリティカルな部分を実行するために必要な時間とカーネルでシステム・コードを実行するために必要な時間の両方に適用されます。

デターミニスティック・システム

*デターミニズム*に影響を与える要因を制御することが可能なシステム。デターミニズムを最大限発揮するためのRedHawk Linux下で利用可能なテクニックは、*シールドCPU*、*固定優先度スケジューリング・ポリシー*、*割り込みハンドリング遅延*、*負荷バランシング*、*ハイパースレッディング*制御ユニットを含みます。

デバイス・ドライバ

オペレーティング・システムが利用を許可するコンピュータ・ハードウェアの部品や周辺機器と直接通信するソフトウェア。デバイス・モジュールまたはドライバとも呼ばれます。

ダイレクトI/O

カーネルのデータ・バッファリングを回避するバッファのないI/O形式。ダイレクトI/Oでは、ファイル・システムはディスクとユーザー提供バッファ間で直接データを転送します。

裁量アクセス制御

ユーザーの裁量で与えられた証明書の有効性を確認するユーザー名、パスワード、ファイル・アクセス・パーミッションに基づくメカニズム。これは例えばIPアドレスといったユーザーが管理できないアイテムに準ずる強制的な制御とは異なります。

実行時間

タスクを完了するために要する時間。RedHawk Linuxで高分解能プロセス・アカウンティング機能を利用すると、各プロセスの実行時間測定は、高分解能タイム・スタンプ・カウンター(TSC)で測定されたシステム時間、ユーザー時間、システムが割り込まれた時間、ユーザーが割り込まれた時間に分類されます

FBS

Frequency-Based Scheduler (FBS) を参照して下さい。

固定優先度スケジューリング・ポリシー

プロセス単位を基本に静的な優先度をユーザーが設定可能なスケジューリング・ポリシー。スケジューラーは固定優先度スケジューリング・ポリシーの1つを使用するプロセスの優先度を決して変更しません。例え他のプロセスが実行可能であっても、最高固定優先度プロセスが実行可能であれば直ぐにCPUを取得します。SCHED_FIFOとSCHED_RRの2つの固定優先度スケジューリング・ポリシーが存在します。

フレイバ

個々の実体の差。RedHawk Linuxは3つのプレビルト・カーネルのフレイバがあり、各々異なる特徴と構成を含んでいます。カスタマイズされたカーネルは違ったフレイバを構成することになります。フレイバの指定はMakefileの最上位に定義され、カーネルが構築されるときにカーネル名称に接尾語として追加されます。(例： <kernelname>**- trace**)

Frequency-Based Scheduler (FBS)

Real-Time Clock and Interrupt Module (*RCIM*)、外部割込みソース、サイクルの完了により提供される高分解能クロックを含む様々なタイミング・ソースに基づき指定の周期でプロセスを開始するために使用されるタスク同期メカニズム。プロセスは優先度ベースのスケジューラを使用してスケジュールされます。*パフォーマンス・モニタ(PM)*と組み合わせて使用する場合、特定アプリケーション用に様々なタスクへプロセッサを割り当てる最良の方法を決定するためにFBSを使用することが可能です。

*NightSim*ツールはFrequency-Based Schedulerおよびパフォーマンス・モニタ用のグラフィカル・インターフェースです。

GRUB

GRand Unified Bootloader。複数のオペレーティング・システム(およびそれらの改良)をロードおよび管理する小さなソフトウェア・ユーティリティ。GRUBはRedHawk Linuxで使われるデフォルトのブートローダーです。

ハイパースレッディング

1つの物理プロセッサでソフトウェア・アプリケーションの複数のスレッドを同時に実行することを可能にするIntel Pentium Xeonプロセッサの機能。1つのプロセッサ実行リソース一式を共有しつつ各プロセッサは2つのアーキテクチャを所有しています。各々のアーキテクチャは1つの論理CPUが結果としてシステムの論理CPUが2倍になったとみなすことが可能です。ハイパースレッディングが有効である単一プロセッサ・システムは2つの論理CPUを持ち、割り込みやバックグラウンド・プロセスからいずれかのCPUをシールドすることが可能になります。ハイパースレッディングは全てのRedHawk Linux i386プレビルト・カーネルではデフォルトで有効になっています。

infoページ

infoページはコマンドまたはファイルに関する詳細な情報を提供します。*man*ページは簡潔かつinfoページよりも情報を少なく提供する傾向があります。infoページは操作可能なメニュー・システムにより対話型となっています。infoページは**info(1)**コマンドを使いアクセスします。

プロセス間通信

あるプロセスが他のプロセスと通信することを可能にする機能。プロセスは同一計算機上またはネットワークを介して接続されている異なる計算機上で実行することが可能です。IPCはあるアプリケーションが他のアプリケーションを制御すること、相互に干渉することなくいくつかのアプリケーションに対し同じデータを共有することを可能にします。IPC方式にはパイプ、*メッセージ・キュー*、*セマフォ*、*共有メモリ*、ソケットが含まれます。

プロセス間同期

協同プロセスが同じ一連のリソースへのアクセスを調整することを可能にするメカニズム。RedHawk Linuxは*再スケジューリング変数*、*ビジーウェイト*、*スリーピーウェイト相互排他*メカニズム、*条件同期*ツールを含む様々なプロセス間同期ツールを提供します。

ジッター

周期的な動作の到達または発進した時間の変化の大きさ。コード・セグメントの実行または割り込みの応答のどちらかで計測された時間のワースト・ケースが標準的な状況よりもはっきりと異なる場合、アプリケーションの性能はジッターに直面していると言います。ジッターは標準的に正しい周期内に動作が全て留まる限り問題の原因にはなりませんが、*リアルタイム・タスク*はジッターを出来る限り最小限に抑えることを概ね必要とします。

ジャーナリング・ファイル・システム

ファイルシステム内の最終的な場所へ書き込む前にディスク処理がジャーナルまたはログと呼ばれるディスクの領域へ順次書き込まれるファイルシステム。もしジャーナル・エントリが働く前にクラッシュが発生した場合、元のデータはディスク上にまだ存在し新しい変更のみが失われます。システムの再起動時、ジャーナル・エントリが再生され中断された更新が大幅に簡素化された修復時間で完了します。RedHawk Linuxのジャーナリング・ファイル・システムはext3, xfs, reiserfsを含みます。

カーネル

より多くの高度な機能が頼る基本的な機能を実行するオペレーティング・システムの重要な部分。LinuxはLinus Torvaldsおよび開発者中心のグループにより開発されたカーネルを基にしています。Concurrentは*デターミニスティックなリアルタイム*処理のために拡張機能を提供するCentOSより配布されるLinuxカーネルを修正しました。RedHawk Linuxはgeneric, debug, traceのフレイバによる3つのプレビルト・カーネルを提供します。これらは**/boot**ディレクトリの中に **vmlinuz-**<kernelversion>**-RedHawk-**<revision.level>**-**<flavor> というファイル名で存在します。

カーネル構成GUI

カーネルを構成するために選択させるグラフィカル・インターフェース。RedhawkLinuxでは、**ccur-config**スクリプトを実行することで選択することが可能なGUIを表示します。

負荷バランシング

CPU全体の負荷のバランスをとるためにいくつかのCPUからプロセスを移動します。

manページ

コマンドまたはファイルを説明する要約および簡潔なオンライン・ドキュメント。manページはシェル・プロンプトで**man**、続いてスペース、その後読みたい項目を入力することにより表示されます。RedHawk LinuxのmanページはCentOS Linuxディストリビューションより提供されるmanページおよびConcurrentで開発された機能を解説するmanページを含んでいます。

メモリ・オブジェクト

対応するメモリを共有することを可能にするために1つ以上のプロセスのアドレス空間へマッピング可能な名前つきストレージ領域。メモリ・オブジェクトは全てのファイル・システム・オブジェクト(例えば、ターミナルやネットワーク・デバイス)ではなく、*POSIX共有メモリ・オブジェクト*、レギュラー・ファイル、いくつかのデバイスを含んでいます。プロセスは、*カーネル*とアプリケーション間でのデータ・コピーを除き、オブジェクト上のアドレス空間の一部をマッピングすることによって直接メモリ・オブジェクト内のデータにアクセスすることが可能です。

メッセージ・キュー

1つ以上の読み取りプロセスによって読まれるメッセージを1つ以上のプロセスが書く事が可能な*プロセス間通信(IPC)*のメカニズム。RedHawk Linuxは*POSIX*と*System V*のメッセージ・キュー機能をサポートしています。

モジュール

システム・レベルの機能を実行するルーチンの集まり。モジュールは必要に応じて実行中の*カーネル*からロードおよびアンロードされる可能性があります。

ミューテックス

変更の同時発生から共有データ構造体を保護するためおよび*クリティカル・セクション*を実装するために便利な*相互排他*デバイス。ミューテックスはアンロック(どのスレッドにも所有されていない)とロック(1つのスレッドに所有されている)の2つの起こりうる状態があります。他のスレッドに既にロックされているミューテックスをロックしようとするスレッドは、所有しているスレッドが最初のミューテックスをアンロックするまで停止します。

相互排他

一連の共同プロセスの1つだけが共有リソースへのアクセスをシリアライズすることにより同時にクリティカル・セクションで実行することが可能となることを保証するメカニズム。3つのメカニズムのタイプ(*ビジーウェイト*を必要とするもの、*スリーピーウェイト*を必要とするもの、この2つの組み合わせ)は一般的に相互排他を提供するために使用されます。

NightProbe

1つ以上の実行プログラム内のプログラムデータの*リアルタイム*のレコーディング、表示、変更を可能にするConcurrentが開発したグラフィカル・ユーザー・インターフェース(GUI)。これはシミュレーション、データ収集、システム制御を含むアプリケーションの開発中や動作中に使用することが可能です。

NightSim

*Frequency-Based Scheduler (FBS)*および*パフォーマンス・モニタ(PM)*のためのグラフィカル・ユーザー・インターフェース(GUI)。

NightStar RTツール

リアルタイム・アプリケーションの実行時の動作をスケジューリング、モニタリング、解析するためにグラフィカル・インターフェースを備えるConcurrentが提供する開発ツールの集まり。このツール群は*NightSim*周期スケジューラ、*NightProbe*データ・モニタ、*NightTrace*イベント・アナライザ、*NightTune*チューナー、*NightView*デバッガを含んでいます。

NightTrace

マルチプロセスおよびマルチプロセッサのユーザー・アプリケーションやオペレーティング・システムの動作の動的な挙動を解析するためにConcurrentが開発したグラフィカル・ツール。NightTrace RTツール群は対話型デバッグ、性能分析ツール、トレース・データ収集デーモン、アプリケーション・プログラミング・インターフェース(API)から構成されます。

NightTune

CPU使用状況、コンテキスト・スイッチ、割り込み、仮想メモリ使用状況、ネットワーク活動、プロセス属性、CPUシールディングを含むシステムとアプリケーション性能の解析のためにConcurrentが開発したグラフィカル・ツール。NightTuneはポップアップ・ダイアログまたはドラッグ＆ドロップを使用して優先度、スケジューリング・ポリシー、プロセス単体またはグループのCPUアフィニティを変更することが可能です。CPUのシールディングやハイパースレッド属性および個々の割り込みのCPU割り当てもまた設定することが可能です。

NightView

C, C++, Fortranで書かれた*リアルタイム*・アプリケーションのためにConcurrentが設計した多目的グラフィカル・ソース・レベル・デバッグおよびモニタリング・ツール。NightView RTは最小限の干渉でローカル・システムまたは異なるターゲットのマルチプロセッサ上で動作しているリアルタイム・プロセスの監視、デバッグ、パッチを当てることが可能です。

非ブロック・メッセージ操作

メッセージの送信または受信の試みに失敗した場合に実行を停止しません。

非ブロック・セマフォ操作

セマフォ値のテスト中は実行を停止しません。

NUMA

Non-Uniform Memory Architecture。異なるメモリのクラスへのアクセス時間が著しく異なる一部のマルチプロセッサで使用されているメモリ・アーキテクチャ。プロセッサは非ローカル・メモリ(ローカルから他のプロセッサまたはプロセッサ間で共有されたメモリ)よりもより高速にそれぞれのローカル・メモリへアクセスすることが可能です。

PAM

Pluggable Authentication Module。この機能のために個々のプログラムを別々に再コンパイルすることなくシステム管理者がアクセスおよび認証ポリシーを設定することが可能な方法。この仕組みの下では、ルートだけが通常許可されている*特権*が必要となるアプリケーションの実行を非ルート・ユーザーに設定することが可能となります。

PCI

Peripheral Component Interface。ビデオカード、サウンド・カード、ネットワーク・インターフェース・カード、モデムのようなプロセッサと周辺機器デバイス間の高速データ・パスを提供する周辺機器バス。PCIは「プラグ＆プレイ」機能、33MHzと66MHzでの動作、32bitと64bitのデータ・パスを提供します。

パフォーマンス・モニタ (PM)

*Frequency-Based Scheduler*上でスケジュールされたプロセスのCPU使用状況の監視を可能にする機能。取得した値は*負荷バランス*や処理の効率を向上するためにプロセッサ間でどのようにプロセスを再配置するのかを判断するために役立ちます。*NightSim*はパフォーマンス・モニタのためのグラフィカル・インターフェースです。

Pluggable Authentication Module (PAM)

*PAM*を参照して下さい。

POSIX

ユーザー空間用の規格と共にUNIXに類似したカーネル・インターフェースのためのセマンティクスとインターフェースの仕様を定める規格。全てのPOSIXに合致しているオペレーティング・システムにサポートされている必要のあるコアPOSIX定義、および特定の機能(例：POSIXメッセージ・キュー)のためのいくつかのオプション規格が存在します。

プリエンプション

CPU上で実行されたプロセスがより高い優先度のプロセスに置き換えられる場合。RedHawkに含まれるカーネル・プリエンプションは例えカーネル空間で動作していても低優先度プロセスがプリエンプトされることが可能であり、結果としてシステム応答が向上されます。プロセス・プリエンプションは再スケジューリング変数の利用を通して制御されます。

優先度継承

*優先度反転*を回避するために必要に応じてあるプロセスの優先度を他へ直ぐに伝えるメカニズム。

優先度反転

高優先度プロセスが低優先度プロセスの実行のために強制的に待たされる場合。

特権

ユーザーまたはプロセスがデリケートな操作やシステムの制限事項を無視することが可能となるメカニズム。スーパーユーザーは全ての(ルートの)特権を所有しています。*ケーパビリティ*を通して、個々のユーザーおよびプロセスに対して特権を有効または無効にすることが可能です。

プロセス

実行されているプログラムの実体。各プロセスはユニークなPID(カーネルのプロセス・テーブル内にあるそのプロセスのエントリ)を持っています。

プロセス・ディスパッチ・レイテンシー

割り込みにより知らされる外部イベントの発生から外部イベントを待っているプロセスがユーザー・モードで最初の命令を実行するまでに経過した時間。

PRT

Concurrentによって開発されたリアルタイム・カーネル・オプションの実装はrt.wiki.kernel.orgで維持されているコミュニティが開発したPREEMPT_RTカーネル・パッチを基にしています。

RCIM

Real-Time Clock and Interrupt Module。複数のアプリケーションで完全にデターミニスティックなイベント同期をするためにConcurrentが設計した多機能PCIカード。RCIMは同期クロック、複数のプログラム可能なリアルタイム・クロック、複数の入出力外部割込みラインを含んでいます。割り込みはRCIMのチェーン接続を利用して相互接続したシステム間で共有(分配)することが可能です。

リアルタイム

実在のイベントに応答し、所定の期限内にイベントを処理することを必要とされる処理手続きを完了すること。実在のイベントへの応答が必要となる計算は期限前に完了する必要があり、さもなければ結果が間違っているとみなされます。指定された時間の制約の範囲内で特定の機能を保証することが可能であるため、RedHawk Linuxは真のリアルタイム・オペレーティング・システムとなります。

再スケジューリング変数

再スケジュール用に単一プロセスの脆弱性を制御する(原則アプリケーションがプロセスごとに割り当てる)データ構造体。

ロウバスト・ミューテックス

アプリケーションのスレッドの1つが*ミューテックス*を保持している間に死んだ場合、回復する機会をアプリケーションへ与えるミューテックス。

RPM

RPMパッケージ・マネージャ。コンピュータ・ソフトウェア・パッケージのインストール、アンインストール、検証、問合せ、更新に使用されるツール、データベース、ライブラリの管理ツール。全ての情報については**rpm(8)**のmanページを参照して下さい。

セマフォ

1つ以上のプロセスがテスト&セットすることが可能であるメモリ位置の値。既にロックされているセマフォをロックしようとするプロセスがロックされるまたはスリープ状態となるため、セマフォは*スリーピーウェイト相互排他*形式となります。RedHawk Linuxは最速性能を得るための単純なインターフェースを提供するPOSIXカウンティング・セマフォ、多くの追加機能(例えば、セマフォ上にいくつの待機者がいるのかを調べる機能、セマフォ一式を操作する機能)を提供する*System V*セマフォを提供します。

共有メモリ

1つ以上のプロセスの仮想アドレス・マップを通してアクセス可能なメモリ。共有メモリを使い、プロセスが通常のオペレーティング・システムのサービスを使い読み書きするよりもより速くデータをやりとりすることが可能です。RedHawk Linuxは*System V*および*POSIX*から派生する規格化された共有メモリ・インターフェースを含んでいます。

シールドCPU

割り込みやシステム・デーモンに関連する予測できない処理から保護されている高優先度タスクの実行を担うCPU。RedHawk Linuxシステムの各CPUはバックグラウンド・プロセス、割り込み、ローカル・タイマー割り込みから個々にシールドすることが可能です。

シールドCPUモデル

特定の重要なリアルタイム機能に品質の高いサービスを保証する方法でタスクおよび割り込みがCPUへ割り当てられるモデル。殆どの割り込みと低優先度タスクは他のCPUへバインドする一方、特に高優先度タスクは1つ以上のシールドCPUへバインドします。高優先度タスクの実行を担うCPUは、割り込みやシステムコールを介してカーネルに入っている他の低優先度プロセスの動作に関連する予測不可能な処理からシールドします。

シールド・プロセッサ

*シールドCPU*を参照して下さい。

スリーピーウェイト

現在ロックされた状態のロックを取得しようとする場合にプロセスをスリープ状態にする*セマフォ*のような*相互排他*の方法。

SMP

対称型マルチプロセッシング。多くの場合は同じメモリを共有し入出力デバイスへのアクセスに対処可能である1つのオペレーティング・システムに管理された2つ以上のプロセッサを使用する演算の方法。アプリケーション・プログラムはシステムのいずれかまたは全てのプロセッサ上で実行することが可能です。

ソフトIRQ

機能の実行が次の利用可能な「セーフ・ポイント」まで遅らせることが可能となる方法。機能を呼び出す代わりに次のセーフ・ポイントで呼び出される原因となる「トリガ」が使用されます。セーフ・ポイントはカーネルがハードウェアまたはソフトウェア割り込みを扱っておらず割り込みのブロックが実行されていない全ての時間です。

スピン・ロック

リソースのための*相互排他*を保証する*ビジー・ウェイト*の方法。スピン・ロックを待ち続けているタスクはスピン・ロックが利用可能になるまでビジー・ループの状態のままとなります。

System V

LinuxやSystem Vシステムを含む多くのUNIKライクなシステムにサポートされる*プロセス間通信(IPC)*オブジェクトのための規格。System V IPCオブジェクトは、System V *メッセージ・キュー*、*セマフォ・セット*、*共有メモリ*領域の3種類があります。

タスクレット

ユーザー空間に復帰またはハードウェア割り込みの後にソフトウェア割り込みを受信した時に実行しているソフトウェア割り込みルーチン。タスクレットは同時に複数のCPU上で実行されませんが、動的に割り当てることが可能です。

TLB

Translation Look-aside Buffer。各仮想アドレス・ページ番号に関連する物理アドレス・ページ番号を記録する仮想メモリ・システムで使用されるテーブル。仮想アドレスに基づくキャッシュのタグと一体となってTLBは使用されます。キャッシュ・アクセスおよび仮想アドレスから物理アドレスへの変換が平行して進むことが可能となるように仮想アドレスはTLBとキャッシュへ同時に渡されます。

トレース・イベント

デバッグおよび性能解析用の*NightTrace*ツールで調査が可能なアプリケーションのソース・コード内またはカーネル内の重要なポイント(トレース・ポイント)について記録された情報。

ワーク・キュー

*ソフトIRQ*や*タスクレット*以外の遅延実行の方法ですが、それらの様式とは異なり、Linuxはカーネル・デーモンのプロセス・コンテキスト内でワーク・キューを処理するためにスリープが可能です。

Index

パス

/bootディレクトリ　11-1
/dev/mqueue　3-2
/etc/pam.d　13-2
/etc/rc.sysinit　2-17
/etc/security/capability.conf　13-2, 13-3
/etc/sysconfig/sbsvme　15-6
/etc/sysconfig/sbsvme-mappings　15-7
/procファイル・システム　1-6
/proc/bus/pci　3-28, 14-1
/proc/ccur　B-3
/proc/driver/btp　15-7, 15-15, 15-16
/proc/driver/graphics-memory　F-4
/proc/interrupts　2-18
/proc/irq/*n*/smp_affinity　2-10, 2-19
/proc/mtrr　F-2
/proc/*pid*/affinity　B-3
/proc/*pid*/mem　9-1–9-4
/proc/*pid/resmem*　B-3
/proc/shield/irqs　2-14, 2-18
/proc/shield/ltmrs　2-14, 7-4
/proc/shield/procs　2-14
/proc/sysvipc/shm　3-15, 3-28
/proc/vmcore　12-2
/usr/lib/libccur_rt　9-3, 14-3
/usr/lib64/libnuma.so　10-11

数値

32bit　1-1, 11-2
64bit
　コード移行　D-1
　カーネル　1-1, 11-2, D-1

A

アフィニティ　2-10, 2-14–2-19, 4-6, 4-14
代替glibc　5-27
AMD Opteronプロセッサ　D-1
非同期I/O　1-10
AUDIT B-3
認証　13-1

B

bar_device_count　14-4
bar_mmap　14-5
bar_munmap　14-5
bar_scan_close　14-4
bar_scan_next　14-3
bar_scan_open　14-3
bar_scan_rewind　14-4
ベース・アドレス・レジスタ(BAR)　3-26, 14-1, B-3
bashコマンド　7-4
ビッグ・カーネル・ロック(BKL)　B-5
I/O空間への共有メモリのバインド　3-22, 3-23, 3-25
プロセスのブロック　5-37–5-41
ブート・コマンド・ライン・パラメータ　H-1
ボトム・ハーフ　14-12
btpモジュール　15-6
カーネルの構築　11-4
ビジー・ウェイト相互排他　5-2, 5-7–5-12

C

キャッシュ・スラッシング　2-22
ケーパビリティ　13-3, B-2, C-1
ccur-config　11-2
ccur-g++　5-27
ccur-gcc　5-27
CD/DVD焼付け　2-34
CentOSディストリビューション　1-1
clock_getres　6-5
clock_gettime　6-5
clock_nanosleep　6-11, 6-12
clock_settime　6-4
クロック
　POSIX　1-11, 6-1, 6-2, 6-4–6-5
　RCIM　1-5, 6-1, 7-1
　システムtime-of-day　6-4, 7-1
　TSC　7-1
クロックソース　7-1
条件同期　5-1, 5-37
カーネルの構成　11-2, B-1
コンソール、シリアルの設定　G-1
カウンティング・セマフォ　1-10, 5-2, 5-12–5-21
CPU

アカウンティング　1-7, 2-11, 7-2, B-2
アフィニティ　2-10, 2-14–2-19, 4-6, 4-13
アイドリング　2-29–2-31, B-2
負荷バランシング　7-3
論理/物理　2-29
再スケジューリング　7-4
シールディング(シールドCPUを参照)
cpuコマンド　2-18, 2-29–2-31
CPU_IDLING　B-2
crashダンプ　B-4
crashユーティリティ　12-8
CRASH_DUMP　B-4
クラッシュカーネル　H-2
プロセッサ間割り込み　B-3, F-1

D

デーモン制御　14-13, 14-14, E-1
DAEMON_CPU_LOCK　B-2
データ管理API(DMAPI)　8-2
データの共有　1-10
デバッグ・カーネル　1-3, 11-2
DEBUG_INFO　B-4
デバッガ　1-6, 1-8, 12-10, B-5
遅延割り込み機能　14-12
DETECT_SOFTLOCKUP　2-35, B-3
デターミニズム　2-2, 2-20, 2-34
デバイス・ドライバ　2-9, 11-5, 14-1
ダイレクトI/O　8-1
ディスクI/O　8-1
DMAPI　8-2
ドキュメント　v
ダンプ　B-4
DVD/CD焼付け　2-34

E

EM64Tプロセッサ　D-1
例
カーネルへのモジュール追加　11-6
認証　13-3, 13-4
ビジー・ウェイト相互排他　5-9
条件同期　5-42
initのCPUアフィニティ　2-17
CPUのシールディング　2-13, 2-17, 2-31–2-34
crashダンプ　12-8, 12-9
デバイス・ドライバ　14-6, 14-9
カーネル構成と構築　11-5
メッセージング　3-7, 3-9, 3-10, A-1
NUMA 10-18
PCI BARスキャン　14-3
PCI-to-VME　15-17
POSIXメッセージ・キュー　A-1
再スケジューリング制御　5-7
物理メモリの予約　2-23, 2-25
runコマンド　4-16
セマフォ　5-34, 5-36
プロセス優先度の設定　4-4
共有メモリ　3-19, 3-21, 3-23
シールドCPU　2-13, 2-17, 2-31–2-34
System Vメッセージ・キュー　A-4

F

FBSCHED　B-3
FBSCHED_PM　B-3
FIFOスケジューリング　4-1, 4-3
ファイル・システム　8-1
浮動小数点演算　2-33
free_pci_device　14-5
Frequency-Based Scheduler (FBS)　1-1, 1-5, B-3
fstat　3-12
ftok　3-27
ftruncate　3-12–3-14

G

get_mempolicy　10-10
glibc　5-27
用語解説　Glossary-1
グラフィクス
割り込み　F-3
サポート　10-5, B-5

H

haldaemon　2-34
高分解能プロセス・アカウンティング　1-7, 2-11, 7-2, B-2
HRACCT 7-2, B-2
ハイパースレッディング　1-8, 2-28–2-34, B-5
HyperTransport　2-27

I

I/O
非同期　1-10
ダイレクト　8-3
ディスク　8-1
同期　1-10
クアッドOpteronのスループット　2-27
ユーザー空間(UIO)　14-14, B-5

iHawkシステム　1-1, 11-2
INHERIT_CAPS_ACROSS_EXEC　13-5, B-2
init　2-15–2-17
プロセス間通信(System V IPCを参照)
プロセス間同期　5-1
割り込み
　/procインターフェース　2-18, 2-19
　プロセッサ間　B-3, F-1
　機能遅延　2-21, 14-12
　無効化　2-10–2-14, 7-2, 7-4
　無効の効果　2-4
　受信の効果　2-5–2-7
　グラフィクス　F-3
　ローカル・タイマー(ローカル・タイマーを参照)
　MTRR　G-1
　NMI　12-11, H-3
　NVIDIA CUDA　G-4
　RCIM　1-5
　応答時間の向上　1-7
　デバイス・ドライバのルーチン　14-11
　シールドCPU　2-10–2-14, 2-31
　ソフトIRQ　4-5, 14-12, B-3
　タスクレット　4-5, 14-12
　TLBフラッシュ　F-5
　ワーク・キュー　14-12, 14-13
インターバル・タイマー　7-3
ioremap　14-11
IPルート・キャッシュ・テーブル　2-35, H-4
IPC(System V IPCを参照)
IRQ　2-10, 2-12, 2-14, 2-19

J

ジャーナリング・ファイル・システム　1-9, 8-1

K

K8_NUMA　B-4
KDB　B-5
kdb　1-8, 10-13, 12-10
KDB_CONTINUE_CATASTROPHIC　B-5
KDB_MODULES　B-5
KDB_OFF　B-5
kdump　12-2
カーネル
　モジュール追加の例　11-6
　ブート　1-3
　構築　11-1
　構成　11-1, B-1
　クラッシュ・ダンプ　12-2, B-4
　デーモン制御　14-13, 14-14, E-1
　デバッグ　1-3, 11-2
　デバッガ　1-6, 1-8, 12-10, B-5
　デバッギング　12-1
　フレイバ　1-3, 11-1, 11-2
　ジェネリック/最適化　1-3, 11-2
　プリエンプション　1-6, B-4
　予約空間　14-11
　トレース　1-3, 11-2
　トレース・イベント　14-15
　トレーシング　1-6, B-5
　チューニング・パラメータ　11-1, 11-3, B-1
　アップデート　1-4
　仮想アドレス空間の予約　14-11, B-4
KEXEC　B-4
kexec　12-10
kgdb　12-10
kgdb/kdb　12-10
ksoftirqd　2-35, 14-13, B-3, H-4

L

LARGE_MMAP_SPACE　B-3
ライブラリ　3-3, 5-2, 5-14, 5-27, 10-11
負荷バランシング　7-3
ローカル・タイマー
　無効化　2-11–2-14, 7-4
　機能性　7-1
LOCK_BREAK_THROTTLE　B-4
LOCK_BREAK_THROTTLE_LIMIT　B-4
低レイテンシー　1-7
少ないメモリ　2-34

M

メールボックス　5-42
mbind　10-10
memmap　2-23, H-2
メモリ・アクセス(NUMA)　2-27, 10-1
メモリ・ロック　4-6, 5-2
メモリ・マッピング　1-10, 9-1, B-3
メモリ・ポリシー(NUMA)　10-2
メモリ常駐プロセス　1-9
メモリ・シールディング(NUMA)　10-3
少ないメモリ　2-34
物理メモリの予約　2-23
MEMSHIELD_ZONELIST_ORDER　10-20, B-4
メッセージ・キュー機構
　POSIX　3-2
　System V　3-4, 3-5
メッセージング　3-1, A-1, B-2
mlock　1-9, 2-20, 4-6
mlockall　1-9, 2-20, 4-6
mmap　1-8, 9-1, 9-4, 14-5, B-3
mpadvise　2-15

mq_close 3-2
mq_getattr 3-2
mq_notify 3-2
mq_open 3-2
mq_receive 3-2
mq_send 3-2
mq_setattr 3-2
mq_unlink 3-2
mqueue 3-2
msgctl 3-3, 3-6, 3-9
msgget 3-3, 3-5, 3-7
msgop 3-6
msgrcv 3-10
msgsnd 3-10
munlock 1-9, 2-20, 4-6
munlockall 1-9, 2-20, 4-6
ミューテックス 5-2, 5-27
　属性オブジェクト 5-23
　コンパイル 5-27
　nopreemptスピン 5-10
　優先度継承 5-23
　pthread 5-21, 5-23
　ロウバスト(堅牢性) 5-22
　スピン 5-7
　状態 5-23
相互排他 5-1, 5-14

N

nanosleep 2-11, 6-11, 7-4
NightProbe 1-2, 1-6
NightSim 1-2, 1-5
NightStar RTツール 1-1, 1-2, 11-2
NightTrace 1-2, 1-3, 1-6, 11-2, 14-15
NightTune 1-2
NightView 1-2, 1-6
NMI割り込み 12-11, H-3
nmi_dump 12-11, H-2
nmi_watchdog 12-11, H-3
NO_HZ B-2, H-3
NO_HZ_ENABLED B-2, H-3
no_pregraph_pgs 10-7, H-3
noatime 2-35
non-uniform memory access (NUMA) 2-27, 10-1
nopreempt_spin_init 5-11
nopreempt_spin_init_thread 5-11
nopreempt_spin_islock 5-11
nopreempt_spin_lock 5-11
nopreempt_spin_mutex 5-10
nopreempt_spin_trylock 5-11
nopreempt_spin_unlock 5-11
NTP_PPS B-2
NUMA 2-27, 10-1, B-4
NUMA 10-19, B-4
numa H-3
numapgsユーティリティ 10-12
NVIDIA B-5
NVIDIAグラフィクスのサポート B-5, F-3

O

ワンショット・タイマー 6-2
Opteron
　プロセッサ D-1
　クアッドI/Oスループット 2-27
最適化されたカーネル 1-3, 11-2

P

PAGE_REPLICATION 10-20, B-4
PAGE_REPLICATION_DYNAMIC 10-20, B-4
ページング 1-9
PAM 1-7, 13-1, B-2
pam_capability 13-2
PCIリソース・アクセス 14-1, B-3
PCI-to-VMEのサポート
　バッファのバインド 15-9
　構成 15-6, B-3
　資料 15-2
　例 15-17
　インストール 15-2, 15-5
　概要 15-1
　ユーザー・インターフェース 15-7
　VMEbusマッピング 15-7, 15-13
性能問題
　キャッシュ・スラッシング 2-22
　プロセッサ間割り込み F-1
　割り込みの遅延 2-21, 14-12
　デバイス・ドライバ 14-11
　ダイレクトI/O 8-4
　ローカル・タイマーの無効 7-2
　ハイパースレッディング 2-30
　クアッドOpteron上のI/Oスループット 2-27
　カーネル・デーモン E-1
　カーネル・トレース 14-15
　メモリ内ページのロック 2-20, 4-6
　ネガティブな影響 2-34
　NUMAプログラミング 2-27, 10-18
　最適化されたカーネル 1-3, 11-2
　優先度スケジューリング 2-21, 4-4, 4-6
　物理メモリの予約 2-23

シールドCPU　2-9–2-11, 4-6, E-1, F-1
ソフトIRQ　4-5, 14-13, E-1
タスクレット　4-5, 14-13, E-1
プロセスの起床　2-21, 5-37–5-41
ワーク・キュー　E-1
パフォーマンス・モニタ　1-1, 1-7, B-3
周期タイマー　6-2
物理メモリの予約　2-22
Pluggable Authentication Modules (PAM)　1-7, 13-1, B-2
POSIXの適合　1-2
POSIX機能
非同期I/O　1-10
クロック・ルーチン　6-4–6-5
クロック　1-11, 6-1, 6-2
カウンティング・セマフォ　1-10, 5-2, 5-12–5-21
メモリのロック　1-9, 2-20, 4-6
メモリ・マッピング　1-10
メッセージ・キュー　3-2, A-1, B-2
pthreadミューテックス　5-21
リアルタイム拡張　1-9
リアルタイム・シグナル　1-11
スケジューリング・ポリシー　4-1, 4-3
セマフォ　1-10, 5-2, 5-12–5-21
共有メモリ　1-10, 3-12–3-15
スリープ・ルーチン　6-11, 6-12
タイマー　1-11, 2-11, 6-2, 6-6–6-10, 7-4
POSIXルーチン
clock_getres　6-5
clock_gettime　6-5
clock_settime　6-4
mlock　1-9, 2-20, 4-6
mlockall　1-9, 2-20, 4-6
mq_close　3-2
mq_getattr　3-2
mq_notify　3-2
mq_open　3-2
mq_receive　3-2
mq_send　3-2
mq_setattr　3-2
mq_unlink　3-2
munlock　1-9, 2-20, 4-6
munlockall　1-9, 2-20, 4-6
pthread_mutex_consistent_np　5-24
pthread_mutex_destroy　5-21
pthread_mutex_getunlock_np　5-24
pthread_mutex_init　5-21
pthread_mutex_lock　5-21
pthread_mutex_setconsistency_np　5-24
pthread_mutex_setunlock_np　5-25
pthread_mutex_trylock　5-21
pthread_mutex_unlock　5-21
pthread_mutexattr_destroy　5-21
pthread_mutexattr_getfast_np　5-25
pthread_mutexattr_gettype　5-21
pthread_mutexattr_init　5-21
pthread_mutexattr_setfast_np　5-26
pthread_mutexattr_setprotocol　5-27
pthread_mutexattr_setrobust_np　5-27
pthread_mutexattr_settype　5-21
pthread_mutexattr_setunlock_np　5-27
sched_get_priority_max　4-11, 4-13
sched_get_priority_min　4-12
sched_getparam　4-11
sched_getscheduler　4-9
sched_rr_get_interval　4-13
sched_setparam　4-10
sched_setscheduler　4-8
sched_yield　4-11
sem_destroy　5-15
sem_getvalue　5-21
sem_init　5-12, 5-14
sem_open　5-16
sem_post　5-20
sem_timedwait　5-19
sem_trywait　5-20
sem_unlink　5-18
sem_wait　5-19
shm_open　3-12, 3-13
shm_unlink　3-12, 3-15
sigqueue　1-11
sigtimedwait　1-11
sigwaitinfo　1-11
timer_create　6-6
timer_delete　6-8
timer_getoverrun　6-10
timer_gettime　6-9
timer_settime　6-8
POSIX_MQUEUE　B-2
POST_WAIT　B-2
post/wait　5-37, B-2
PPSAPI　B-2
PPSAPI_SERIAL　B-2
PREALLOC_GRAPHICS_PAGES　B-3
プリアロケート・グラフィクス・ページ　10-5
PREEMPT　B-4
プリエンプション　1-5, 1-6, 2-8, 5-3, B-4
優先度
カーネル・デーモン　14-13, 14-14
プロセス　4-1, 4-2
優先度継承　1-7, 5-23
優先度反転　1-7
PROC_CCUR_DIR　B-3
PROC_PCI_BARMAP　B-3
PROC_PID_AFFINITY　B-3
PROC_PID_RESMEM　B-3
プロセス
CPUへの割付け　2-14–2-17
ブロック　5-37–5-41
協同　5-37
ディスパッチ・レイテンシー　2-2, 2-3
実行時間クォンタム　4-4–4-5, 4-9, 4-13, 4-14, 7-3

メモリ常駐　1-9
優先度　4-1, 4-2
スケジューリング　4-1, 7-3
同期　1-10, 5-1
起床　2-21, 5-37–5-41
プロセス・スケジューラ　4-2
PROCMEM_ANYONE　9-4, B-3
PROCMEM_MMAP　9-4, B-3
PROCMEM_WRITE　B-3
プロファイリング　7-3
クアッドOpteron上のプログラムドI/O　2-28
psコマンド　4-3, 7-2
pthread_mutex_consistent_np　5-24
pthread_mutex_destroy　5-21
pthread_mutex_getunlock_np　5-24
pthread_mutex_init　5-21
pthread_mutex_lock　5-21
pthread_mutex_setconsistency_np　5-24
pthread_mutex_setunlock_np　5-25
pthread_mutex_trylock　5-21
pthread_mutex_unlock　5-21
pthread_mutexattr_destroy　5-21
pthread_mutexattr_getfast_np　5-25
pthread_mutexattr_gettype　5-21
pthread_mutexattr_init　5-21
pthread_mutexattr_setfast_np　5-26
pthread_mutexattr_setprotocol　5-27
pthread_mutexattr_setrobust_np　5-27
pthread_mutexattr_settype　5-21
pthread_mutexattr_setunlock_np　5-27
ptrace　1-6, B-4
PTRACE_EXT　B-4
関連資料　v

R

rcim　H-4
RCIM_CLOCKSOURCE　B-2
RCIM_IRQ_EXTENSIONS　B-3
RCIM_PPS　B-2
RCU　7-4, B-2
RCU_ALTERNATIVE　7-4, B-2
リード・コピー・アップデート(RCU)　7-4, B-2
Real-Time Clock and Interrupt Module (RCIM)　1-5, 6-1, B-2, B-3, H-4
リアルタイム・クロック・タイマー　6-2
リアルタイム機能　1-4
リアルタイム・プロセス・スケジューリング　4-1
リアルタイム・スケジューラ　1-6
リアルタイム・シグナル　1-11
RedHawk Linux
ケーパビリティ　C-1
資料一式　v
カーネル・パラメータ　11-1, 11-3, B-1
カーネル　1-3, 11-1, 11-2
POSIXの適合　1-2
リアルタイム機能　1-4
スケジューラ　4-2
更新　1-4
関連する資料　v
REQUIRE_RELIABLE_TSC　B-2
REQUIRE_TSC　B-2
resched_cntl　5-4
resched_lock　5-5
resched_nlocks　5-6
resched_unlock　5-6
RESCHED_VAR　B-2
再スケジューリング制御　5-3–5-7, 7-4
再スケジューリング変数　5-3, B-2
物理メモリの予約　2-23
rhash_entries　2-35, H-4
ロウバスト・ミューテックス　5-22
ロール・ベース・アクセス制御　1-7, 13-2
ラウンドロビン・スケジューリング　4-1, 4-4
RTCタイマー　6-2
runコマンド　2-15–2-17, 4-2, 4-14, 10-7

S

SBSテクノロジー　15-1
SBSVME　15-6, B-3
SCHED_FIFO　4-1, 4-3
sched_get_priority_max　4-12, 4-13
sched_get_priority_min　4-12
sched_getparam　4-11
sched_getscheduler　4-9
SCHED_OTHER　4-1, 4-4
SCHED_RR　4-1, 4-4
sched_rr_get_interval　4-13
sched_setaffinity　2-15
sched_setparam　2-21, 4-10
sched_setscheduler　2-21, 4-8
SCHED_SMT　B-3
SCHED_SMT_IDLE　2-35, B-3
sched_yield　4-11
リアルタイム・スケジューラ　1-6
ポリシーのスケジューリング　4-1, 4-3
優先度のスケジューリング　4-2
sem_destroy　5-15
sem_getvalue　5-21
sem_init　5-12, 5-14
sem_open　5-16
sem_post　5-20
sem_timedwait　5-19
sem_trywait　5-20

sem_unlink　5-18
sem_wait　5-19
セマフォ
　データ構造体　5-29
　POSIXカウンティング　5-2, 5-12–5-21
　System V　5-2, 5-28–5-37
semctl　5-28, 5-34
semget　5-28, 5-30, 5-31
semop　5-28, 5-29, 5-36
シリアル・コンソール構成　G-1
server_block　5-39
server_wake1　5-40
server_wakevec　5-41
set_mempolicy　10-10
shコマンド　7-4
共有メモリ　1-10
　NUMA　10-9
　概要　3-1
　POSIX　3-12–3-15
　System V　3-15–3-28
共有リソース　5-1
SHIELD　B-2
shieldコマンド　2-12–2-14, 2-17, 7-4
シールドCPU
　プロセッサ間割り込み　F-1
　例　2-13, 2-17, 2-31–2-34, 10-18
　インターフェース　2-12
　カーネル・デーモン　E-1
　カーネル・パラメータ　B-2
　NUMAインターフェース　10-3
　概要　1-4, 2-1
　性能　2-9–2-11, 4-6
　ユニプロセッサ　2-34
shm_open　3-12, 3-13
shm_unlink　3-12, 3-15
shmat　3-16, 3-23, 15-20
SHMBIND　B-3
shmbind　3-22, 15-13, 15-20
shmconfig　3-16, 3-25, 10-9, 15-13, 15-21
shmctl　3-16, 3-21
shmdefine　3-16, 3-25
shmdt　3-16, 3-23
shmget　3-15, 3-19, 3-22, 3-27
sigqueue　1-11
sigtimedwait　1-11
sigwaitinfo　1-11
スリープ・ルーチン　5-37, 6-11, 6-12
スリープ/ウェイクアップ/タイマーのメカニズム　5-37
スリーピーウェイト相互排他　5-2
SOFTIRQ_PREEMPT_BLOCK　B-3
SOFTIRQ_PRI　14-13, B-3
ソフトIRQ　4-5, 14-12, 14-13, B-3, E-2
スピン・ロック
　ビジーウェイト相互排他　1-8, 5-1, 5-2, 5-7–5-12
　条件同期　5-42
　マルチスレッド・デバイス・ドライバ　14-14
　nopreempt　5-10
　プリエンプション　1-5, 1-7
spin_init　5-8
spin_islock　5-9
spin_lock　5-8
spin_mutex　5-7
spin_trylock　5-9
spin_unlock　5-9
ssh　13-5
straceコマンド　7-4
スワップ　1-9
同期I/O　1-10
構文記法　iv
システム・プロファイリング　7-3
システム・セキュリティ　13-1
システム・アップデート　1-4
System V IPC
　メッセージ・キュー　3-1, 3-3–3-11, A-4
　セマフォ　5-2, 5-28–5-37
　共有メモリ　3-1, 3-15–3-28
System.mapファイル　11-4

T

タスクレット　4-5, 14-13
スレッド・ライブラリ　5-14
ティックレス・カーネル　B-2, H-3
タイム・スタンプ・カウンタ(TSC)　7-1
時間構造体　6-3
time-of-dayクロック　6-4, 7-1
timer_create　6-6
timer_delete　6-8
timer_getoverrun　6-10
timer_gettime　6-9
timer_settime　6-8
タイマー
　ローカル　2-14, 7-1, 7-4
　POSIX　1-11, 2-11, 6-2, 6-6–6-10, 7-4
　RCIM RTC　6-2
　システム　7-1
タイムシェアリング・スケジューリング　4-1, 4-4
topコマンド　4-3, 7-2
TRACE　B-5
カーネル・トレース・イベント　14-15
トレース・カーネル　1-3, 11-2
トレース・ポイント　1-6, 14-15
TSC　7-1

U

UIO　14-14, B-5
ユニプロセッサ　2-34
システム更新　1-4
ユーザー認証　13-1
ユーザー・レベル・スピン・ロック　1-8
usermap　1-8, 9-3, 9-4, B-3

V

仮想アドレス空間の予約　14-11
vmalloc　14-11, H-4
VMALLOC_RESERVE　14-11
vmcore　12-2, 12-8
VME-to-PCIのサポート(PCI-to-VMEのサポートを参照)
vmlinux　12-2, 12-8

W

プロセスの起床　2-21, 5-37–5-41
ワーク・キュー　14-12, 14-13

X

X86_64_ACPI_NUMA　B-4
X86_HT　2-30, B-5
xfs　1-9, 8-1, B-4
XFS_DMAPI　8-2
XFS_FS　B-4
XFS_RT　B-4