Linuxのスケジューラを読もう

2003年6月20日
VA Linux Systemsジャパン(株)
技術部 マネージャー
箕浦 真

VA LINUX SYSTEMS JAPAN

Linuxのスケジューラを読もう

- ■前書き
- ■カーネル基礎知識
- ■スケジューラの基礎
- ■スケジューラを読む
- ■2.5系での強化
- ■まとめ

VA LINUX SYSTEMS JAPAN

カーネルソースを読むということ

- なぜ?
  - 生活に必要な機能を追加したいから?
  - お仕事で必要だから?
  - 不審な動作を見付けた?
  - そこにコードがあるから?
- なにが特殊?
  - 別に特殊なことはない
  - ちょっと複雑
  - いろんな要素が並列･非同期に動作
  - デバッガで追いにくい (複数コンテキスト)

VA LINUX SYSTEMS JAPAN

Linuxカーネルツアー

- お近くのミラーからソースゲット
  - カーネルソースパッケージでもいいけど
- grep(1)は常に味方
- find . –name *.[hcS] ¦ xargs grep –n hoge
- Tag jumpの活用
  - make TAGS (emacs用)
  - make tags (vi用)
- エディタのマークジャンプ機能

Linuxカーネルソースツリー

Documentation	文書
Makefile, Rules.make	
arch	アーキテクチャ(i386、ia64、sparc…)毎のファイル
drivers	デバイスドライバ
fs	ファイルシステム群とVFS
include	ヘッダファイル群
init	main.c
ipc	SYSV IPC
kernel	カーネル本体
lib	共通関数群
mm	メモリ管理システム
net	ネットワーク処理
scripts	カーネルbuildスクリプト等

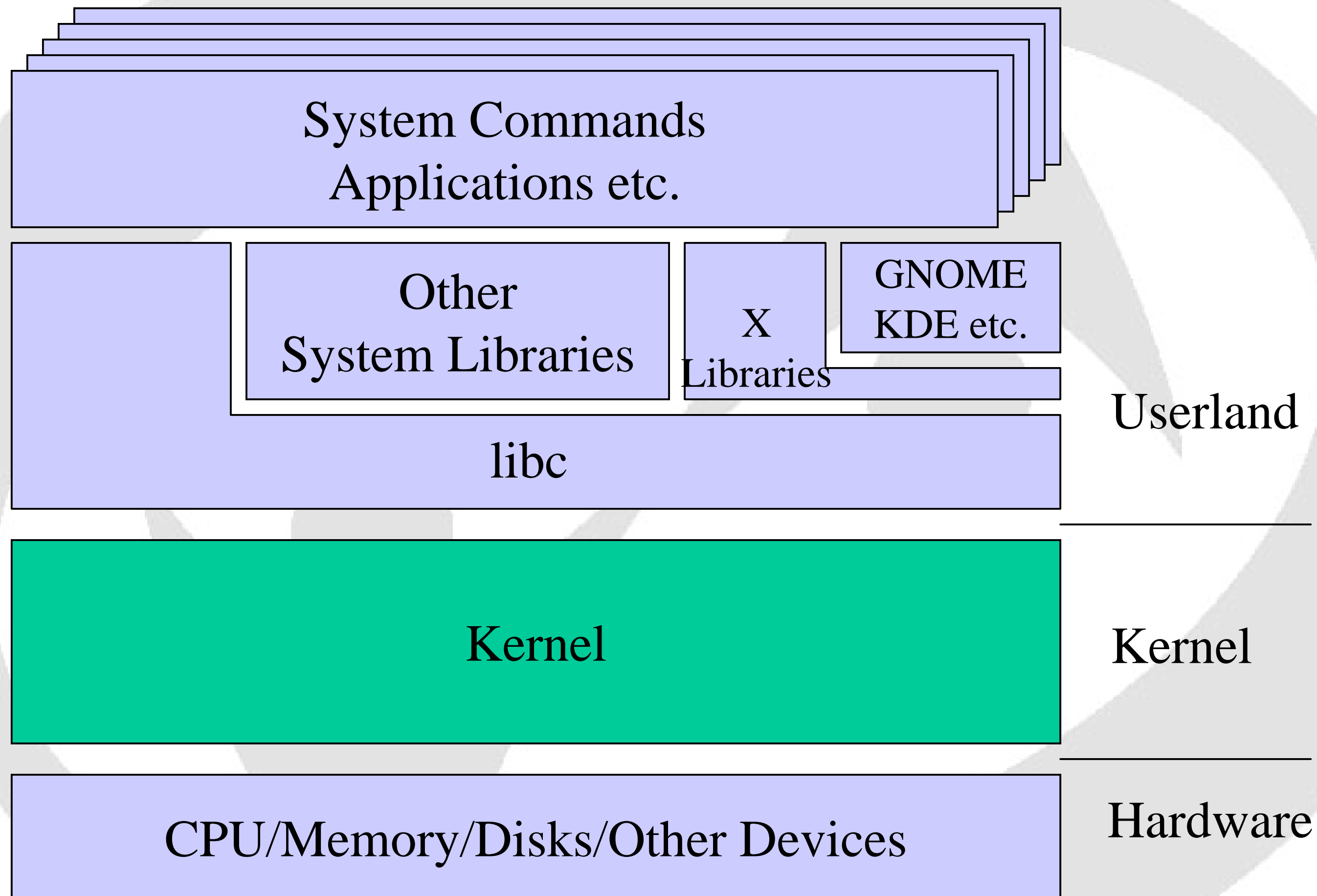
VA LINUX SYSTEMS JAPAN

カーネル基礎知識

■OSとカーネルの構造
■プロセスとカーネルスレッド
■トップハーフとボトムハーフ
■カーネル内排他処理
■jiffies
■コンテキストスイッチ

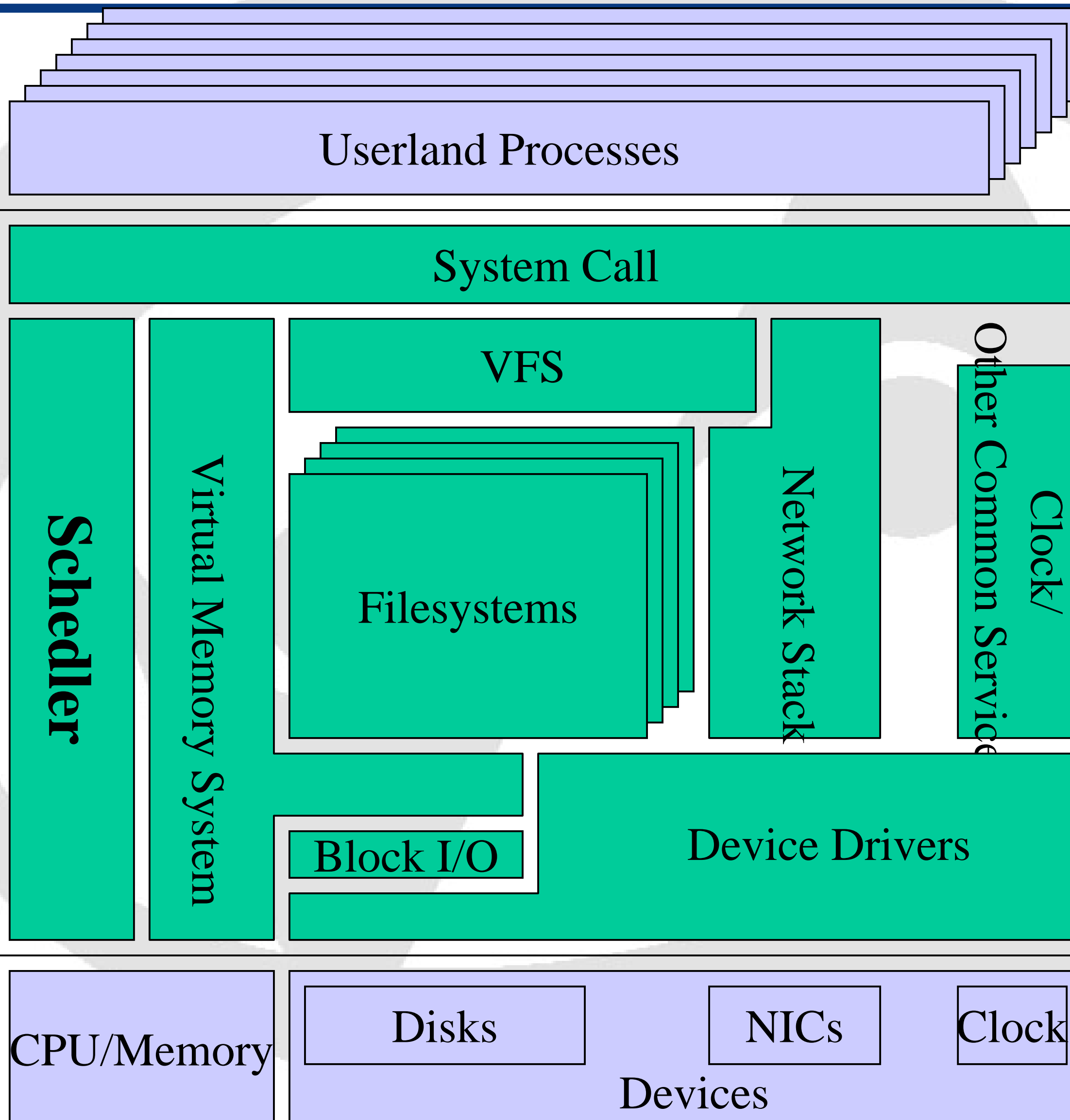
VA LINUX SYSTEMS JAPAN

オペレーティングシステム構造



VA LINUX SYSTEMS JAPAN

カーネルの構造



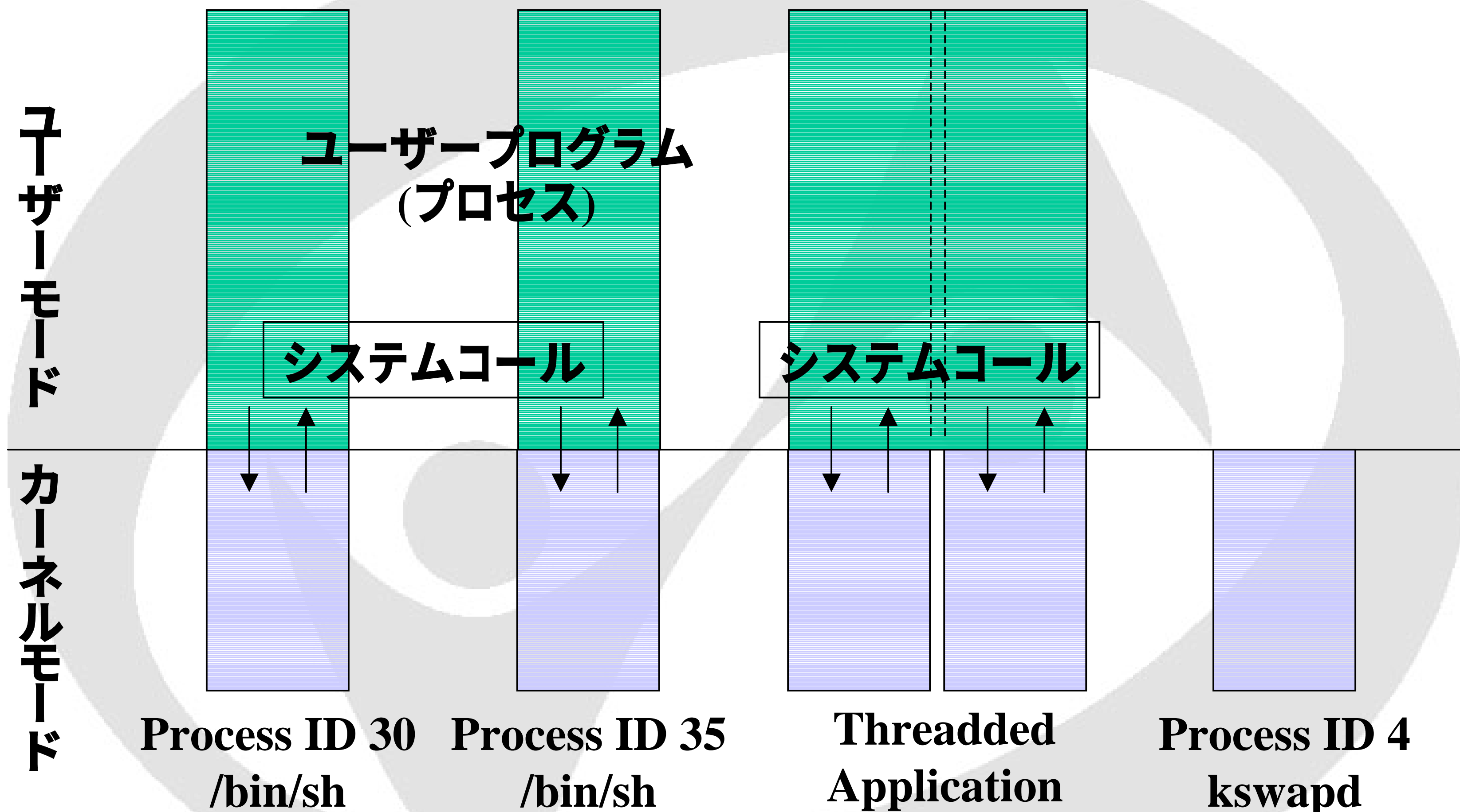
VA LINUX SYSTEMS JAPAN

プロセスとカーネル

- プロセス: プログラムのインスタンス
- カーネルによる資源配分の対象
- システム内に複数存在、それぞれ不可侵な実行環境を与えられる
- カーネルへの働きかけ: システムコール
- 他のプロセスへの働きかけ: IPC
  - signal、pipe、socket、SYSV-IPC、…

VA LINUX SYSTEMS JAPAN

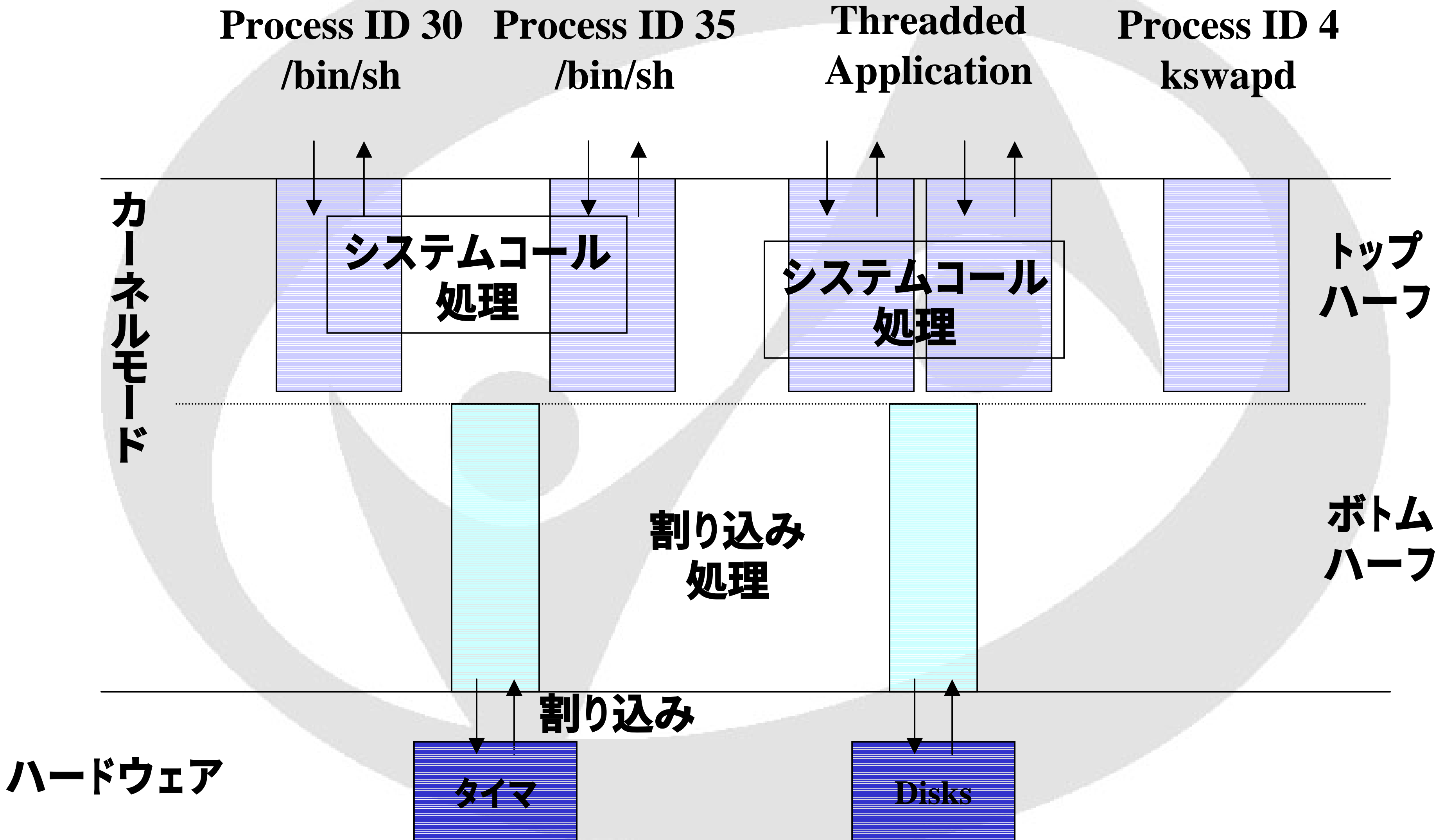
カーネルスレッド



■以下『プロセス』と呼ぶ

VA LINUX SYSTEMS JAPAN

トップハーフとボトムハーフ



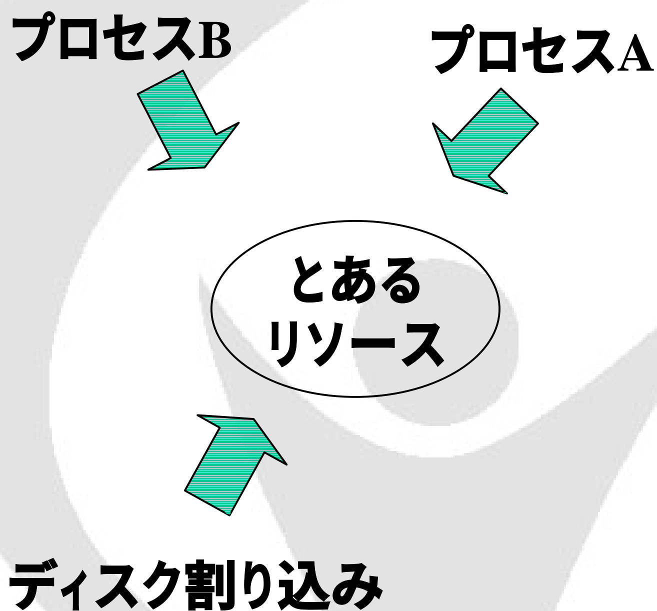
VA LINUX SYSTEMS JAPAN

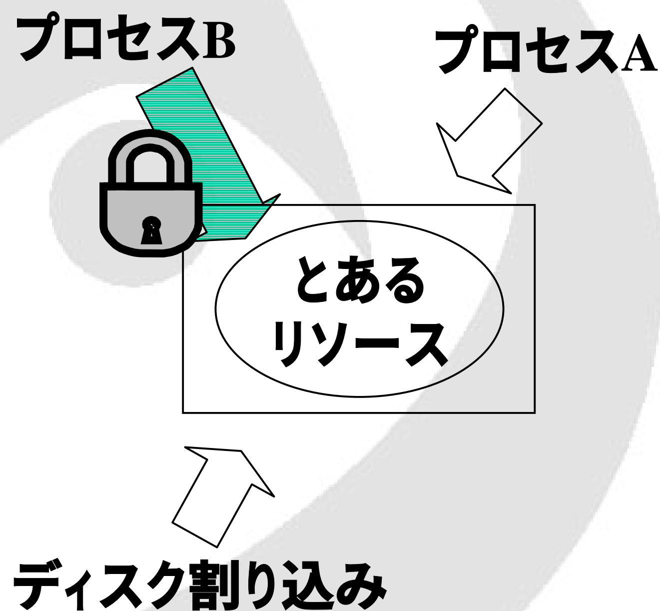
トップハーフとボトムハーフ

- ■トップハーフ
  - ●プロセスコンテキスト、アッパーハーフ
  - ●「待ち」に入る (= 他のプロセスにCPUを譲る) ことができる
- ■ボトムハーフ
  - ●割り込みコンテキスト、ロウワーハーフ
  - ●待ちに入るどころか、一刻も早く処理を終了させる必要

VA LINUX SYSTEMS JAPAN

カーネル内排他処理





- 排他の手法: spinlock、semaphore、per-CPU data、RCU (2.5～)、…

VA LINUX SYSTEMS JAPAN

jiffies

- システムのチックカウント
- 1秒間にHZ回更新される

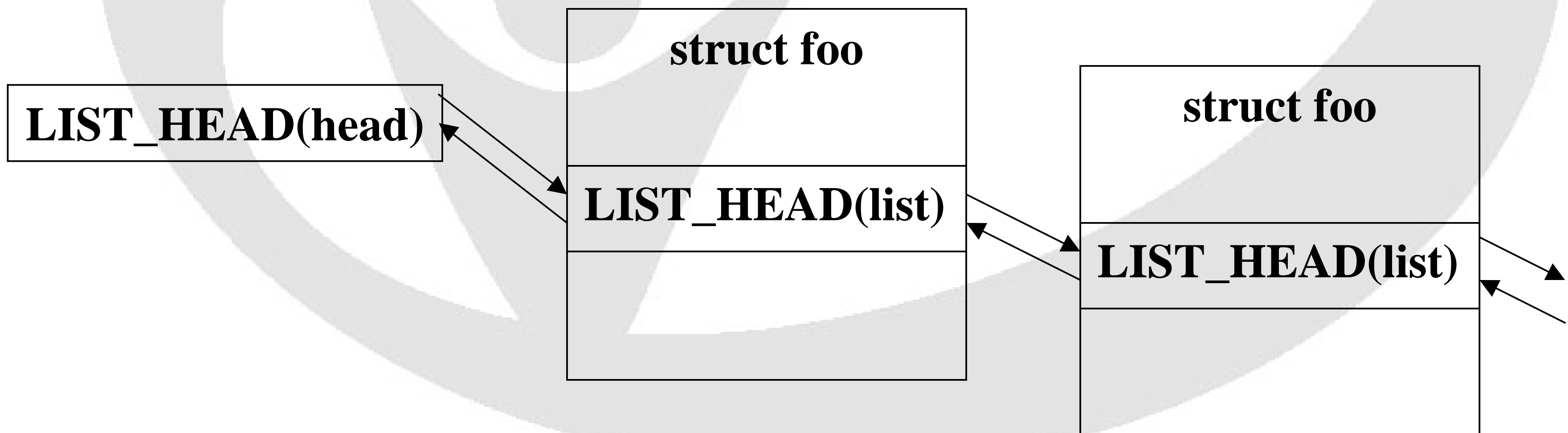
VA LINUX SYSTEMS JAPAN

リスト構造

■カーネル内でのリストの保持方法 <linux/list.h>

```
struct list_head {
    struct list_head *next, *prev;
}
#define LIST_HEAD(name) struct list_head name = LIST_HEAD_INIT(name)
```

その他list_add、list_del、…



VA LINUX SYSTEMS JAPAN

コンテキストスイッチ

- ■ あるプロセスから次のプロセスへの遷移
  - ●メモリ空間の切り替え
  - ●CPUレジスタの退避と復帰
- ■ 自発的なコンテキストスイッチ
  - ●I/Oの開始 (→終了までCPUを譲る)
  - ●wait(2) (→子プロセス終了までCPUを譲る)
  - ●sleep(3)、poll(2)、…
- ■ 非自発的なコンテキストスイッチ (preempt)
  - ●より高い優先度のプロセスが発生

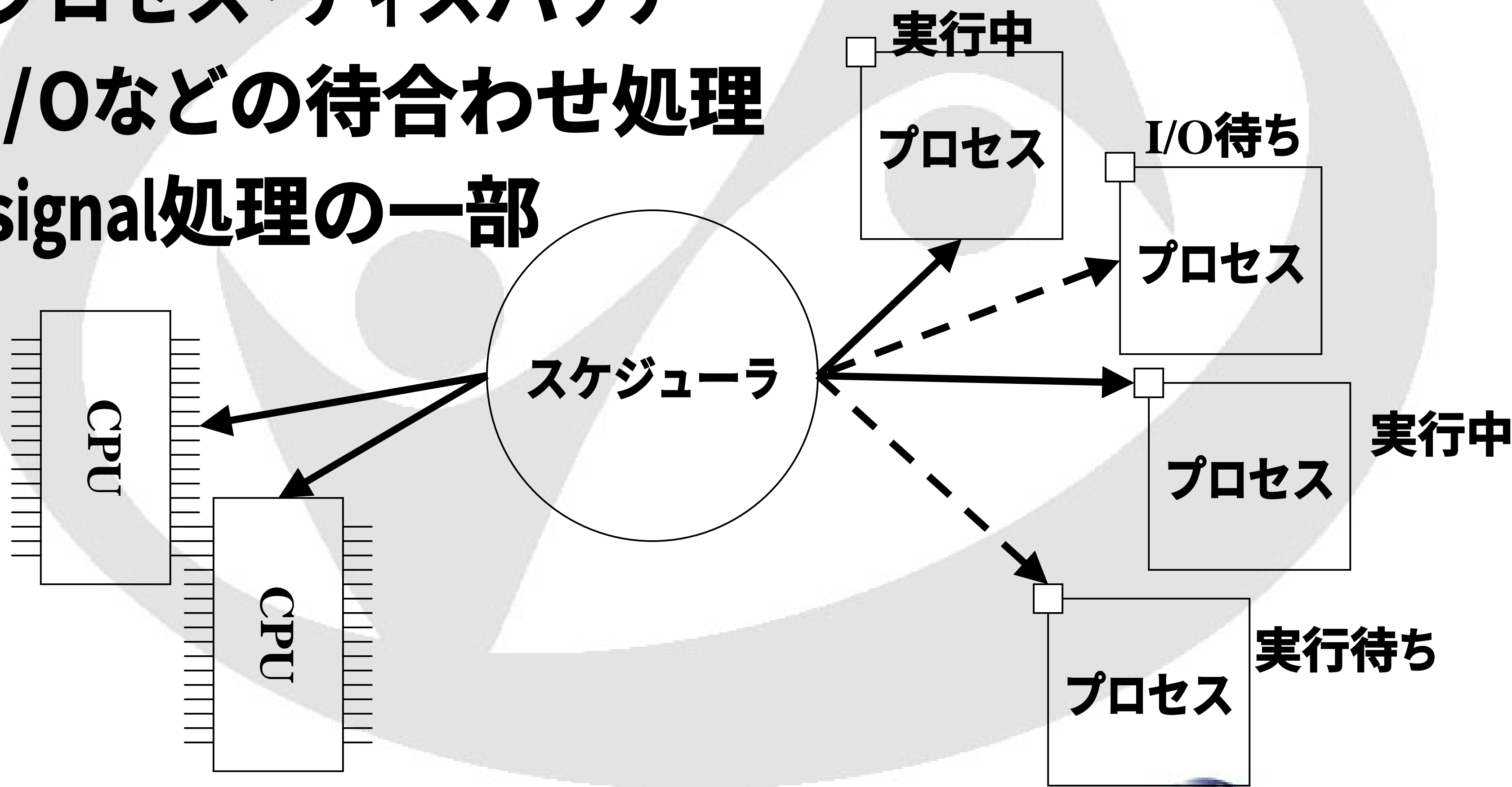
VA LINUX SYSTEMS JAPAN

スケジューラの基礎

- スケジューラとは
- スケジューラに関する諸問題

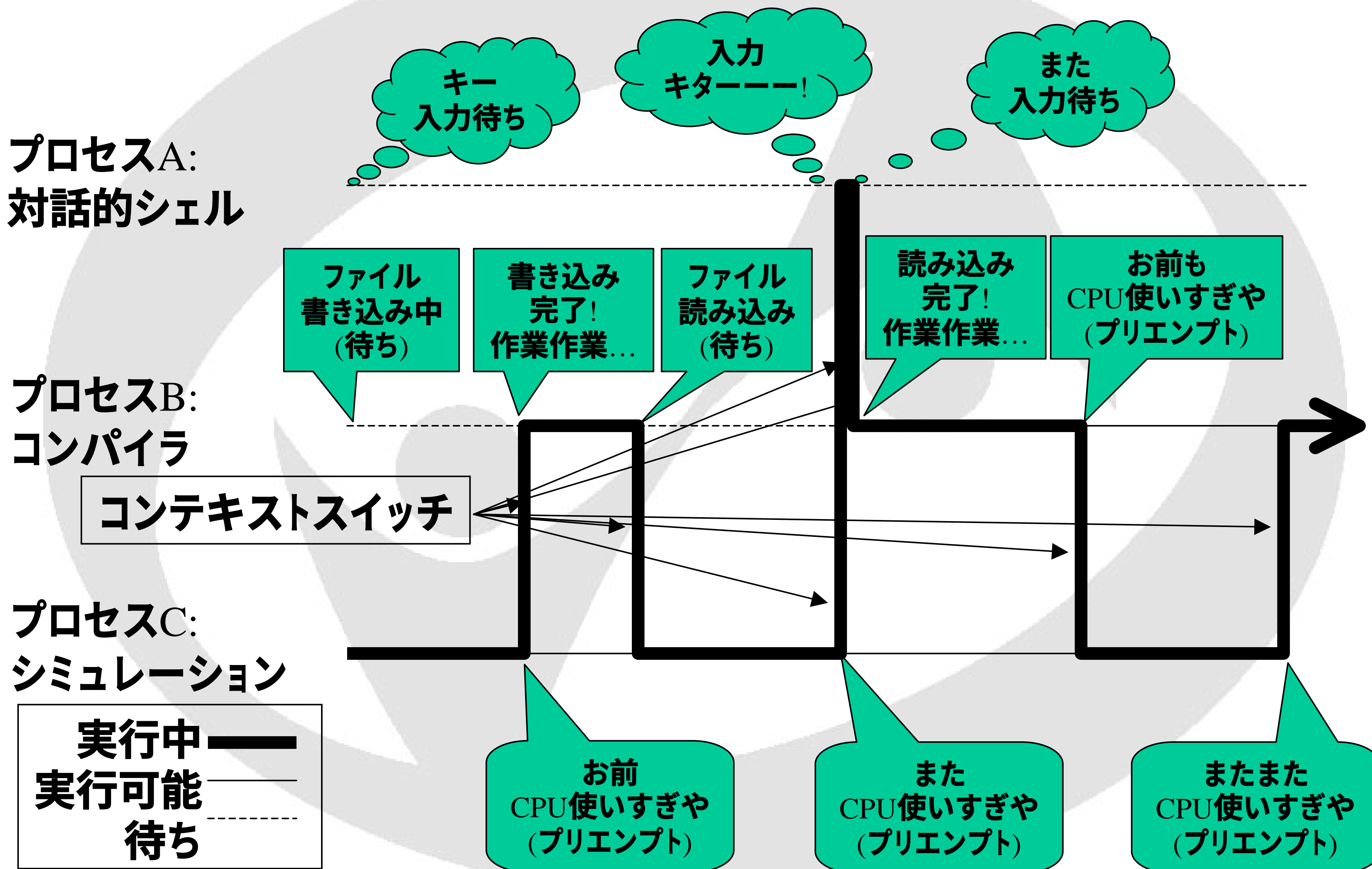
スケジューラとは

- 各プロセスにCPU資源 (CPU時間) を配分
- プロセス･ディスパッチ
- I/Oなどの待合わせ処理
- signal処理の一部



VA LINUX SYSTEMS JAPAN

スケジューラとは



スケジューラに関する諸問題

- 「公平」なプロセススケジューリング
  - シミュレーションプロセス (CPUバリバリ)、ネットワークデーモン (ネットワークからの要求待ち)、インタラクティブシェル (人間からの要求待ち) などに対する「公平性」とは?
  - CPUの公平性 (暇なCPUと忙しいCPU)
- 単純さ
  - 公平なスケジューリングのために頻繁に呼ばれる
  - 公平性にこだわるあまりこったアルゴリズムにすると、性能が落ちる可能性

VA LINUX SYSTEMS JAPAN

スケジューラを読む

- スケジューラに関するデータ構造
- プロセスの生成とスケジューラ
- スケジューラの入り口
- schedule()関数、goodness関数
- プロセスアカウンティング
- プリエンプト処理
- まちあわせ処理

VA LINUX SYSTEMS JAPAN

前提

- linux-2.4.20
- Uni-processor
- i386

VA LINUX SYSTEMS JAPAN

スケジューラに関するデータ構造

- まずはデータ構造を把握
- ざっと眺めたときに目につく構造体、近隣で定義されている構造体、グローバル変数など
- 各データ構造の関係の把握

VA LINUX SYSTEMS JAPAN

task_struct (<linux/sched.h>)

- プロセスに関するあらゆる情報を管理
- 重要メンバ
  - volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
  - int sigpending;
  - volatile long need_resched;
  - long counter;
  - long nice;
  - unsigned long policy;
  - int processor;
  - unsigned long cpus_runnable, cpus_allowed;
  - struct list_head run_list;
  - long per_cpu_utime[NR_CPUS], per_cpu_stime[NR_CPUS];

プロセスの状態

- #define TASK_RUNNING 0
- #define TASK_INTERRUPTIBLE 1
- #define TASK_UNINTERRUPTIBLE 2
- #define TASK_ZOMBIE 4
- #define TASK_STOPPED 8

VA LINUX SYSTEMS JAPAN

スケジューリングポリシー

■スケジューリングクラス

```
#define SCHED_OTHER     0 (通常)
#define SCHED_FIFO      1 (リアルタイム)
#define SCHED_RR        2 (リアルタイム)
```

■フラグ

```
#define SCHED_YIELD     0x10
```

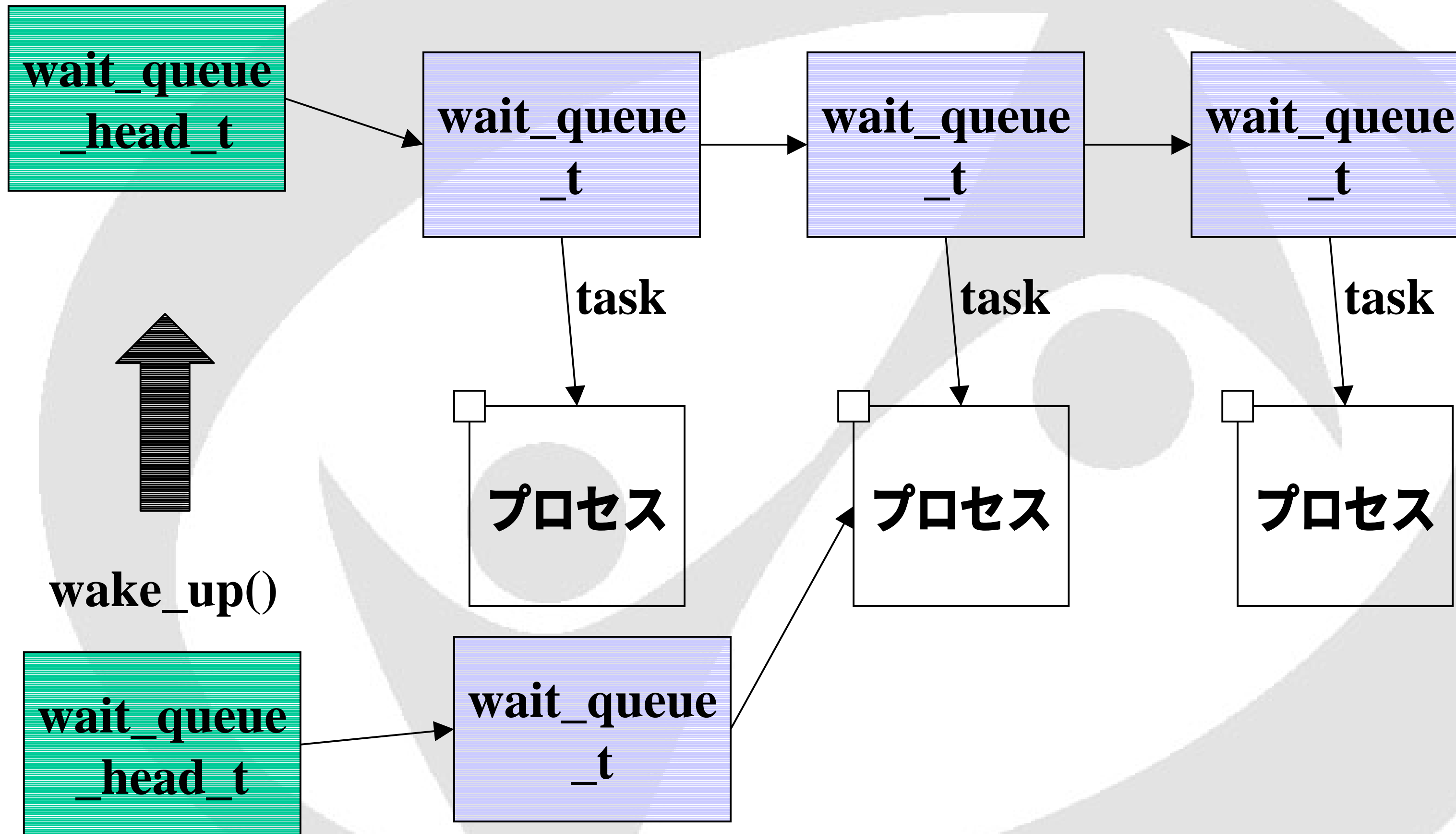

VA LINUX SYSTEMS JAPAN

waitqueue (<linux/wait.h>)

- あるリソースを待つプロセスのリスト
- リソースに対してwait_queue_head_tを定義し、各プロセスはそこにwait_queue_tを登録する
- wait_queue_tの重要メンバ
  - unsigned int flags;
  - struct task_struct * task;
  - struct list_head task_list;
- wait_queue_head_tの重要メンバ
  - wq_lock_t lock;
  - struct list_head task_list;

VA LINUX SYSTEMS JAPAN

waitqueue (<linux/wait.h>)



- 複数のwaitqueueに繋げることもできる (see sys_poll)

VA LINUX SYSTEMS JAPAN

runqueue (kernel/sched.c)

- sched.cのstatic変数
- static LIST_HEAD(runqueue_head);
- →struct task_structのrun_listメンバ
- queue?

VA LINUX SYSTEMS JAPAN

aligned_data (schedule_data)

- sched.cのstatic変数/構造体
- (名前センスないね)
- 重要メンバ
  - struct task_struct * curr;
  - cycles_t last_schedule;

VA LINUX SYSTEMS JAPAN

init_task (idle task)

- アイドル時に実行される仮想的なプロセス
- i386の場合 cpu_idle() (arch/i386/kernel/process.c)
- 典型的にはCPUを止める (hlt命令)
- CPU毎に用意 (bootプロセッサとそれ以外では実行パスが異なる)
- 通常外 (ps(1)コマンドなど) からは見えない

VA LINUX SYSTEMS JAPAN

プロセスの生成とスケジューラ

- fork(2) →sys_fork() →do_fork() (fork.c)
- task_structの確保と各メンバの初期化 (エラー処理を除きほぼ一本道)
  - state = TASK_UNINTERRUPTIBLE
  - counter = 親のcounter
  - run_listはどこにも繋がない
- 最後にwake_up_processで新しいプロセス(子プロセス)を起床

VA LINUX SYSTEMS JAPAN

wake_up()

- wake_up_process() → try_to_wake_up(p, 0)
- try_to_wake_up (*p, int synchronous)

```
{
    p->state = TASK_RUNNING;
    add_to_runqueue(p);
    if (synchronous ¦¦ ～(略)～)
      reschedule_idle(p)
}
```


VA LINUX SYSTEMS JAPAN

reschedule_idle()

■ reschedule_idle()

```
{
    tsk = cpu_curr(this_cpu)  ←実行中のプロセス
    if (preemption_goodness(tsk, p, this_cpu) > 0)
      tsk->need_resched = 1;
}
```

■ preemption_goodness()

- 2つのプロセスのgoodness値(優先度)を比較

■ need_reschedについては後で

VA LINUX SYSTEMS JAPAN

スケジューラの入口

- init/main.c の初期化の流れ
  - kernel_thread() で/sbin/initプロセス (PID 1)を生成
  - PID 0 : 各種初期化後、cpu_idle()呼出し
  - PID 1 : /sbin/initを起動 (init())
- cpu_idle() (arch/i386/process.c)

```
while(1) {
  while (!current->need_resched) idle();
  schedule();
}
```


VA LINUX SYSTEMS JAPAN

schedule()関数 (1/5)

■ 1. SCHED_RRの処理

```
if (unlikely(prev->policy == SCHED_RR))
  if (!prev->counter) {
          prev->counter = NICE_TO_TICKS(prev->nice);
          move_last_runqueue(prev);
  }

#define TICK_SCALE(x)        ((x) >> 2)
#define NICE_TO_TICKS(nice) (TICK_SCALE(20-(nice))+1)
```

■ task_structのcounterメンバについては後で

VA LINUX SYSTEMS JAPAN

schedule()関数 (2/5)

■実行中のプロセスの状態変更

```
switch (prev->state) {
  case TASK_INTERRUPTIBLE:
          if (signal_pending(prev)) {
                  prev->state = TASK_RUNNING;
                  break;
          }
  default:
          del_from_runqueue(prev);
  case TASK_RUNNING:;
}
prev->need_resched = 0;
```


VA LINUX SYSTEMS JAPAN

schedule()関数 (3/5)

■次に実行するプロセスの選択

```
repeat_schedule:
    next = idle_task(this_cpu);
    c = -1000;
    list_for_each(tmp, &runqueue_head) {
      p = list_entry(tmp, struct task_struct, run_list);
      if (can_schedule(p, this_cpu)) {
              int weight = goodness(p, this_cpu, prev->active_mm);
              if (weight > c)
                      c = weight, next = p;
      }
    }
```

■goodness()関数は後で

VA LINUX SYSTEMS JAPAN

schedule()関数 (4/5)

■コンテキストスイッチの準備

```
sched_data->curr = next;
task_set_cpu(next, this_cpu);

if (unlikely(prev == next)) {
   /* We won't go through the normal tail, so do this by hand */
   prev->policy &=  SCHED_YIELD;
   goto same_process;
}

kstat.context_swtch++;
```

schedule()関数 (5/5)

- コンテキストスイッチ

```
    switch_to(prev, next, prev);
    prev->policy &=   SCHED_YIELD;

same_process:
    if (current->need_resched)
      goto need_resched_back;            // schedule()の入口
    return;
```

- switch_to()から帰ってきたときの状態に注意

goodness()関数 (1/2)

- 前半部

```
int goodness(struct task_struct * p, int this_cpu, struct mm_struct
    *this_mm)
{
    int weight;

    weight = -1;
    if (p->policy & SCHED_YIELD)
      goto out;
```

- SCHED_YIELD 自発的にCPUを明け渡している (sched_yield(2)システムコール)

VA LINUX SYSTEMS JAPAN

goodness()関数 (2/2)

■後半部

```
    if (p->policy == SCHED_OTHER) {    // 通常(non-RT)プロセス
      weight = p->counter;
      if (!weight)
              goto out;
      if (p->mm == this_mm || !p->mm)   // 同一メモリ空間 (スレッド)
              weight += 1;
      weight += 20 - p->nice;
      goto out;
    }
    weight = 1000 + p->rt_priority;
out:
    return weight;
```


VA LINUX SYSTEMS JAPAN

プロセスアカウンティング (1/2)

- counterメンバってなんだっけ →grep
- update_process_times() (timer.c)でカウントされている
- UPのとき、do_timer()から呼ばれている
- do_timer(): タイマー処理。1秒間にHZ回の割合で定期的に呼ばれる

```
void do_timer(struct pt_regs *regs)
{
    (*(unsigned long *)&jiffies)++;
    update_process_times(user_mode(regs));
    ソフト割り込み関連処理 (略)
}
```


VA LINUX SYSTEMS JAPAN

プロセスアカウンティング (2/2)

- update_process_times

```
if (--p->counter <= 0) {
        p->counter = 0;
        if (p->policy != SCHED_FIFO) {
                p->need_resched = 1;
        }
}
if (p->nice > 0)
        kstat.per_cpu_nice[cpu] += user_tick;
else
        kstat.per_cpu_user[cpu] += user_tick;
kstat.per_cpu_system[cpu] += system;
```


VA LINUX SYSTEMS JAPAN

プリエンプト処理

- need_reschedメンバーってなんだっけ? →grep
- 決まり文句

```
if (current->need_resched)
    schedule();
```

- 登場個所: idleループ、システムコール後、softirq処理後など
- need_reschedの設定: wake_up()の延長(reschedule_idle)、fork()、sched_setscheduler(2)…

VA LINUX SYSTEMS JAPAN

待合わせ処理

■ 例: wait_on_inode

```
    add_wait_queue(&inode->i_wait, &wait);
repeat:
    set_current_state(TASK_UNINTERRUPTIBLE);
    if (inode->i_state & I_LOCK) {
      schedule();
      goto repeat;
    }
    remove_wait_queue(&inode->i_wait, &wait);
    current->state = TASK_RUNNING;
```

■ この間いつI_LOCKが落ちてwakeupされても構わない

VA LINUX SYSTEMS JAPAN

Linuxスケジューラの特徴

- nice値→タイムスライス値、タイムスライス値→実行優先度の単純な計算式と単一のリスト構造によるランキュー (キュー?)
- タイムスライス値を使い切るまで原則的にプリエンプトしない
- 単純なリアルタイムスケジューリング機能
- コンテキストスイッチのたびにランキューをなめるアルゴリズム→キューが短いのが前提
- 対話型プロセスに対するボーナスがない
- 独特のwait queueデータ構造

VA LINUX SYSTEMS JAPAN

スケジューラに関するトピック

- Linux-2.5でのスケジューラ強化
  - O(1)スケジューラ
  - カーネルレベルプリエンプション
- 他のUnixのスケジューラ
  - Solaris
  - 4.4BSD

VA LINUX SYSTEMS JAPAN

O(1)スケジューラ (1/10)

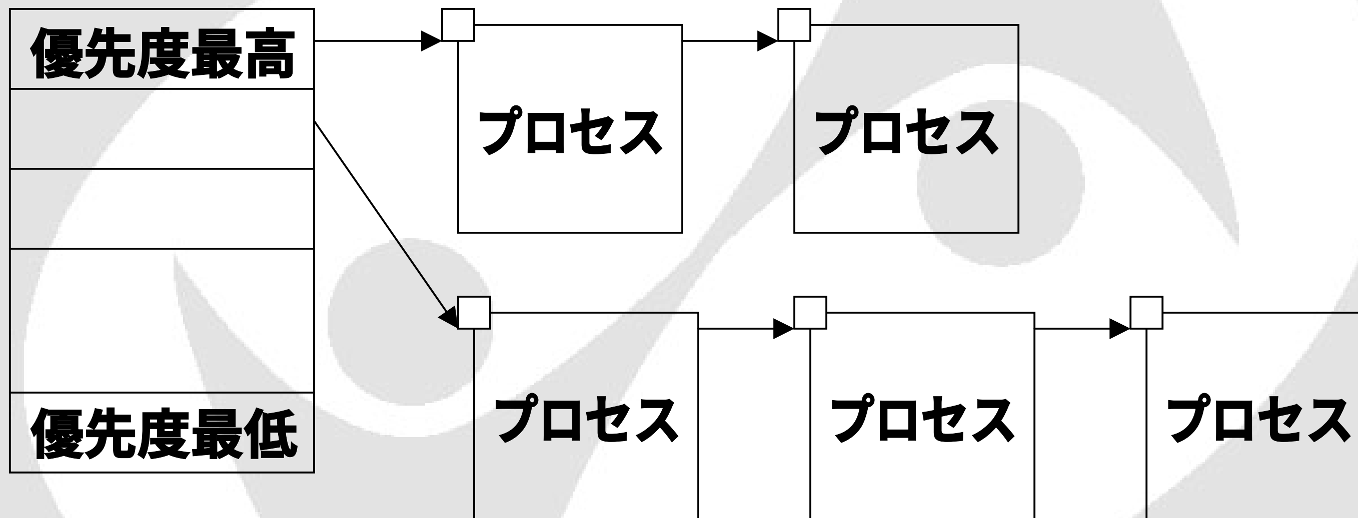
■データ構造

```
static struct runqueue runqueues[NR_CPUS] _cacheline_aligned;
struct runqueue {
    unsigned long nr_running, nr_switches, expired_timestamp,
                  nr_uninterruptible;
    task_t *curr, *idle;
    struct mm_struct *prev_mm;
    prio_array_t *active, *expired, arrays[2];
    int prev_nr_running[NR_CPUS];
    task_t *migration_thread;
    struct list_head migration_queue;
    atomic_t nr_iowait;
}
```


VA LINUX SYSTEMS JAPAN

O(1)スケジューラ

■ runqueue構造



VA LINUX SYSTEMS JAPAN

O(1)スケジューラ (2/10)

■schedule()関数 1.準備

```
need_resched:
    preempt_disable();
    prev = current;
    rq = this_rq();

    prev->sleep_timestamp = jiffies;
```


VA LINUX SYSTEMS JAPAN

O(1)スケジューラ (3/10)

■ schedule()関数 2.実行中のプロセス関連

```
switch (prev->state) {
case TASK_INTERRUPTIBLE:
  if (unlikely(signal_pending(prev))) {
              prev->state = TASK_RUNNING;
              break;
  }
default:
  deactivate_task(prev, rq);
case TASK_RUNNING:
  ;
}
```


VA LINUX SYSTEMS JAPAN

O(1)スケジューラ (4/10)

- schedule()関数 3. 次のプロセスの選択

```
pick_next_task:
    if (unlikely(!rq->nr_running)) {
      next = rq->idle;
      rq->expired_timestamp = 0;
      goto switch_tasks;
    }
    array = rq->active;
    if (unlikely(!array->nr_active)) {
      rq->active = rq->expired;
      rq->expired = array;
      array = rq->active;
      rq->expired_timestamp = 0;
    }
    idx = sched_find_first_bit(array->bitmap);
    queue = array->queue + idx;
    next = list_entry(queue->next, task_t, run_list);
```


VA LINUX SYSTEMS JAPAN

O(1)スケジューラ (5/10)

■ schedule()関数 4. コンテキストスイッチ

```
switch_tasks:
    clear_tsk_need_resched(prev);
    if (likely(prev != next)) {
        rq->nr_switches++;
        rq->curr = next;
        prepare_arch_switch(rq, next);
        prev = context_switch(rq, prev, next);
        finish_task_switch(prev);
    }
    preempt_enable_no_resched();
    if (test_thread_flag(TIF_NEED_RESCHED))
        goto need_resched;
```


VA LINUX SYSTEMS JAPAN

O(1)スケジューラ (6/10)

■キューからのプロセスの削除

```
static inline void deactivate_task(struct task_struct *p, runqueue_t *rq)
{
    nr_running_dec(rq);
    if (p->state == TASK_UNINTERRUPTIBLE)
      rq->nr_uninterruptible++;
    p->array->nr_active--;
    list_del(&p->run_list);
    if (list_empty(p->array->queue + p->prio))
      __clear_bit(p->prio, p->array->bitmap);
    p->array = NULL;
}
```


VA LINUX SYSTEMS JAPAN

O(1)スケジューラ (7/10)

■キューへのプロセスの追加

```
static inline void activate_task(task_t *p, runqueue_t *rq)
{
    long sleep_time = jiffies - p->last_run - 1;
    if (sleep_time > 0) {
        int sleep_avg = p->sleep_avg + sleep_time;
        if (sleep_avg > MAX_SLEEP_AVG)
                sleep_avg = MAX_SLEEP_AVG;
        if (p->sleep_avg != sleep_avg) {
                p->sleep_avg = sleep_avg;
                p->prio = effective_prio(p);
        }
    }
    enqueue_task(p, rq->active);
    nr_running_inc(rq);
}
```

```
#define MAX_SLEEP_AVG          (10*HZ)
```


VA LINUX SYSTEMS JAPAN

O(1)スケジューラ (8/10)

■ 実行優先度: effective_prio()関数

```
static int effective_prio(task_t *p)
{
    int bonus, prio;
    if (rt_task(p))    return p->prio;
    bonus = MAX_USER_PRIO*PRIO_BONUS_RATIO*p->

      sleep_avg/MAX_SLEEP_AVG/100 -
                 MAX_USER_PRIO*PRIO_BONUS_RATIO/100/2;
    prio = p->static_prio - bonus;
    if (prio < MAX_RT_PRIO)   prio = MAX_RT_PRIO;
    if (prio > MAX_PRIO-1)    prio = MAX_PRIO-1;
    return prio;
}
```

USER_PRIO : 0〜39
PRIO : 0〜139
(RTが0〜99、その他が100〜139

VA LINUX SYSTEMS JAPAN

O(1)スケジューラ (9/10)

■実行優先度の計算式

- sleep時間によるボーナス

(p->sleep_avg/MAX_SLEEP_AVG – 0.5)

   * MAX_USER_PRIO * PRIO_BONUS_RATIO (%)

- これにnice値を優先度に変換した値を加える

p->static_prio - bonus

VA LINUX SYSTEMS JAPAN

O(1)スケジューラ (10/10)

■スリープ時間のカウント: scheduler_tick()

```
void scheduler_tick(int user_ticks, int sys_ticks)
{
    int cpu = smp_processor_id();
    runqueue_t *rq = this_rq();
    task_t *p = current;

(略)
    if (p->sleep_avg)
       p->sleep_avg--;
(略)
}
```


VA LINUX SYSTEMS JAPAN

カーネルレベルプリエンプション

- idleループ、システムコール後、softirq処理後に加えてハード割り込み処理後にもneed_reschedチェック、schedule()呼び出し → いつでもプリエンプト
- Big Kernel Lock削減により実現
- 応答性改善に効果
- プリエンプト抑制: preempt_disable/enable

VA LINUX SYSTEMS JAPAN

他のUnixのスケジューラ (1/2)

- ■ Solarisのスケジューラ
  - ●CPU毎のランキュー(ディスパッチキュー)に加えて、リアルタイムプロセス用のグローバルキュー
  - ●プロセッサのパーティショニング
- ■ 4.4BSDのスケジューラ
  - ●優先度毎のグローバルランキュー (O(1)スケジューラに似ている)、毎秒全プロセスの優先度を再計算
  - ●FreeBSD-5.1 (最新版)では2つのスケジューラを選択可能 (SCHED_4BSD、SCHED_ULE)、新スケジューラはCPU毎にキューを用意

VA LINUX SYSTEMS JAPAN

他のUnixのスケジューラ (2/2)

- 起き上がってきたときの優先度を設定できる

  tsleep (void *ident, int priority, const char *wmesg, int timeo)

  → I/O完了後すぐに処理を再開可能
- wait channelはプロセスに1つ
- 待合わせは基本的にmutex lockで → 条件判断と待ちをatomicに行なうため

VA LINUX SYSTEMS JAPAN

おしまい

■ご質問をどうぞ