

Linuxカーネルの読み方

Naoya Ito

担当章

- リバース proxy
- キャッシュ
- **Linux**
- **Web**サーバー



Linux カーネルを読む

- 「よーし、**Linux** カーネル読むぞー」

早速カーネルを読む!

```
Putty
[naoya@karamatsu naoya]% hexdump -C /boot/vmlinuz-2.6.18-8.el5 | head -12
00000000  7f 45 4c 46 02 01 01 ff  eb 3d 00 00 00 00 00 00  |.ELF....=.....|
00000010  03 00 3e 00 01 00 00 00  00 01 00 00 00 00 00 00  |..>.....|
00000020  50 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |P.....|
00000030  00 00 00 00 40 00 38 00  02 00 40 00 00 00 00 00  |...@.8...@....|
00000040  00 00 00 00 00 00 00 66  ea 1c 01 00 00 c0 07 00  |.....f.....|
00000050  01 00 00 00 07 00 00 00  00 20 00 00 00 00 00 00  |.....|
00000060  00 00 00 80 ff ff ff ff  00 00 00 00 00 00 00 00  |.....|
00000070  fc 89 1b 00 00 00 00 00  00 d0 4c 00 00 00 00 00  |.....L.....|
00000080  00 00 20 00 00 00 00 00  04 00 00 00 00 00 00 00  |.....|
00000090  c0 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |.....\.....|
000000a0  00 00 00 00 00 00 00 00  5c 00 00 00 00 00 00 00  |.....|
000000b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |.....|
[naoya@karamatsu naoya]%
[17:53] 0 emacs 1 ack 2 hexdump 3 ssh 4 apache
```

- 「なんとかわいいバイナリ」
- 「**0x000000c4** は俺の嫁」
- 「いやいや、そこは **0x0000006d** でしょう? なおや氏」

... という話ではなく

- ~~Linux~~ カーネルの読み方
- **Linux** カーネルのソースコードの読み方

お話したいこと

- **Linux**カーネルを読む
- カーネルのソースを読んで良かったこと
- **Linux** カーネルに触れたきっかけ
- カーネルソースの読み方

カーネルのソースを
読んで良かったこと

以前の私#1

- 負荷が過剰
 - 「このサーバーはディスクが重そう」
 - (実際どこが正確な原因なんだろう...)
 - (そもそも"負荷"って何だろう...)

以前の私#2

- **load average** が **10.00**
- 「**Google** で調べると**CPU**数で割って **1.00** 以下なら **OK** らしい。」
- (なんで**CPU**数で割るんだろう...)
- (そもそも **load average** って何だ...)

以前の私#3

- **Apache** がアクセス過多
 - 「**prefork** から **worker** にすればマルチスレッド"になって速くなるかも」
 - (なんでスレッドだと速いのかしら...)
 - (とりあえず収まったから良いか...)

以前の私#4

- **I/O** が高くてディスクがもたない
 - 「サーバー増やしてみる? **SCSI** のディスクに
変えてみようか」
 - (**I/O** ってどうやったら減らせるの...?)
 - (**SCSI** にしたところで大して速くならない...)

根拠がない

- 重そう
- **Google**で調べると
- 速くなるかも
- 変えてみよう

根拠がないので対症療法しか取りようがない

現在の私#1

- 負荷が過剰
 - OS の出力する統計情報から原因を特定
- **load average** が **10.00**
 - **load average** を上昇させるプロセスを特定
 - プロセスの挙動からボトルネックを特定

現在の私#2

- **Apache** がアクセス過剰
 - なぜ応答を返せないか原因を統計から特定
 - プロセス **or** スレッドは両者の差異点がどこかを考慮して選択
- **I/O** が高くてディスクが持たない
 - **I/O** 分散計画をキャッシュを前提に立てる

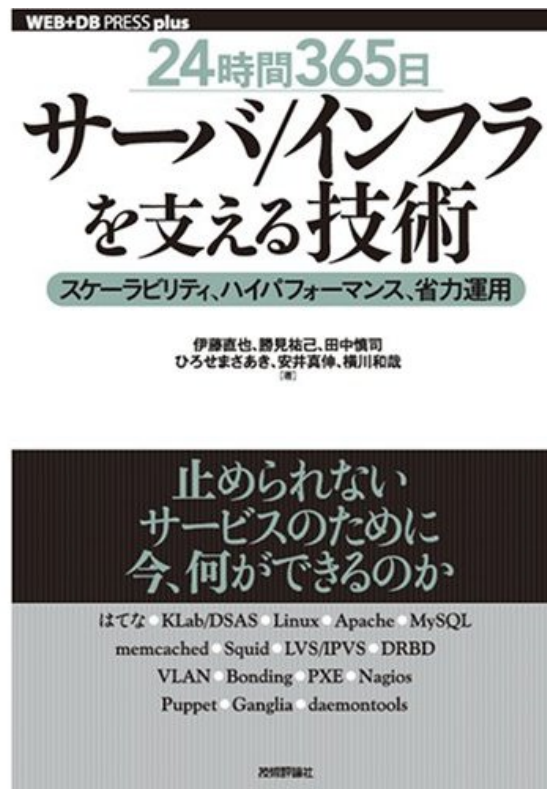
カーネルのソースに触れたことで

- **OS** の動作原理や統計情報の意味をソースコードレベルで理解することができた
- 根拠を持って負荷分散戦略を立てられるようになった
- スケーラブルなアプリケーション設計に自信が持てるようになった

抽象レイヤが透けて見える

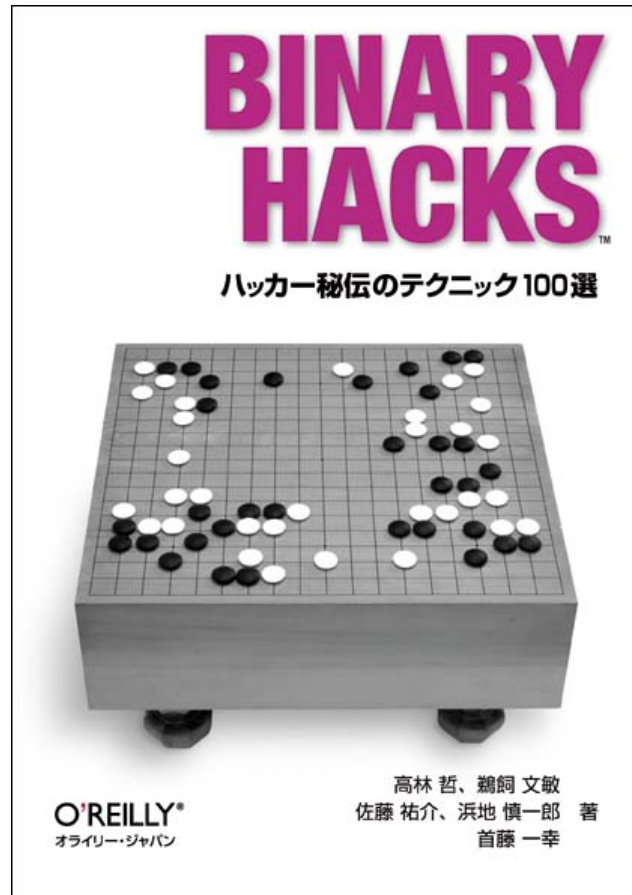
結果

■ 『サーバー/インフラを支える技術』 執筆



Linux カーネルに 触れたきっかけ

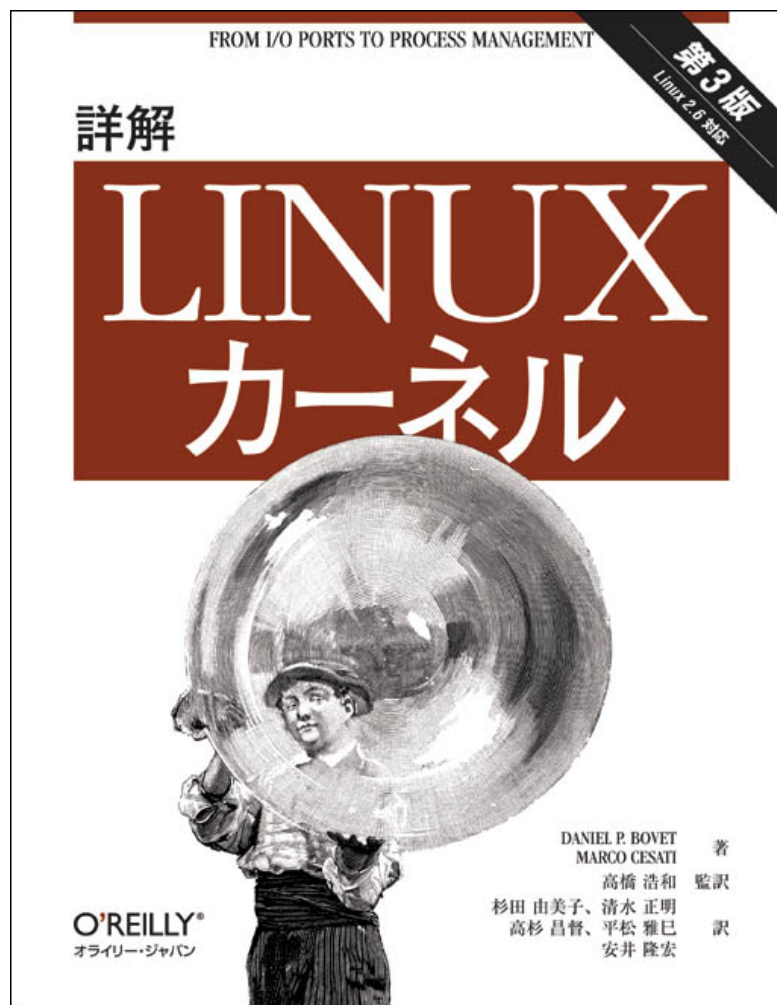
バイナリアンになりたくて



苛立ち

- ローレイヤのことが分からない
 - 分かりたい
- 何に手をつけて良いかが分からない

きっかけ



Linuxカーネル本の概論を読んだ

- **3**版発売決定
 - **2**版をずっと「積ん読」にしていた

**せっかく買ったしこのままお蔵
入りも勿体ない。1章だけでも
読んでみるか...**

知りたかったことが書かれていた

- プロセスとは何か
- スレッドとは何か
- 仮想メモリとは何か
- ファイルがどうキャッシュされるか
- システムコールの裏側

ソースを開く

- **load average** とは何かを調べる
 - 半端無く難しいと思い込んでいた
 - 書籍片手に **grep** したら意外と分かった

Linux カーネルの設計

- モノリシックカーネル
- 複雑な設計パターンはあまりない
 - ややオブジェクト指向 (**VFS** など)
 - ただし量は多い ... **500M**ステップ

カーネル開発者に感謝

- ソースコードの裏には幾多の試行錯誤
 - 開発者 >>>> ソース読者
- 素晴らしい学習教材に感謝

俺流

カーネルソースの読み方

使っている道具

- **GNU Emacs**
- **GNU GLOBAL (gtags)**
- **App::Ack (ack)**
- 書籍数冊
- 紙とペン

Emacs

```
switch_mm(oldmm, mm, next);

if (unlikely(!prev->mm)) {
    prev->active_mm = NULL;
    rq->prev_mm = oldmm;
}
/*
 * Since the runqueue lock will be released by the next
 * task (which is an invalid locking op but in the case
 * of the scheduler it's an obvious special-case), so we
 * do an early lockdep release here:
 */
#ifdef __ARCH_WANT_UNLOCKED_CTXSW
    spin_release(&rq->lock.dep_map, 1, _THIS_IP_);
#endif

/* Here we just switch the register state and the stack. */
switch_to(prev, next, prev);

-EE-:---F1 sched.c 29% (1898,0) (C/lah Gtags Abbrev)---3:17午後 0.68

sigdelsetmask(&set, ~_BLOCKABLE);
spin_lock_irq(&current->sigband->siglock);
current->blocked = set;
recalc_sigpending();
spin_unlock_irq(&current->sigband->siglock);

if (restore_sigcontext(sc, regs, sw))
    goto give_sigsegv;

/* Send SIGTRAP if we're single-stepping: */
if (ptrace_cancel_bpt (current)) {
    siginfo_t info;

    info.si_signo = SIGTRAP;
    info.si_errno = 0;
    info.si_code = TRAP_BRKPT;
    info.si_addr = (void __user *) regs->pc;
    info.si_trapno = 0;
    send_sig_info(SIGTRAP, &info, current);

-EE-:---F1 signal.c 42% (303,0) (C/lah Gtags Abbrev)---3:17午後 0.68

[15:17] 0 emacs 1 perl 2 ssh 3 ssh 4 apache
```

GNU GLOBAL

- ソースコードタグシステム
 - 関数の定義場所、呼び出し場所にジャンプ
 - **Emacs** と連携

```
if (likely(prev != next)) {  
    rq->nr_switches++;  
    rq->curr = next;  
    ++*switch_count;  
  
    context_switch(rq, prev, next);  
} else {  
    ...  
}
```

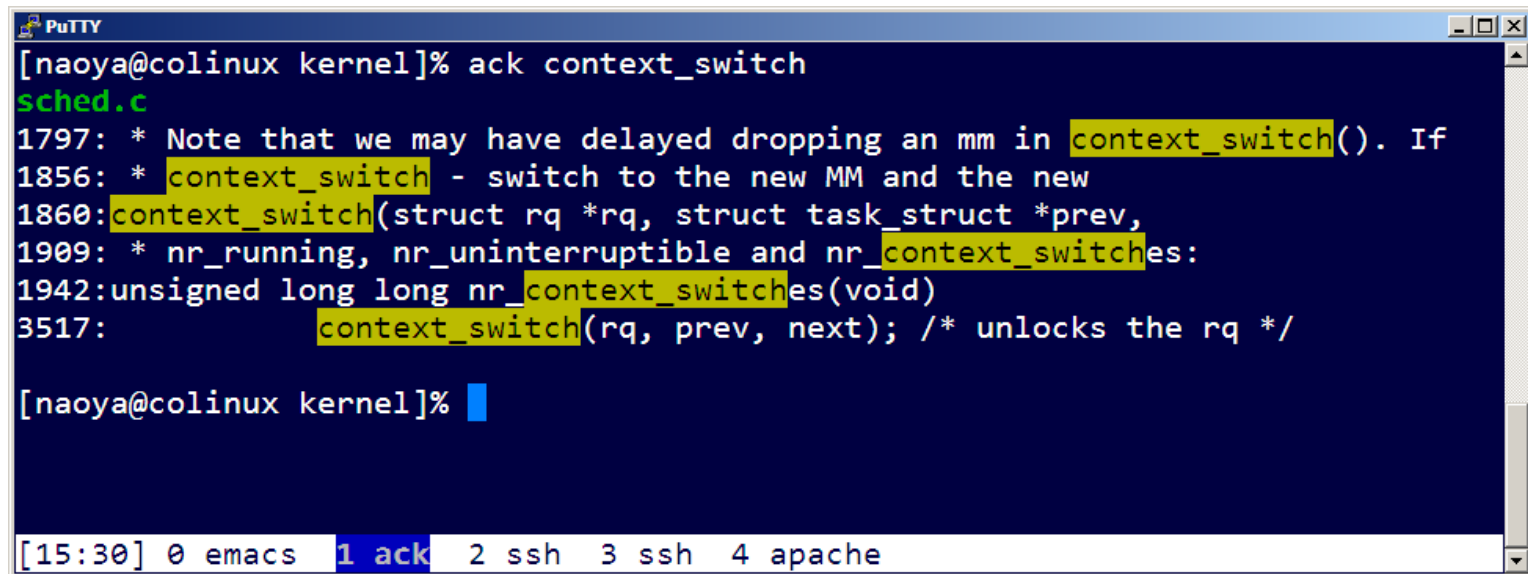
sched.c 3517行目

```
static inline void  
context_switch(...)  
{  
    struct mm_struct *mm, *oldmm;  
  
    prepare_task_switch(rq, prev, next);  
    mm = next->mm;  
    ..  
}
```

sched.c 1860行目

App::Ack

- **grep** よい良いソースコード検索ツール
- `% sudo cpan App::Ack`



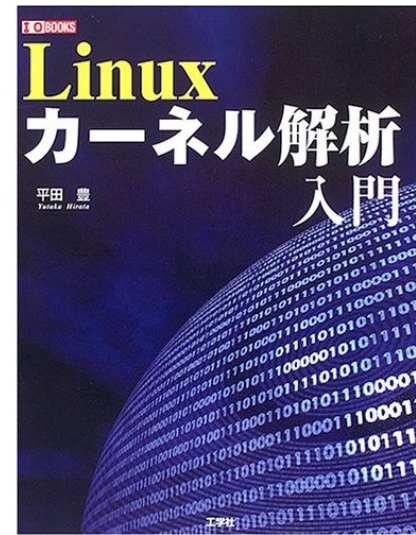
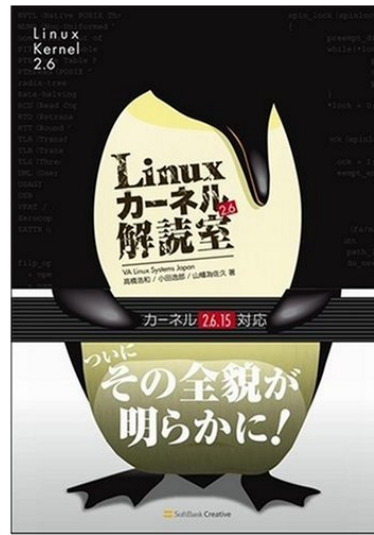
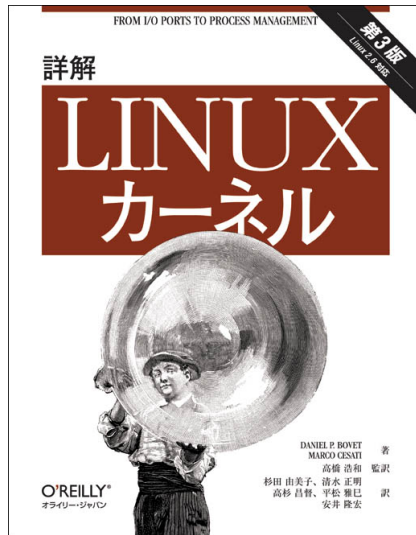
```
PutTY
[naoya@colinux kernel]% ack context_switch
sched.c
1797: * Note that we may have delayed dropping an mm in context_switch(). If
1856: * context_switch - switch to the new MM and the new
1860: context_switch(struct rq *rq, struct task_struct *prev,
1909: * nr_running, nr_uninterruptible and nr_context_switches:
1942: unsigned long long nr_context_switches(void)
3517:     context_switch(rq, prev, next); /* unlocks the rq */

[naoya@colinux kernel]%
```

[15:30] 0 emacs 1 ack 2 ssh 3 ssh 4 apache

書籍数冊

- 別の書籍の同テーマの解説を読む



紙とペン

- 処理の流れをメモしながら進める
 - コールスタックが深いと迷子になりやすい

Linux Kernel Hack Japan

- <http://hira.main.jp/wiki/index.php>
 - "ひら"さんによる膨大な覚書き
 - ひらメソッド
 - 分からない時のヒントに

ソースを読む**5**つのコツ

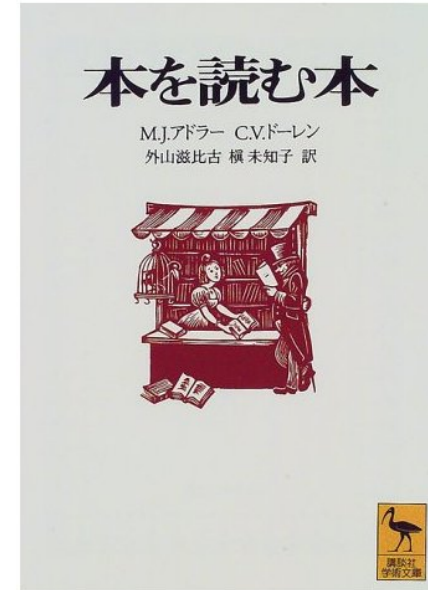
- 読む対象を絞る
- いきなり読まない
- 概観を知った上で読む
- 重要なデータ構造を把握する
- テストプログラムのトレースを活用する

1. 読む対象を絞る

- 隅々まで読もうとしない
 - **500M**ステップも読めない
 - (自分もまだ数%です)
- テーマを決めて読む
 - 「**load average** の計測方法を調べる」
 - 「コンテキストスイッチの仕組みを調べる」
 - 「**select (2)** の実装を調べる」

2. いきなり読まない

- まず全体を把握する
 - ソースツリーの構造
 - 本の目次を読むのと同じ



3. 概観を知った上で読む

- 書籍で知りたい箇所の概観を掴む
 - スケジューラの概要 ... ふむふむ、なるほど
→ ソース読む
 - 大枠が分かっているならば迷子にならない

4. 重要なデータ構造を把握しておく

- 重要なデータ構造
 - `struct task_struct`
 - `struct mm_struct`
 - VFS の `struct file_operations`
 - ...

5. テストプログラムを活用

- 例: スレッド生成周りの処理が知りたい
 - `pthread_create()` するプログラムを作る
 - プログラムを `strace` する
 - システムコールの呼ばれ方 → 実装箇所の当たりが付く

例: **usleep(3)** の精度が知りたい

- <http://d.hatena.ne.jp/naoya/20080122>
- まずテストプログラムで実験
- **strace** で **nanosleep(2)** の呼び出しを確認
- 書籍の索引で **nanosleep** を引く
 - タイマ割り込み周りの章を読む
 - タイマ割り込み周りのコードの当たりがつく
- **ack** や **GNU GLOBAL** を駆使して
sys_nanosleep() の実装/周辺を芋づる式に調べていく (道筋の記録を忘れずに)
- **nanosleep(2)** の母体になっている動的タイマが割り込み周期に依存していることが分かる → **4ms**

読んで分かった必要な知識

- C言語
- gcc 拡張
- GNU開発ツール
- CPU の機能 (x86)
- オブジェクト指向
- 簡単なパターン (Template Method など)
- Writing a Simple File System
 - http://www.geocities.com/ravikiran_uvs/article



番外編

- 手を動かしたい方は...
 - カーネルモジュール開発が面白い
 - <http://d.hatena.ne.jp/naoya/20071201>
 - フィボナッチ数列を計算するデバイス (アホ)

```
% cat /proc/fib  
10  
% cat /dev/fib  
55
```

まとめ

- 巨大なソースに当たるにはコツがあります
 - 堀から埋めていきましょう
- カーネルのソースを読むと負荷分散に強くなります。自信が持てます
- カーネルのソースを読むことで**OS**の動作原理を知ることができます

最後に

- カーネル? 何の役に立つの?
 - 「ただ知りたかった」
 - カーネルへの興味が負荷分散に繋がるとは思っていなかった

**役に立つかどうかは知らなければ分からない。
知りたいと思う動機を大切に。**

Thank you!