Linux カーネルの歩き方

Linux World C&D/Tokyo 2001

2001年10月24日

VA Linux Systems Japan
高橋　浩和
taka@valinux.co.jp

目次

- Linuxカーネルを読むツボ
- Linuxカーネルの基本メカニズム概要

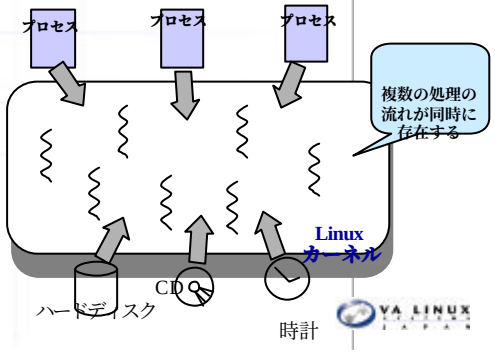
Linuxカーネルを読むツボ

- コンピュータのアーキテクチャの基礎知識
- OS(オペレーティングシステム)の基礎知識
- Linuxカーネルコードの取っ掛かり

コンピュータアーキテクチャを知る

- CPUアーキティクチャの基本
- ハードウェアの基本アーキテクチャ
  - CPU、バス、各種コントローラ
- 非同期処理の感覚を磨く
  - 割り込み
  - プリエンプション
  - CPU同士、CPUとI/Oコントローラの並列動作
- CPU、メモリ、バス、ネットワークなどの速度感覚

カーネルは並列処理する



並列動作による不思議なコード

一見不思議に見えるコードが散在する

```
if ( テーブルに目的のAに対応する箱がない) {
        Aを保存するための箱Bを確保
        箱BをAで初期化
        if (テーブルに目的のAに対応する箱がある) {
                箱Bを解放
                return
        }
        箱Bをテーブルに登録
}
```

何故二度判定する必要があるのか？

既存情報からの知識習得

過去の遺物からOSの基本を学ぼう

- UNIX 7 th Edition、UNIX 32V
- MINIX

既存情報からの知識習得

UNIXカーネル実装の入門書も悪くない

- The Design of The UNIX Operationg System (UNIX カーネルの設計)
- Lions' Commentary on UNIX
- Operating system — Design and Implementation … タネンバーグ著

✧他のOS解説本、各種論文は後からゆっくりと

Linuxカーネルの取っ掛かり

- お勧めの個所
  - プロセススケジューラ
    - コード量が小さく読みやすいが、マルチタスクを実現するための基本機能が濃縮されている
  - デバイスドライバ
    - 身近であり親しみやすい。
    - 機能がデバイスドライバの中に閉じており、カーネルの他の機能への影響が殆ど無い。カーネル本体とのシンプルなインターフェイスのみ。
  - モジュールプログラミング

Linuxカーネルの取っ掛かり

- 良く見かける失敗
  - カーネルのBOOT処理から読み始める
    - あらゆるハードウェア、カーネル内データを初期化するのみ。苦労する割に実りは少ない。
  - open システムコールから読み始める
    - UNIX(Linux)カーネル内で、最も複雑な処理の一つ。次々と新しいデータ構造・関数が登場し、蜘蛛の巣状に広がっていく。

良く目にする勘違い

- Linuxカーネルの中からは、当然システムコールを呼び出せない。
- カーネルの中では、Cの標準ライブラリも使えない
- Linuxカーネルは main() 関数から動き出すわけではない

どのカーネルを読むべきか

- 安定版カーネルと開発版カーネル
- マイナーバージョンが偶数の版は安定版
  - 2.0.x、2.2.x、2.4.x など
- マイナーバージョンが奇数の版は開発版
  - 2.1.x、2.3.x など

安定版カーネルでも、リリース番号が１０以下のカーネルは、不安定版カーネルと呼んだほうが良い◎

本当に安定した、安定版カーネルがお勧め

どのカーネルを読むべきか

- 基本はどのカーネルでも同じ。古い版の方がシンプル。
- シンプルなカーネルを読んだ後で、新しいカーネルを読むほうが読みやすい。気合で２.４カーネルから読み始めてもOK

Linuxカーネルソース構造

- 大雑把なソースコード構造の把握
  - Linux/kernel … カーネル基本機能
  - Linux/fs … ファイルシステム
  - Linux/mm … メモリ管理
  - Linux/arch … CPUアーキテクチャ依存部
  - Linux/include … ヘッダファイル
  - Linux/drivers … デバイスドライバ

Linuxカーネルソース構造

- 特に重要なファイルをおさえておこう。
  - プロセススケジューラ
    - linux/kernel/sched.c
  - 割り込み・ソフトウェア割り込み
    - linux/arch/i386/kernel/irq.c
    - linux/kernel/softirq.c
  - 時計
    - linux/kernel/timer.c
- カーネルソースの何処が重要かがわかると、理解の速度があがる。
  - とはいえ、２００万行を越えるソースにあたりがつくようになるまでが苦行の道だが…

読み進めるコツ

- 何か一つ目標を作りましょう
- ソースコードを読むだけではなく、実際にいじってみよう
- 想像力は重要。想像力で仮定を立て、その仮定を確認するという手段で読み進める。
- 最後は気合と根性

余裕が出来てきたら……

- 他のOS(BSDなど)のソースなどもみてみよう
- 他のOS用に書かれた論文にも目を通しましょう

カーネルを知っていると

- 気の利いたソフトウェア開発、システム設計ができるようになる
  - 負荷や性能を見通した無駄の少ない設計ができるようになる。

Linuxカーネル基本メカニズム

カーネルの基本メカニズム

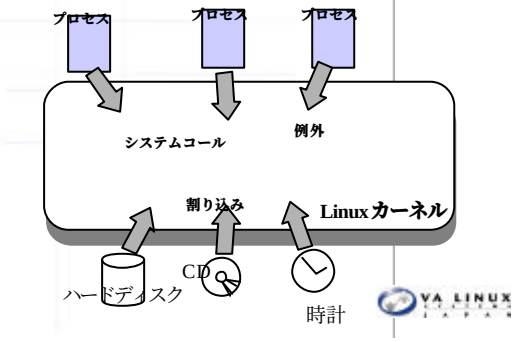
Linuxカーネルを動かすためのパーツの理解

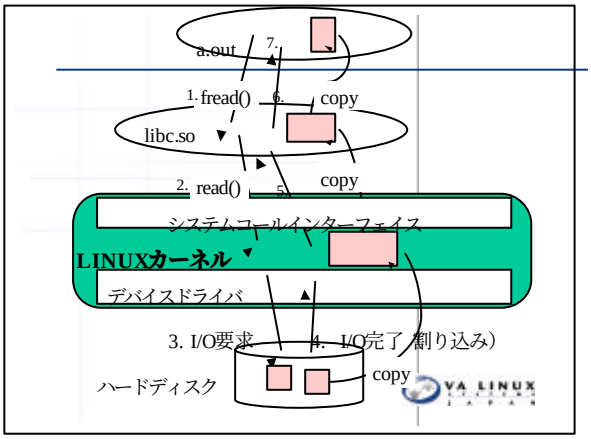
- システムコール
- プロセススケジューラ
- 割り込み処理
- 同期と排他
- 時計

システムコールと割り込み

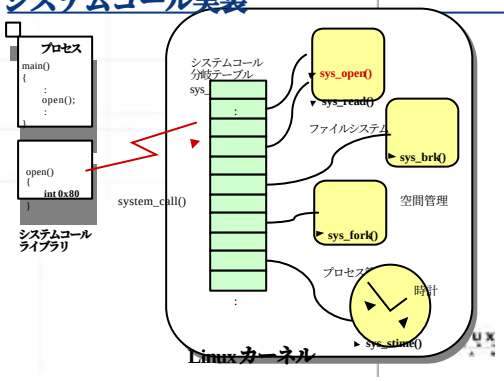
- システムコールは、プロセスからLinuxカーネルへの要求
  - プロセスのコンテキスト上で、Linuxカーネル内システムコールを実行（モノリシックカーネル方式）伝統UNIXと同じポピュラーな枯れた実装方式。
- 割り込みは、ハードウェアからLinuxカーネルへの要求
  - プロセスコンテキストに強制的に割り込んで実行

システムコールと割り込み

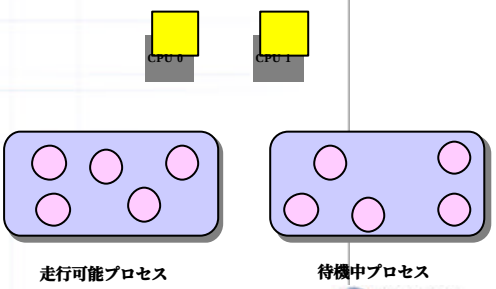




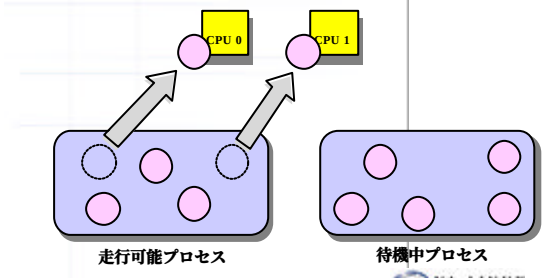
システムコール実装



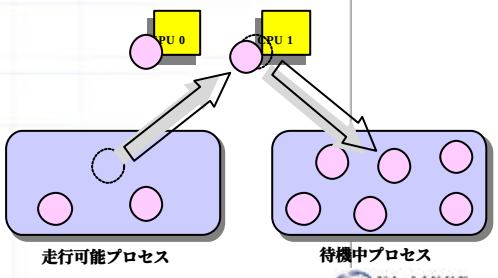
スケジューラの仕事



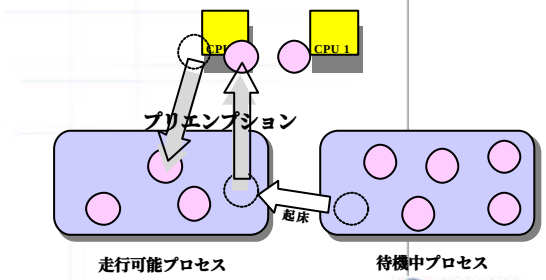
スケジューラの仕事



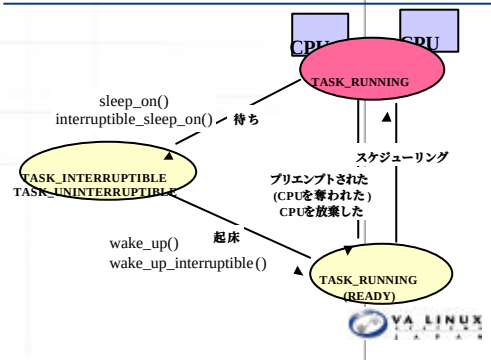
スケジューラの仕事



スケジューラの仕事



プロセスの状態遷移



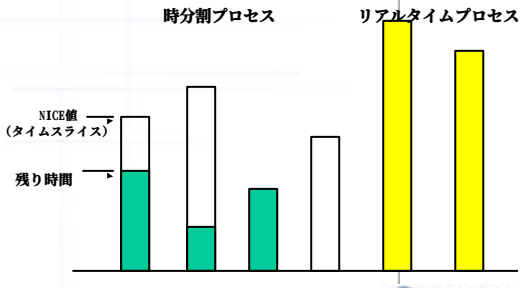
スケジューリングの方針

- 応答性(レスポンス)とスループットの両立
  - プリエンプションとタイムスライス
- スケジューリングのオーバヘッドの最小化
  - 再スケジューリング頻度を減らす
  - プロセスのCPU間移動を減らす
- 公平原則
  - 全てのプロセスに公平にCPUを割り当てる

応答性確保とスループット向上

- タイマがCPUに割り当てられているプロセスのクァンタム(CPU残り時間)を減らす。それに伴いプライオリティが下がっても、クァンタムを使い切るまでプリエンプトさせず、プロセス切り替えの回数を抑える
- 待機中プロセスが走行可能状態になった場合、即座に最も高いプライオリティのプロセスをCPUに割り当てる (プリエンプト：実行権の奪い取り)
- 実行中プロセスがCPUを手放した場合も、最も高いプライオリティのプロセスにCPUを割り当てる

プロセスの選択基準



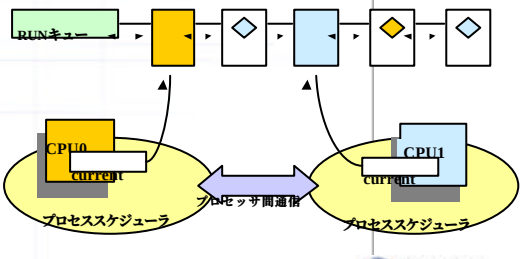
公平原則

- 幾らプライオリティが低くとも、走行可能状態のプロセスには必ずCPUの割り当てがあることを保証
  - 走行可能状態のプロセス全てがクァンタムを使い切るまで、プロセスには再度新しいクァンタムを与えない。
  - 古くの伝統UNIXのスケジューラでは高負荷時にCPU飢餓状態のプロセスが発生する。

CPU間移動のオーバヘッド

- プロセスのCPU間移動を最小限にする。
  - 別のCPU上で動作することのペナルティ
    - キャッシュ
    - TLB
  - CPU間通信のオーバヘッド
- なるべく同じプロセスを同じCPUに割り付ける
  - 少しぐらいプライオリティが逆転しても、他のCPU上のプロセスに対して、プリエンプト要求を出さない。
- NUMA型システムでは、ペナルティは更に大きい

スケジューラの実装



重要な関数

- schedule()
  - スケジューラ本体。各CPU上で独立して動作。常に走行可能状態プロセス群を監視し、最も動作するに相応しいプロセスにCPUの実行権を与える。
- schedule_idle()
  - スケジューリング要求を出す
- switch_to()
  - プロセスの切り替えを行う。現在実行中のプロセスのコンテキスト(CPU状態)を保存し、次に実行するプロセスのコンテキストをロードする。

Linuxスケジューラの問題点

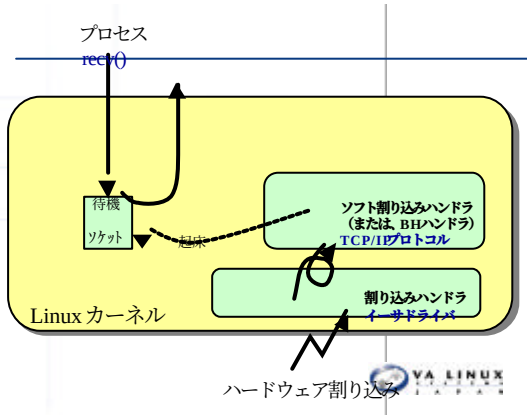
- あまりスケーラブルではない。大規模SMPシステムでは効率が悪い
  - CPU数の増加にあわせて、プロセスのCPU間移動によるオーバヘッドが大きくなる。
  - 単純なスケジューリングキューの構造。各CPU上で動作するスケジューラが同じデータ構造を参照
  - プロセッサ間通信の遅延による弊害

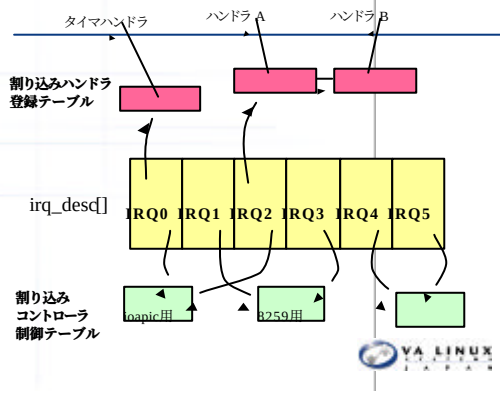
割り込み

- ハードウェアからの要求を受け付けるための機構
  - 割り込みを受け付けたCPUは、実行中のプロセスを一時中断し、割り込みハンドラを起動する
- 一つのIRQに複数の割り込みハンドラ登録可能
- 同じIRQに対するハンドラは、システム上に一つだけ存在可能

割り込み

- 割り込みレベル(優先度)が無い。伝統UNIXは各割り込みに優先度を割り当てることが可能
- どのCPU上でも任意の割り込みハンドラを実行可能。各CPU上で異なるIRQのハンドラを並列実行可能
- 応答性確保のため、割り込みハンドラとソフトウェア割り込みハンドラの二段階で処理





重要な関数

- do_IRQ(), handle_IRQ_event()
  - 割り込みハンドラ共通のエントリ関数。IRQ種別に対応する割り込みハンドラを呼び出す
- request_irq ()
  - 指定したIRQ用の割り込みハンドラを登録する

Linux割り込みハンドラの注意点

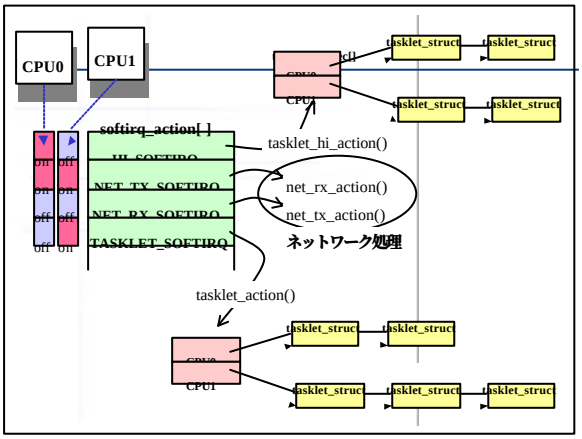
- 割り込みレベルが無い
  - 割り込みに優先度を持たせることができない
  - 複数の割り込みがIRQの数だけネストする可能性がある。システムコール処理、ソフトウェア割り込み処理の上に何重にもネストする。
- 割り込みハンドラの実装には注意が必要
  - 短時間で処理を完了し、ソフトウェア割り込みハンドラに処理を委ねること
  - 割り込みスタックの使用量を最小限にすること

ソフトウェア割り込み

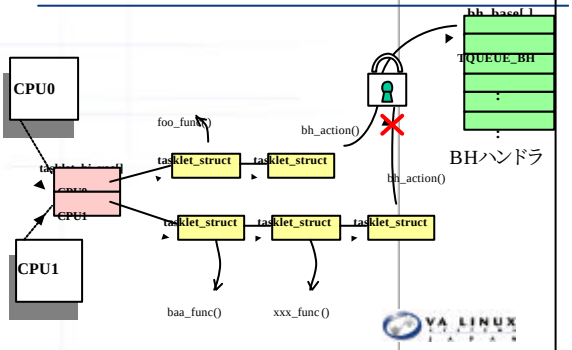
- 割り込み処理の遅延実行
  - 割り込みハンドラ終了時呼び出し
  - カーネルスレッド(ksoftirqd)による実行。各CPU毎に1スレッド用意
- ハードウェア割り込みハンドラが動作したCPU上で、ソフト割り込みハンドラを起動
- 複数のソフトウェア割り込みハンドラが、各CPU上で並列動作可能（同種のハンドラでもOK）
- 旧版との互換性のため、複雑な構造になっている

各種ハンドラ

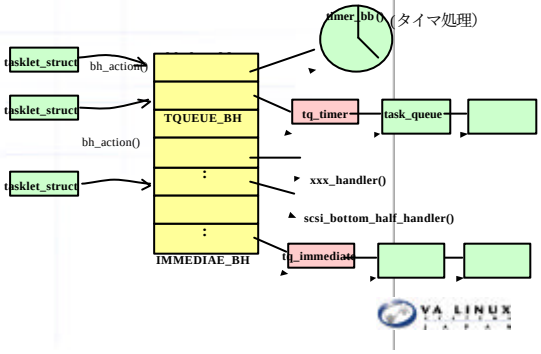
- ソフトウェア割り込みハンドラ
  - 複数のCPU上で並列実行可能。排他処理はハンドラ自身の責任
- タスクレット
  - 後述タスクキューのSMP対応版。各CPU毎に実行させるべきハンドラのキューを用意
- BHハンドラ
  - 旧型ソフトウェア割り込みハンドラ。複数のCPU上で並列動作することができないハンドラを登録
- タスクキュー
  - 固定数のハンドラしか登録できないBHハンドラを拡張し、汎用化したもの
  - 割り込み処理遅延以外の目的でも利用可能



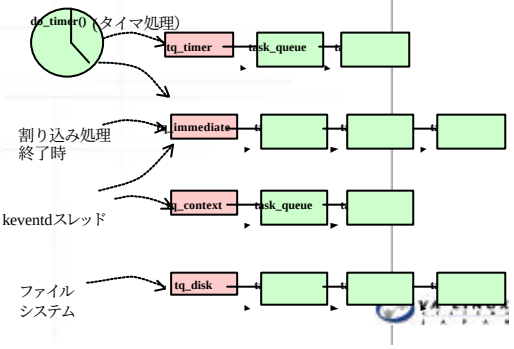
タスクレット



BH(ボトムハーフ)ハンドラ



タスクキュー



重要な関数

- cpu_raise_softirq()
  - ソフトウェア割り込みの要求
- do_softirq()
  - ソフトウェア割り込みハンドラの実行
- tasklet_schedule()
  - タスクレットへのハンドラ登録
- tasklet_action()
  - タスクレットの実行

重要な関数

- mark_bh()
  - BHハンドラ起動要求
- bh_action()
  - BHハンドラの実行
- queue_task()
  - タスクキューへのハンドラ登録
- run_task_queue()
  - タスクキューの実行

利用上の注意点

- 割り込みハンドラ、ソフトウェア割り込みハンドラ内から利用できるカーネル機能には制限がある
  - 待機状態になることができない。待機できるのはプロセスだけ
  - 動的資源確保は原則禁止

同期と排他

- プリエンプト可能とノンプリエンプト
- 割り込み禁止
- プロセスのカーネル内待ち合わせ
- CPU間同期とスピンロック
- アトミック(不可分)操作

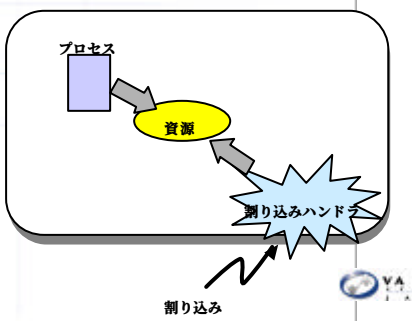
プリエンプト可能性

- ノンプリエンプト
  - プロセスが自らCPUの実行権を手放さない限り、他のプロセスにCPUを奪い取られることが無い。ユニプロセッサ時には、排他処理を非常に単純化できる。
- カーネル内ノンプリエンプト
  - カーネル内処理の吐いた処理を単純化することが可能。特にユニプロセッサ時に効果が大きい。
- カーネル内プリエンプト可能
  - 応答性を高めることが可能。ただしカーネル内構造が複雑化する。

割り込み禁止

- CPU割り込み禁止
  - プロセスコンテキストと割り込みコンテキストの両方から操作される資源を保護する。
  - 排他区間走行中に割り込みらがが発生すること(割り込みハンドらが走行すること)を禁止する。
- ある特定のデバイスの割り込み(IRQ)のみ禁止
- 他のCPU上には割り込みが発生する。擬似的に全てのCPUでの割り込みを禁止する仕組みは提供されているが性能は非常に悪い。

割り込み禁止



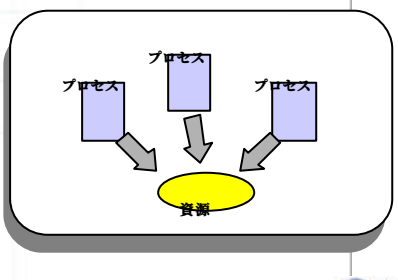
割り込み制御関数

- CPU割り込み禁止
  - _cli(), _sti()
  - save_flages(), restore_flags()
- ある特定の割り込み(IRQ)のみ禁止
  - disable_irq(), enable_irq()
- 擬似的に全CPUでの割り込みを禁止
  - cli(), sti()
  - 後述するスピンロックを利用して実現

スリープとウェイクアップ

- 伝統UNIXから実装されている最もポピュラーなカーネル内でのプロセス間同期機構
  - カーネル内処理を行うにあたり条件がまだ成立していない場合、プロセスは条件成立までウェイトチャネルで待機する
  - 条件成立時にウェイトチャネルで待っているプロセス群を全て起床する

スリープとウェイクアップ



スリープとウェイクアップ 例)

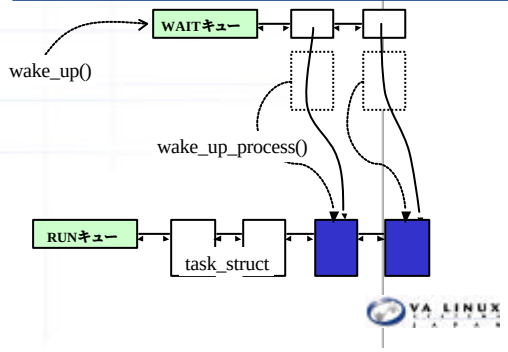
```
while (オブジェクトA = = BUSY) {
        waitflag = ON;
        スリープ(ウェイトチャネル);
}
オブジェクトA = BUSY;
        :
      <オブジェクトAの操作>
        :
オブジェクトA = FREE;
if (waitflag = = ON) {
        waitflag = OFF;
        ウェイクアップ(ウェイトチャネル)
}
```

While文であることに注意

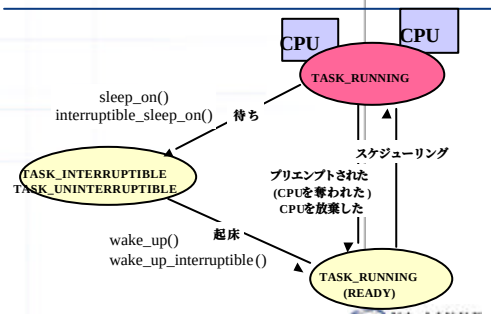
スリープとウェイクアップ関数

- sleep_on()
  - プロセスが条件成立を待って、待機する。
- interruptible_sleep_on()
  - sleep_on()と同じだが、シグナルで強制起床させることが可能
- wake_up()
  - 指定されたウェイトチャネルで待機しているプロセスを起床する
- wake_up_interruptible()
  - シグナルによる強制起床

通常の待ち



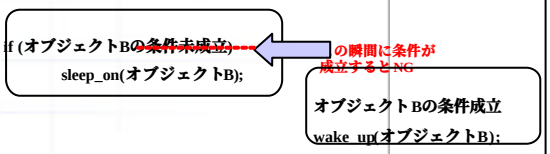
プロセス状態遷移



スリープとウェイクアップ関数

- sleep_on()関数を利用せず、同等の処理を展開した形式で記述してある個所が多い
  - 割り込み、ソフトウェア割り込みからのwake_up()処理、SMP時の他CPUからのwake_up()時に対応するため。WAITキュー操作と条件成立処理をアトミックに行えないため、その逃げの処理

sleep_on()では実現できない場合



展開されたsleep_on()関数

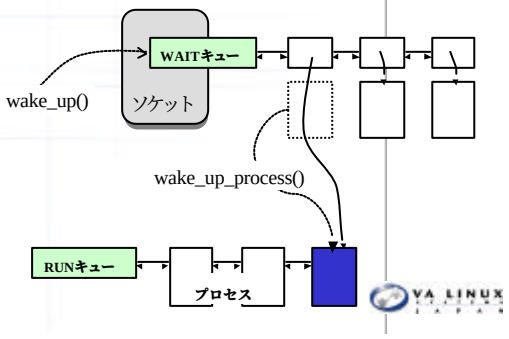
```
wait_for_something(オブジェクトB)
{
        struct wait_queue  wait = { current, NULL };
        add_wait_queue(オブジェクトBのwaitキューhead);
        current->state = TASK_INTERRUPTIBLE;
        if (オブジェクトBの条件未成立)
                schedule();
        current->state = TASK_RUNNING;
        remove_wait_queue(オブジェクトBのwaitキューhead);
}
```

この瞬間に条件が成立してもOK

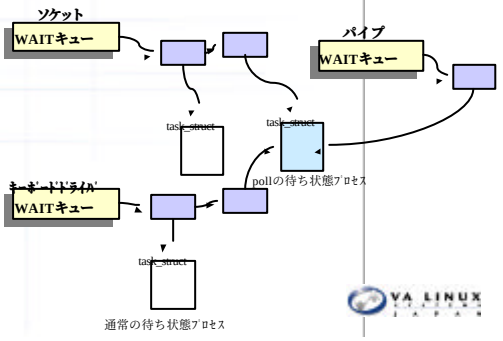
Linuxでの拡張

- Linuxでは、複数のウェイトチャネルで待てるように拡張されている
  - select(), poll()システムコールの実装で有利
- wake_up()時に、プロセス一つだけを起こすことも可能。
  - 多くのプロセスから同時に高頻度でアクセスされるオブジェクトに対して有効に機能
  - ただし、現在の実装では、起床順序はFIFO

EXCLUSIVE属性の待ち



select/poll待ち

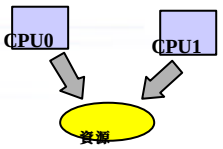


プロセッサ間の排他

- スピンロック
  - CPU間で同期を取るための最もポピュラーな方法
  - ある資源操作時、他のCPUがある資源を利用している間、ビジーウェイトする
- アトミック(不可分)操作

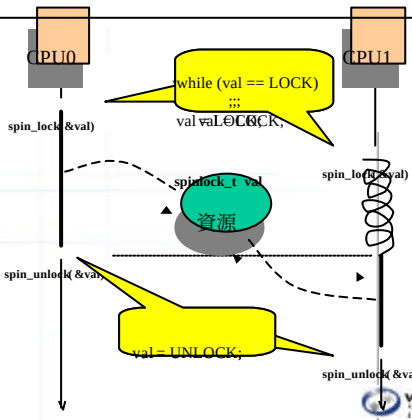
- 複数の資源を同時にロックしなければならなこともある
  。

プロセッサ間の排他



プロセスからのみ参照される資源

- 単純スピンロックで排他
- spin_lock()
  - 資源(ロックフラグ)をロックする
- spin_unlock()
  - ロックの解除
- spin_trylock()
  - ロックを試みるが、既にロックされていたらエラーとなる



割り込みからも参照される資源

- 単純スピンロックに加え割り込みも禁止する
- spin_lock_irq(), spin_lock_irqsave()
  - 資源(ロックフラグ)をロックする
- spin_unlock_irq(), spin_lock_irqrestore()
  - ロックの解除

参照頻度の高い資源のロック

- 更新頻度より、参照頻度の高い資源のロック
- 複数のCPUが同時に資源を参照することが可能。
- 資源を更新するためにロックした場合は、通常のスピンロックと同様に動作
- read_lock(), read_unlock()
  - Readロックする。複数のReadロックが可能。Writeロックされている場合はビジーウェイトする
- write_lock(), write_unlock()
  - Writeロックする。Readロックまたは他のWriteロックがある間、ビジーウェイトする

アトミック操作

- CPUアーキテクチャレベルで、アトミック処理可能なものは、スピンロックを利用せず実現
- Atomic_read(), atomic_set(), atomic_add(), atomic_sub(), atomic_inc(), atomic_dec(), atomic_inc_and_test(), atomic_dec_and_test() など

```
struct buffer_head * getblk(kdev_t dev, int block, int size)
{
        struct buffer_head * bh;
        int isize;
repeat:
        spin_lock(&lru_list_lock);
        write_lock(&hash_table_lock);
        bh = __get_hash_table(dev, block, size);
        if (bh)  goto out;
        isize = BUFSIZE_INDEX(size);
        spin_lock(&free_list[isize].lock);
        bh = free_list[isize].list;
        if (bh) {
                __remove_from_free_list(bh, isize);
                atomic_set(&bh->b_count, 1);
        }
        spin_unlock(&free_list[isize].lock);
```

各種スピンロックで細かくガード

アトミック更新

ジャンボロック

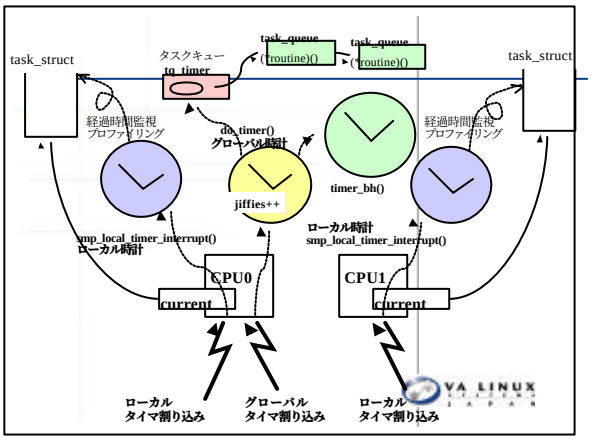
- 大きな変更を必要とせずSMP対応を行うために導入された。カーネル内排他処理を単純化できる。
- ジャンボロックのネストが可能
- 殆どspin_lockに書き直されたが、一部ジャンボロックが残る
- lock_kernel()
  - Linuxカーネル全体をロックする。既にロックされていたらビジーウェイトする。
- unlock_kernel()
  - カーネルジャンボロックの解除

スピンロック利用の注意

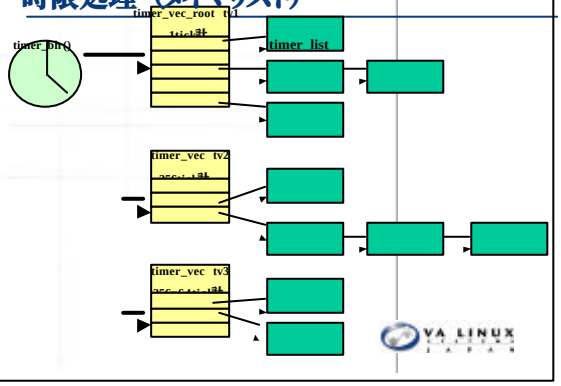
- スピンロックを確保したままスリープしてはならない
- 排他区間は可能な限り短くする
  - 性能劣化を最低限に抑える
- デッドロックの回避
  - 複数の資源をロックする場合、ロック順序を揃える
  - ロックしなければならない資源の数を絞り込む
  - 非同期な割り込み・ソフトウェア割り込み、他のCPUの動きまでも考慮

時計

- 二つの時計
  - 割り込みハンドラdo_timer()、時刻を刻む
  - BHハンドラtimer_bh()、時限処理の実行
- システムグローバルなタイマとCPUローカルなタイマ（３つめの時計）
  - local_timer_interrupt() 実行中プロセスのプロファイリング処理を行う
- ３つ目の時計
  - CPU毎の時計smp_local_timer_interrupt()



時限処理（タイマリスト）



重要な関数

- add_timer()
  - タイマ(時限処理)の登録
- del_timer()
  - タイマ登録の解除
- run_timer_list()
  - 登録されているタイマの実行