

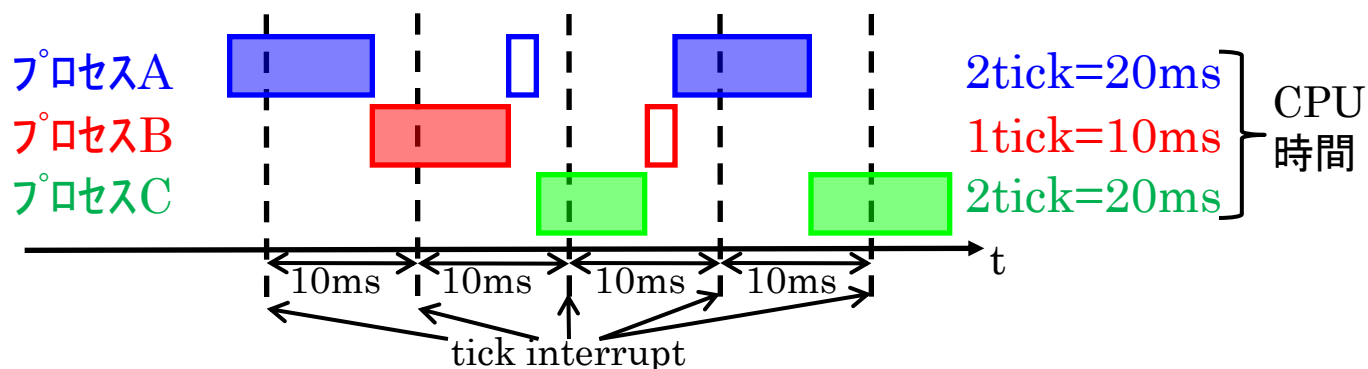
CE Linux Forum Japan Technical Jamboree #34 2010/9/3

tracepoints 利用法 ～cputime モニタリング～

(株)日立製作所 中央研究所
茂岡 知彦 (SHIGEOKA Tomohiko)

- LinuxやUnixではプロセスのCPU時間統計はサンプリング方式
- 組み込み機器ではサンプリングによるCPU時間測定は不正確なことがある
- 性能のチューニングには正確な情報が必要
～「正確に測れないものは改善できない」～
- カーネルトレースは事後にのみ正確なCPU時間を計算できるが、リアルタイムで計測出来ず

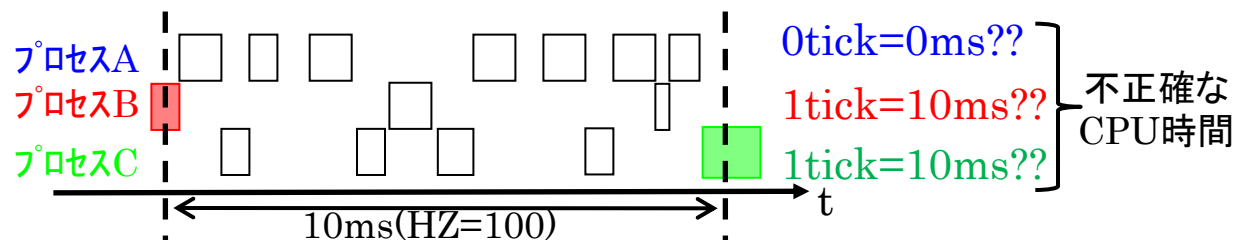
- サンプルングでCPU時間を算出
 - 定期的な割り込み(tick)の発生時に実行中のプロセス
⇒ 割り込み間隔時間CPUを占有(と仮定)
 - プロセス数少、タスクスイッチ頻度少ならばOK



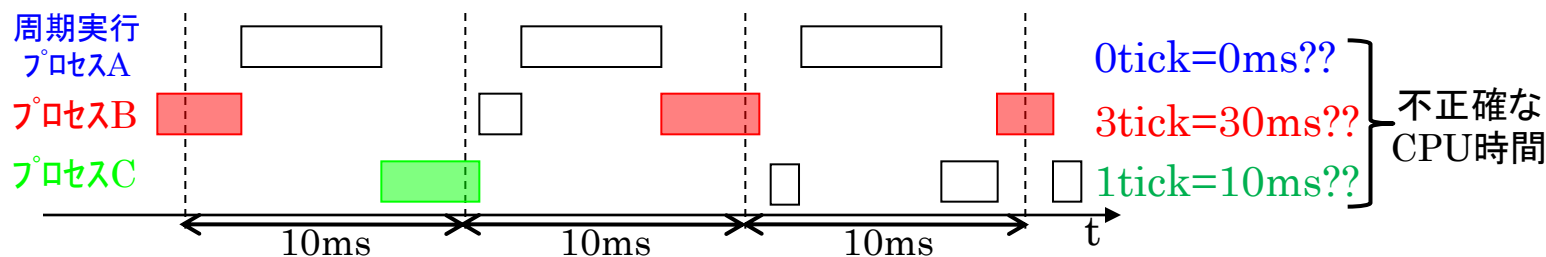
- 組込みや最近のサーバ環境では？
 - プロセス数多、割り込み・タスクスイッチ頻度多
 - 仮想化すると他のVMが実はCPUを使っている

サンプリングによる問題点

- 標本化定理によると。。。 (CPU時間はアナログ信号ではないが)
 - サンプリング周波数は信号周波数の2倍以上
 - HZの1/2以上のタスクスイッチはめずらしくない

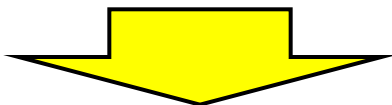


- 組込みでは周期実行タスクがよく使われる
 - 周期によってはサンプリングでまったくヒットしない



⇒タスクスイッチの間隔でCPU時間を測れば正確

- CELFジャンボリー#15 上床@Toshibaさん発表[1]
 - netcpurate(タスクスイッチなどに計測コードを追加,netpoll送信)
⇒(LAN経由で)その場でわかるがカーネルを改造
- CONFIG_VIRT_CPU_ACCOUNTING[2]
 - ppc64,s390などの(仮想化)特定環境用の測定機能
- カーネルトレース(LKST,LTTng, ftraceなど)の利用
 - トレースデータを事後処理して計測
⇒性能がその場でわからない、リアルタイム計測に不向き



tracepoints(v2.6.28から)をフックしてリアルタイム計測
フックから呼ばれるモジュールは本体改造不要

- カーネル内のイベントをフックする標準フレームワーク
 - ソースへの埋め込みが必要[4]
 - 有効・無効や各種条件フィルタは実行時設定

埋め込み箇所:subsys/file.c →
宣言:include/trace/subsys.h

```
#include <linux/tracepoint.h>
DECLARE_TRACE(subsys_myevent,
    TP_PROTO(int arg, struct task_struct *p),
    TP_ARGS(arg,p));
```

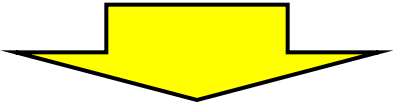
```
#include <trace/subsys.h>
DEFINE_TRACE(subsys_myevent);
void func(void)
{
    ...
    trace_subsys_myevent(arg, task);
    ...
}
```

※赤字はイベント名、
対応コードがマクロで展開

- カーネル内にすでに埋め込み済み
 - 割込み、スケジューラ、システムコール、メモリ管理その他
(v2.6.33.5で150以上のイベントを定義済み)

- 各種トレース用のプローブ(トレーサ)が標準装備
 - 実行時に呼び出すプローブは動的に設定
 - blk,function,function,sched_switch他多数
 - 汎用ログバッファ機能(ringbuffer)有

```
#include <trace/subsys.h>
static void probe_subsys_myevent(int arg, struct task_struct *p)
{
    printk(KERN_INFO "myevent! arg=%d tsk=%p\n", arg, p);
}
static int __init myevent_probe_init(void)
{
    int ret;
    ret = register_trace_subsys_myevent(probe_subsys_myevent);
    ...
}
static void __exit myevent_probe_exit(void)
{
    unregister_trace_subsys_myevent(probe_subsys_myevent);
    tracepoint_synchronize_unregister();
}
```

- 通常のプロセスCPU時間統計
 - tickによるサンプリング(最小時間単位1tick)
 - ユーザ(user)時間、カーネル(sys)時間をカウント
 - 割込みハンドラ実行中は除外
 - その他(realなど)
 - 今回
 - 単純化のためプロセス実行中かどうかのみ考慮
 - リアルタイム観測用に動作中プロセスを毎秒集計(topコマンド風)
- 
sched_switchイベントをフック

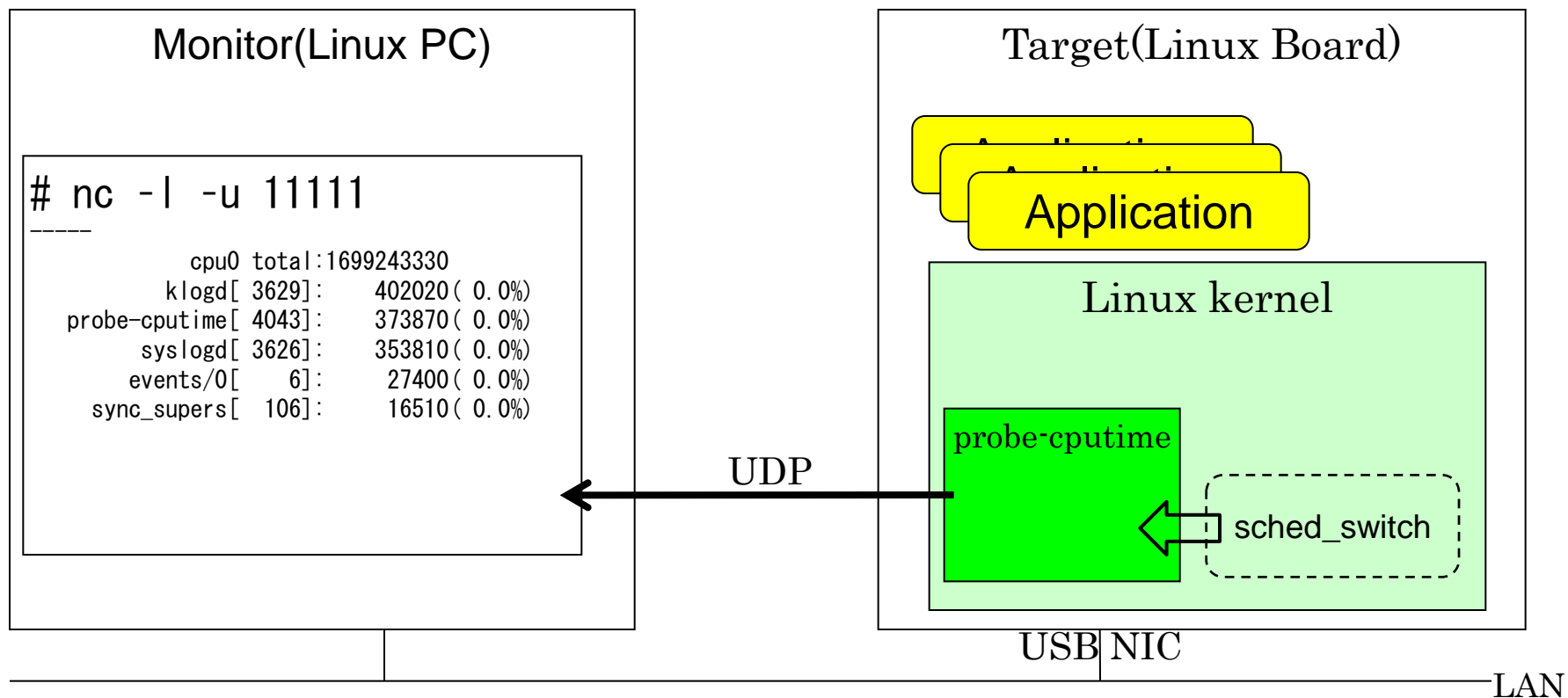
- netpoll/netconsole
 - カーネル内からudpでデータを簡単に送信
 - 割込み不使用、オーバヘッド、副作用が少ない
 - 普通のNICドライバはほぼ対応
- 今回
 - USB接続NICを使用
 - USB接続NICはnetpoll非対応(USBホストが割込み要)
 - カーネルスレッドでソケットを使用[3]
 - MTU以下の少量データをUDP送信しオーバヘッド削減

- ユーザランドAPIとの違い
 - ファイルディスクリプタを使わない

ユーザランドAPI @sys/socket.h	カーネル内API @include/linux/inet.h	説明
socket()	sock_create()	ソケット作成。fdなし。struct socketを使用。
inet_aton()	in_aton()	dot区切り数値文字列をバイナリ値に変換。
connect()	sock->ops->connect()	ソケットに関連するconnect()等と呼ぶ。
sendmsg()	sock_sendmsg()	メッセージ送信。事前にset_fs(KERNEL_DS)する。
recvmsg()	sock_recvmsg()	メッセージ受信。事前にset_fs(KERNEL_DS)する。
close()	sock_release()	ソケットを開放。

- カーネル内のプロセス毎の情報
 - 通常のCPU時間統計はstruct task_structに格納
 - 情報追加はカーネル本体改造(バイナリ非互換)
 - ⇒別のデータ構造でモジュール側で保持
- 毎秒1回集計
 - LAN経由で観測するためカーネルスレッドで毎秒1回集計
 - ソートしてTop10のみ抽出(送信データ削減のため)

- 観察はnc(netcat)コマンド実行のみ
 - text formattingなどはターゲットで実施



probe-cputime.c (1)

```
1 /* Copyright (c) 2010 Hitachi, Ltd. */
2 /* TODO: add error check */
3 #include <linux/version.h>
4 #include <linux/init.h>
5 #include <linux/module.h>
6 #include <linux/kthread.h>
7 #include <linux/string.h>
8 #include <linux/types.h>
9 #include <linux/delay.h>
10 #include <linux/smp_lock.h>
11 #include <linux/sort.h>
12 #include <asm/atomic.h>
13 #include <linux/netdevice.h>
14 #include <linux/ip.h>
15 #include <linux/in.h>
16 #include <linux/inet.h>
17 #include <net/udp.h>
18 #include <trace/events/sched.h>
19
20 #define MODULENAME "probe-cputime"
21 MODULE_DESCRIPTION("Tracepoint module for
22   measuring cputime.");
23 MODULE_LICENSE("GPL");
24
25 #define UDP_HOST "192.168.220.1"
26 #define UDP_PORT 11111
27 #define INTERVAL 1000
28 #define NCPU 2
29 #define NBUFS 2
30 #define NPROCS 256
31 struct cputime_struct {
32     struct task_struct *tsk;
33     pid_t pid;
34     u32 cputime;
35     u32 padding;
36     char comm[TASK_COMM_LEN];
37 } cputime_data[NCPU][NBUFS][NPROCS];
38 static atomic_t cur_buf_idx[NCPU];
39
40 static int cmp_cputime(const void *a, const void *b)
41 { /* for sort function */
42     u32 x = ((struct cputime_struct*)a)->cputime;
43     u32 y = ((struct cputime_struct*)b)->cputime;
44     if (x < y) return 1; if (x > y) return -1; return 0;
45 }
46
47 static void add_cputime(struct task_struct *tsk, pid_t pid,
48     u32 cputime)
49 { /* TODO: USE HashTable */
50     int cpu = smp_processor_id();
51     int buf_idx = atomic_read(&cur_buf_idx[cpu])%NBUFS;
52     int i;
53
54     /* total time */
55     cputime_data[cpu][buf_idx][0].cputime += cputime;
56     if (pid == 0) { /* idle task */
57         cputime_data[cpu][buf_idx][1].cputime += cputime;
58         return;
59     }
60     for (i = 2; i < NPROCS; i++) { /* other task */
61         struct cputime_struct *e =
62             &cputime_data[cpu][buf_idx][i];
63         if (e->pid == pid) {
64             e->cputime += cputime; break;
65         }
66         if (e->tsk == NULL) {
67             e->tsk = tsk; e->pid = pid;
68             strncpy(e->comm, tsk->comm, 16);
69             e->cputime = cputime; break;
70         }
71     }
72 }
```

cputime加算

probe-cputime.c (2)

```
71 static int helper_running = 0;
72 static struct task_struct *helper_thread;
73 static char send_buf[1024];
```

kernel thread function

L75-L165

```
74
75 static int helper_func(void *arg)
76 { /* function for thread */
77     int err, optval;
78     static struct socket *sock;
79     struct sockaddr_in addr;
80
81     lock_kernel();
82     current->flags |= PF_NOFREEZE;
83     daemonize(MODULENAME);
84     allow_signal(SIGKILL);
85     helper_running = 1;
86     unlock_kernel();
87
88     /* setup socket */
89     err = sock_create(AF_INET, SOCK_DGRAM,
90                     IPPROTO_UDP, &sock);
91
92     /* TODO:fix host address */
93     memset(&addr, 0, sizeof(struct
94         sockaddr));
95     addr.sin_family = AF_INET;
96     addr.sin_addr.s_addr =
97     in_aton(UDP_HOST);
98     addr.sin_port = htons(UDP_PORT);
99
100     optval = 1;
101     sock_setsockopt(sock, SOL_SOCKET,
102                    SO_REUSEADDR, (char*)&optval,
103                    sizeof(int));
```

udp socket

```
100 do {
101     int cpu;
102     int buf_lastpos = 0;
103
104     if (signal_pending(current)) break;
105     msleep(INTERVAL);
106     for (cpu = 0; cpu < num_online_cpus(); cpu++) {
107         int i;
108         int bufidx = (atomic_inc_return(&cur_buf_idx[cpu])-
109                     1)%NBUFS;
110
111         /* sort */
112         sort(cputime_data[cpu][bufidx], NPROCS, sizeof(struct
113             cputime_struct), cmp_cputime, NULL);
114
115         /* format text */
116         buf_lastpos += sprintf(send_buf+buf_lastpos,
117                               sizeof(send_buf)-buf_lastpos,
118                               "%23s:%10u%#n", cputime_data[cpu][bufidx][0].comm,
119                               cputime_data[cpu][bufidx][0].cputime);
120         u32 totaltime_h =
121         cputime_data[cpu][bufidx][0].cputime>>10;
122         for (i = 1; i < 11; i++) {
123             u32 cputime_h =
124             cputime_data[cpu][bufidx][i].cputime>>10;
125             if (cputime_h == 0) {
126                 buf_lastpos += sprintf(send_buf+buf_lastpos,
127                                       sizeof(send_buf)-buf_lastpos,
128                                       "%#n"); break;
129             }
130             buf_lastpos += sprintf(send_buf+buf_lastpos,
131                                   sizeof(send_buf)-buf_lastpos,
132                                   "%16s[%5d] :%10u(%2d.%1d%)%#n",
133                                   cputime_data[cpu][bufidx][i].comm,
134                                   cputime_data[cpu][bufidx][i].pid,
135                                   cputime_data[cpu][bufidx][i].cputime,
136                                   (cputime_h*100/totaltime_h),
137                                   (cputime_h*1000/totaltime_h % 10));
138         }
139     }
140 }
```

loop block
L100-L160

probe-cputime.c (3)

```
133     /* clear old buffer */
134     memset(cputime_data[cpu][bufidx], 0,
sizeof(struct cputime_struct)*NPROCS);
135     snprintf
(cputime_data[cpu][bufidx][0].comm, 16,
"cpu%d total", cpu);
136     snprintf
(cputime_data[cpu][bufidx][1].comm, 16,
"idle");
137     }
138     snprintf(send_buf+buf_lastpos,
sizeof(send_buf)-buf_lastpos,
139     "-----%n");
140
141     if (sock->sk) { /* send */
142         mm_segment_t oldfs;
143         int sz;
144         struct iovec iov = {
145             .iov_base = send_buf, .iov_len =
strlen(send_buf),
146         };
147         struct msghdr msg = {
148             .msg_flags = 0, .msg_name = &addr,
149             .msg_namelen = sizeof(struct
sockaddr_in),
150             .msg_control = NULL,
151             .msg_controllen = 0,
152             .msg_iov = &iov, .msg_iovlen = 1,
153             .msg_control = NULL,
154         };
155         oldfs=get_fs(); set_fs(KERNEL_DS);
156         sz = sock_sendmsg(sock, &msg,
iov.iov_len);
157         set_fs(oldfs);
158     }
159 } while (1);
160 }
161
```

```
162     sock_release(sock); sock = NULL;
163     helper_running = 0; helper_thread = NULL;
164     return 0;
165 }
166
167 static u32 last_ctxsw_time[NCPU];
168 static void probe_sched_switch(struct rq *rq, struct
task_struct *prev, struct task_struct *next)
169 { /* tracepoints probe */
170     unsigned long flags;
171     pid_t pid = prev->pid;
172     int cpu = smp_processor_id();
173     u32 now = get_cycles();
174     local_irq_save(flags);
175     if (last_ctxsw_time[cpu]) {
176         add_cputime(prev, pid, (now-last_ctxsw_time[cpu]));
177     }
178     last_ctxsw_time[cpu] = now;
179     local_irq_restore(flags);
180 }
181
182 static int __init probe_cputime_init(void)
183 { /* module init */
184     int ret = 0;
185     if (num_online_cpus() > NCPU) return -ENOMEM;
186     ret = register_trace_sched_switch(probe_sched_switch);
187     WARN_ON(ret); if (ret) return ret;
188     helper_thread = kthread_create(helper_func, NULL, MODULENAME
"helper");
189     kthread_bind(helper_thread, num_online_cpus()-1);
190     wake_up_process(helper_thread);
191     return 0;
192 }
193 module_init(probe_cputime_init);
```

send!

probe for
sched_switch

probe-cputime.c (4)

```
195 static void __exit probe_cputime_exit(void)
196 { /* module clear */
197     int err;
198     if (helper_thread) { /* terminate thread */
199         lock_kernel();
200         err = kill_pid(find_vpid(helper_thread->pid), SIGKILL, 1);
201         unlock_kernel();
202         if (err >= 0) while (helper_running) msleep(100);
203     }
204     unregister_trace_sched_switch(probe_sched_switch);
205     tracepoint_synchronize_unregister();
206 }
207 module_exit(probe_cputime_exit);
```


- 評価環境
 - Atom D510(1.66GHz,1core,noHT), USBLAN(100Mbps)
 - kernel2.6.33.2, CentOS5, HZ100, PREEMPT_VOLUNTARY
- 評価内容
 - データ集計とネットワーク送信のオーバヘッドを測定
 - probe-cputimeなしとあり
 - 負荷なしとあり(dd if=/dev/zero |dd|dd if=/dev/null)

- 高負荷時の影響

- dd if=/dev/zero bs=1K count=100K|dd|dd of=/dev/null

項目	probe-cputimeあり	probe-cputimeなし	影響度
real	7.57s	7.40s	+2.8%
user	0.64s	0.62s	+3.4%
sys	6.91s	6.77s	+2.1%

- ctxswは60798回/s

```
cpu0 total:1699243330
  dd[ 4214]: 885568160 (52.1%) ← 引数なし
  dd[ 4215]: 634279990 (37.3%) ← of=/dev/null
  dd[ 4213]: 178221570 (10.4%) ← if=/dev/zero
  klogd[ 3629]: 402020 ( 0.0%)
  probe-cputime[ 4043]: 373870 ( 0.0%) ← 0.02%
  syslogd[ 3626]: 353810 ( 0.0%)
  events/0[ 6]: 27400 ( 0.0%)
  sync_supers[ 106]: 16510 ( 0.0%)
```

- 低負荷時の影響⇒ほとんどなし

- dd if=/dev/zero bs=1K count=1000K of=/dev/null

項目	probe-cputimeあり	probe-cputimeなし
real	1.40s	1.40s
user	0.44s	0.45s
sys	0.96s	0.94s

- ctxswは36回/s、カーネルスレッドの負荷は同等(0.02%)

```
cpu0 total:1699211580
  dd[10904]:1697940180 (99.9%)
  klogd[ 3629]: 386170 ( 0.0%)
  probe-cputime[10890]: 357770 ( 0.0%) ← 0.02%
  syslogd[ 3626]: 336960 ( 0.0%)
  init[ 1]: 147980 ( 0.0%)
  events/0[ 6]: 24660 ( 0.0%)
  sync_supers[ 106]: 17860 ( 0.0%)
```

- tracepointsの負荷はイベント発生回数次第
 - 超大量タスクスイッチ60000回/sで2%は優秀
- 今回はプロセス数の影響は不明
 - データ構造などの工夫(hashtable等)が必要
 - データ集計のCPU負荷に影響
- 少量のUDPデータ送信は誤差程度

- tracepointsは主要なカーネル内イベントを簡単にフック可能
- tracepointsの負荷もあまり心配ない
- 標準のトレーサ以外にもtracepointsはいろいろ活用できる

- [1] “Linux NetPollの活用”,
http://www.celinuxforum.org/CelfPubWiki/JapanTechnicalJamboree15?action=AttachFile&do=get&target=celf_netpoll.pdf
- [2] “CONFIG_VIRT_CPU_ACCOUNTINGまとめ”,
<http://mkosaki.blog46.fc2.com/blog-entry-453.html>
- [3] “Simple UDP Server”,
http://kernelnewbies.org/Simple_UDP_Server
- [4] “Using the Linux Kernel Tracepoints”,
[\\$KERNELSRC/Documentation/trace/tracepoints.txt](#)