

Copyright 2025 by Concurrent Real-Time, Inc. All rights reserved.

本書またはその一部は、Concurrent Real-Timeの社員、顧客、エンドユーザーによるConcurrent Real-Timeの製品を使用することを目的とします。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。

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。Concurrent Real-Time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。

Concurrent Real-Time, Inc.およびそのロゴはConcurrent Real-Time, Inc.の登録商標です。当社のその他すべての製品名はConcurrent Real-Time, Inc.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
March 2016	780	RedHawk Linux Release 7.2
March 2016	800	RedHawk Linux Release 7.2A
June 2017	810	RedHawk Linux Release 7.3
June 2018	820	RedHawk Linux Release 7.5
October 2019	900	RedHawk Linux Release 8.0
November 2020	920	RedHawk Linux Release 8.2
March 2021	930	RedHawk Linux Release 8.2a
December 2021	940	RedHawk Linux Release 8.4
June 2022	940	RedHawk Linux Release 8.4a
September 2022	960	RedHawk Linux Release 8.4b
March 2023	1000	RedHawk Linux Release 9.1Beta
June 2023, March 2024	1020	RedHawk Linux Release 9.2
August 2025	1030	RedHawk Linux Release 9.6

注意事項 :

本書は、Concurrent Real-Time, Inc.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。

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。Part 1の情報はリアルタイム・ユーザー向けです。Part 2はシステム管理者向けです。Part 3は附録、用語解説、索引で構成されます。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を説明します。
- 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を説明します。シールドCPUモデルについても説明します。
- 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を説明します。
- 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を説明します。
- 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を説明します。(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、条件同期ツールが含まれます)
- 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を説明します。
- 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・タイマーを説明します。
- 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 実行手順を説明します。
- 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します。
- 10章:「Non-Uniform Memory Access (NUMA)」は、特定のシステム上で利用可能なNUMAサポートを説明します。

Part 2 – 管理者

- 11章:「カスタム・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構築方法について説明します。

- 12章:「カーネル・デバッグング」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を説明します。
- 13章:「**PAM**ケーパビリティ」は、RedHawk LinuxのPAM認証機能を説明します。
- 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を説明します。
- 15章:「**PCI-to-VME** サポート」は、RedHawkがサポートするPCI-VME間ブリッジを説明します。
- 16章:「**PRT**カーネル・オプション」は、RedHawkのオプションであるPREEMPR_RTリアルタイム・セマンティクスを備えた一連のPRTカーネルを説明します。

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・プログラムを含みます。
- 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を含みます。
- 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)をリストアップします。
- 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をリストアップします。
- 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します。
- 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を説明します。
- 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を説明します。
- 付録H:「RedHawk起動コマンド・ライン・パラメータ」は、RedHawk特有の起動パラメータを表示します。

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 Real-TimeのWEBサイトにて参照または入手することが可能です。

<http://redhawk.concurrent-rt.com/docs/>

RedHawk Linux Operating System Documentation	Pub. Number
<i>RedHawk Linux Release Notes</i>	0898003
<i>RedHawk Linux User's Guide</i>	0898004
<i>Real-Time Clock & Interrupt Module (RCIM) User's Guide</i>	0898007
<i>RedHawk Linux FAQ</i>	N/A
Optional RedHawk Product Documentation	
<i>RedHawk Linux Frequency-Based Scheduler (FBS) User's Guide</i>	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・シールドディング	1-4
プロセッサ・アフィニティ	1-4
ユーザー・レベル・プリエンブション制御	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の修正	1-6
カーネル・トレース機能	1-6
ptrace拡張	1-6
カーネル・プリエンブション	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
カーネルのコア・ダンプ/クラッシュおよびライブ解析	1-8
ユーザー・レベル・スピン・ロック	1-8
usermapと/procの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	1-8
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	1-9
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	1-10
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
プリエンブション禁止の効果	2-7
オープン・ソース・デバイス・ドライバの影響	2-8

シールドリ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ドリング	2-9
割り込みからのシールドリング	2-10
ローカル割り込みからのシールドリング	2-10
CPUシールドリングのインターフェース	2-11
shieldコマンド	2-11
shieldコマンド例	2-13
終了ステータス	2-14
shieldコマンド拡張機能	2-14
cpuctlおよびcpustatシステムコール	2-14
CPUシールドリングの/procインターフェース	2-15
systemdシールド・サービス	2-15
CPUへの割り込み割り当て	2-16
systemdシールド・サービス	2-16
/procインターフェース	2-17
管理割り込みに関するカーネル起動オプション	2-17
CPUへのプロセス割り当て	2-17
runコマンド	2-18
/procインターフェース	2-18
sched_setaffinity()	2-18
mpadvise()	2-19
initへのCPUアフィニティ割り当て	2-20
シールドCPUの設定例	2-20
デターミニズムを高める手順	2-23
メモリのページをロック	2-23
プログラム優先度の設定	2-24
遅延割り込み処理の優先度設定	2-24
別プロセスの起床	2-24
キャッシュ・スラッシングの回避	2-25
物理メモリの予約	2-26
NUMAノードへのバインディング	2-30
4-WayシステムのI/Oスループット	2-30
ハイパースレッディングの理解	2-31
システム構成	2-33
推奨されるCPU構成	2-33
標準的なシールドCPUモデル	2-34
割込みの分離を使ったシールド	2-34
ハイパースレッドのシールド	2-35
浮動小数点/整数の共有	2-36
データ・キャッシュの共有	2-36
単一プロセッサのシールド	2-37
メモリ不足状態の回避	2-37
Linuxのデターミニズムに関する既知の問題	2-37

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	3-24
shmdefineユーティリティ	3-24
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
デッドライン・スケジューリング(SCHED_DEADLINE)	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
低優先度スケジューリング(SCHED_IDLE)	4-5
性能向上のための手続き	4-5
優先度設定方法	4-5
割り込みルーチン	4-5
SCHED_FIFO vs SCHED_RR	4-6
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	4-7
プロセス・スケジューリング・インターフェース	4-7
POSIXスケジューリング・ルーチン	4-7
sched_setschedulerルーチン	4-8
sched_getschedulerルーチン	4-9
sched_setparamルーチン	4-9
sched_getparamルーチン	4-10
sched_yieldルーチン	4-11
sched_get_priority_minルーチン	4-11
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	4-13

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	5-8
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	5-11
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
sem_initルーチン	5-14
sem_destroyルーチン	5-15
sem_openルーチン	5-16
sem_closeルーチン	5-18
sem_unlinkルーチン	5-18
sem_waitルーチン	5-19
sem_timedwaitルーチン	5-19
sem_trywaitルーチン	5-20
sem_postルーチン	5-20
sem_getvalueルーチン	5-20
POSIX ミューテックスの基礎	5-21
ロウバスト・ミューテックス	5-22
優先度継承	5-22
ユーザー・インターフェース	5-22
pthread_mutex_consistent	5-23
pthread_mutexattr_getprotocol	5-23
pthread_mutexattr_getrobust	5-24
pthread_mutexattr_setprotocol	5-24
pthread_mutexattr_setrobust	5-24
POSIX ミューテックス・プログラムのコンパイル	5-25
System Vセマフォ	5-25
概要	5-25
System Vセマフォの利用	5-26
semgetシステムコール	5-29
semctlシステムコール	5-30
semopシステムコール	5-32
条件同期	5-34
postwaitシステムコール	5-34
serverシステムコール	5-35
server_block	5-36
server_wake1	5-36
server_wakevec	5-37
条件同期ツールの適用	5-38

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
プロセス実行時間のクォンタムおよび制限	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
ダイレクト・ディスクI/O	8-2
 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
run(1)を利用したNUMAサポート(プロセス用)	10-3
shmconfig(1)を利用したNUMAサポート(共有メモリ領域用)	10-5
システムコール	10-7
ライブラリ機能	10-7
情報提供ファイルおよびユーティリティ	10-7

ノード統計値	10-8
マッピングされたページのノードID	10-8
numastatを利用したNUMA成功/失敗統計値	10-9
NUMAバランシング	10-10
NUMAバランシングの有効化	10-11
シールドイングの相互作用	10-11
シールドイングの制限	10-11
性能ガイドライン	10-12
タスク全体のNUMA mempolicy	10-12
共有メモリ領域	10-13
構成	10-13
 11章 カスタム・カーネルの構成および構築	
序文	11-1
カーネル・パッケージの構築手順	11-1
Rocky互換システムの手順	11-2
Ubuntuベース・システムの手順	11-3
xconfig	11-4
3rdパーティ・ドライバ・モジュールの構築	11-5
動的カーネル・モジュール・サポート	11-6
 12章 カーネル・デバッグング	
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	12-2
VMcore構成	12-2
kdump構成の更新	12-4
scp VMcore生成の構成	12-5
NFS VMcore生成の構成	12-6
sysctl(1) kdumpオプション	12-8
crashを利用したダンプの解析	12-8
ダンプ・ファイルの解析	12-9
実行中システムの解析	12-9
ヘルプの入手	12-10
NMI割り込み	12-10
NMIウォッチドッグ	12-11
 13章 PAMケーパビリティ	
序文	13-1
PAMサービス・ファイル	13-2
PAM構成ファイル	13-2
ロール・ベース・アクセス制御	13-4
ロール	13-4
グループ	13-5
ユーザー	13-5
実例：リアルタイム・ユーザー向けPAMケーパビリティの構成	13-5
通常使用されるサービスの割り当て	13-6
リアルタイム・ロールの割り当て	13-6
リアルタイム・ユーザーの割り当て	13-7
リアルタイム・ケーパビリティの確認	13-7

14章 デバイス・ドライバ

デバイス・ドライバの種類理解	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
bar_scan_open(3)	14-3
bar_scan_next(3)	14-3
bar_device_count(3)	14-4
bar_scan_rewind(3)	14-4
bar_scan_close(3)	14-4
free_pci_device(3)	14-5
bar_mmap(3)	14-5
bar_munmap(3)	14-5
カーネル・スケルトン・ドライバ	14-6
サンプル・ドライバの機能理解	14-6
ドライバのテスト	14-9
カーネル・レベル・デバイス・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
ソフトIRQとタスクレット	14-13
ワーク・キュー	14-13
優先度の理解	14-13
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関数	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
bigphysareaパッチ	15-11
btpモジュールのアンロード	15-11
bt_bindとrem_addr_pパラメータ	15-12

ローカル・メモリ	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
VMEバス・マッピングの表示	15-15
VMEバス・マッピングの生成	15-15
VMEバス・マッピングの削除	15-16
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
 16章 PRTカーネル・オプション	
PRTとは?	16-1
RedHawk vs PRT	16-1
PRTの注意事項	16-2
PRTカーネル・フレイバー	16-2
追加リソース	16-3
 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	
 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-2
データ型のサイズ	D-2
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-4

API	D-4
呼び出し規約	D-5
条件付コンパイル	D-5
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-6
メモリのアライメントおよび構造体のパディング	D-6

付録E シールドCPU上のカーネル・レベル・デーモン

付録F シールドCPU上のプロセッサ間割り込み

概要	F-1
グラフィクス割り込み	F-1
RedHawkページ・プール・ドライバ	F-2
ユーザー・アドレス空間のTLBフラッシュ割り込み	F-3

付録G シリアル・コンソールの設定

付録H RedHawkの起動コマンド・ライン・パラメータ

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 Real-Time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、自立走行車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はConcurrent Real-TimeのiHawkシステムに各々含まれています。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。

x86アーキテクチャのシステムについては、一般的なCentOS(Community Enterprise Operating System®)が含まれています。ARM64アーキテクチャではUbuntuが提供されます。ベースのディストリビューション(CentOSまたはUbuntu)は、以下、ベースLinuxディストリビューションと称します。

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。カーネルを除き、ベースLinuxディストリビューションの全てのコンポーネントは標準的な方法で動作します。それらは修正されていないLinuxユーティリティ、ライブラリ、コンパイラ、ツール、インストーラを含んでいます。オプションの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・モニタを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 Real-Time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 Real-Time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「リアルタイム機能」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。

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・シールドディンクは、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および4.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 Real-Time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。Intel x86とARM64の両アーキテクチャ上のLinuxは、Concurrent Real-Timeのx86およびARM64のiHawkシステムが動作するそれらのプラットフォーム上で動作するようパッケージ・アプリケーションが設計された事実上のバイナリ標準を定義しました。

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 Real-Time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シールドリ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のシールドリ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リアルタイム有りと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 *	vmlinux-kernelversion-RedHawk-x.x	vmlinux-kernelversion-RedHawk-x.x-trace	vmlinux-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
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
* <i>kernelversion</i> 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。 <i>x.x</i> はRedHawkのバージョン番号を示します。 例: vmlinux-3.10.25-rt23-RedHawk-6.5.			

システム・アップデート

「RedHawk Linux updates」はConcurrent Real-Time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 Real-TimeはベースLinuxディストリビューションにアップデートをダウンロードすることを推奨しません。Concurrent Real-Time以外のソースからのアップグレードのインストール(特にgccとglibcに対して)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のアップデートは必要であればインストールすることは可能です。

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、本書の後續の章にて提供します。オンラインで読まれている方は、参照用語上をクリックすることで直ぐにその情報を表示することが可能です。一部の機能は全アーキテクチャに適用されておらず、一部は全アーキテクチャでサポートされていません。そのような場合、章の中に適用されていないもしくはサポートされていないことを示す注記があります。

プロセッサ・シールドイング

Concurrent Real-Time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・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章ではシールドイング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 Real-Time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・プリエンブション制御

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ブ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ブ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・レベルのプリエンブ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 Real-Time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ち構えているプロセスを瞬時にサスペンドすることが可能です。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つの外部割り込み、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NighthTraceにより利用できます。カーネル・トレースに関する情報はNighthTraceの資料を参照してください。

ptrace拡張

Linuxのptraceデバッグ・インターフェースは、NightViewデバッガ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・プロセスが現在停止状態ではないプロセスのメモリを読み書きする機能
- デバッガがデバッグ中のプロセスのシグナルの一部だけをトレースする機能
- デバッガがデバッグ中のプロセス内の新しいアドレスで効率的に実行を再開する機能で、これは条件付きイベント・ポイントを実装するためにNightViewデバッガで使用されます

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 Real-Time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root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。

カーネルのコア・ダンプ/クラッシュおよびライブ解析

kexec-toolとcrash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。**crash**ユーティリティはライブ・システムでも使用することが可能です。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と/procの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こと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プロセッサの機能です。これは1つの物理プロセッサをオペレーティング・システムに2つの論理プロセッサ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をサポート)

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の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。**mlock(2)**, **mlock2(2)**, **munlock(2)**, **mlockall(2)**, **munlockall(2)**のPOSIXシステムコールおよびRedHawk Linuxの**mlockall_pid(2)**と**munlockall_pid(2)**システムコールは物理メモリにあるプロセスの仮想アドレス空間の全てまたは一部をロックおよびアンロックすることが可能です。

RedHawkはカーネル・デーモンがメモリ移動できないようにするために**mlock2(2)**, **mlockall(2)**, **mlockall_pid(2)**のシステムコールに渡すことが可能な追加のフラグも提供します。回避することを意図するフラグとカーネルの動作は**noautomigrate(7)**で文書化されています。詳細はmanページを参照して下さい。

RedHawk Linuxの**sigbus_pagefaults(7)**デバッグ・サポートは、ユーザー・ページをロックした後でも発生し続ける可能性のある予期せぬページ・フォルトを探すために使用することも可能です。

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 Real-Time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・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な実行、同じく割り込みに対してデターミニスティック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シールドリングは、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ドリ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、コンピューター・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ー・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。(「プロセス・ディスパッチ・レイテンシー」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・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ドリングは、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ない、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。

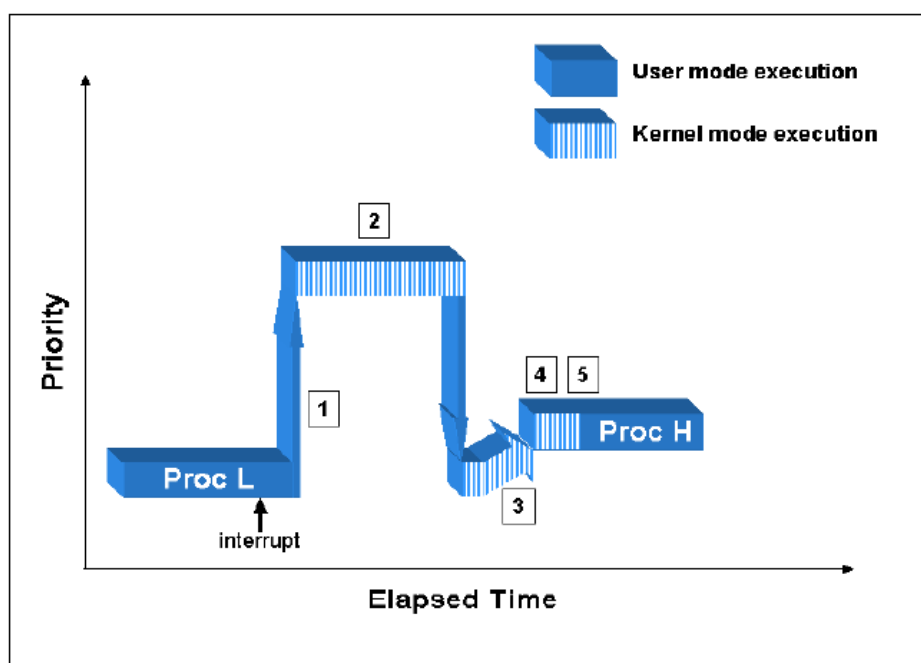
用語 *プロセス・ディスパッチ・レイテンシー* は、割り込みによって通知される外部イベント発生から外部イベント待ちプロセスがユーザー・モードでの最初の命令を実行するまでの時間経過を意味します。リアルタイム・アプリケーションにとって予想される最悪のプロセス・ディスパッチ・レイテンシーは基準の鍵になります。それは、デッドラインを満足することを保証するリアルタイム・アプリケーション性能を左右する最悪の応答時間であるためです。

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ため、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、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。

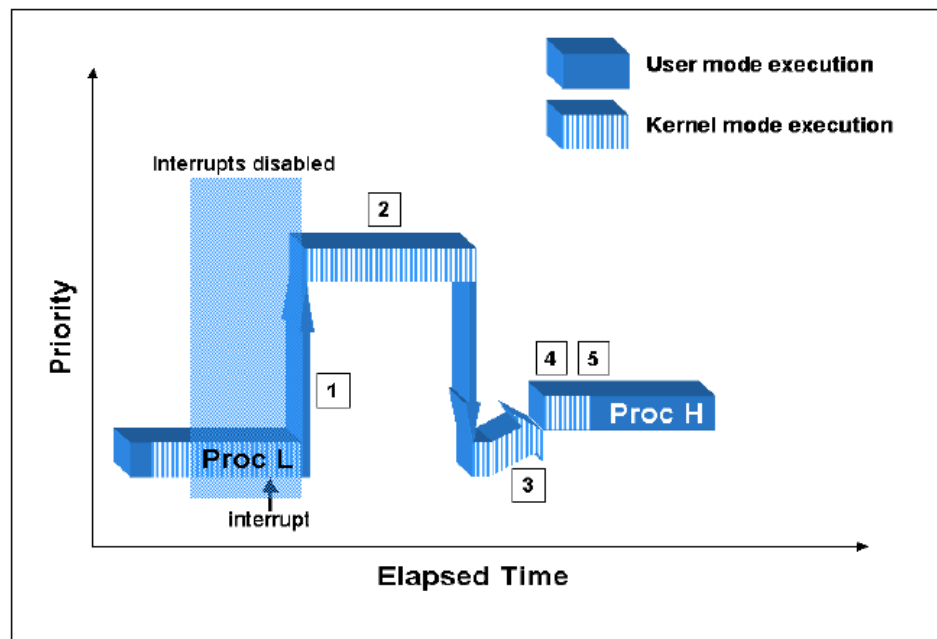
割り込み禁止の効果

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られなければなりません。

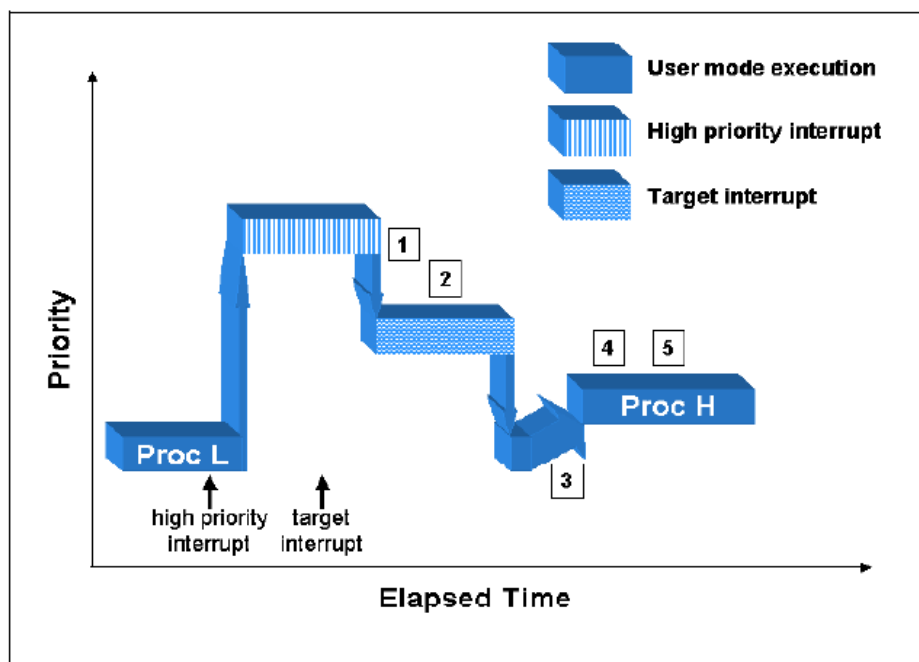
図2-2 割り込み無効によるプロセス・ディスパッチ・レイテンシーへの影響



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。

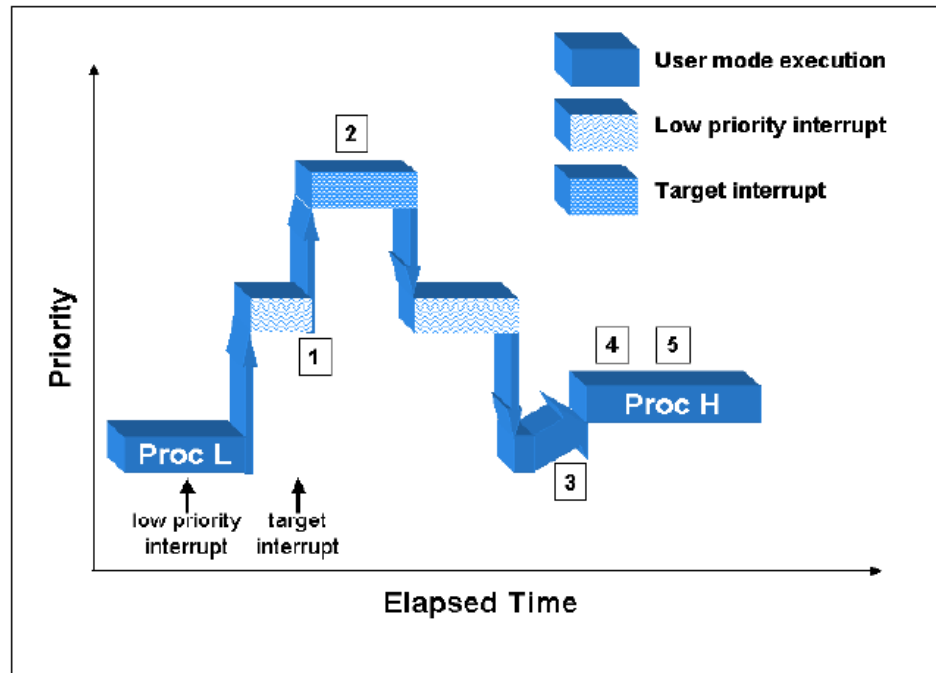
図2-3 高優先度割り込みによるプロセス・ディスパッチ・レイテンシーへの影響



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



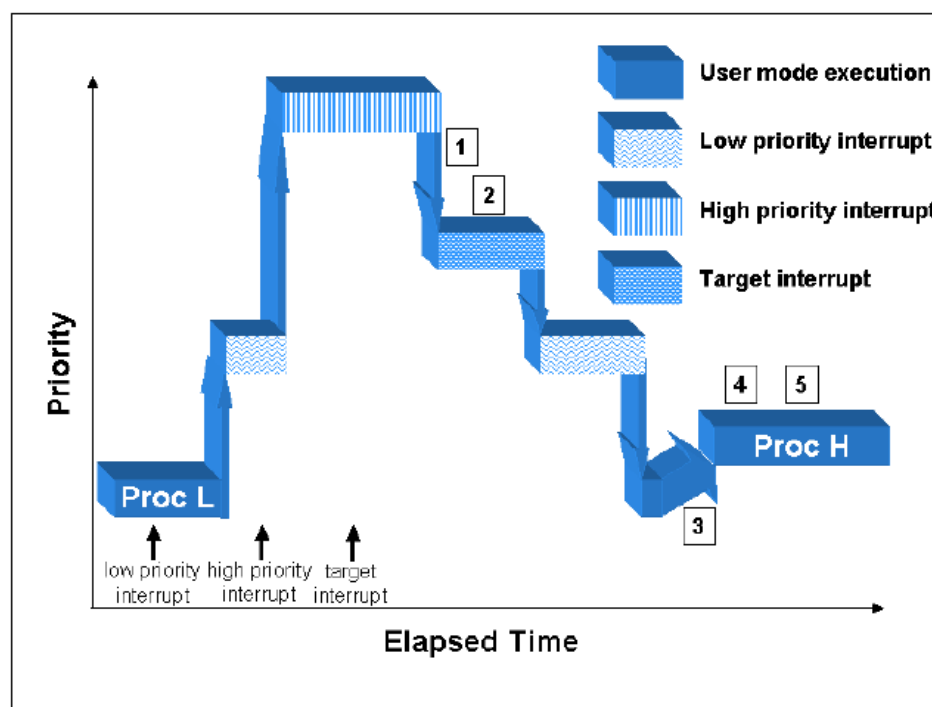
プロセス・ディスパッチ・レイテンシーに対する明確な影響で割り込みの無効化と割り込み受信とで最大の違いの1つは、割り込みが実行しているアプリケーションに対して非同期かつ予測不可能な時に発生することです。これは利用可能なシールドの様々なレベルを理解することが重要です。

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

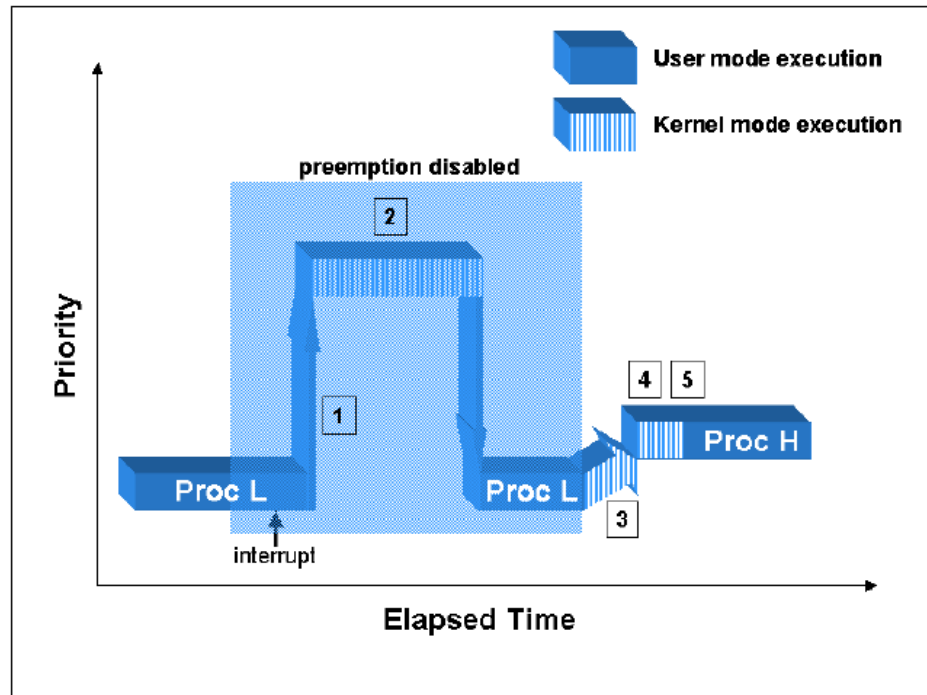


プリエンプション禁止の効果

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。

シールドリ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シールドリ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。

シールドリングを有効にする場合、すべてのシールドリ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。各々のシールドリ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ドリ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ドリング特性の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ドリングs
- 割り込みからのシールドリング
- ローカル割り込みからのシールドリング

各々の特性はCPU単位で個別に選択可能です。各々のシールドリ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ドリング

このシールドリング特性は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ブ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「プリエンブション禁止の効果」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い必要があります。

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Eで説明しています。

割り込みからのシールドイング

このシールドイ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、このシールドイ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「実際の」アフィニティ設定に反映されるように**shield(1)**コマンドによって、または**/proc/irq/<irq-no>/smp_affinity_list**を通して修正されます。

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ドイング

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章の中で説明されています。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となります。

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シールド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。
--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。
--procfs, -P	様々なシールド・マスクへのアクセスに古いprocfsインターフェースを使用することを強制します。
--help, -h	利用可能なオプションと使用方法を説明します。
--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。

NOTE

shield コマンドはユーザーのCPUアフィニティ設定を上書きしません。CPUアフィニティ設定がプロセスまたはIRQが他のCPUで実行するのを妨げる場合、シールドされたCPUで実行し続けます。これはエラーとは見なされません。

例えば、プロセスがCPU 1〜5で実行するように割付けました。そのCPUアフィニティ設定は次のようになります：

```
1-5 user 1-5 actual 1-5 effective - bash
```

その後、CPU 3がシールドされます。プロセスのCPUアフィニティは変更され、CPU 3はそのactualとeffectiveアフィニティから除外されますが当初の意図はユーザー・アフィニティ設定に記憶されます。

```
1-5 user 1-2,4-5 actual 1-2,4-5 effective - bash
```

今、CPU 1〜5がシールドされていると仮定します。プロセスには実行可能なCPUアフィニティに他のCPUがありません。プロセスはシールドされたCPUで実行し続けます。

```
1-5 user 1-5 actual 1-5 effective - bash
```

シールドされたCPUから移動するには、プロセスのCPUアフィニティを変更する必要があります。

NOTE

システムの全てのCPUがシールドされた場合、同じ効果が見られます。全てのCPUをシールドすると全てのシールドが無効となって次の警告が出力されます：

```
WARNING: All processors are inactive or shielded
from interrupts! This is the same as having no cpus
shielded from interrupts.
```

```
WARNING: All processors are inactive or shielded
form processes! This is the same as having no cpus
shielded from processes.
```

管理割り込みは**shield**コマンドで移動されない割り込みクラスの実例です。最近の殆どのNIC, RAID, NVMEデバイスはCPU毎の割り込みを生成します。CPU毎の割り込みは管理割り込みとして分類されます。そのCPUアフィニティは1つのCPUのみ設定されるため、**shield**コマンドでそれらの割り込みは移動されません。

プロセスまたはIRQが他のCPUで実行するのをCPUアフィニティ設定が妨げる場合、CPUアフィニティ設定を変更する必要があります。IRQおよびプロセスのCPUアフィニティ設定の変更の詳細については2-16ページの「CPUへの割り込み割り当て」と2-17ページの「CPUへのプロセス割り当て」をそれぞれ参照して下さい。

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、終了ステータスは0です。誤記または範囲外の数字などの構文エラーがある場合、**shield** コマンドはエラー(終了ステータス1)のみを返します。ユーザーはオプションなしの**shield** コマンドを実行してシステムの状態を確認する必要があります。

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。

*CPU***LIST** 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CPU 0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CPU 1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。

cpuctlおよびcpustatシステムコール

cpuctl関数は一部のCPUセットにシールドもしくはダウン操作のいずれかを実行します。詳細については**cpuctl(2)**のmanページを参照して下さい。

```
#include <sys/cpuctl.h>
#include <cpuset.h>

int cpuctl(int cmd, unsigned int sz, void *mask);

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

詳細については**cpuctl(3)**のmanページを参照して下さい。

cpustat関数はシステムのCPUに関する基本的な問合せを行います。詳細については**cpustat(2)**のmanページを参照して下さい。

```
#include <sys/cpustat.h>
#include <cpuset.h>

int cpustat(int numcmd, unsigned int int_sz, int * num);
int cpustat(int maskcmd, unsigned int set_sz, cpuset_t * mask);
int cpustat(int tblcmd, unsigned int tbl_sz,
            struct cpustat_cpuid *tbl);

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

CPUシールドディングの/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：ビットマスクを使用
/proc/shield/procs-list	プロセスのシールド：CPUリストを使用
/proc/shield/irqs	IRQのシールド：ビットマスクを使用
/proc/shield/irqs-list	IRQのシールド：CPUリストを使用
/proc/shield/tmrs	ローカル・タイマーのシールド：ビットマスクを使用
/proc/shield/tmrs-list	ローカル・タイマーのシールド：CPUリストを使用

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。

上記の接尾辞-listが付いた/procファイルはシールドされたCPUのリストを返します。ファイルへ書き込む時、CPUのリストが期待されます。例：

```
echo 10-11,8 > /proc/shield/procs
cat /proc/shield/procs
8,10-11
```

上記の接尾辞-listなしのファイルは16進数値をASCIIで返します。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。設定された各ビットのポジションは、そのビットによりシールドされている論理CPUの番号です。

例：

```
001 - 0ビット目が設定されているのでCPU #0がシールド
002 - 1ビット目が設定されているのでCPU #1がシールド
004 - 2ビット目が設定されているのでCPU #2がシールド
006 - 1ビット目と2ビット目が設定されているのでCPU #1と#2がシールド
```

ファイルへ書き込む時、16進数値のASCIIが期待されます。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。

追加の情報についてはshield(5)のmanページを参照してください。

systemdシールド・サービス

systemd shieldサービスは起動時に選択されたCPUに対してシールド属性を設定するために使用することが可能です。本サービスはシステムがシールドされた状態を再起動後も維持することが必要である場合に便利です。変更は構成ファイル/etc/sysconfig/shield内で行われます。本ファイル自身がドキュメントとなっています。

次の例では、CPU 10～12と15がプロセスからシールドされ、CPU 17～19は全てからシールドされます。サービスが再起動されるので次のリブートの代わりに直ぐに実行されます。

/etc/sysconfig/shieldファイルを編集してこれらの変数を設定して下さい：

```
SHIELD_PROCESSES=10-12,15
SHIELD_ALL=17-19
```

シールド・サービスの再開で変更を実施して下さい：

systemctl restart shield

最後のコマンドの状態を見るには：

systemctl status shield

CPUのシールド状況を確認するにはオプションなしで**shield**コマンドを実行して下さい：

shield

CPUへの割り込み割り当て

IRQのCPUアフィニティはオペレーティング・システムにより設定されます。IRQのCPUアフィニティを変更する2つの起こり得る理由は：

1. IRQが1つ以上のシールドされたCPUにより処理されることが要求される。
2. IRQが1つ以上シールドされたCPUから移動されることが要求される。

CPU毎の割り込み(管理割り込みとも呼ぶ)はシールドされたCPUから移動される必要がある可能性のある割り込みのクラスです。管理割り込みのCPUアフィニティはこれらの手法のいずれかで変更される可能性があります、後述の「管理割り込みに関するカーネル起動オプション」項はその割り込みクラスに関する具体的なソリューションであることに注意して下さい。

systemdシールド・サービス

systemd shieldサービスは起動時に選択されたCPUに対してシールド属性を設定するために使用することが可能です。これは特定のCPUにIRQを割り当てることも可能です。変更は構成ファイル**/etc/sysconfig/shield**で行われます。本ファイル自身がドキュメントとなっています。

例えば、次の行を**shield**サービス構成ファイルに追加することで**enp4s0f0**割り込みをCPU 0～4に割り当て、割り込み番号55, 60, 61をCPU 0と2に割り当てることが可能です。最初の割り当てには「=」の前に「+」記号はありませんが、残り全てはあることに注意して下さい。

```
IRQ_ASSIGN="0-4:enp4s0f0;"
IRQ_ASSIGN+="0,2:55; 0,2:60; 0,2:61;"
```

コマンドでシールド・サービスを再開して下さい：

systemctl restart shield

変更がシステムに対して行われますが再起動は必要ではありません。コマンドで再開の状況を確認して下さい：

systemctl status shield

/proc/<irq-no>/smp_affinity_listを読むことで変更を確認することが可能です。

この変更は再起動後も維持することに注意して下さい。

/procインターフェース

特定のIRQのCPUアフィニティは対応するファイルに書き込むことで設定することが可能です。本ファイルはCPUのリストを必要とします。

次の例では、IRQ番号11はCPU 0～10, 13で実行するように設定されます：

```
echo "0-10,13" > /proc/irq/11/smp_affinity_list
```

この変更は再起動後は維持しないことに注意して下さい。

管理割り込みに関するカーネル起動オプション

CPU毎割り込みは管理割り込みとして分類されます。最近の殆どのNIC, RAID, NVMEデバイスはCPU毎の割り込みを生成します。管理割り込みはリアルタイム性能に影響を及ぼす可能性がありますので管理割り込みの移動が必要となる場合があります。

各IRQの個々のディレクトリ下にある**managed**という名前の

proc/irq

ファイルは、IRQが移動するのが可能かどうかを示していることに注意して下さい。managed=0の場合、IRQは割り当てられたCPUを移動することは可能ですが、managed=1の場合は出来ません。全てのIRQは移動するのがシステムのデフォルトです。

カーネル起動パラメータ**msi_affinity_mask**を設定すると、起動時に全てのMSI(X)管理割り込みに対してアフィニティ・マスクを設定します。

```
msi_affinity_mask=<cpulist>
```

cpulistはCPUのリストを設定する必要があります。このリストに範囲またはカンマ区切りのリストを含めることが可能です。**cpulist**はCPU 0を含める必要があることに注意して下さい。

カーネル起動オプションは**blscfg(1)**コマンド(Ubuntu)システムでは**ccur-grub2(1)**を介して追加することが可能です。例：

```
blscfg -kopt-add msi_affinity_mask=0-5,7 <kernel-index>
blscfg -kopt-add irq_affinity =0-5,7 <kernel-index>
```

新しく追加されオプションである**-C**オプションで確認することが可能です：

```
blscfg -C <kernel-index>
```

変更を有効にするには再起動が必要となることに注意して下さい。

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があります。

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。**/proc**インターフェースもまた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カ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、適切なカ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。

runコマンド

runコマンドは4章の「**run**コマンド」および**run(1)**のmanページに記載されています。**run**コマンドはCPUのリストにプロセスを割り付けるために使用することが可能です。簡単な例を次に示します：

次の例では、プロセス15のCPUアフィニティをCPU5で実行するように設定します：

```
run -b 5 -p 15
cat /proc/15/affinity-list
5 user 5 actual 5 effective - bash
```

/procインターフェース

特定のプロセスのCPUアフィニティは対応する**/proc/<proc-id>/affinity-list**ファイルへの書き込みによって設定することが可能です。本ファイルはCPUのリストを必要とします。

次の例では、プロセスID 55で識別されるプロセスをCPU 3で実行するように設定します。そのuser, actual, effectiveはその後にCPU 3に設定されます：

```
echo 3 > /proc/55/affinity-list
cat /proc/55/affinity-list
3 user 3 actual 3 effective - bash
```

この変更は再起動後は維持しないことに注意して下さい。

sched_setaffinity()

sched_setaffinity()はIDが**pid**のスレッドのCPUアフィニティ・マスクを**mask** で指定された値に設定します。**pid** がゼロの場合、呼び出したスレッドが使用されます。引数**cpusetsize** は**mask** で指し示されたデータのバイト単位の長さです。

sched_getaffinity()はIDが`pid`のスレッドのアフィニティ・マスクを`mask`で指し示された構造体に書き込みます。引数`cpusetsize`は`mask`の大きさをバイト単位で指定します。

概要

```
#define _GNU_SOURCE
#include <sched.h>

int sched_setaffinity (pid_t pid, size_t cpusetsize,
                      const cpu_set_t *mask);

int sched_getaffinity (pid_t pid, size_t cpusetsize,
                      cpu_set_t *mask);
```

詳細については**sched_setaffinity(2)**および**sched_getaffinity(2)**に関するmanページを参照して下さい。

mpadvise()

mpadviseは、様々なマルチプロセッサの機能を実行します。CPUは`cpuset_t`オブジェクトへのポインタを指定することで識別し、それは1つ以上のCPUの組み合わせを指定します。ここでは制御コマンドのみを示します。CPUの設定に関する詳細な情報については**mpadvise(3)**および**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 -lcurr_rt ...
```

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

initへのCPUアフィニティ割り当て

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CPU 1,2,3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CPU 0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CPU 0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ため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。

CPU 1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CPU 1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)コマンド経由：

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

cpu(1)コマンド経由：

```
$ cpu
cpu  chip  core  ht  ht-sibling  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**ファイルシステム経由：

```
cat /proc/shield/irqs-list
1
```

全ての割り込みがCPU 1上での実行から排除されます。この例では、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、CPU 1にRCIMの割り込みの割り付けとCPU 1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RCIMはIRQ 17に割り当てられています。そのIRQ番号が分かったら、RCIMへの割り込みをIRQ 17の`/proc`ファイルを介してシールド・プロセッサに割り付けることが可能であることに注意して下さい。以下のコマンドはCPU 1にIRQ 17のCPUアフィニティ・マスクを設定します。

```
echo 1 > /proc/irq/17/smp_affinity_list
```

IRQのための“`smp_affinity_list`”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付き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 17のアフィニティ用`smp_affinity_debug_list` `/proc`ファイルは変更が有効になっ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_debug_list
1 user 1 actual 1 effective
```

“user”で返された値は、IRQのCPUアフィニティ用にユーザーにより指定されたCPUのリスト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がリストから取り除かれた後のCPUのリストとなります。もしユーザーがシールドCPUと非シールドCPUの両方を含むアフィニティ・リストを設定した場合、シールドCPUはIRQのアフィニティ・リストから取り除かれる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・リストの中に非シールドCPUがない場合、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、CPU 1は割り込みからシールドされていますが、このアフィニティ・リストはCPU 1のみが割り込みの処理を許可されていることを示すのでCPU 1はIRQ 17を処理します。

`smp_affinity_debug` `/proc`ファイルは、CPUアフィニティをビットマスク形式で示すことに注意して下さい：

```
cat /proc/irq/17/smp_affinity_debug
002 user 002 actual 002 effective
```

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・プロセスからシールドされたCPUにおいて、シールドされたCPUだけを指定するCPUアフィニティ・リストを持つプロセスだけがシールドされたプロセッサ上で実行することが許可されます。プロセスのアフィニティ・リストの中に何らかの非シールド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”をリアルタイム優先度で実行し、CPU 1上で動作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()`」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確認するために使用することが可能です：

```
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からです。この例では、CPU 1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CPU 1だけがプログラムの実行を許可されたことを示すため、CPU 1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**mlock(2)**, **mlock2(2)**, **munlock(2)**, **mlockall(2)**, **munlockall(2)**, **mlockall_pid(2)**, **munlockall_pid(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。これらのシステムコールを利用するには、**CAP_IPC_LOCK**カーナビリティを所有する必要があります。**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**カーナビリティも必要となる可能性があります。カーナ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、様々なカーネル・メモリ管理サービスはアプリケーションのアプリ性能を高めるために自動的にアプリケーションのメモリを方々に移動します。これらの自動的な移動を無効にするため、RedHawkは**mlock(2)**, **mlockall(2)**, **mlockall_pid(2)**システムコールで使用される追加のフラグを提供します。カーネルのメモリ移動やこの動作を回避するフラグは**noautomigrate(7)**のmanページに記載されています。

更に**run**コマンドは**--lock**オプションを使って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リングされたプロセスは、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を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カーパビリティを持っていないと注意してください(これらのルーチンを利用するためのカ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。結果的に優れたプロセス・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の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 のいず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(x86アーキテクチャ)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ータ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な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`/etc/grub2.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アダプター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`/etc/grub2.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以降において‘\$’は予約語となりますので、‘\’(バックスラッシュまたは円記号)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2048byte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**/etc/grub2.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。

/usr/bin/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	0000000000000000-000000000009fbff : System RAM
000000000009fc00-000000000009ffff : reserved	000000000009fc00-000000000009ffff : System RAM
00000000000a0000-00000000000bffff : Video RAM area	00000000000a0000-00000000000bffff : Video RAM area
00000000000c0000-00000000000c7fff : Video ROM	00000000000c0000-00000000000c7fff : Video ROM
00000000000c8000-00000000000cbfff : Adapter ROM	00000000000c8000-00000000000cbfff : Adapter ROM
00000000000f0000-00000000000fffff : System ROM	00000000000f0000-00000000000fffff : System ROM
0000000000100000-0000000000d7feffff : System RAM	0000000000100000-0000000000cffeffff : System RAM
0000000000100000-000000000005c9521 : Kernel code	0000000000100000-000000000005c9521 : Kernel code
000000000005c9522-00000000000954137 : Kernel data	000000000005c9522-00000000000954137 : Kernel data
00000000d7ff0000-00000000d7ffefff : ACPI Tables	00000000cfff0000-00000000cfffffff : ACPI Tables
00000000d7fff000-00000000d7fffff : ACPI Non-volatile Storage	00000000ff7c0000-00000000ffffffff : reserved
00000000ff7c0000-00000000ffffffff : reserved	0000000100000000-000000011ffffffff : System RAM
0000000100000000-000000017ffffffff : System RAM	0000000120000000-000000013ffffffff : reserved
.	0000000140000000-000000017ffffffff : 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 사용할 수 있습니다。

以下の例は物理アドレス0x3e000000にアクセス制限なしおよび6602のキーを伴った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(1)** (-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事が出来ます。

```
# 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

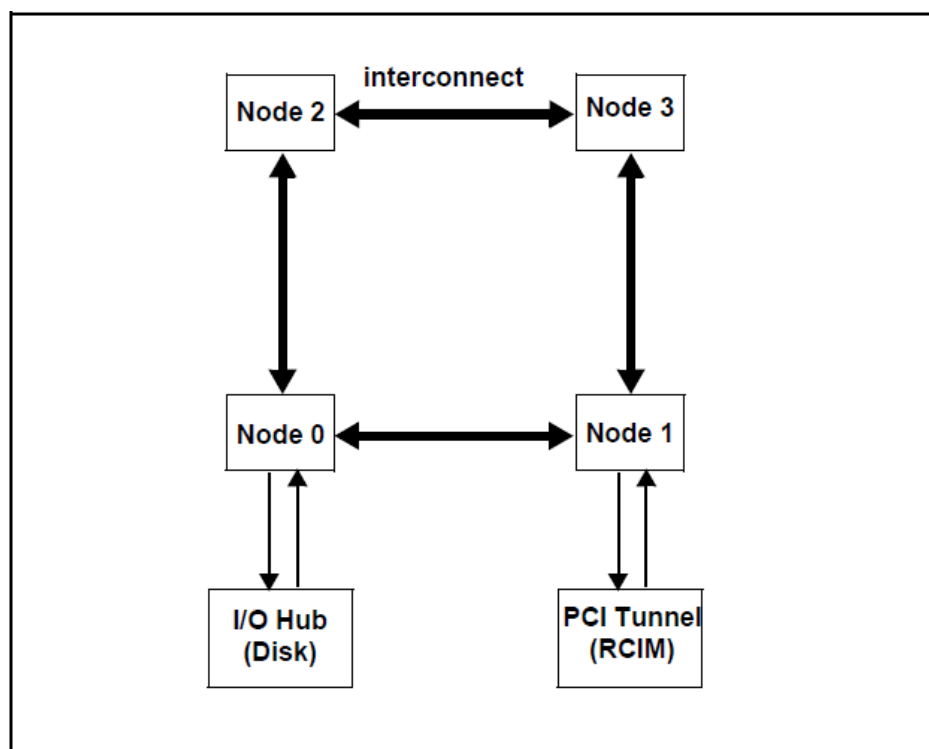
Non-Uniform Memory Access (NUMA) のシステム上では、ほかのシステムよりも一部のメモリ領域へのアクセスに時間が掛かります。NUMA システム上のメモリはノードに分けられ、ノードはメモリ領域と NUMA ノードのメモリ領域と同じ物理バス上に存在する全ての CPU で定義されます。もしこのタイプのシステム上で実行中のプログラムが NUMA 対応でない場合、十分に実行することが出来ません。

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。

4-Way システムの I/O スループット

NUMA をサポートするクアド対称型マルチ・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プロセッサ相互接続(例: Intel QPI/UPI および AMD HyperTransport™)を介してどの CPU からで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 NUMA I/Oスループット・レイアウト



NUMA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、プロセッサ相互接続バス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ター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、最新のIntel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できます。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。

- ハイパースレッディングをサポートするIntelシステム・アーキテクチャ。
- カーネル構成GUI上の「Symmetric multi-processing support」にある利用可能なSMPカーネル・チューニング機能を通してマルチ処理サポートを有効にする構成にする必要があります。マルチ処理とハイパースレッディングはすべてのRedHawk x86_64定義済みカーネルではデフォルトで有効化されています。
- 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。

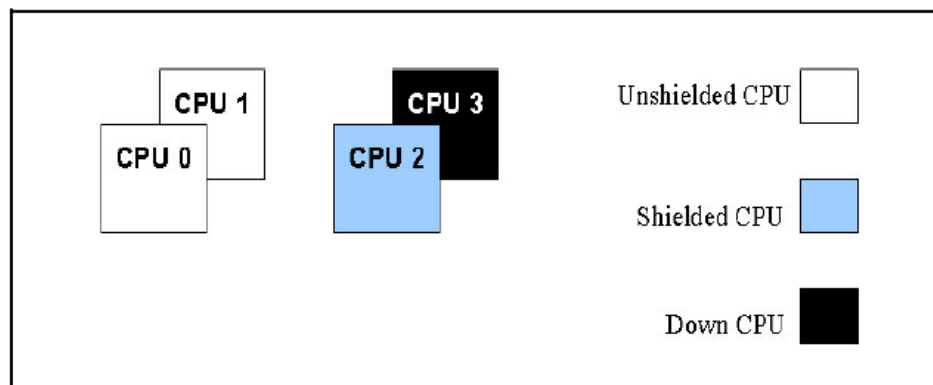
標準的なシールドCPUモデル

このモデルは、プログラム実行においてデター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「シールドリ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、CPU 3はダウン状態となりCPU 2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CPU 2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、

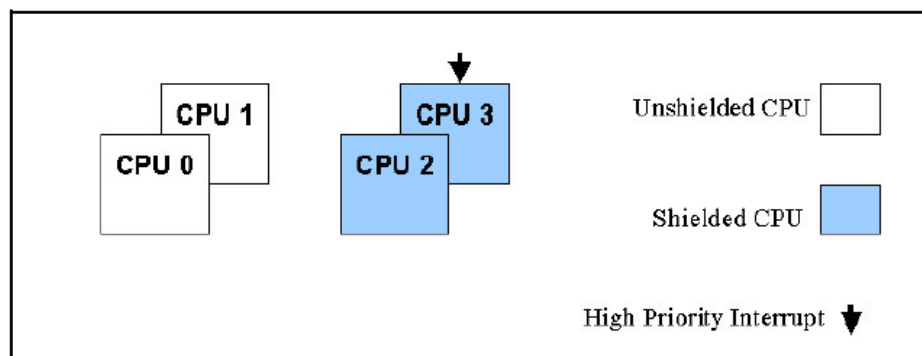
```
# shield -a 2
# cpu -d 3
```

割り込みの分離を使ったシールド

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ドディングを図2-9に図示します。

Figure 2-9 割り込みを分離したシールドディング



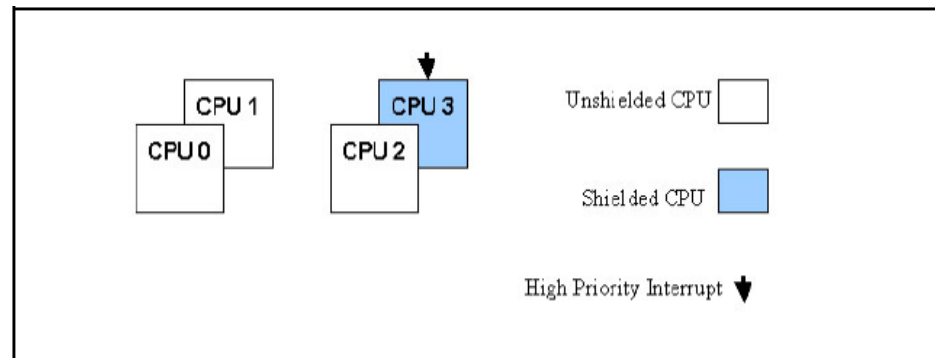
このアプローチの利点は、(CPU 3上で動作する)割り込みルーチンとシブリングCPU上の高優先度タスク (CPU 2上で動作する割り込み待ちプログラム)の実行との間で僅かながらの並列処理を提供することです。割り込みルーチンがCPU 3上で実行している唯一のコードであるため、この割り込みルーチンは通常L1キャッシュに完全に保持され、そのコードはキャッシュの中に留まって、割り込みルーチンに対し最適な実行時間を提供します。その割り込みルーチンはシブリングCPU上の割り込み待ちタスクを起こすためにプロセッサ間割り込みを送信する必要があるため、小さな代償を払うことにはなります。この余分なオーバーヘッドは2 μ 秒未満です。

割り込み分離によるシールド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ド

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ドディング構成を図2-10に図示します。

図2-10 ハイパースレッドのシールドリング



このサンプルでは、CPU 3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。CPU 2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式を実行するために構成されています。プリエンブションや割り込みブロックが無効の時、CPU 3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、CPU 2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CPU 3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っても依存します。

浮動小数点 / 整数の共有

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の両方のシプ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つのシプリング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プリング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。

データ・キャッシュの共有

この構成はアプリケーションが生産者/消費者型ア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のシプリングに割り付ける必要があります。2つのシプリング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つのシプリング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つのシプリングCPU上のプロセスのためにもう一つのシプリング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。

単一プロセッサのシールド

この構成はハイパースレッド・シールドリング構成の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、単一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、単一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シールドリ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 Real-Time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しないでください。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。
- 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。

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)**)を使って見ることも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・キュ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され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 사용되는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
- メッセージのテキスト・サイズ
- メッセージのテキスト・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
- 最後のメッセージ送信者のプロセス識別番号 (PID)
- 最後のメッセージ受信者のPID
- 最後のメッセージの送信時間
- 最後のメッセージの受信時間
- 最後の変更時間

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(1)**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ったら、**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 と一緒に使用は、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と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(1)**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 <i>buf</i> 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 <i>buf</i> 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(1)**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カ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カ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カ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。

msgid 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、*msgid* 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**設定されていない**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**設定されている**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1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、 <i>name</i> 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、 <i>mode</i> 引数により指定された許可ビットが設定されます。 <i>name</i> 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 <i>name</i> 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。 O_CREAT が指定されない場合は、このビットは無視されます。
O_TRUNC	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、 <i>name</i> 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：

<i>name</i>	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、 <i>name</i> 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(1)**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を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に示すメンバーを含みます。

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>**ヘッダー・ファイルの中にあります。

この構造体の`shm_perm`メンバー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>**ヘッダー・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(1)**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。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 と一緒に使用は、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と一緒に使用して簡単に設定することが可能です。使用例：

```
shmids = shmget (key, size, (IPC_CREAT | 0400));
shmids = 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(1)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 shm_id *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 <i>shmid</i> に対応するデータ構造体、ポインタ <i>buf</i> 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 <i>shmid</i> 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 <i>shmid</i> 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・ユーザーもしくは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・ユーザーもしくは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(1)**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カーナビリティを所有

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なも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：

<i>shmid</i>	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
<i>paddr</i>	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。例えば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	<p>以下で構成される<code>DEVSTR</code> 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</p> <p><code>vendor_id:device_id</code></p> <p>--bindを使用すると役に立ちます。 <code>DEVSTR</code> 上の情報に関しては--bindを参照してください。</p>
--bind=BARSTR, -b BARSTR	<p>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。 <code>BARSTR</code> は以下で構成されます。</p> <p><code>vendor_id:device_id:bar_no[:dev_no]</code></p> <p><code>vendor_id</code> と <code>device_id</code> 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、<code>/usr/share/hwdata/pci.ids</code>、<code>lspci -ns</code>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。</p> <p><code>bar_no</code> 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 <code>bar_no</code>: Memory at ...”と表示されます)。メモリ領域だけをバインドすることが可能です。</p> <p><code>dev_no</code> 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: <code>dev_no</code>.”と表示されます)。</p> <p>このオプションを使用するためにユーザーは <code>CAP_SYS_RAWIO</code>権限を持っている必要があります。</p>
--size=SIZE, -s SIZE	<p>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。</p>
--physical=ADDR, -p ADDR	<p>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<code>ADDR</code> を指定します。このオプションは廃止されたので、--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 <code>CAP_SYS_RAWIO</code>権限を持っている必要があります。</p>
--user=USER, -u USER	<p>共有メモリ領域所有者のログイン名称を指定します。</p>
--group=GROUP, -g GROUP	<p>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。</p>
--mode=MODE, -m MODE	<p>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<code>mode</code> 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。</p>
--help, -h	<p>利用可能なオプションと使用方法について説明します。</p>
--version, -v	<p>コマンドのバージョンを印字します。</p>

/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
```

実施例

この見本では、RCIM上の物理メモリ領域を**lspci(8)**を使って確認し、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)を探ることが可能で、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 Real-Time 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      shmids 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
```

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つはバックグラウンドでCPUに負荷をかけるプロセス用(SCHED_RATCH)、リアルタイム・アプリケーション用に2つの固定優先度(SCHED_FIFOとSCHED_RR)と動的優先度(SCHED_DEADLINE)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。例えば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。

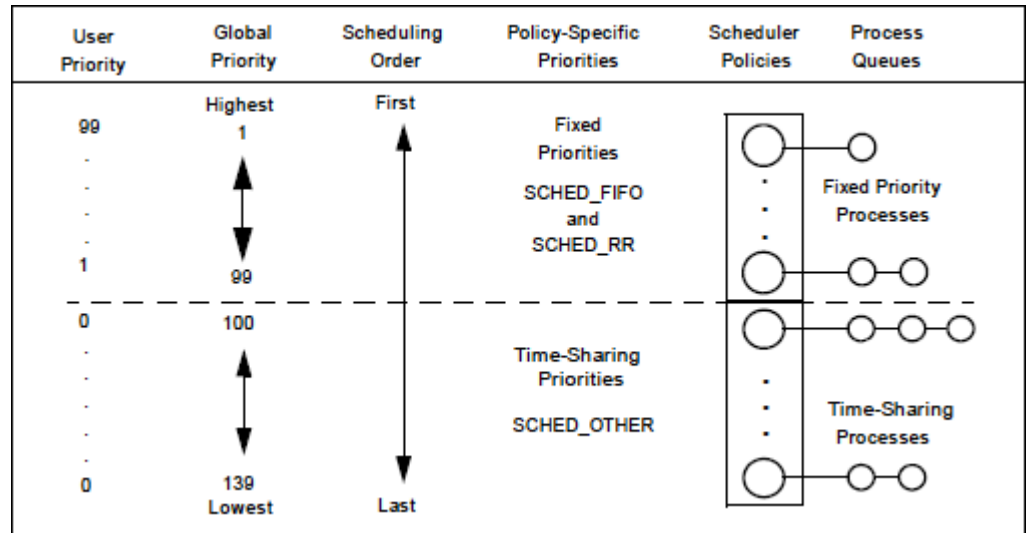
IEEE規格1003.1bに基づくシステムコール一式は、プロセスのスケジューリング・ポリシーおよび優先度へのダイレクトなアクセスを提供します。このシステムコール一式に含まれているのは、プロセスがスケジューリング・ポリシーおよび優先度を取得もしくは設定することを許可するシステムコールで、特定のスケジューリング・ポリシーに関連する優先度の最小値・最大値を取得し、ラウンドロビン(SCHED_RR)・スケジューリング・ポリシーに基づいてスケジューリングされたプロセスのタイム・クオンタムを取得。

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リング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リングされるためには、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値を判断するために**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の範囲内のユーザー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と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は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を注意してください。

スケジューリング・ポリシー

Linuxはプロセスをスケジュールする方法を制御するスケジューリング・ポリシーを5種類定義します：

SCHED_DEADLINE	周期的なソフト・リアルタイム・タスク向けのポリシー
SCHED_FIFO	ファーストイン・ファーストアウト(FIFO)・スケジューリング・ポリシー
SCHED_RR	ラウンドロビン(RR)・スケジューリング・ポリシー
SCHED_OTHER	デフォルトのタイムシェアリング・スケジューリング・ポリシー
SCHED_BATCH	対話の少ないジョブを長時間実行
SCHED_IDLE	CPUがアイドルである時に実行

デッドライン・スケジューリング(SCHED_DEADLINE)

SCHED_DEADLINEはRedHawkカーネルでサポートされますが、それよりもConcurrentはシールドを利用するリアルタイム・アプローチの使用を強く推奨します。

NOTE

SCHED_DEADLINEとSCHED_RRスケジューリング・クラスの両方が正しく機能するにはローカル・タイマー次第ですので、これらはローカル・タイマー・シールドとは互換性がありません。どちらかのスケジューリング・クラスが設定されたプロセスをローカル・タイマーがシールドされたCPUにバインドする場合、そのプロセスは期待通りにはスケジュールされません。

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リング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せん。

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・クオ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・クオ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・クオンタムを使い切らずに終了します。タイム・クオンタムの長さは**sched_rr_get_interval(2)**で取り出すことが可能です。タイム・クオンタムの長さは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・クオンタムを割り当てられます。

NOTE

SCHED_DEADLINEとSCHED_RRスケジューリング・クラスの両方が正しく機能するにはローカル・タイマー次第ですので、これらはローカル・タイマー・シールドとは互換性がありません。どちらかのスケジューリング・クラスが設定されたプロセスをローカル・タイマーがシールドされたCPUにバインドする場合、そのプロセスは期待通りにはスケジューリングされません。

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・クオ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リングする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。

低優先度スケジューリング(SCHED_IDLE)

SCHED_IDLEは静的優先度が0でのみ使用することが可能です(プロセスのナイス値はこのポリシーには影響なし)。

本ポリシーは非常に低優先度(SCHED_OTHERもしくはSCHED_BATCHポリシーでナイス値が+19以下)で実行中のジョブが対象です。

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・クオンタムを所有している間にタイム・クオ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・クオンタムに従い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・クオンタムは**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・クオンタムが割り当てられます。プロセスのタイム・クオンタムは**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、プログラマー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カーバ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・オーバー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およびアンロックするために**mlock(2)**, **mlock2(2)**, **munlock(2)**, **mlockall(2)**, **munlockall(2)**各システムコールおよびRedHawk Linuxの**mlockall_pid(2)**, **munlockall_pid(2)**システムコールを使用します。

RedHawkはカーネル・デーモンがメモリ移動するのを防ぐシステムコール**mlock(2)**, **mlockall(2)**, **mlockall_pid(2)**に渡すことが可能な追加のフラグも提供します。

これらのフラグやそれを回避することを意図するカーネルの動作は**noautomigrate(7)**に記載されています。詳細については**man**ページを参照して下さい。

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4-13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ラ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SCHED_RRへ変更する、もしくは、(2)SCHED_FIFOもしくはSCHED_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スーパーユーザーもしくはCAP_SYS_NICEカ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：

<i>pid</i>	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プロセス識別番号(PID)。現在のプロセスを指定するには <i>pid</i> の値をゼロに設定します。
<i>policy</i>	<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。 <i>policy</i> 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)・スケジューリング・ポリシー
SCHED_RR	ラウンドロビン(RR)・スケジューリング・ポリシー
SCHED_OTHER	デフォルトのタイムシェアリング・スケジューリング・ポリシー
SCHED_BATCH	対話の少ないジョブを長時間実行
SCHED_IDLE	CPUがアイドルである時に実行
<i>p</i>	<i>pid</i> 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4-11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・クオ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・クオ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・クオ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4-13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)・スケジューリング・ポリシー
SCHED_RR	ラウンドロビン(RR)・スケジューリング・ポリシー
SCHED_OTHER	デフォルトのタイムシェアリング・スケジューリング・ポリシー
SCHED_BATCH	対話の少ないジョブを長時間実行
SCHED_IDLE	CPUがアイドルである時に実行

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スーパーユーザーもしくはCAP_SYS_NICEカ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：

<i>pid</i>	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 <i>pid</i> の値をゼロに設定します。
<i>p</i>	<i>pid</i> 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4-9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4-11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：

<i>pid</i>	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 <i>pid</i> の値をゼロに設定します。
<i>p</i>	<i>pid</i> 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：

<i>policy</i>	ファイル内に定義されるスケジューリング・ポリシー。 <i>policy</i> 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)・スケジューリング・ポリシー
SCHED_RR	ラウンドロビン(RR)・スケジューリング・ポリシー
SCHED_OTHER	デフォルトのタイムシェアリング・スケジューリング・ポリシー

SCHED_BATCH	対話の少ないジョブを長時間実行
SCHED_IDLE	CPUがアイドルである時に実行

数値的に高い優先度値を持つプロセスは数値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)・スケジューリング・ポリシー
SCHED_RR	ラウンドロビン(RR)・スケジューリング・ポリシー
SCHED_OTHER	デフォルトのタイムシェアリング・スケジューリング・ポリシー
SCHED_BATCH	対話の少ないジョブを長時間実行
SCHED_IDLE	CPUがアイドルである時に実行

数値的に高い優先度値を持つプロセスは数値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・クオンタムを取得することが可能です。タイム・クオンタムとはカーネルがプロセスにCPUを割り当てて一定の時間です。CPUが割り当てられたプロセスがそのタイム・クオ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：

<i>pid</i>	タイム・クオ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 <i>pid</i> の値をゼロに設定します。
<i>tp</i>	<i>pid</i> で識別されるプロセスのラウンドロビン・タイム・クオンタムが返される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・クオンタムは*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/スレッド一式的環境を変更します。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・クオ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	タイムシェアリング・スケジューリング
SCHED_BATCH or batch	対話の少ないジョブを長時間実行
SCHED_IDLE or idle	CPUがアイドルである時に実行

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カーバビリティは更に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 list	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ー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ーは有効なユーザー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。

実施例

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CPU 0-3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CPU 3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 Real-Time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。

NOTE

再スケジューリング変数はARM64アーキテクチャではサポートされていません。本章で説明する高速ブロック/ウェイクアップ・サービス(`postwait()`と`server_block()/server_wake()`)およびビジーウェイト相互排他は再スケジューリング変数を使用するため、これもサポートされません。

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ク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ミューテックスの基礎」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

再スケジューリング変数はARM64アーキテクチャではサポートされていないことに注意して下さい。

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。再スケジューリング無効、クォンタム終了、プリエンブション、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、カーネルは再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。再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：

<i>cmd</i>	実行される操作
<i>arg</i>	<i>cmd</i> 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カ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、クオンタム終了、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、アンロックであると判断されます。

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

再スケジューリング変数はARM64アーキテクチャではサポートされていないため、ビジーウェイト相互排他もまたARM64アーキテクチャではサポートされていないことに注意して下さい。

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进行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**を返します。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**-Intrace**もしくは**-Intrace_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れ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**を少なくとも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とも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あることに注意して下さい。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**もしくは**-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インターフェースは以下を含みます：

- セマフォをゼロもしくは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・セマフォの値を取得します

これらのインターフェースを使用するため、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。

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 Real-Time 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：

<i>sem</i>	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
<i>pshared</i>	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 <i>pshared</i> 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 <i>pshared</i> 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、 <i>pshared</i> 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。
<i>value</i>	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。

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：

<i>sem</i>	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。

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：

<i>name</i>	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 <i>name</i> 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で埋め込めません。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。
<i>oflag</i>	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： <p>O_CREAT <i>name</i> 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<i>mode</i> 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<i>value</i> 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、<i>mode</i> と<i>value</i> 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。</p> <p>もし<i>name</i> 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。</p> <p>O_EXCL もしO_CREATが設定され、<i>name</i> 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。</p> <p>もしO_CREATとO_EXCL以外のフラグ・ビットが<i>oflag</i> 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。</p>
<i>mode</i>	次の例外を含む <i>name</i> 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 <i>mode</i> 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 <i>mode</i> 引数を指定します。
<i>value</i>	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 <i>value</i> 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。

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：

<i>sem</i>	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。

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：

<i>name</i>	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 <i>name</i> 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で埋め込めません。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。

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。

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。

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：

<i>sem</i>	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。

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：

<i>sem</i>	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。

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：

<i>sem</i>	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
<i>sval</i>	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ミューテックスの基礎

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・ミューテックス機能が含まれます。**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(3)	矛盾するミューテックスの矛盾をなくす
pthread_mutexattr_getprotocol(3)	プロトコルを返す
pthread_mutexattr_getrobust(3)	ロウバスト・レベルを返す
pthread_mutexattr_setprotocol(3)	プロトコルを設定
pthread_mutexattr_setrobust(3)	ロウバスト・レベルを設定

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。

これを実装するには、EOWNERDEADとENOTRECOVERABLEの2つerrnoコードが利用できます。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、その時々により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次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作成します。

pthread_mutex_consistent(3) 矛盾したミューテックスの矛盾をなくす

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_getprotocol(3) プロトコルを返す
pthread_mutexattr_getrobust(3) ロウバスト・レベルを返す

属性の設定：

pthread_mutexattr_setprotocol(3) プロトコルを設定
pthread_mutexattr_setrobust(3) ロウバスト・レベルを設定

pthread_mutex_consistent

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(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_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

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(const pthread_mutexattr_t *attr,
int *robustness)
```

利用可能なレベル：

PTHREAD_MUTEX_ROBUST

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

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」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_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

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(pthread_mutexattr_t *attr, int
robustness)
```

robustness は、PTHREAD_MUTEX_ROBUSTもしくはPTHREAD_MUTEX_STALLEDになります。これらの定義については「pthread_mutexattr_getrobust」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のための拡張部分を提供する代替の**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**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**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はありません。RedHawkの以前のバージョン上で**ccur-gcc**または**ccur-g++**でコンパイルされた既存のバイナリは、現在のRedHawkのバージョン上で再コンパイルおよび/もしくは移植する必要があります。これはロウバストおよび/もしくは優先度継承の機能が使用された場合に特に当てはまります。移植作業では、移植の大半はPスレッド関数や変数名から単に古い接尾辞_**np**や_**NP**の削除だけとなります。

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、いくつか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する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>ヘッダー・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する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>ヘッダー・ファイルを内部的に含む<sys/sem.h>ヘッダー・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>ヘッダー・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>ヘッダー・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(1)**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、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を作成される、もしくは既存のものが見つかる**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をします。
IPC_EXCL	IPC_CREAT と一緒に使用は、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と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(<i>semids</i>)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(1)**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(1)**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カーナビリティを所有

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なも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ー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ーのテンプレートと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>`ヘッダー・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

再スケジューリング変数はARM64アーキテクチャではサポートされていないため、条件同期もまたARM64アーキテクチャではサポートされていないことに注意して下さい。

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には、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트는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* 配列で使用するエントリの数は、*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：

<i>options</i>	この引数の値はゼロである必要があります。
<i>r</i>	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。
<i>timeout</i>	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：

<i>server</i>	起こされるサーバ・プロセスのグローバル・プロセスID
<i>r</i>	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：

<i>servers</i>	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
<i>nservers</i>	配列のエレメント数を指定する整数値
<i>r</i>	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ブロックします。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**dequeue**)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**の機能を使用して、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_glpid);
        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。

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	<p>一般的な400nsのクロック信号のティックを1ず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。</p> <p>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。</p>
POSIX	<p>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ずつインクリメントされます。</p>

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、もしくはコード・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。
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**カ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：

<i>which_clock</i>	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。
<i>setting</i>	<i>which_clock</i> へ設定する時間を指定する構造体へのポインタ。 <i>which_clock</i> 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：

<i>which_clock</i>	時間を取得するクロックの識別子。 <i>which_clock</i> の値は CLOCK_REALTIME または CLOCK_MONOTONIC にすることが可能です。
<i>setting</i>	<i>which_clock</i> 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：

<i>which_clock</i>	分解能を取得するクロックの識別子。 <i>which_clock</i> の値は CLOCK_REALTIME または CLOCK_MONOTONIC にすることが可能です。
<i>resolution</i>	<i>which_clock</i> 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：

<i>which_clock</i>	タイマーに使用されるクロックの識別子。 <i>which_clock</i> の値は CLOCK_REALTIME である必要があります。
<i>timer_event_spec</i>	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： <div> <p>NULL タイマー満了時にSIGALRMがプロセスへ送信されます</p> <p><i>sigev_notify</i>=SIGEV_SIGNAL <i>sigev_signo</i> 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。</p> <p><i>sigev_notify</i>=SIGEV_THREAD 指定した<i>sigev_notify</i> 機能はタイマー満了時に <i>sigev_value</i>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。</p> <p><i>sigev_notify</i>=SIGEV_THREAD_ID <i>sigev_notify_thread_id</i> の番号にはタイマー満了時に <i>sigev_signo</i>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。</p> <p><i>sigev_notify</i>=SIGEV_NONE タイマー満了時に通知は配信されません。</p> </div>

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。

<i>created_timer_id</i>	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：

<i>timer_id</i>	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：

<i>timer_id</i>	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)。
-----------------	---

<i>flags</i>	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 <i>it_value</i> 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 <i>it_value</i> で指定された値に到達する時に満了となります。
<i>new_setting</i>	<p>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。</p> <p>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 (<i>it_interval</i>) 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。</p> <p>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 (<i>it_interval</i>) 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。</p> <p>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 <i>flags</i> 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。</p> <p>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。</p>
<i>old_setting</i>	<p>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。 <i>old_setting</i> 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。</p>
<p>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、<i>errno</i>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。</p>	

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：

<i>timer_id</i>	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)。
<i>setting</i>	繰り返し間隔とタイマーの残り時間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：

<i>timer_id</i>	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：

<i>req</i>	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。 <i>req</i> 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 <i>req</i> 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。
<i>rem</i>	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 <i>rem</i> 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：

<i>which_clock</i>	使用するクロックの識別子。 <i>which_clock</i> の値は CLOCK_REALTIME または CLOCK_MONOTONIC となります。
<i>flags</i>	以下のいずれかを指定する整数値： <div> <div>TIMER_ABSTIME</div> <div><i>rqtp</i> で指定された時間は <i>which_clock</i> 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。</div> <div>0</div> <div><i>rqtp</i> 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。</div> </div>
<i>rqtp</i>	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、 <i>rqtp</i> 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。 <p>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ありません。</p>
<i>rmtp</i>	TIMER_ABSTIME が指定されていない場合、 <i>rmtp</i> 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 <i>rmtp</i> が NULL の場合、残り時間は設定されません。 <i>rmtp</i> 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** は少なくと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。

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`lkernel-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 Real-Time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・クオ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・クオ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・タイマーのシールドリングは、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・シールドリングは非シールドCPUへ重要ではない仕事を移動するプロセス・シールドリングと連動します。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提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。

プロセス実行時間のクォンタムおよび制限

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リング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リング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リング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上にスケジューリング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。シールドイングは **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。

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。ユーザー割り当てお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。

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。

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ー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** コマンドを指定、*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(3)** 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(3)** 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(3)** 変数が返す値の整数倍である必要があります。

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・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。

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に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：

- ファイルは少なくと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が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 Real-Time 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 Real-Time 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 Real-Time 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：

PROC_MEM_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 Real-Time 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。

PROC_MEM_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 Real-Time 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。

Non-Uniform Memory Access (NUMA)

最新のIntelおよびAMDの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。

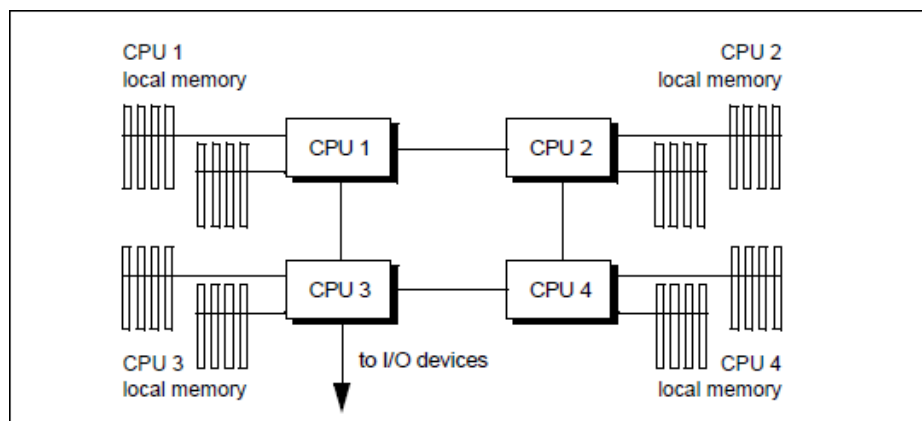
NOTE

NUMAはARM64アーキテクチャには対応していません。本章で説明する機能はNUMA特有であるため、ARM64アーキテクチャではサポートされません。

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またはIntelの最新のマルチ・プロセッサ・システムはNUMAアーキテクチャを搭載しています。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



最新のAMDまたはIntelの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、リアルタイム・アプリケーションのリモート・メモリ・アクセス量を減らすためにプログラムのページが割り当てられる場所からプログラムがノードを指定することを可能にします。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サポートは最新のiHawkシステム上で利用可能です。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CPU 0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CPU 1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

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。

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ド[~]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[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[interleave]=[~]*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ド[~]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には **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ー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。

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。

```
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 0L
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 upto lru act
map dsk
0x402000: pfn: 0x39ece8 node: 0 mapcnt: 3 flags: ref upto lru act
map dsk
0x403000: pfn: 0x36c52e node: 2 mapcnt: 3 flags: ref upto 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

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。

カーネルのNUMAバランシング・サポートを有効にする場合、物理メモリにあるプロセスの仮想アドレス空間の全てまたは一部をロックするために**mlock(2)**, **mlock2(2)**, **mlockall(2)**システムコールおよびRedHawk Linuxの**mlockall_pid(2)**システムコールを利用することが可能であることに注意して下さい。さらにカーネルが自動的にメモリを移動するのを防ぐために**mlock2(2)**, **mlockall(2)**, **mlockall_pid(2)**システムコールへフラグを渡すことも可能です。回避することを意図するフラグとカーネルの動作は**noautomigrate(7)**で文書化されています。詳細はmanページを参照して下さい。

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
```

シールドイングの相互作用

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・シールドイ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。

シールドイングの制限

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リングした時、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、既存の共有テキスト・ページと書き込みデータ・ページ上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、CPU 2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CPU 2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CPU 3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. CPU 2とCPU 3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CPU 2のNUMAノードを使うMPOL_BINDメモリ・ポリシーのCPU 2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CPU 3のNUMAノードを使うMPOL_BINDメモリ・ポリシーのCPU 3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

最新のAMDとIntelプロセッサはNUMAアーキテクチャを搭載しています。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・オプションであることに注意して下さい。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。

最新のAMDまたはIntelのシステム上で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。

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。

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・カーネルでは有効になっていません。

ZRAM_NUMA

有効にした場合、このパラメータは圧縮RAMディスクに関連する全てのメモリを特定のNUMAノードの中に含めることを保証するメカニズムを提供します。詳細については次のファイルを参照して下さい：/usr/src/linux-*RedHawk*/Documentation/blockdev/zram-numa.txt

BLK_DEV_RAM_NUMA

有効にした場合、このパラメータはRAMディスクに関連する全てのメモリを特定のNUMAノード内に含めることを保証するメカニズムを提供します。詳細については次のファイルを参照して下さい：/usr/src/linux-*RedHawk*/Documentation/blockdev/ramdisknuma.txt

カスタム・カーネルの構成および構築

本章ではカスタムRedHawk Linuxカーネルの構成、構築およびインストールの方法に関する情報を提供します。

また、**xconfig**、3rdパーティ・ドライバの構築、動的カーネル・モジュール・サポートを最後に簡単に考察しています。

序文

traceカーネルはデフォルトでインストールされるカーネルで、**Concurrent**は大抵のケースで推奨しています。**standard**カーネルはインストール・メディアからインストールすることが可能です。これはカーネル・トレース機能がないことを除きトレース・カーネルと全く同一です。

カーネルの構成または再構成は、カーネル構成を再定義する処理で、その結果、新しい定義に従い新しいカーネルを生成します。

一般に供給されるカーネルは調整可能なパラメータと殆どのシステムに適しているデバイス・ドライバを伴い生成されます。しかしながら、個別の要求のためにいずれかの調整可能なパラメータを変更、ドライバを追加または削除、もしくはパッチを適用したい場合にカーネルの再構成を選択することが可能です。その場合はカーネルの再構築、インストール、再起動が必要になります。

RedHawkは2つのカーネル(**standard**と**trace**)とカスタム・カーネル・パッケージを構築するための1つのソース・ツリーを提供します。ソース・ツリー内の**configs**ディレクトリはRedHawkのプレビルド・カーネルと一致する2つの構成ファイルを保持します。

カーネル・ソース・パッケージ(**ccur-kernel-source**)はカスタム・カーネルを作成するためだけに必要とされることに注意して下さい。これはデフォルトではインストールされません。これはインストール・メディアで提供されますので、そこからインストールすることが可能です。

カーネル・パッケージの構築手順

本項ではカスタム・カーネル・パッケージを構成、構築、インストールする手順について説明します。

以降の手順は次に示す順番で実行する必要があります。手順はまず**Rocky**と同種のシステム向け、続いて**Ubuntu**が動作中のシステム向けに提供します。手順は非常に似ていますが、コマンドやファイル名称が異なります。

xconfigの実行とカスタム・カーネル・パッケージの作成にはパッケージ依存があることに注意して下さい。依存のリストはRelease Notesの「既存の問題」項で見つけることが可能です。後続のコマンドが正常に終了した場合、警告は安全に無視することが可能です。

Rocky互換システムの手順

カーネル・ソース・パッケージをインストールした際、次の手順内で言及される *kern-name* は `/usr/src` ツリー以下に生成されるディレクトリで使用する名称で引用します。名称の例が **linux-6.12.33-RedHawk-9.6** である場合、6.12.33 は kernel.org のバージョンで 9.6 は RedHawk のバージョンとなります。従って、このディレクトリ名称は RedHawk のリリース毎に変化します。

kern-id はパッケージが生成された日付を含んでいます。

Rocky または Rocky 互換のいずれかのシステム上でカスタム・カーネル・パッケージを構築するには、次の手順に従ってください：

1. **ccur-kernel-source** パッケージがシステムにインストールされていることを確認して下さい。ない場合、パッケージは RedHawk のインストール・ディスク、アップデート・ディスク、または RedHawk のネットワーク・リポジトリからインストールすることが可能です。

```
rpm -q ccur-kernel-source
```

2. RedHawk Release Notes の「既知の問題」項にある「カスタム・カーネル構築に必要な追加パッケージ」を探して下さい。必要なパッケージが表示されたら、それらをすぐにインストールして下さい。入力を指示されたら次の手順へ移りそれらをインストールすることを選択することも可能です。

```
dnf install packages
```

3. ソース・ディレクトリにディレクトリを変更して下さい。

```
cd /usr/src/linux-kern-name-source
```

4. **configs** ディレクトリ内にある構成ファイルの1つから **.config** ファイルを生成して下さい。この構成ファイルはカスタム・カーネルを構築するために変更するベースとなるカーネル構成となります。選択肢は **standard** もしくは **trace** です。

```
cp configs/x86_64/configfile .config
```

5. 構成を変更するには次のコマンドのいずれかを実行して下さい。GUI の使用に関するヘルプは「**xconfig**」項を参照して下さい。

```
make menuconfig (テキスト・ベースのメニュー) もしくは  
make xconfig (グラフィック)
```

6. 次を実行してパッケージを作成して下さい：

```
make binrpm-pkg
```

7. カーネル・パッケージがあるディレクトリへ移動してインストールして下さい：

```
cd /usr/src/linux-kern-name-source/rpmbuild/RPMS/x86_64  
sudo dnf install ccur-kernel-custom-kern-id.x86_64.rpm
```

オプションで次のカーネル・パッケージもインストールすることが可能です：

```
ccur-kernel-custom-devel-kern-id.x86_64.rpm  
(サード・パーティ・ドライバの構築が必要)
```

ccur-kernel-custom-debuginfo-kern-id.x86_64.rpm
(クラッシュ・ダンプの調査が必要)

8. カスタム・カーネルをデフォルト・カーネルとして起動させることが望ましい場合、デフォルトとして構成するために**blscfg**の**-d**オプションを使用して下さい。*kern-id-no*を見つけるにはオプションなしでコマンドを実行して下さい。その後、再起動して下さい。

blscfg -d kern-id-no
reboot

Ubuntuベース・システムの手順

カーネル・ソース・パッケージをインストールした際、次の手順内で言及される*kern-dirname*は**/usr/src**ツリー以下に生成されるディレクトリで使用される名称で引用します。名称の例が**linux-6.12.33-RedHawk-9.6**である場合、6.12.33はkernel.orgのバージョンで9.6がRedHawkのバージョンとなります。従って、このディレクトリ名称はRedHawkのリリース毎に変化します。

Ubuntuシステムで使用される*kern-name*は*kern-dirname*と同じになりますが、全て小文字になります。例：**linux-6.12.33-redhawk-9.6**

*kern-id*はパッケージが生成された日付を含んでいます。

Ubuntuオペレーティング・システムを使用している場合にカスタム・カーネル・パッケージを構築するには、次の手順に従って下さい：

1. **ccur-kernel-source**パッケージがシステムにインストールされていることを確認して下さい。システムにない場合、パッケージはRedHawkのインストール・ディスク、アップデート・ディスク、またはRedHawkのネットワーク・リポジトリからインストールすることが可能です。

apt list ccur-kernel-source

2. RedHawk Release Notesの「既知の問題」項にある「カスタム・カーネル構築に必要な追加パッケージ」を探して下さい。必要なパッケージが表示されたら、それらをすぐにインストールして下さい。入力を指示されたら次の手順へ移りそれらをインストールすることを選択することも可能です。

apt install packages

3. ソース・ディレクトリにディレクトリを変更して下さい。

cd /usr/src/linux-kern-dirname

4. **configs**ディレクトリ内にある構成ファイルの1つから**.config**ファイルを生成して下さい。この構成ファイルはカスタム・カーネルを構築するために変更するベースとなるカーネル構成となります。選択肢は**standard**もしくは**trace**です。

cp configs/x86_64/configfile .config

5. 構成を変更するには次のコマンドのいずれかを実行して下さい。GUIの使用に関するヘルプは「**xconfig**」項を参照して下さい。

make menuconfig (テキスト・ベースのメニュー) もしくは
make xconfig (グラフィック)

6. 次を実行してパッケージを作成して下さい：

make bindeb-pkg

7. カーネル・パッケージがあるディレクトリへ移動してインストールして下さい：

```
cd /usr/src/linux-kern-dirname/debian-pkgs  
apt install ccur-linux- image-kern-name-custom_kern-id_amd64.deb
```

オプションで次のカーネル・パッケージもインストールすることが可能です：

```
ccur-linux-headers-kern-name-custom_kern-id_amd64.deb  
(サード・パーティ・ドライバの構築が必要)
```

```
ccur-linux-headers-kern-name-custom-dbg_kern-id_amd64.deb  
(クラッシュ・ダンプの調査が必要)
```

8. カスタム・カーネルをデフォルト・カーネルとして起動させることが望ましい場合、デフォルトとして構成するために**ccur-grub2**の**-d**オプションを使用して下さい。*kern-id-no*を見つけるにはオプションなしでコマンドを実行して下さい。その後、再起動して下さい。

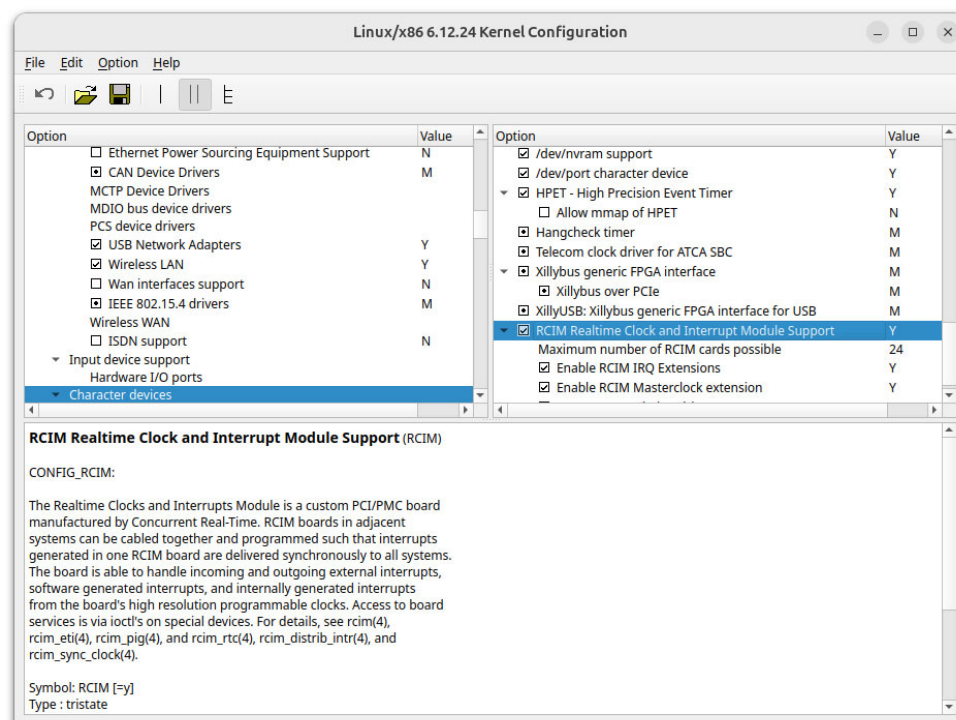
```
ccur-grub2 -d kern-id-no  
reboot
```

xconfig

カスタム・カーネルの構築手順の1つの中で、カーネルの構成を変更するオプションが提供され、RedHawk Linuxカーネルの多くの異なる側面をカスタマイズすることが可能な**xconfig**という名称のグラフィカル構成インターフェースが現れます。

カーネル構成GUIの例については下のイメージを参照して下さい。

図11-1 カーネル構成GUI



「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。

「Save」がグレー表示の場合、「Save As」オプションを使用して.configファイルの上書きを選択することが可能です。

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。

3rdパーティ・ドライバ・モジュールの構築

3rdパーティ・ドライバ・モジュールを構築するには、システムにたった1つのカーネル・パッケージをインストールする必要があります：Rocky互換システムには**ccur-kernel-trace-devel-kern-id.x86_64**、Ubuntuシステムには**ccur-kernel-headers-trace-kern-id_amd64**です。

3rdパーティ・ドライバを構築するには、適切なカーネル・パッケージが前述のとおりシステムにインストールされていることを確認し、ベンダーの指示に従ってください。

動的カーネル・モジュール・サポート

動的カーネル・モジュール・サポート(DKMS: Dynamic Kernel Module Support)は、いつカーネルがアップグレードされてもモジュールの構築、インストール、更新の処理を自動化します。インストールされていない場合、**dkms**パッケージはRockyについてはEPELリポジトリから、Ubuntuについては標準リポジトリからインストールすることが可能です。

将来的には、RedHawkで生成される全てのRedHawkの3rdパーティ・ドライバ・ディスクはDKMSのサポートを含むようになります。(DKMS構成ファイルを提供する)DKMSをサポートする全ての3rdパーティ・ドライバは、新しいカーネルがインストールされる時に構築、インストールされます。

利用可能なオプションや**dkms.config**構成ファイルの書式に関する更なる情報については**dkms**のmanページを参考にすることが可能です。DKMSプロジェクトは<https://github.com/dell/dkms>にあります。

12 カーネル・デバッグング

本章はRedHawk Linuxが提供するカーネル・デバッグングやクラッシュ・ダンプ解析のためのツールについて説明します。

概要

標準的な**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。カーネル・バージョンと一致する**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。

NOTE

本章内で説明する標準的な**kexec-tools**ユーティリティはARM64アーキテクチャではサポートされません。1つの例外が実行中のシステムを解析するために使用可能な**crash**ツールです。後述の「実行中システムの解析」を参照して下さい。

クラッシュ・ダンプを調査するには**ccur-kernel-trace-debuginfo**(Ubuntuでは**ccur-kernel-trace-dbg**)のカーネル・パッケージがシステムにインストールされている必要があります。もしシステム上にない場合、RedHawkインストール・ディスクかアップデート・ディスクまたはRedHawkのネットワーク・リポジトリからインストールが可能です。

VMcore生成イベント

kdumpサポートが構成され有効化されている場合、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」セクションを参照して下さい。
- **/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/sysrq-trigger
```

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構成

RedHawkカーネルはデフォルトで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せん。メモリを予約するため、**crashkernel=...**の**grub**起動オプションをカーネル起動オプションに付け加える必要があります。

殆どのシステムで**grub**オプション**crashkernel=256M**は動作します。しかしながら、一部のシステムにおいてこれは無駄である可能性がある一方、他のシステムではそれが十分ではなく**kdump**の操作に失敗する可能性があります。

次の表は、システムのRAMサイズに応じて予約する推奨の**crashkernel**メモリを示します。

表12-1 推奨のクラッシュ・カーネル・サイズ

RAM (GB)	<i>crashkernel</i> メモリ (MB)
1 - 4	192
4 - 64	256
64 -	512

推奨の**crashkernel**メモリ・サイズは十分ではない可能性があります。確認する唯一の方法は、それを試すことです。十分ではない場合、更にメモリを予約して再度試して下さい。

次のコマンドは**crashkernel**用にメモリが予約されたことを確認するために使用することが可能です：

```
dmesg | grep Reserving
```

crashkernel=...の**grub**オプションは次のように追加することが可能です：

1. **/etc/default/grub** ファイルの **GRUB_CMDLINE_LINUX** 行に **crashkernel=...** 行を全体的に付加。これはシステムインストールされた全てのカーネルに作用し、本オプションはカーネルrpmが更新された場合でも存続します。
2. 特に**blscfg(1)** コマンド経由。 **crashkernel=...** オプションを1つ以上のカーネルに追加することが可能です。本grubオプションは更新用のカーネルrpmをシステムにインストールする際に上書きされることに注意して下さい。

変更を有効にするにはこの後にシステムを再起動する必要があります。

パニックまたは他のクラッシュ状態に直面しているカーネル用にgrubオプションが指定された場合、**vmcore**クラッシュ・ファイルは**/var/crash**以下に収納されます。

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(8)**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(8)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(8)**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、次の項目を修正して下さい：
 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.xの`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.xベースのシステムにすることを推奨します。RedHawk Linux 7.x0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`ファイルを作成する代わりに、リモート**`kdump`**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,rsiz=8192,wsiz=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。

次の黒丸項目内の**kdumpctl**または**systemctl**コマンドを発行する前にサーバー・システム上でこの対象となる**crash**ディレクトリを生成する必要があります。例えば、サーバー・システムにおいてrootで次のコマンドを発行します：

```
# mkdir -p /kdump/var/crash
```

- 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.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割り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。

NOTE

クラッシュ・ダンプを調査するには**ccur-kernel-trace-debuginfo**(Ubuntuでは**ccur-kernel-trace-dbg**)のカーネル・パッケージがシステムにインストールされている必要があります。もしシステム上にない場合、RedHawkインストール・ディスクかアップデート・ディスクまたはRedHawkのネットワーク・リポジトリからインストールが可能です。

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、少なくとも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12-2ページ「VMcore構成」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(8)**のmanページを見るには**man -M /usr/ccur/man crash**を実行して下さい。

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**のいずれかです。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`/etc/grub2.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(`/etc/grub2.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。

Pluggable Authentication Modules(PAM)は認証とセキュリティ用のフレームワークを提供します。ベースLinuxディストリビューションはPAMモジュール一式を提供する一方、RedHawkオペレーティング・システムはケーパビリティ・モジュールの独自バージョンをインストールします。

本章ではRedHawkのケーパビリティ・モジュールを使ってケーパビリティを割り当てる構成ファイルおよび手順を取り扱います。

序文

Pluggable Authentication Modulesを表すPAMは、認証プログラムの再コンパイルを必要としない認証ポリシーの設定方法を提供します。

モジュールはいつでも生成または追加して利用することが可能です。PAMを使用するプログラムは再コンパイルを必要とせずに新しいモジュールを即時使用することが可能です。

認証が必要なプログラムはそのサービス名称を定義し、**/etc/pam.d**ディレクトリに自身のPAM構成ファイルをインストールする責務があります。

PAMはベースLinuxオペレーティング・システムの一部としてインストールされます。RedHawkオペレーティング・システムは、独自の**pam_capability.so**モジュールと独自の**capability.conf**ファイルを**pam_capability(8)**と**capability.conf(5)**のmanページと共にインストールします。結果として、ケーパビリティに関するInternet上の説明や情報は当てにできないことに注意して下さい。RedHawkは独自のよりシンプルで使い易いケーパビリティの実現を提供します。

RedHawkバージョンは**exec()**システムコール全体でケーパビリティを継承するためにカーネルも変更しています。本機能はカーネル構成GUIの「General Setup」以下でアクセス可能な**INHERIT_CAPS_ACROSS_EXEC**構成オプションを介して利用可能です。全てのRedHawk Linuxカーネルはデフォルトで本オプションが有効化されていることに注意して下さい。

本章はPAMサービス・ファイルおよび**capability.conf**ファイルへの必要な変更を取り扱います。最後にリアルタイム・ケーパビリティを持つリアルタイム・ユーザーに割り当てる例を章の最後で提供します。

PAMサービス・ファイル

/etc/pam.d以下の各ファイルはユーザーがシステムにログインするために利用可能なサービスに対応しています。各アプリケーションまたはサービスは対応するファイルを持っており、大抵は対応するサービスと同じような名前になっています。例えば、**login**ファイルは**login**サービスを定義し、**sshd**ファイルは**ssh**サービスを定義します。

ケーパビリティをユーザーに割り当てる場合、**/etc/pam.d**以下のファイルを変更する必要があります。ユーザーが変更されていないサービスを使用してシステムにログインする場合、特別なケーパビリティの割り当ては行われません。

PAM構成ファイル

このファイルについてここでは簡単に説明します。詳細については**pam.d(5)**のmanページを参照して下さい。

PAM構成ファイル内の行の項目は次の書式を使用します：

```
module-types module-control module-path module-arguments
```

PAM標準で定義される4種類の**module-types** が存在します：

<i>auth</i>	例えばパスワードの要求およびチェックによりユーザーのIDを確認します。
<i>account</i>	アカウントの期限切れやログイン制限時間等に関するチェックのような非認証ベースのアカウント管理を実行します。
<i>password</i>	パスワードを更新するために使用します。
<i>session</i>	セッションの開始および終了時に実行する動作(例えば、ユーザーのホーム・ディレクトリをマウント)を定義します。

module-control フラグはサービスに対するユーザーの認証を全体的な目的とする成功もしくは失敗の重要性を決定します。

module-path は"/で始まる場合はPAMモジュールの完全なファイル名称、そうではない場合はデフォルト(**/lib64/security**)のモジュール位置からの相対パスのいずれかです。

module-arguments は任意のPAMの特定の動作を変更するために使用可能なスペース区切りのトークンのリストです。次のRedHawkの**pam_capability.so**モジュールについては同じ行に指定することが可能です。

conf=conf_file	構成ファイルの位置を指定します。本オプションが指定されていない場合はデフォルト位置は /etc/security/capability.conf となります。
debug	syslog を介してデバッグ情報を記録します。デバッグ情報は syslog authpriv クラスに記録されます。通常、本ログ情報は /var/log/secure ファイルに集約されます。

NOTE

ケーパビリティが使用されている場合、**/etc/pam.d/su**ファイルは**su -l nobody daemon**のような呼び出しが**nobody**ユーザーにリストされたケーパビリティだけを**daemon**に与え、呼び出しユーザーからは余分なケーパビリティを与えないことを確実に行うセキュリティ措置として変更される必要があります。

/etc/security/capability.confファイルに定義されたロールをシステムにログインするユーザーに割り当てるには、次の行を**/etc/pam.d/sshd**にディレクトリ以下の各々の適切なPAM対応アプリケーションに追加する必要があります：

```
session required pam_capability.so
```

ユーザーがシステムに入ることが可能な方法は沢山ありますので、いくつかのサービスは通常変更する必要があります。以下は通常使用されるPAM対応ファイルのリストの一部です：

- **gdm-passwd**: GNOMEディスプレイ・マネージャーにログイン。本機能は最近のLinuxディストリビューションではもう動作しません：Red Hat互換ディストリビューションでは8.2以降、Ubuntuでは20.04以降。下のNOTEを参照して下さい。
- **login**: システムにログイン(非グラフィカル・ログイン)。
- **sshd**: リモートからログイン。
- **su**および**sudo**: あるユーザーから他のユーザーへ切り替え。

ご使用のアプリケーション次第で他も必要となる可能性があります。例えば、VNC (Virtual Network Computing)を実行するには次も必要です：

- **runuser-l**: 代わりのユーザーとグループIDでコマンドを実行し、ログイン・シェルとして(-lオプションで)シェルを開始。

NOTE

systemdとXサーバーのセキュリティ変更に起因して、最近のどのLinuxディストリビューションに含まれるグラフィカル**login**においてもはやケーパビリティは自動的に許可されません。ユーザーは**ssh**またはグラフィックでログインを利用するか、ケーパビリティなしで再度同じユーザーになるために**su**を利用のどちらかが可能です。

セッション行をユーザーが必要とする各PAM対応アプリケーションに追加すること、または通常使用されるPAM対応アプリケーションの殆どに含まれる重要なPAM対応アプリケーションにそれを追加することが可能です。本章最後の実例は後者の方式を使用しています。13-5ページの「実例：リアルタイム・ユーザー向けPAMケーパビリティの構成」を参照して下さい。

デフォルトのケーパビリティ構成ファイルの代わりにセッション行に他のファイルを指定することが可能です。本例では、**/etc/security/capability.conf**の代わりに**/root/ssh-capability.conf**構成ファイルを使用しています。

```
session required pam_capability.so \
conf=/root/ssh-capability.conf
```

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。
/etc/security/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。

capability.conf(5)ファイルは3種類のエントリ(ロール、ユーザー、グループ)を認識します。

ロール

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ケーパビリティ一式
 は**/usr/include/linux/capability.h**カーネル・ヘッダー・ファイルの中、または
`_cap_names[]`文字配列を使うことで見る事が可能です。これらは付録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：

<code>all</code>	(<code>cap_setcap</code> を除く)全ケーパビリティ
<code>cap_fs_mask</code>	ファイルシステムに関する全ケーパビリティ
<code>none</code>	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がある職務に
 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
```

必要最小限のケーパビリティはアプリケーション次第です。最小の設定を見つけ出すのはテ
 ストと時間が必要になります。1つの戦略は全てのケーパビリティを削除してから必要に応じ
 てそれらを戻すことです。

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
 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。

次の例ではrootとほぼ同等の管理用ロール'admin'を設定します：

```
role      admin      all
```

本例ではリアルタイム・ユーザー・ロール'rtuser'を設定します：

```
role      rtuser      cap_ipc_lock \
                        cap_sys_rawio \
                        cap_sys_admin \
                        cap_sys_nice \
                        cap_sys_resource
```

ロールを定義したら、**capability.conf**ファイルでユーザーまたはグループに割り当てる事が可能です。

グループ

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      groupnam      rolename
```

例えば、poweruserロールをグループに割り当てるには、**capability.conf**ファイルのGROUPSセクションに次を入力して下さい：

```
group      hackers      poweruser
```

ユーザー

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、リストにあるユーザーと一致しないユーザーまたはリストにあるグループのメンバーシップを持つユーザーにデフォルトのロールを割り当てる事が可能です：

```
user       *              default_rolename
```

例えばユーザー'joe'にdesktopuserロールを割り当てるには、**capability.conf**ファイルのUSERSセクションに次を入力して下さい：

```
user       joe            desktopuser
```

実例：リアルタイム・ユーザー向けPAMケーパビリティの構成

本例では、通常使用されるPAMサービスのグループにケーパビリティ・サポートを割り当てる方法を示します。リアルタイム用ケーパビリティをロールに定義してそのロールをユーザーに割り当てます。

通常使用されるサービスの割り当て

ケーパビリティ・モジュールを持つセッション行は、ユーザーがシステムにログインする必要がある各PAM対応アプリケーションに追加する必要があります。各サービス・ファイルに個々にまたは通常使用されるサービスのグループとして追加することが可能です。

/etc/pam.dディレクトリ内の主要なファイルは通常使用されるサービスに含まれているので、これらのサービスにセッション行を追加すると実質的にこれらの2つのファイルを含む全ての通常使用されるサービスに追加したことになります。そのファイルはUbuntuオペレーティング・システムとRHELベース・ディストリビューションとは異なります。

Ubuntuの主要ファイルは次のとおり：

```
common-session
common-session-noninteractive
```

RHELベース・ディストリビューションの主要ファイルは次のとおり：

```
system-auth
passwd-auth
```

これらの主要ファイルを使用するには、最初に更新の場合に備えてそれらを保存するため次のとおりそれらをコピーおよびリンクして下さい。keyf1とkeyf2を上述の適切なディストリビューション固有名称に置き換えて下さい。

```
cd /etc/pam.d
cp ${keyf1} ${keyf1}-local
cp ${keyf2} ${keyf2}-local
ln -sf ${keyf1}-local ${keyf1}
ln -sf ${keyf2}-local ${keyf2}
```

次のセッション行は、上述の手順で生成・リンクされたファイルの**-local**バージョンの両方に追加することが可能です。

```
session required pam_capability.so
```

VNC (Virtual Network Computing)を利用したいユーザーは更に**runuser-l**サービスに上記セッション行を追加する必要があることに注意して下さい。

リアルタイム・ロールの割り当て

本例では'rtuser'ロールにリアルタイムで通常必要となるケーパビリティが割り当てられます。全てのケーパビリティのリストは**/usr/include/linux/capability.h**の中にあります。

殆どのケースにおいて、本例で使用されるケーパビリティで十分事足ります。しかしながら、多かれ少なかれケーパビリティはアプリケーション次第で必要となる可能性があります。

/etc/security/capability.confファイルのROLESセクションに次のような行を追加して下さい。

```
role      rtuser      cap_ipc_lock \
          cap_sys_rawio \
```



```
cap_sys_admin \
cap_sys_nice \
cap_sys_resource
```

リアルタイム・ユーザーの割り当て

リアルタイム・ロールを定義した後、ユーザーをそのロールに割り当てるのが可能です。次の例ではリアルタイム・ユーザー'joe'と'ami'が/etc/security/capability.confファイルのUSERSセクションに追加されています：

```
user      joe      rtuser
user      ami      rtuser
```

リアルタイム・ケーパビリティの確認

上記手順に従ったら、変更されたPAMサービスの1つ(このケースは**su**)を介してシステムにログインし/**proc/self/status**ファイルを観察することでケーパビリティが承諾されたことを確認することが可能です。

例：

```
$ cat /proc/self/status | grep Cap
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
CapBnd: 0000003fffffffff
CapAmb: 0000000000000000

$ su - ami
$ cat /proc/self/status | grep Cap
CapInh: 0000000001a24000
CapPrm: 0000000001a24000
CapEff: 0000000001a24000
CapBnd: 0000003fffffffff
CapAmb: 0000000000000000
```

capsh(1)コマンドはケーパビリティを出力するまたは/**proc/self/status**で示される数字をデコードするために利用することが可能です：

```
$ capsh --print | grep Current
Current:=cap_ipc_lock,cap_sys_rawio,cap_sys_admin,
cap_sys_nice,cap_sys_resource+eip

$ capsh --decode=1a24000
0x0000000001a24000=cap_ipc_lock,cap_sys_rawio,cap_sys_admin,
cap_sys_nice,cap_sys_resource
```


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バスは、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 <ccur/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/ccur/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/ccur/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 <ccur/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 <ccur/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 <ccur/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、この呼び出しの後にはもう使用することはできません。

概要

```
#include <ccur/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 <ccur/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 <ccur/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 <ccur/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ー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(`rtc0`)を使用します。`rtc0`は、Concurrent Real-Time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1 μ 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.c**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・デバイス・ドライバの開発

後に続くセクションで、カーネル・レベル・デバイス・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「カスタム・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「RedHawkの起動コマンド・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。直接機能呼び出す代わりに、その後に機能が呼び出されるように「トリガー」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)

ソフトIRQとタスクレット

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のスケジューリング優先度は、grub行のブート・オプション「softirq.pri=」を使って変更することが可能です。リアルタイム・システムでは、デフォルトの優先度は高い値が設定されており変更すべきではありません。これはリアルタイムに最適化されたシステムではデーモンは全てのsoftirq処理を実行するためです。非リアルタイムのシステムではそうではなくデフォルトでゼロが設定されています。

ワーク・キュー

「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。

優先度の理解

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。Isuio(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/urio0, /dev/urio1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/urio/urioX 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本章では扱いませんが、ヘルプは<https://www.kernel.org/doc/html/latest/driver-api/urio-howto.html>で見ることが可能です。

Concurrent Real-Time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/urio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 Real-Time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NightTraceの資料を参照して下さい。

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。

NOTE

PCI-to-VMEバス・ブリッジはARM64アーキテクチャではサポートされていません。

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 Real-Time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ます、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이지ー・チェーン、バス・グラン
ト、未使用カード周辺の割り込み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コンピューター・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・ケーブルのそれを取り外します。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なりません。**/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
- /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関数

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)
(次ページに続きます)	
Note: <ol style="list-style-type: none"> 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。 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_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
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: <ol style="list-style-type: none"> 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。 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。

もしアンバインドおよび解放数が可能なバインド・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'パラメータに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**bt**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

<code>btd</code>	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。
<code>phys_addr_p</code>	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 ヘッダー・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・マッピングの生成

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)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`(「VMEバス・マッピング」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：

- デバッグング
- 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を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ヘ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ヘ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・ドライバで使用する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へ出力するバイト数。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。

PRTカーネル・オプション

本章ではRedHawkシステムで利用可能なPRTカーネル・オプションについて説明します。

NOTE

RedHawk PRTカーネルはどのRedHawkのメジャー・バージョンの初期リリースでは利用できませんが、その後の最初のRedHawkのマイナー・アップデートに含まれます。従って、例えばRedHawk 7.3に関してはRedHawk PRTカーネルはRedHawk 7.3.1に含まれます。

PRTとは？

RedHawk 7.3は、既定のRedHawk標準カーネル、RedHawkトレース・カーネル、RedHawkデバッグ・カーネルに加えて3つの新しい”PRT”カーネル・オプションが利用可能となります。PRTカーネルはRedHawkの通常のリアルタイム機能全てを含みますが、更にコミュニティに開発されたPREEMPT_RTリアルタイム・セマンティクスも含みます。

PREEMPT_RTの追加は実質的にPRTカーネルのリアルタイム動作が変わり、RedHawkカーネルのシールドリングによるリアルタイム・モデルが最適ではない可能性のある特定のソフト・リアルタイム・タスク(例：単一ソケット・単一コアのシステムまたは数千のスレッドを伴うアプリケーション)にとって相応しいものになる可能性を秘めています。

RedHawk vs PRT

RedHawkカーネルのシールドリングは特定のリソースをリアルタイム活動に分離して専念させるためにユーザーにシステムのリソースを分けるよう要求します(この時適切に調整すると、このアプローチはそのハードウェアが到達可能である最高のリアルタイム性能をもたらします)。しかしながら、このアプローチはプロセスの分離およびシールドに積極的に関与させること、およびどの部分を分離、シールドする必要があるかについて十分に適した決定をする事ができるようにアプリケーションを理解する事もユーザーに要求します。

PRTカーネルの主な目的は、RedHawkのシールドリング用に設計されていないアプリケーションであってもあるレベルのリアルタイム性能に達する事を可能とするために手動のチューニング処置を極力排除する事です。例えば、数百の競合スレッドで構成されているアプリケーションはRedHawkカーネルよりもPRTカーネルのほうがより機能する可能性があります。しかしながら、最高のリアルタイム性能は依然としてRedHawkのシールドリング用に明確に設計されたアプリケーションによって実現されます。

PRTの注意事項

PREEMPT_RTはLinuxカーネルの多くの分野に重要な変更を行っておりますが、恐らくその根本的な変更の殆どはカーネル・スピン・ロックの大部分が完全にプリエンプト可能な状態になっている事です。カーネル・スピン・ロック中にプリエンプションを許可する事は危険な常態となる可能性があり、PREEMPT_RTで適切に動くよう完全に設計されていないデバイス・ドライバでPRTカーネルを使用する場合は注意する必要があります。

PREEMPT_RTは、優先度の最も高い実行可能なプロセスが常にシステムのプロセッサ上で実行されていることを確実にするためLinuxプロセス・スケジューラもまた根本的に変更しています。これを保証するため、スケジューラは実行可能なプロセス一式を絶えず再評価し、利用可能となるCPUへプロセスを瞬時に入れ替える事が必要となります。このスケジューリング動作は結果としてCPU移動が非常に大きな数になりキャッシュ・スラッシングが発生する可能性があり、PRTカーネルで到達可能なリアルタイム性能に制限をかける可能性があります。

一方、これらの注意事項はあるにしても、RedHawkカーネルに対してRedHawkのシールドディンク用に明確に設計されていないアプリケーションがPRTカーネルを使って総合的に勝るリアルタイム性能を達成する可能性があります。

PRTカーネル・フレイバー

以下の3つのPRTカーネル・オプションは既存のRedHawk 7.3のシステム上へのインストールが可能です：

PRT標準	PREEMPT_RTリアルタイム・セマンティクスを含めて修正されたRedHawk標準カーネルのバージョン。このPRT標準カーネルは最も最適化され、PRTカーネルの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。
PRT Trace	PREEMPT_RTリアルタイム・セマンティクスを含めて修正されたRedHawkトレース・カーネルのバージョン。PRTトレース・カーネルは標準的なPRT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します。
PRT Debug	PREEMPT_RTリアルタイム・セマンティクスを含めて修正されたRedHawkデバッグ・カーネルのバージョン。PRTデバッグ・カーネルはPRTトレース・カーネルの全ての機能をサポートし、更に実行時間での検証を含みかつカーネル・レベル・デバッグのサポートを提供します。

Concurrent Real-TimeのNetwork Update Utility (NUU)は、既存のRedHawk 7.3のシステム上にPRTカーネルをダウンロードしてインストールするために使用する事が可能です。あるいは、最新のRedHawk製品アップデートのコピーを要求するためにConcurrent Real-Timeと連絡を取ることも可能です。

追加リソース

本章はPRTカーネルの基本的な紹介のみを提供しますが、最も有名であるReal-TimeLinux Wikiを含む様々なオンライン・リソースがPREEMPT_RT開発およびユーザー・コミュニティのために設けられています。

Real-Time Linux Wiki
<http://rt.wiki.kernel.org>

PREEMPT_RTに関する最新の情報については前述のReal-TimeLinux Wikiのリンクを閲覧、または単純に文字列“preempt_rt”をインターネットで検索してください。

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なので、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 MAXMSG 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 = MAXMSG;
    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 handler(int signo, siginfo_t *info, void *ignored)
{
    psignal(signo, "handler:\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);
}

```

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 Real-Time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。一部の機能はプレビルト・カーネルだけでなく特定のアーキテクチャカーネルに特有であることに注意して下さい。そうである場合、アーキテクチャの仕様が括弧内に記載されます。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(アクティブなハイパーテキスト・リンク)
- 他の適切なConcurrent Real-Time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*/ プラットフォーム	ページ/ 参考文献
シールドCPU				
CPUシールドイング有効	General Setup	SHIELD	Y / all	2-1ページ
CPU停止有効		CPU_DOWNING	Y / all	2-31ページ
再スケジューリング変数	General Setup	RESCHED_VAR	Y / all (x86_64)	5-3ページ
時間管理				
ティックレス・システム有効	Timers Subsystem	NO_HZ	Y / all	H-1ページ
高分解能 プロセス・アカウンティング	General Setup	HRACCT	Y / all	7-2ページ
TSC信頼性	Processor Type and Features	REQUIRE_TSC	Y / all (x86_64)	7-2ページ
RCIMサポート	Device Drivers	RCIM	M / all	RCIM User's Guide (0898007)
POSIXメッセージ・キュー	General Setup	POSIX_MQUEUE	Y / all	3-2ページ
Post/Waitサポート	General Setup	POST_WAIT	Y / all (x86_64)	5-34ページ
exec間でカーネル・ヒラティ継承	General Setup	INHERIT_CAPS_ACROSS_EXEC	Y / all	13-1ページ
プロセス・スケジューリング	Processor Type and Features	SCHED_SMT	Y / all (x86_64)	2-37ページ
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)
Auditing	General Setup	AUDIT	Y / all (x86_64) N / all (ARM64)	RedHawk-FAQ
SBS VMEbus-to-PCI	Device Drivers	SBSVME	M / all (x86_64)	15-1ページ
* 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*/ プレビルトカーネル	ページ/ 参考文献
/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	PCI Support	PROC_PCI_BARMAP	Y / all	14-1 ^{ページ}
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	9-1 ^{ページ}
Interrupt Processing				
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	General Setup	SHMBIND	Y / all	3-16 ^{ページ}
XFSファイルシステム				
XFS有効	File Systems	XFS_FS	M / all (x86_64) N / all (ARM64)	8-1 ^{ページ}
リアルタイム・サブボリューム サポート		XFS_RT	Y / all (x86_64) N / all (ARM64)	
カーネル・プリエンプション	Preemption Model	PREEMPT	Y / all	1-6 ^{ページ}
ptrace拡張	General Setup	PTRACE_EXT	Y / all	1-6 ^{ページ}
システム・ダンプ				
kdumpクラッシュ・ダンプ 有効	Processor Type and Features	KEXEC	Y / all (x86_64)	12-1 ^{ページ}
カーネル・クラッシュ・ダンプ 有効		CRASH_DUMP	Y / kdump (x86_64)	
デバッグ・シンボル生成	Kernel Hacking	DEBUG_INFO	Y / all	
* 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*/ プラットフォーム・カーネル	ページ/ 参考文献
NUMAサポート	Processor Type and Features	NUMA	Y / all (x86_64)	10-1ページ
		AMD_NUMA	Y / all (x86_64)	
		X86_64_ACPI_NUMA	Y / all (x86_64)	
	Memory Management Options	MEMSHIELD_ZONELIST_ORDER	Y / all (x86_64)	
	General Setup	NUMA_BALANCING	Y / all (x86_64)	10-10ページ
	RAM block device support	BLK_DEV_RAM_NUMA	Y / all (x86_64)	
カーネル・トレーシング				
カーネル・トレーシング有効	General Setup	XTRACE	Y / trace, debug; N / generic	n/a
クラッシュ・ダンフから xtraceデータを抽出		XTRACE_CRASH	Y / debug, trace; N / generic (x86_64)	n/a
NVIDIAグラフィックス のサポート	Device Drivers	NVIDIA	M / all (x86_64)	Release Notes (0898003)
UIOサポート	Device Drivers	UIO	M / all (x86_64) N / all (ARM64)	14-14ページ
* 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				

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。Linuxに実装されたケーパビリティの最新のリストや各ケーパビリティが許可する操作または挙動については、<http://man7.org/linux/man-pages/man7/capabilities.7.html> を参照して下さい。

CAP_AUDIT_CONTROL

- 本ケーパビリティは、カーネル監査の有効および無効、監査フィルター・ルールの変更、監査ステータスとフィルタリング・ルールの検索を許可します。

CAP_AUDIT_READ

- 本ケーパビリティは、マルチキャストnetlinkソケットを介した監査ログの読み出しを許可します。

CAP_AUDIT_WRITE

- 本ケーパビリティは、カーネル監査ログへの記録の書き込みを許可します。

CAP_BLOCK_SUSPEND

- 本ケーパビリティは、システムのサスペンドをブロック(**epoll(7)** **EPOLLWAKEUP**, */proc/sys/wake_lock*)することが可能な機能の使用を許可します。

CAP_CHOWN

- 本ケーパビリティは、ファイルのUIDおよびGIDの任意の変更 (**chown(2)**)を参照して下さい)を許可します。

CAP_DAC_OVERRIDE

本ケーパビリティは次を許可します：

- ファイルの読み取り、書き込み、実行許可のチェックの回避。(DACは「Discretionary Access Control」の略称) 詳細については **acl(5)**を参照して下さい。
- 任意のプロセスのマッピングを調査するために**numapgs(1)**の使用。次のケーパビリティも必要となります：CAP_SYS_NICE, CAP_IPC_LOCK, CAP_SYS_PTRACE, CAP_SYS_ADMIN
- 一致するユーザーIDを持つ任意のプロセスが**pagemap(1)**によって提供される利用可能な全てのマッピング情報へのアクセス。一致しないユーザーIDのプロセスに関しては、**pagemap(1)**を参照して下さい。次のケーパビリティも必要となります：CAP_SYS_ADMIN, CAP_SYS_NICE, CAP_SYS_PTRACE

CAP_DAC_READ_SEARCH

本ケーパビリティは次を許可します：

- ファイルの読み取り権限のチェックとディレクトリの読み込み/実行権限のチェックの回避。詳細については**acl(5)**を参照して下さい。
- **open_by_handle_at(2)**の起動。

CAP_FOWNER

本ケーパビリティは次を許可します：

- ファイルのUIDに一致するプロセスのファイルシステムUIDを通常必要となる操作に関する権限のチェックの回避。但し、**CAP_DAC_OVERRIDE**と**CAP_DAC_READ_SEARCH**に該当するこれらの操作は除きます。
- 任意のファイルに関する拡張ファイル属性(**chattr(1)**を参照して下さい)の設定。
- 任意のファイルに関するAccess Control List(ACL)の設定。
- ファイル削除に関するディレクトリのスティッキー・ビットの無視。
- **open(2)**と**fcntl(2)**システム・コールで任意のファイルに対して**O_NOATIME**の指定。

CAP_FSETID

本ケーパビリティは次を許可します：

- ファイル修正時にSet-User-IDとSet-Group-IDのモード・ビットの消去の無効。
- ファイルシステムまたは呼び出し元プロセスの任意の補足GIDに一致しないGIDのファイルに対してSet-Group-IDビットの設定。

CAP_IPC_LOCK

本ケーパビリティは次を許可します：

- **mlock(2), mlock2(2), mlockall(2), mmap(2), shmctl(2)** システム・コールを介したメモリのロック。
- 任意のプロセスのマッピングを調査するために**numapgs(1)**の使用。次のケーパビリティも必要となります：
CAP_DAC_OVERRIDE, CAP_SYS_NICE, CAP_SYS_PTRACE, CAP_SYS_ADMIN

CAP_IPC_OWNER

- 本ケーパビリティは、ユーザーにSystem VのIPCオブジェクト(共有メモリ・セグメント、メッセージ・キュー、セマフォ配列)に関する操作に対して権限のチェックの回避を許可します。**ipcs(1)**を参照して下さい。

CAP_KILL

- 本ケーパビリティは、**ioctl(2)**のKDSIGACCEPT操作の使用を含むシグナルの送信(**kill(2)**を参照して下さい)に対する権限のチェックの無効を許可します。

CAP_LEASE

- 本ケーパビリティは、任意のファイルに関するリースの設置(**fcntl(2)**を参照して下さい)を許可します。

CAP_LINUX_IMMUTABLE

- 本ケーパビリティは、**FS_APPEND_FL**と**FS_IMMUTABLE_FL**のi-nodeフラグの設定(**chattr(2)**を参照して下さい)を許可します。

CAP_MAC_ADMIN

- 本ケーパビリティは、Smack Linux Security Module(LSM)に実装されたMandatory Access Control(MAC)の無効を許可します。

CAP_MAC_OVERRIDE

- 本ケーパビリティは、MACの構成または状態の変更を許可します。Smack LSMに実装されています。

CAP_MKNOD

- 本ケーパビリティは、**mknod(2)**を使用する特別なファイルの生成を許可します。

CAP_NET_ADMIN

本ケーパビリティは、以下のネットワーク管理操作を許可します：

- インターフェース構成
- IPファイヤウォール、マスカレード、アカウントティングの管理
- ルーティング・テーブルの変更
- 任意の透過プロキシ(Transparent Proxy)用アドレスにバインド
- TOS (Type Of Service)の設定
- ドライバの統計情報を消去
- プロミスキャス・モードの設定
- マルチキャストの有効化
- 次のソケット・オプションを設定するために**setsockopt(2)**を使用：**SO_DEBUG, SO_MARK, SO_PRIORITY**(0~6の範囲外の優先度用), **SO_RCVBUFFORCE, SO_SNDBUFFORCE**

CAP_NET_BIND_SERVICE

- 本ケーパビリティは、インターネット・ドメインの特権ポート(ポート番号1024未満)へのソケットのバインドを許可します。

CAP_NET_BROADCAST

- 本ケーパビリティは現在使用されていませんが、ブロードキャスト・ソケットの作成とマルチキャストの接続待機を許可することを目的としています。

CAP_NET_RAW

本ケーパビリティは次を許可します：

- RAWおよびPACKETソケットの利用。
- 透過プロキシの任意のアドレスにバインド。

CAP_SETGID

本ケーパビリティは次を許可します：

- プロセスのGIDおよび補足のGIDリストの任意の操作。
- UNIXドメイン・ソケットを介してソケットの認証情報を渡す時にGIDの模造。
- ユーザー名前空間(**user_namespaces(7)**を参照して下さい)にマッピングしたグループIDの書き込み。

CAP_SETFCAP

- 本ケーパビリティは、ファイルのケーパビリティの設定を許可します。

CAP_SETPCAP

本ケーパビリティは次を許可します：

- ファイルのケーパビリティがサポートされていない場合：任意の他のプロセスとの間で呼び出し元の許可されたケーパビリティのセットに任意のケーパビリティの付与または削除。(CAP_SETPCAPのこの特性はファイルのケーパビリティをサポートするように構成されたカーネルの場合は利用できません。それはCAP_SETPCAPがこのようなカーネルとは全く異なるセマンティクスを持っているためです。)
- ファイルのケーパビリティがサポートされている場合：呼び出し元スレッドのバウンディング・セットから自身の継承セットに任意のケーパビリティの追加；バウンディング・セットから(**prctl(2)** PR_CAPBSET_DROPを介して)ケーパビリティの削除および *securebits* フラグの削除。

CAP_SETUID

本ケーパビリティは次を許可します：

- UNIXドメイン・ソケットを介してソケットの認証情報を渡す時にUIDの模造。
- ユーザー名前空間にマッピングしたユーザーIDの書き込み。
(**namespaces(7)**を参照して下さい)

- プロセスUIDの任意の操作。(setuid(2), setreuid(2), setresuid(2), setfsuid(2))

CAP_SYS_ADMIN

本ケーパビリティは、以下のシステム管理操作を提供します：

- 次を含むある範囲のシステム管理操作の実行：quotactl(2), mount(2), umount(2), swapon(2), setdomainname(2)
- 特権を持つsyslog(2)操作の実行。その操作を許可するためにCAP_SYS_LOGを使用する必要があります。
- VM86_REQUEST_IRQ vm86(2)コマンドの使用。
- 任意のSystem VのIPCオブジェクトに対するIPC_SETとIPC_RMID操作の実行。
- RLIMIT_NPROCリソース制限の無効化。
- trustedとsecurityの拡張属性に対する操作の実行(xattr(7)を参照して下さい)。
- lookup_dcookie(2)の使用。
- IOPRIO_CLASS_RTを指定するためioprio_set(2)の呼び出し。
- UNIXドメイン・ソケットを介してソケットの認証情報を渡す時にPIDの模造。
- ファイルを開くシステムコール(accept(2), execve(2), open(2), pipe(2))において/proc/sys/fs/file-max (システム全体でファイルを開く数の制限)の超過。
- clone(2)やunshare(2)を使って新しい名前空間を生成するCLONE_*フラグの採用。(Linux 3.8以降、ユーザーの名前空間の生成はどのケーパビリティも必要ありません)
- perf_event_open(2)の呼び出し。
- 特権を持つperfイベント情報にアクセス。
- (target名前空間でCAP_SYS_ADMINを必要とする) setns(2)の呼び出し。
- fanotify_init(2)の呼び出し。
- bpf(2)の呼び出し。
- KEYCTL_CHOWNとKEYCTL_SETPERMのkeyctl(2)操作の実行。
- madvise(2)のMADV_HWPOISON操作の実行。
- 呼び出し元制御端末以外の端末の入力キューに文字を挿入するためにTIOCSTIをioctl(2)で使用。
- 廃止されたnfsservctl(2)システムコールを使用。
- 廃止されたbdflush(2)システムコールを使用。
- 様々な特権を持つブロック・デバイスのioctl(2)操作の実行。
- 様々な特権を持つファイルシステムのioctl(2)操作の実行。
- 多くのデバイス・ドライバで管理操作の実行。
- /sys/kernel/debug/rcu/rcudataファイルへの書き込み。
- nvidiaのプリアロケート・ページの構成。
- pw_post(2)の使用。本ケーパビリティがない場合は、実ユーザーまたは実グループIDが互換であれば一部の他のプロセスのpw_post(2)を停止することが可能です。
- nodemaskの変更

- RCIMのファームウェアの変更。
- RCIMの拡張試験の有効化。
- 一部のプロセスのマッピングを調査するために**numapgs(1)**を使用。次に示すカーパビリティも必要となります：
CAP_DAC_OVERRIDE, CAP_IPC_LOCK, CAP_SYS_PTRACE, CAP_SYS_NICE
- 一致するユーザーIDを持つ任意のプロセスに対して**pagemap(1)**で提供される利用可能な全てのマッピング情報へのアクセス。ユーザーIDが一致しないプロセスについては**pagemap(1)**を参照して下さい。次に示すカーパビリティも必要となります：**CAP_SYS_NICE, CAP_DAC_OVERRIDE, CAP_SYS_PTRACE**

CAP_SYS_BOOT

- 本カーパビリティは、**reboot(2)**と**kexec_load(2)**の使用を許可します。

CAP_SYS_CHROOT

- 本カーパビリティは、**chroot(2)**の使用を許可します。

CAP_SYS_MODULE

- 本カーパビリティは、カーネル・モジュールのロードとアンロードを許可します(**init_module(2)**と**delete_module(2)**を参照して下さい)。

CAP_SYS_NICE

本カーパビリティは次を許可します：

- プロセスのナイス値(**nice(2)**, **setpriority(2)**)の引き上げおよび任意のプロセスのナイス値の変更。
- 呼び出し元プロセスに対してリアルタイム・スケジューリング・ポリシーの設定および任意のプロセスに対してスケジューリング・ポリシーや優先度を設定(**sched_setscheduler(2)**, **sched_setparam(2)**, **sched_setattr(2)**)。
- 任意のプロセスに対してCPUアフィニティの設定(**sched_setaffinity(2)**)。
- 任意のプロセスに対してI/Oスケジューリング・クラスや優先度の設定(**ioprio_set(2)**)。
- 任意のプロセスに**migrate_pages(2)**の適用およびプロセスが任意のノードに移動することの許可。
- 任意のプロセスに**move_pages(2)**の適用。
- **mbind(2)**と**move_pages(2)**を使ってMPOL_MF_MOVE_ALLフラグの使用。
- **fbsget(2)**と**fbsconfigure(3)**の使用。
- **/proc/irq*/smp_affinity***ファイルー式に値の書き込み。
- **/proc/shield**以下のファイルの変更。
- **local_irq(2)**の使用。
- 一部の他のプロセスで**mlockall_pid(2)**の使用。
- **procstat(2)**の使用。
- **cpucntl(2)**を介しての変更。
- 任意のプロセスのマッピングを調査するために**numapgs(1)**の使用。次に示すカーパビリティも必要となります：
CAP_DAC_OVERRIDE, CAP_IPC_LOCK, CAP_SYS_PTRACE, CAP_SYS_ADMIN

- 一致するユーザーIDを持つ任意のプロセスに対して**pagemap(1)**で提供される利用可能な全てのマッピング情報へのアクセス。ユーザーIDが一致しないプロセスについては**pagemap(1)**を参照して下さい。次に示すケーパビリティも必要となります：
CAP_DAC_OVERRIDE, CAP_SYS_PTRACE, CAP_SYS_ADMIN

CAP_SYS_PACCT

- 本ケーパビリティは、**acct(2)**の使用を許可します。

CAP_SYS_PTRACE

本ケーパビリティは次を許可します：

- **ptrace(2)**を使用する任意のプロセスのトレース。
- 任意のプロセスに**get_robust_list(2)**の適用。
- **process_vm_readv(2)**および**process_vm_writev(2)**を使用する任意のプロセスのメモリとの間でデータの転送。
- **kcmp(2)**を使用するプロセスの調査。
- 任意のプロセスのマッピングを調査するために**numapgs(1)**の使用。次に示すケーパビリティも必要となります：
CAP_DAC_OVERRIDE, CAP_IPC_LOCK, CAP_SYS_NICE, CAP_SYS_ADMIN
- 一致するユーザーIDを持つ任意のプロセスに対して**pagemap(1)**で提供される利用可能な全てのマッピング情報へのアクセス。ユーザーIDが一致しないプロセスについては**pagemap(1)**を参照して下さい。次に示すケーパビリティも必要となります：
CAP_DAC_OVERRIDE, CAP_SYS_PTRACE, CAP_SYS_NICE
- 全てのRedHawk拡張に**ptrace(2)**の使用。

CAP_SYS_RAWIO

本ケーパビリティは次を許可します：

- I/Oポートの操作(**iopl(2)**および**ioperm(2)**)。
- **/proc/kcore**へのアクセス。
- **FIBMAP**を**ioctl(2)**操作で使用。
- x86モデル固有のレジスタ(MSR、**msr(4)**を参照して下さい)にアクセスするためにデバイスのオープン。
- **/proc/sys/vm/mmap_min_addr**の更新。
- **/proc/sys/vm/mmap_min_addr**で指定された値以下のアドレスにメモリ・マッピングの生成。
- **/proc/bus/pci**にファイルのマッピング。
- **/dev/mem**と**/dev/kmem**のオープン。
- 様々なSCSIデバイス・コマンドの実行。
- **hpsa(4)**および**cciss(4)**デバイスに関するいくつかの操作の実行。
- 他のデバイスにおいてある範囲のデバイス固有の操作の実行。
- 他のプロセスのアドレス空間の一部の**mmap(2)**
- 再スケジューリング変数の生成。
- **shmbind(2)**の使用
- 一部の他のプロセスで**mlockall_pid(2)**の使用。

CAP_SYS_RESOURCE

本ケーパビリティは次を許可します：

- ext2ファイルシステムで予約済み空間の使用。
- ext3ジャーナリングを制御する**ioctl(2)**呼び出しの実行。
- ディスク・クォータ制限の無効化。
- リソース制限の無効化(**setrlimit(2)**を参照して下さい)。
- **RLIMIT_NPROC**リソース制限の無効化。
- コンソールの割り当てでコンソールの最大数の無効化。
- キーマップの最大数の無効化。
- リアルタイム・クロックから64Hzを超える割り込み；
/proc/sys/kernel/msgmnb(**msgop(2)**および**msgctl(2)**を参照して下さい)の制限を超えるSystem Vメッセージ・キューに対して**msg_qbytes**制限の引き上げ。
- **F_SETPIPE_SZ**を**fcntl(2)**コマンドを使ってパイプの能力を設定する時に/proc/sys/fs/pipe-max-size制限の無効化。
- /proc/sys/fs/pipe-max-sizeで指定された制限を超えてパイプの能力を向上させるために**F_SETPIPE_SZ**の使用。
- POSIXメッセージ・キュー生成時に/proc/sys/fs/mqueue/queues_max制限の無効化(**mq_overview(2)**を参照して下さい)。
- **prctl(2)**で**PR_SET_MM**の操作。
- **CAP_SYS_RESOURCE**付きプロセスが最後に設定した値よりも低い値を/proc/PID/oom_score_adjに設定。

CAP_SYS_TIME

本ケーパビリティは次を許可します：

- システム・クロック (**settimeofday(2)**, **stime(2)**, **adjtimex(2)**)およびリアルタイム(ハードウェア)・クロックの設定。
- 任意の/proc/masterclockファイルへの書き込み。
- Pulse-Per-Second(PPS)サポートへの変更。

CAP_SYS_TTY_CONFIG

本ケーパビリティは次を許可します：

- 本ケーパビリティは、**vhangup(2)**の使用および仮想端末上で様々な特権を持つ**ioctl(2)**操作の使用を許可します。

CAP_SYSLOG

本ケーパビリティは次を許可します：

- 特権を持つ**syslog(2)**の操作。操作で必要となる特権に関する情報については**syslog(2)**を参照して下さい。
- /proc/sys/kernel/kptr_restrictの値が1の時に/procおよび他のインターフェースを介して公開されたカーネルのアドレスの参照(**proc(5)**にある**kptr_restrict**の解説を参照して下さい)。

CAP_WAKE_ALARM

本ケーパビリティは次を許可します：

- 本ケーパビリティは、システムを起こす何かの起動を許可します(**CLOCK_REALTIME_ALARM**の設定および**CLOCK_BOOTTIME_ALARM**タイマー)。

NOTE

序文

- 唯一64bitのデバイス・ドライバだけは、64bitオペレーティング・システムで使うことが可能です。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。

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。ヘッダー/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ります。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__` または `__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

ANSIデータ型	サイズ(Byte)
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
<code>intptr_t</code>	ポインタを格納するための符号付き整数型
<code>uintptr_t</code>	ポインタを格納するための符号なし整数型
<code>ptrdiff_t</code>	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
<code>size_t</code>	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
<code>ssize_t</code>	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
<code>int64_t</code>	64-bit符号付き整数
<code>uint64_t</code>	64-bit符号なし整数
<code>int32_t</code>	32-bit符号付き整数
<code>uint32_t</code>	32-bit符号なし整数
<code>int16_t</code>	16-bit符号付き整数
<code>uint16_t</code>	16-bit符号なし整数
<code>int8_t</code>	8-bit符号付き整数
<code>uint8_t</code>	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`~0x7fffffff`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	可変引数を使用されている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。

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シールドイ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CPU 0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のスケジューリング優先度は、grub行ブート・オプション「softirq.pri=」を使って変更することが可能です。リアルタイム・システムでは、デフォルトの優先度は高い値が設定されており変更すべきではないことに留意して下さい。それはリアルタイムに最適化されたシステムでは、そのデーモンは全てのsoftirqの処理を実行するためです。非リアルタイム・システムではそうではなく、デフォルトでゼロに設定されています。

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。

aiol/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・システムで使用するワーク・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)で使用するワーク・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
kworker/*
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。

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。

グラフィクス割り込み

グラフィクスまたはCUDAベースのアプリケーション実行中は様々なプロセッサ間割り込みが発生します。

NVIDIAグラフィクス・ドライバは、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メモリ・バッファを割り当ておよび設定をします。いつバッファが割り当てもしくは解放されようとも、プロセッサ間割り込みはTranslation Lookaside Buffer (TLB)キャッシュ内のフラッシュを動作させます。これらのプロセッサ間割り込みはシールドCPU上で実行中のデターミニスティック・アプリケーションに影響を及ぼす可能性があります。これらの影響は次の動作中に発生する可能性があります：

- XサーバーやVNCサーバー等を含むグラフィカル・アプリケーションの開始または停止
- CUDAアプリケーションの開始または停止
- 非グラフィカルTTYからグラフィカルTTYへの切り替え

プロセッサ間割り込みは、事前に割り当てられたキャッシュ禁止ページのプールを利用することで排除または縮小される可能性があります。ドライバの割り当てを満たす必要のあるページはそのページ・プールから直接取得し、それらのページが解放される時にページ・プールへ戻されます。これはCPUページ・マッピングを同期するために追加のフラッシュ操作を実行する必要性を取り除きます。

Concurrent Real-Timeより出荷されるNVIDIAドライバは、新しくページを事前に割り当てるよう改良された方式に依存します。現在は次項で説明される「RedHawkページプール・ドライバー」を使用します。

NVIDIA CUDAは、CPU上で必要とするわずかな時間で多くの複雑な計算問題を解くためにNVIDIAグラフィクス処理ユニット(GPU)に搭載される並列計算エンジンを利用する汎用並列計算アーキテクチャです。

CUDAアプリケーションはNVIDIA GPUと接続するためにキャッシュ禁止バッファを利用するので、同じ事前割当てグラフィクス・ページプール・ページは、CUDAアプリケーションを実行しているシステムのシールドCPUのジッターを大幅に削減するのにも役立ちます。特別なCPUアプリケーションのコーディングまたは構成は必要ありません。

NOTE

9.1以前のRedHawkのバージョンはカーネルに組み込まれるNVIDIAドライバが標準装備でした。RedHawk 9.1はNVIDIAアドオン・ドライバー・メディアを標準装備する新しいページプール・事前割り当てドライバを単独で使用します。

9.1以前のRedHawkのバージョンは、承認済みNVIDIAドライバと新しいページプール事前割り当てドライバをインストールするためにNVIDIAアドオン・ドライバー・メディアも使用することも可能です。インストール・スクリプトは以前のNVIDIA RedHawkインストールを削除します。

RedHawkページ・プール・ドライバ

RedHawkはページを管理、割り当て、他のドライバに供給するページ・プール・ドライバを提供します。全てのメモリはページ・プール・ドライバが事前に割り当てられることで提供されるので、メモリを解放または割り当て時にページプール・ドライバを利用するドライバはプロセッサ間割込みの原因とはなりません

現時点で、Concurrent Real-Timeのメディアと一緒に出荷されるNVIDIAドライバのみがページ・プール・ドライバを使用することが許可されています。ページ・プール・ドライバはNVIDIAドライバより依頼されるページの割り当てや提供について責任を負います。これはNVIDIAドライバをロードする時はいつでも動的にロードされるモジュール式ドライバです。本ドライバはカーネル・ソースに含まれていますが、カーネルとは関係なく出荷されます。このような方法で古いシステムにおいて新しいNVIDIAドライバを対応することが可能となります。

ページ要求された時にページプールが空の場合、ページは動的に割り当てられてプロセッサ間割込みが通常どおり発生します。これらのページが解放される時、システムに解放される代わりにページ・プール・ドライバに戻されます。

PREALLOC_PAGEPOOL_PAGESカーネル・パラメータに正の値が設定されている場合、この値はプールに事前に割り当てられるページ数を表します。20480ページの値が全てのプレビルトRedHawk Linuxカーネルに設定されています。しかしながら、ページ・プール・ドライバがロードされるまでページ割り当ては発生しません。PREALLOC_PAGEPOOL_PAGESに0(ゼロ)が設定されている場合、ページプール割り当ては無効となります。

代わりにカーネル起動パラメータ(*preallocated_pages=<numpages>*)を設定することで静的にコンパイルされた値を無効にすることが可能です。*numpages*は事前割り当てされるページ数を表します。カーネル起動パラメータ*pagepool_disabled*を使ってページプール割り当てを無効にすることも可能です。カーネル起動パラメータの設定についてはRockyは**blscfg(1)**のmanページ、Ubuntuは**ccur-grub2(1)**を参照して下さい。

/proc/driver/pagepool/stats ファイルは、ページプールの使用率を表示および実際に使用している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/pagepool/stats
Pre-allocated pages:    20480
Total allocated pages:  23550
Pages in use:           0
Pages to be zeroed:     0
Maximum pages used:     20000
```

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。これは値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。ファイルに書き込まれるページの値はPages in useフィールドの現在の値以上である必要があることに注意して下さい。

以下の例はプールの事前割り当てページ数を5120へ下げます。

```
$ echo 5120 > /proc/driver/pagepool/stats
```

本ファイルへ書き込むことでページ数の値を変更する手法は起動後は持続しないので、ユーザーは本ファイルへ書き込むためのCAP_SYS_ADMINカーナビリティを持つことが必要です。

非現実的な大きな値を指定すると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、ファイルを読み出すことでページ割り当てを検証することが可能です。

システムで事前割り当てされたページ数を表示および変更するために**pagepool-memory(1)**ユーティリティも使用する事が可能です。詳細については**pagepool-memory(1)**のmanページを参照してください。

ユーザー・アドレス空間の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パ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10 μ 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。

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。

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. ハングアップ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：

- **/etc/grub2.cfg**を修正して下さい：

1. 以下の行を### END /etc/grub.d/00_header ### sectionセクションの前に追加して下さい：

```
serial --speed=115200 --unit=0 --word=8 --parity=no --
stop=1
terminal_input console serial
terminal_output console serial
```

2. シリアル・コンソールを介して起動する各カーネルに対して次のgrubオプションをそのカーネルのlinux16またはlinuxの行に付け足して下さい：

```
console=ttyS0,115200
```

- 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。
- 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「ハングアップ」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
```


RedHawkの起動コマンド・ライン・パラメータ

ここに含まれているのは、RedHawkで特有の起動コマンド・ライン・パラメータです。起動パラメータの完全なリストは**kernel-params.txt**ファイル内にあります。

起動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。起動コマンドはカーネル起動時に自動的に包含するために**/etc/grub2.cfg**へ追加、またはカーネル起動している時に起動コマンド・ラインに指定することが可能です。

カーネル起動パラメータはカーネル・ソース・ツリーの**Documentation/admin-guide**ディレクトリにある**kernel-params.txt**ファイル内に定義されています。

以下で使用されている**kern-name**は、カーネル・ソース・パッケージをインストールしたときに**/usr/src/**ツリー以下に生成されているディレクトリの名称であることに注意して下さい。名称の例が**linux-6.12.33-RedHawk-9.6**である場合、6.12.33はkernel.orgのバージョンで9.6はRedHawkのバージョンとなります。従って、このディレクトリ名称はRedHawkのリリース毎に変化します。

kernel-params.txtファイルへのパス形式は次のようになります：

`/usr/src/linux-<kern-name>-source/Documentation/admin-guide/`

Ubuntuシステムでのパスは次のようになります：

`/usr/src/linux-<kern-name>/Documentation/admin-guide/`

次のアルファベット順に並んだパラメータのリストは、改良されたリアルタイム性能を提供するためにRedHawkが追加または変更したLinuxカーネル・パラメータの全てが含まれています。これらに関する詳細については上述の**kernel-params.txt**ファイルを参照して下さい。

- `acpi_no_setmode_retry`
- `adjtimex.debug.on`
- `enable_managed_irqs`
- `fbs.intr.serialize`
- `idle.forcespin`
- `idle.shielded`
- `jrcu.ehz`
- `jrcu.hz`
- `jrcu.pri`
- `kvm.kvmrt_gs_hc_host_enabled`
- `kvmrt_gs_hc_guest_enabled`
- `kvmrt_gs_syscall_enabled`

- masterclock
- no_rcim_bar2_resize
- no-stagger-sched-tick
- serial_bound
- slub_cpu_partial
- smi_sniffer
- softirq.nodaemonize
- softirq.pri
- stop_machine_warning
- tsc
- tsc.recalibrate.async
- tsc.recalibrate.none
- tsc.recalibrate.sync
- vfio-pci.addr
- watchdog.verbose
- workqueue.pri
- workqueue.shield.nomigrate
- xilinx.fpga.aspm.off

