ORACLE®

Solaris の DTrace を使用した性能分析手法

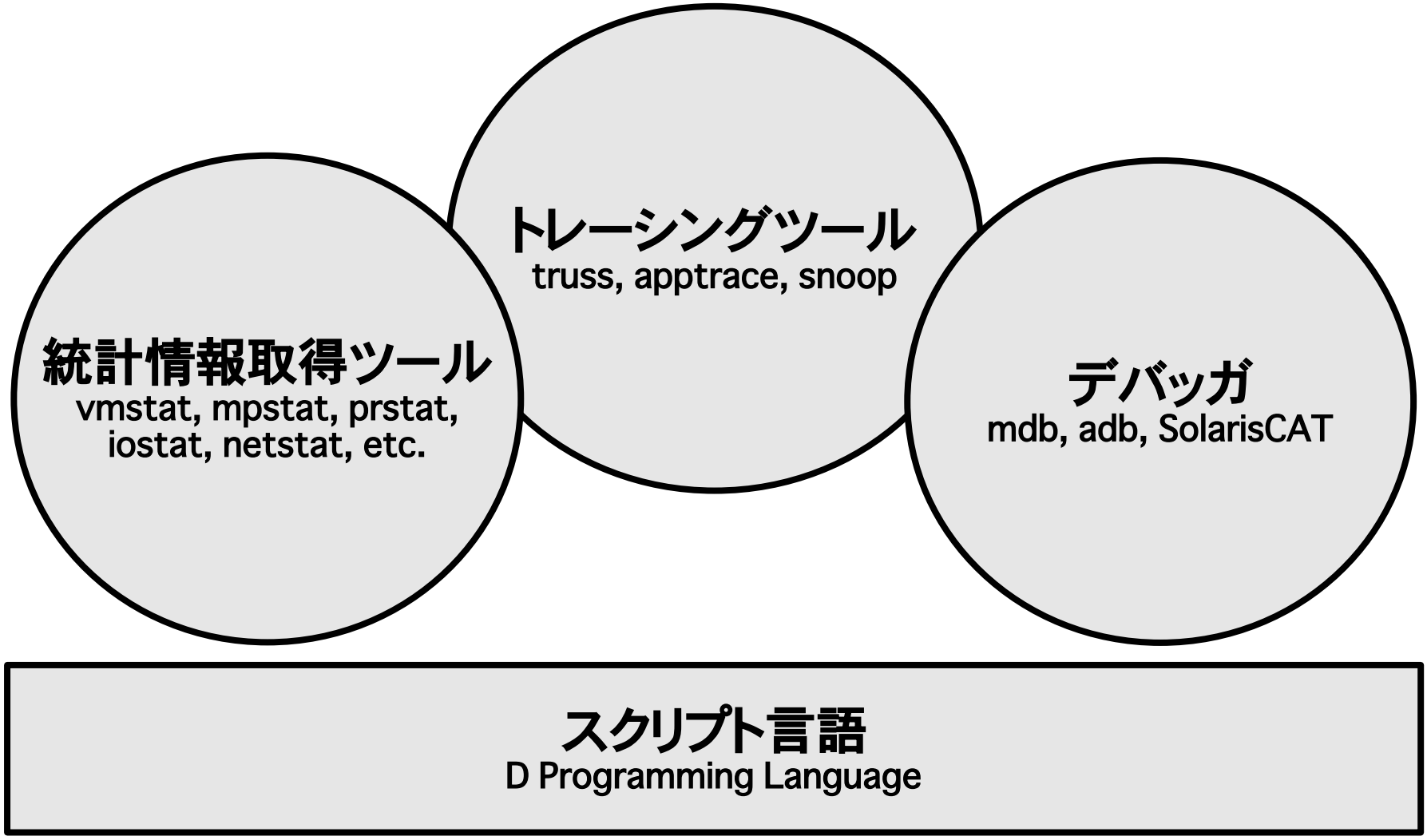
日本オラクル株式会社　システム事業統括
ソリューションスペシャリスト　本間大輔

以下の事項は、弊社の一般的な製品の方向性に関する概要を説明するものです。また、情報提供を唯一の目的とするものであり、いかなる契約にも組み込むことはできません。以下の事項は、マテリアルやコード、機能を提供することをコミットメント（確約）するものではないため、購買決定を行う際の判断材料になさらないで下さい。オラクル製品に関して記載されている機能の開発、リリースおよび時期については、弊社の裁量により決定されます。

DTrace とは?

- DTrace は、トレーシングツール
  - プロセスやシステムで発生したイベント (関数呼び出し、システムコール、I/O 等) を時系列で記録
  - e.g. truss, apptrace, snoop
- DTrace は、統計情報取得ツール
  - 発生したイベントの情報を一定期間毎に集計・記録する
  - e.g. vmstat, mpstat, prstat, iostat, netstat, etc.
- DTrace は、デバッガ
  - 動いているプログラムの詳細情報をドリルダウン
  - e.g. mdb, adb, SolarisCAT
- DTrace は、スクリプト言語
  - D programming language

DTrace とは?



トレーシングツールとしての DTrace

- トレーシングツール
  - プロセスやシステムで発生したイベント（関数呼び出し、システムコール、I/O 等）を時系列で記録する
- プロセスのトレーシング
  - あるプロセスが発行した I/O を時系列に記録したい
  - プロセスが fork する都度、時間を記録したい
- システムのトレーシング
  - 特定の IP アドレス向けに送信されたパケットのログを取る
  - ZFS の関数のコールフローを取得したい

統計情報取得ツールとしての DTrace

- 統計情報取得ツール
  - 発生したイベントの情報を一定期間毎に集計・記録する
- プロセスの統計情報
  - あるプロセスが毎秒何回システムコールを発行しているか
  - あるプロセスが毎秒何 bytes のメモリをアロケートしたか
- システムの統計情報
  - カーネル内の特定の変数の値を 1 秒毎に出力する
  - 1 分間でカーネルタスクが幾つ実行開始されたか数える

デバッガとしての DTrace

- デバッガ
  - 動いているプログラムの詳細情報をドリルダウンする

- 主な機能
  - 関数、メソッドのコールスタックをダンプする : ustack(), jstack(), stack()
  - レジスタの中身をダンプする : uregs[], regs[]
  - 関数の引数をダンプする : argN, args[]
  - 関数がどのパスからリターンしたかを調べる : args[0]
  - 関数の返り値を調べる : args[1]
  - プロセスを停止させる : stop()
  - DTrace には、ソースコードレベルの解析やコアダンプの解析機能はなく、動作中のプログラムを『止めずに解析』する事に特化したデバッガです

スクリプト言語としての DTrace

- DTrace のスクリプト
  - 指定したイベントが発生した時に実行したい処理を登録出来る
  - どんな時に、何を調べたいかを AWK ライクな記法で定義可能
- ex.) 『プロセス ID が 100 のプロセスで bar() という関数が呼ばれたら』、『コールスタックを 20 段ダンプする』

```
#!/usr/sbin/dtrace -qs
pid100::bar:entry
{
    ustack(20)
}
```

DTrace のその他の特徴

- 稼働中のシステムでも使える安全性
  - 注意深く定義されたプローブポイント
  - 専用の仮想マシン上で作動
    - 危険性のあるコードや不正なコードの実行を防ぐ
    - メモリ操作を仮想マシンレベルで管理
    - CPU を占有するビジーループの防止
    - ゼロ除算の検知
    - デバイスがマップされているアドレスへのアクセス防止
    - メモリがマップされていないアドレスへのアクセス防止
- オープンソースでマルチプラットフォーム
  - http://src.opensolaris.org/
  - Solaris, OpenSolaris, Mac OS X, FreeBSD, NetBSD

DTrace の主な適用範囲

- 稼働中のシステムでのトラブルシューティング
  - アプリケーションの動作解析
  - システムの稼働情報収集
  - 再現システムやデバッグモジュール無しで問題解析可能
- パフォーマンスチューニング
  - システムのボトルネック解析
  - アプリケーションの性能プロファイル取得
  - 今までのツールでは取得出来なかったデータが入手可能
- 開発中のプログラムの動的デバッグ
  - プログラムの実行時情報へのアクセス
  - 動作プロファイルの作成
  - テスト用のコードを減らす事が可能

DTrace を使用する上での基礎知識

- DTrace の実行
- DTrace コマンドの舞台裏
- DTrace のプログラムの基本形
- DTrace のプログラムを実行する
- スクリプトをファイルに保存して実行
- probe description とは
- DTrace の主なプロバイダ
- プローブの一覧
- predicate とは
- action statements とは
- DTrace の主なアクションとサブルーチン
- DTrace の主な組み込み変数とマクロ変数
- 収集したデータの集計 (集積体)

DTrace の実行

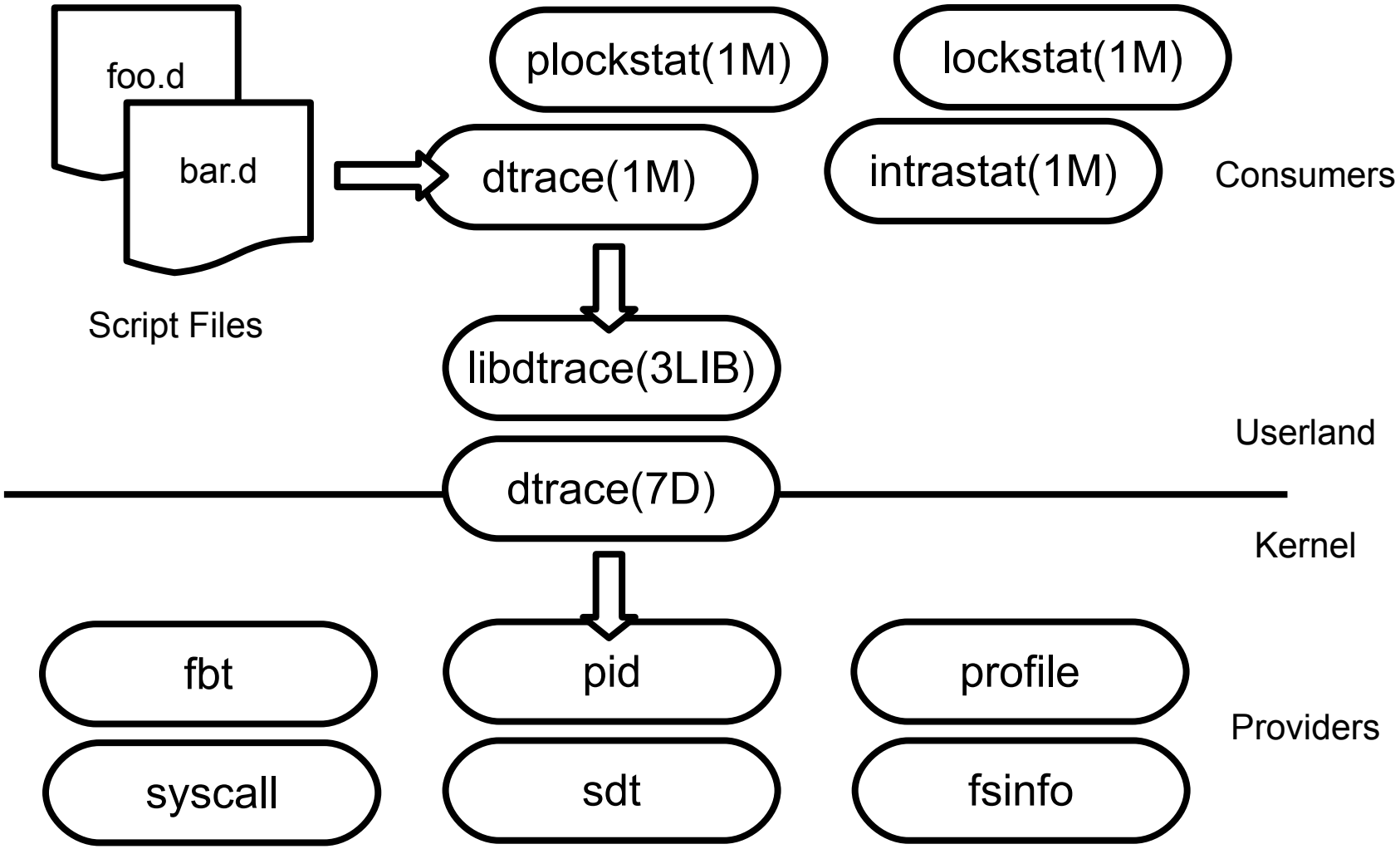
- DTrace を実行するには...
  - /usr/sbin/dtrace (UNIX コマンド)
  - chime という GUI のツールもあります
- DTrace の実行に必要な権限
  - dtrace コマンドの実行は root ユーザか、以下の権限が必要
  - dtrace_proc : プロセスをトレースする権限
  - dtrace_user : dtrace_proc + syscall, profile プロバイダの実行
  - dtrace_kernel : カーネル空間のトレースを実行する為の権限

```
# usermod -K defaultpriv=basic,dtrace_proc,dtrace_user <user_name>
```

dtrace コマンドの舞台裏

- プロバイダ
  - fbt, pid, profile, syscall, etc.
  - DTrace の各々の機能を実装しているカーネル内モジュール
- 仮想デバイス
  - dtrace(7D)
  - /dev/dtrace/dtrace
  - /kernel/drv/sparcv9/dtrace
- ライブラリ
  - libdtrace(3LIB)
  - /usr/lib/libdtrace.so.1

dtrace コマンドの舞台裏（図）



DTrace のプログラムの基本形

- probe descriptions/predicate/{ action statements }
- ex.) Java のシステムコールを抜き出すスクリプト

```
#!/usr/sbin/dtrace -qs

syscall:::entry                 /* probe descriptions */
/execname == “java”/            /* predicate */
{
  printf(“%s\n”, probefunc) /* action statements */
}
```

DTrace のプログラムを実行する

- ワンライナー

```
# dtrace -qn 'probe desc/pred/{action stmts}'
```

  - ex.) Java のシステムコールを抜き出すプログラム

```
# dtrace -qn 'syscall:::entry/execname == “java”/
{printf(“%s\n”, probefunc)}'
```

  - ex.) Hello World

```
# dtrace -qn 'BEGIN { trace(“Hello World”); exit(0) }'
```

  - ex.) 一番短い DTrace プログラム

```
# dtrace -n ':::'
```

スクリプトをファイルに保存して実行

- DTrace スクリプト

```
#!/usr/sbin/dtrace -qs
probe descriptions/predicate/{action statements}
probe descriptions/predicate/{action statements}
```

- シェルスクリプト

```
#!/bin/sh
dscript1='probe descriptions/predicate/{action statements}'
dscript2='probe descriptions/predicate/{action statements}'
dtrace -qn "${dscript1} ${dscript2} ..."
```

probe description とは

- probe description : 監視したいイベントの指定
  - ex.) syscall::read:entry ==> read() システムコール呼び出し
  - ex.) fbt:zfs:zio_read:entry ==> zio_read() 関数呼び出し
  - ex.) proc:::create ==> プロセスの生成
  - ex.) pid100:libc.so.1:malloc:return ==> malloc() からの復帰

- プロバイダ名:モジュール名:関数名:プローブ名　の形式
  - プロバイダ : プローブの実装モジュール
  - モジュール : カーネルモジュール、ライブラリ等
  - 関数 : プログラム内の関数
  - プローブ : 詳細なイベント名

DTrace の主なプロバイダ

- fbt : カーネル内の関数を追跡
- pid : プロセスの関数を追跡
- syscall : システムコールの呼び出しを追跡
- io : Disk I/O の発生を追跡
- plockstat : プロセス内のロックを追跡
- profile : 指定したインターバル毎に処理を起動
- proc : プロセスの生成等を追跡
- sched : スケジューラの動きを追跡
- ip : ネットワークの送受信を追跡

プローブの一覧

- dtrace -l でプローブの一覧を見る事が出来ます
  - 動的に作成されるプローブ以外のプローブをリストします

```
# dtrace -l
   ID   PROVIDER            MODULE                          FUNCTION NAME
    1     dtrace                                                     BEGIN
    2     dtrace                                                     END
    3     dtrace                                                     ERROR
    4        fbt                bl                          bl_open entry
    5        fbt                bl                          bl_open return
    6        fbt                bl                        bl_ioctl entry
    7        fbt                bl                        bl_ioctl return
    8        fbt                bl                     bl_getinfo entry
    9        fbt                bl                     bl_getinfo return
   10        fbt                bl                        bl_open entry
   11        fbt                bl                        bl_open return
   12        fbt                bl                        bl_ioctl entry
   13        fbt                bl                        bl_ioctl return
   14        fbt                bl                           _info entry
...
```

predicate とは

- predicate : 条件式によるフィルター
  - /条件式/　の形式
  - probe description に加えて、更に詳細な条件を設定します
  - ex.) /pid == 100/ ==> プロセス ID が 100 なら...
  - ex.) /i != 0/ ==> 変数 i の値が 0 でなかったら...
  - ex.) /n > 5/ ==> 変数 n の値が 5 より大きかったら...

```
io:::start/execname == "mysqld"/ { do_something }
io:::start/execname == "java"/ { do_other_thing }
```

action statements とは

- action statements : 実行したい処理
  - probe description と predicate に合致したイベントが発生した時に実行したい処理を記述します
  - ex.) { trace(“Hello World”) }
  - ex.) { printf(“%s\n”, some_text); exit(0) }
  - ex.) { i = timestamp }
  - ex.) { @foo[bar] = count() }
  - ex.) { jstack(20) }
  - ex.) { system(“echo foo”) }
- action statements には、DTrace に組み込まれたアクション、サブルーチンの呼び出し、変数の参照、変数への代入、算術演算等が可能です。

DTrace の主なアクションとサブルーチン

- trace() : 端末出力
- printf() : 書式付き出力
- exit() : DTrace プログラムの終了
- ustack() : プロセスのスタックのダンプ
- stack() : カーネルのスタックのダンプ
- jstack() : Java のスタックのダンプ
- tracemem() : メモリの中身のダンプ
- copyin() : ユーザプロセスからデータを読み込む
- copyinstr() : ユーザプロセスから文字列データを読み込む
- system() : UNIX コマンドの実行
- stop() : プロセスを停止させる

DTrace の主な組み込み変数とマクロ変数

- execname
  - プローブが発動した時に稼働していたプロセスの名前
  - /execname == “httpd”/
- timestamp
  - 現在のタイムスタンプ
  - self->ts = timestamp;
- pid
  - プローブが発動した時に稼働していたプロセスの ID
- $target
  - dtrace -p <PID> で渡したプロセス ID が格納される
  - /pid == $target/

組み込み変数とマクロ変数の続き

- curthread
  - プローブ発動時に、その CPU を使用していたスレッド
- probefunc
  - 発動したプローブの関数名
- probename
  - 発動したプローブの名前
- argN, args[]
  - プローブの引数
- その他
  - probeprov, probemod, ppid, uid, gid, psinfo, zonename, vtimestamp, ustackdepth, tid, etc.

収集したデータを集計する（集積体）

- 集積体
  - 連想配列に似たデータ型
  - リアルタイムで自動的にデータが集計され格納される
  - @name[key,key,...] = aggfunc(arg)　の形式
  - 集計方法は集積関数（aggfunc() の部分）で指定（総和、平均、数え上げ等）
  - 集計のキーは key の部分で指定（プロセス名毎に集計したい場合は key に execname を指定する等）
  - 集計済みのデータのみが格納されるため、省スペース
  - 集積体に格納されたデータの出力は printa() アクションを使用

トレースしたデータを集計する（集積体）

- 集積関数
  - count() : 回数の数え上げ
  - sum() : 値の合計
  - avg() : 平均値
  - max() : 最大値
  - min() : 最小値
- ex.) システムコールの発生回数を名前別に集計

```
# dtrace -qn 'syscall:::entry
/execname == “mysqld”/
{ @[probefunc] = count() }'
```

集積体の表示

- 集積体を表示する
  - printa(“%d, %d, %@d”, @agg)
- 集積体の上位 N 番のデータだけを取り出す
  - trunc(@agg, N)
  - 下位 N 番の場合は trunc(@agg, -N)
- ex.) システムコールの発生回数を集計して、上位 3 つのシステムコールを表示する

```
# dtrace -qn 'syscall:::entry/execname == “mysqld”/
{ @[probefunc] = count() }
END{ trunc(@, 3); printa(“%s is called %@d times.\n”, @) }'
```

集積体の出力例

```
# dtrace -qn 'syscall:::entry
{
  @[probefunc] = count()
}
END
{
  trunc(@, 3);
  printa(“%s is called %@d times.\n”, @)
}'
^C
pollsys is called 77 times.
p_online is called 256 times.
ioctl is called 1334 times.
```

DTrace の活用例

- CPU 負荷が高い場合のドリルダウン方法

CPU 負荷が高いとき...

- mpstat コマンドを使用して、CPU が律速要因になっていると確認出来る場合
  - idl が 0 だったり、sys が 20% を超えていたり、icsw や syscl の値が多い場合は要注意です

```
% mpstat 1
...
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0   42   0    0  5493  105 20448 5717 1850  792    0 64786   63  30   0   7
  1   27   0    0  4872   10 20024 5355 1728  823    0 65545   63  29   0   8
  2   42   0    0  4971   19 20767 5444 1781  758    0 63488   60  28   0  12
  3   17   0    0  4836    9 19881 5253 1749  757    0 60687   60  29   0  11
  4   32   0    0  5292   12 21156 6040 1873  804    0 66033   62  30   0   8
  5   16   0    0  5170   18 20603 5845 1809  794    0 65853   63  28   0   9
  6   52   0    0  5104    8 21172 5521 1813  751    0 64467   59  29   0  12
  7   41   0    0  5492   24 21896 5918 1821  773    0 65941   60  29   0  11
```

CPU 負荷が高いとき...

- prstat コマンドを使用して、プロセスの消費している CPU 時間が多い場合
  - “CPU の値” == 100 / “CPU 数” の場合は要注意

```
% prstat -Ln 10 -p `pgrep -x mysqld` 1 | while read i; do echo ${i}; done
PID USERNAME  SIZE   RSS STATE  PRI NICE      TIME  CPU PROCESS/LWPID
6124 mysql    2513M 1594M cpu4     7    5   0:04:08 4.7% mysqld/19
6124 mysql    2513M 1594M cpu5     7    5   0:04:10 4.6% mysqld/14
6124 mysql    2513M 1594M cpu0     6    5   0:04:10 4.5% mysqld/21
6124 mysql    2513M 1594M cpu7     6    5   0:04:09 4.5% mysqld/16
6124 mysql    2513M 1594M sleep    7    5   0:04:08 4.5% mysqld/20
6124 mysql    2513M 1594M run      6    5   0:04:07 4.5% mysqld/22
6124 mysql    2513M 1594M run      6    5   0:04:08 4.5% mysqld/23
6124 mysql    2513M 1594M cpu3     7    5   0:04:09 4.5% mysqld/13
6124 mysql    2513M 1594M run      6    5   0:04:08 4.5% mysqld/25
6124 mysql    2513M 1594M run      6    5   0:04:06 4.4% mysqld/18
Total: 1 processes, 26 lwps, load averages: 12.64, 11.31, 6.66
```

CPU 負荷の主な要因

- ユーザ関数
  - (mpstat で usr が高い場合)
  - 最も呼び出し回数の多い関数を調べる
  - 最も時間を消費した関数を見つける
    - inclusive CPU time
    - exclusive CPU time
- カーネルサービス
- システムコール

pid プロバイダ

- プロセスの関数呼び出しを調査したい場合は pid プロバイダを使用します
- 主なプローブ
  - entry : 関数の呼び出しを捕捉します
  - return : 関数からの復帰を捕捉します
  - 殆どの関数に関して entry/return のプローブが自動的に作成され、pidプロセス ID:ファイル名:関数名:entry の形式でプローブを呼び出す事が出来ます
- 主な変数
  - probefunc : pid プロバイダの probefunc には関数名が入ります
  - argN : entry プローブの場合は、関数の引数
  - arg0 : return プローブの場合は、関数がどこから return したか
  - arg1 : return プローブの場合は、関数の返り値

最も呼び出し回数の多い関数を調べる

- pid プロバイダの中で、probefunc 毎に集計します

```
# dtrace -qn 'pid$target:::entry { @[probemod, probefunc] = count() }
tick-10sec {trunc(@, 20); printa("%15s\t%30s\t\t%@d\n", @); exit(0)}' \
-p `pgrep -x mysqld`
...
      libc.so.1                  clear_lockbyte
      libc.so.1              mutex_unlock_queue
         mysqld                  dfield_get_ty
         mysqld            dtuple_get_nth_fie
         mysqld                  buf_frame_alig
         mysqld              dict_table_is_comp
         mysqld                   ut_align_down           9863
         mysqld                 mach_read_from_1          1073076
         mysqld              dict_field_get_col          1636696
         mysqld        dict_index_get_nth_field          1681825
         mysqld                 mach_read_from_2          1750883
```

mach_read_from_2() が
1750883 回

mach_read_from_2 の原因は?

- pid プロバイダで mach_read_from_2() 関数の呼び出しを補足して、ustack() でスタックをダンプ

```
# dtrace -qn '
pid$target::mach_read_from_2:entry
{
  @[ustack(20)] = count()
}
tick-10sec
{
  trunc(@, 3);
  exit(0)
}' -p `pgrep -x mysqld` | c++filt
```

mach_read_from_2 の原因は?

- 実行結果...

```
# dtrace -qn 'pid$target::mach_read_from_2:entry { @[ustack(20)] = count() }
tick-10sec { trunc(@, 3); exit(0) }' -p `pgrep -x mysqld` | c++filt
...
              mysqld`mach_read_from_2
              mysqld`page_rec_get_n_recs_before+0x11a
              mysqld`btr_cur_search_to_nth_level+0xc17
              mysqld`btr_estimate_n_rows_in_range+0x439
              mysqld`unsigned long long
ha_innobase::records_in_range(unsigned,st_key_range*,st_key_range*)+0x2c3
              mysqld`unsigned long long
check_quick_keys(PARAM*,unsigned,SEL_ARG*,unsigned char*,unsigned,int,unsigned char*,unsigned,int)+0x7c8
              mysqld`TRP_RANGE*get_key_scans_params(PARAM*,SEL_TREE*,bool,bool,double)+0x398
              mysqld`int SQL_SELECT::test_quick_select
(THD*,Bitmap<64U>,unsignedlong long,unsigned long long,bool)+0x10dc
              mysqld`int mysql_update
(THD*,TABLE_LIST*,List<Item>&,List<Item>&,Item*,unsigned,st_order*,unsigned long long,enum_duplicates,bool)+0x5ef
              mysqld`int mysql_execute_command(THD*)+0x1c58
              mysqld`bool Prepared_statement::execute(String*,bool)+0x2a0
              mysqld`bool Prepared_statement::execute_loop(String*,bool,unsignedchar*,unsigned char*)+0x4f2
              mysqld`void mysql_stmt_execute(THD*,char*,unsigned)+0xd9
              mysqld`bool dispatch_command(enum_server_command,THD*,char*,unsigned)+0xe36
              mysqld`bool do_command(THD*)+0x105
              mysqld`handle_one_connection+0x3b6
              libc.so.1`_thrp_setup+0x9b
              libc.so.1`_lwp_start
```

mach_read_from_2 の原因は?

```
# dtrace -qn 'pid$target::mach_read_from_2:entry
{ @[ustack(20)] = count() }
tick-10sec { trunc(@, 3); exit(0) }' -p `pgrep                    c++filt
...
              mysqld`mach_read_from_2
              mysqld`page_rec_get_n_recs_befo
              mysqld`btr_cur_search_to_nth_le
              mysqld`btr_estimate_n_rows_in_ra
              mysqld`unsigned long long ha_i                          ()
              mysqld`unsigned long long ch    _quick_
              mysqld`TRP_RANGE*get_key_scans_params()
              mysqld`int SQL_SELECT::test_quick_select()
              mysqld`int mysql_update()
              mysqld`int mysql_execute_command()
              mysqld`bool Prepared_statement::execute()
              mysqld`bool Prepared_statement::execute_loop()
              mysqld`void mysql_stmt_execute()
              mysqld`bool dispatch_command()
              mysqld`bool do_command()
              mysqld`handle_one_connection+0x3b6
              libc.so.1`_thrp_setup+0x9b
              libc.so.1`_lwp_start
```

prepared statement の UPDATE からの SELECT

最も時間を消費した関数を見つける

- inclusive CPU time
  - 関数内で呼び出している他の関数（子関数）の実行時間も含んだ (inclusive) 実行時間
  - foo() が bar() を呼び出している場合に、bar() の実行時間も foo() の実行時間の一部とする
  - main() の実行時間が一番大きく出る
- exclusive CPU time
  - 関数内で呼び出している他の関数（子関数）の実行時間を除いた (exclusive) 実行時間
  - foo() が bar() を呼び出している場合に、bar() の実行時間は foo() の実行時間に含めない
  - 小さな関数に分割された処理が埋もれ易い

inclusive CPU time

- pid プロバイダの entry/return で timestamp の差分を取ります
  - 再帰関数やコンパイラの最適化に影響を受けます

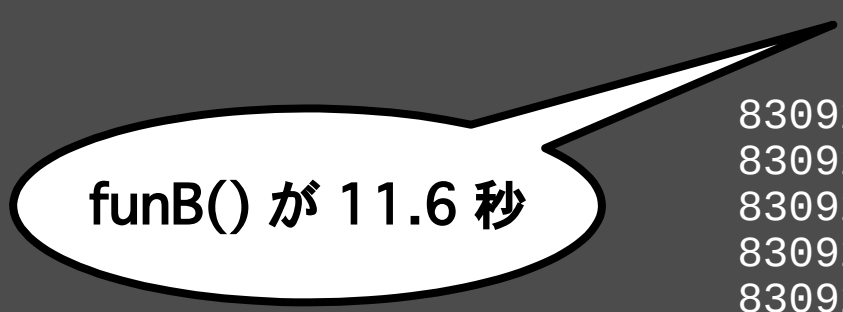
```
dtrace -qn '
pid$target:::entry
{
  self->ts[probefunc] = timestamp;
}
pid$target:::return
{
  @time[probefunc] = sum(timestamp - self->ts[probefunc]);
}
END
{
  printa(@time);
}' -p `pgrep -n mysqld`
```

inclusive CPU time（実行結果）

- 単位はナノ秒です

```
# dtrace -qn '
pid$target:::entry{ self->ts[probefunc] = timestamp;}
pid$target:::return{
  @time[probefunc] = sum(timestamp - self->ts[probefunc]);
}
END{ trunc(@time, 10); printa(@time);}' -c ./a.out

  sleep                                                 10023117940
  funB                                                  11636520954
  funC                                                  11636544440
  rtld_db_preinit                                   8309240860342215
  rd_event                                          8309240861228652
  memmove                                           8309240861722816
  rtld_db_postinit                                  8309240862390474
  setup                                             8309240862429679
  _setup                                            8309240862441209
  main                                              8309252499334308
```



exclusive CPU time

- ustackdepth 変数を利用して、スタックの深さ毎に CPU の消費時間を集計します

```
# dtrace -qn '

pid$target:::entry {
  self->ts[ustackdepth] = timestamp;
  self->el[ustackdepth] = 0;        /* elapsed time */
  self->ad[ustackdepth] = 0;        /* adjustment */
}

pid$target:::return {
  self->el[ustackdepth +1] = timestamp - self->ts[ustackdepth +1];
  self->ad[ustackdepth] -= self->el[ustackdepth +1];
  @time[probefunc] = sum(self->el[ustackdepth +1] + self->ad[ustackdepth +1]);
}

END { printa(@time); }' -p `pgrep -n mysqld`
```

exclusive CPU time（実行例）

- 単位はナノ秒です

```
# dtrace -qn '
pid$target:::entry {
  self->ts[ustackdepth] = timestamp;
  self->el[ustackdepth] = 0;  /* elapsed time */
  self->ad[ustackdepth] = 0;  /* adjustment */
}
pid$target:::return {
  self->el[ustackdepth +1] = timestamp - self->ts[ustackdept    1];
  self->ad[ustackdepth] -= self->el[ustackdepth +1];
  @time[probefunc] = sum(self->el[ustackdepth +1] + self->ad    stackdepth +1]);
}
END { trunc(@time, 5); printa(@time); }' -c ./a.out

  funB                                                   656253
  sleep                                                 1016151
  funA                                               1601471452
  ___nanosleep                                      10007146550
  rtld_db_preinit                                8308230445500287
```

funA() が 1.6 秒

CPU の解析に関するまとめ

- DTrace を使用すると、CPU を誰がどの様に使用しているか、詳細にドリルダウンする事が出来ます
- DTrace はシステムコール、カーネルサービス、ユーザプロセス等、CPU を消費する処理の全てを解析する事が出来ます
- ユーザプロセスの解析には DTrace の pid プロバイダを使用します

おわりに

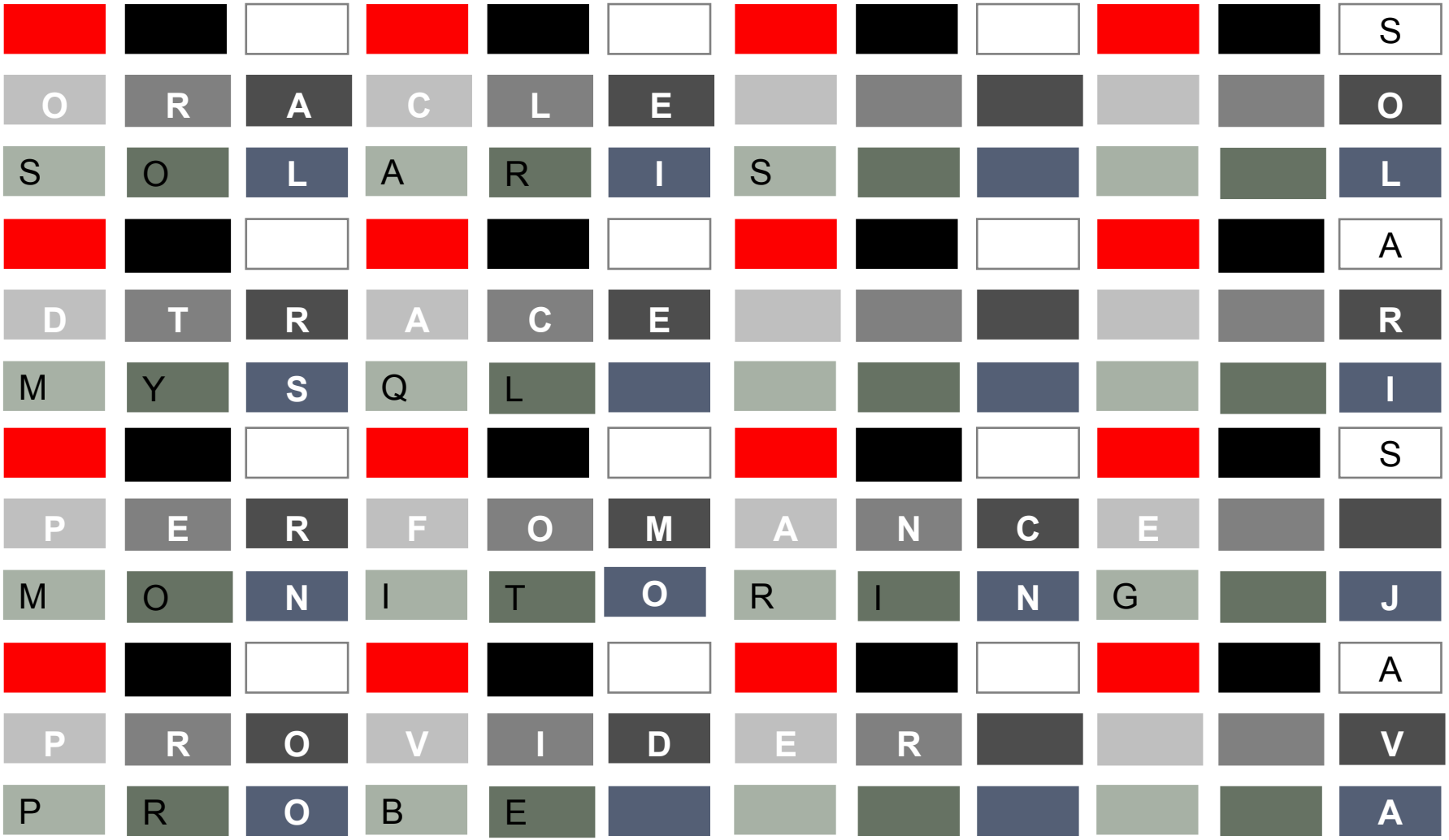
- DTrace の基本的な機能と、その使い方をご紹介しました。
- 本資料末尾に CPU 以外の解析方法も掲載してありますので、是非ご覧下さい。
- DTrace は Solaris 10 以降の環境であれば漏れなく使用可能です。是非、お試し下さい！

DTrace Toolkit

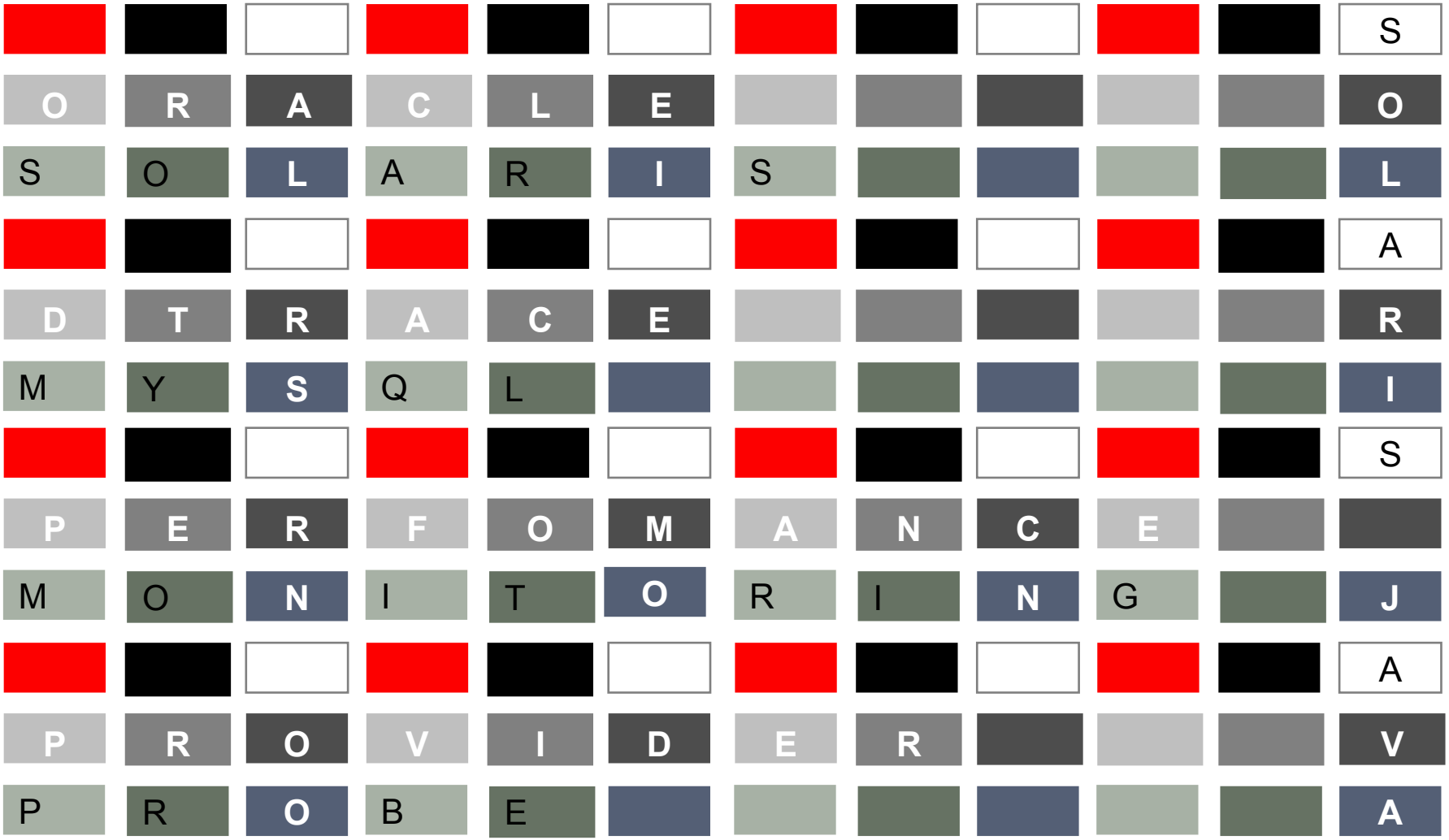
- http://hub.opensolaris.org/bin/view/Community+Group+dtrace/dtracetoolkit
- 200 以上の DTrace のスクリプトが入っており、そのままでも使えますし、スクリプトのサンプルとしても大変参考になります

```
# ls
anonpgpid.d       j_who.d           pridist.d         sh_stat.d
bitesize.d        js_calldist.d     procsystime       sh_syscalls.d
connections       js_calls.d        putnexts.d        sh_syscolors.d
cpudists          js_calltime.d     py_calldist.d     sh_wasted.d
cputimes          js_cpudist.d      py_calltime.d     sh_who.d
cputypes.d        js_cputime.d      py_cpudist.d      shellsnoop
cpuwalk.d         js_execs.d        py_cputime.d      shortlived.d
crash.d           js_flow.d         py_flow.d         sigdist.d
creatbyproc.d     js_flowinfo.d     py_flowinfo.d     stacksize.d
...
```

Q & A



ご清聴ありがとうございました



ORACLE

あなたにいちばん近いオラクル

Oracle Direct

まずはお問合せください

Oracle Direct　検索

システムの検討・構築から運用まで、ITプロジェクト全般の相談窓口としてご支援いたします。
システム構成やライセンス/購入方法などお気軽にお問い合わせ下さい。

Web問い合わせフォーム

専用お問い合わせフォームにてご相談内容を承ります。

http://www.oracle.co.jp/inq_pl/INQUIRY/quest?rid=28

※フォームの入力には、Oracle Direct Seminar申込時と同じ
ログインが必要となります。
※こちらから詳細確認のお電話を差し上げる場合がありますので、ご登録さ　れている連絡先が最新のものになっているか、ご確認下さい。

フリーダイヤル

0120－155－096

※月曜~金曜　9:00~12:00、13:00~18:00
（祝日および年末年始除く）

付録

- DTrace を使用する上での Tips
- DTrace の活用例 (続き)

性能解析に必須のコマンドランキング

- vmstat : システム全体の統計
- mpstat : CPU の統計
- iostat : Disk I/O の統計
- prstat : プロセスのアクティビティの統計
- dtrace : 多機能動的解析
- dladm : NIC の統計
- netstat : ネットワークの統計
- snoop : ネットワークのトレース
- truss : プロセスのトレース
- kstat : 全ての統計情報の情報もと
- plockstat, lockstat, intrstat, etc.

勝手に

vmstat, mpstat,
iostat, prstat の
見方を把握したら
次は DTrace!!

DTrace 小史

- 2001/10 : プロジェクトの開始
  - 開発開始から、ほぼ 10 年が経っています
- 2003/11 : Solaris Express 11/03 にてベータリリース
- 2005/03 : Solaris 10 3/05 にて正式リリース
  - 既にプロダクションシステムで 5 年以上の稼働実績
- 2006/09 : The Wall Street Journal's 2006 Technology Innovation Awards の Gold Award を受賞
  - 他にも InfoWorld や Usenix の Award も受賞しています
- 作者は Bryan Cantrill, Mike Shapiro, Adam Leventhal
- DTrace : Dynamic Trace == 動的追跡

DTrace を使用する上での Tips

- プロセスを分析する際の定石
- 時間を測定する
- 定期的に集計する
- スクリプトの開始と終了を補足する
- カーネルの変数にアクセスする
- プログラムの変数にアクセスする
- 関数の引数にアクセスする
- CPU 毎にプローブを実行する
- ファイルディスクリプタからファイル名を取り出す

プロセスを分析する際の定石

- execname でフィルタリング
  - /execname == “mysqld”/
- DTrace と一緒にコマンドを実行する
  - # dtrace <options> -c <command>
  - # dtrace -qn 'syscall:::entry/pid == $target/ { printf("%s\n", probefunc) }' -c "/bin/echo foo"
- 動作中のプロセスのプロセス ID を渡す
  - # dtrace <options> -p <pid>
  - # dtrace <options> -p `pgrep -n mysqld`
- pid でフィルタリング
  - /pid == 100/
  - /pid == $target/

プロセスを分析する際の定石の続き

- ex.) echo コマンドが発行したシステムコールの表示

```
# dtrace -qn 'syscall:::entry/pid == $target/
{ printf("%s\n", probefunc) }' -c "/bin/echo foo"
mmap
foo
munmap
setcontext
getrlimit
getpid
setcontext
sysi86
ioctl
fstat64
write
rexit
```

- pid プロバイダを使用する
  - pid100::somefunc:entry { do something ... }
  - pid$target::anotherfunc:return { dump data ... }

時間を測定する

- 時間の情報を保持している 3 つの変数
  - walltimestamp : 絶対時間 (タイムスタンプ用)
  - timestamp : 相対時間 (経過時間測定用)
  - vtimestamp : 相対 CPU 時間 (CPU 消費時間測定用)
- ex.) %Y と walltimestamp を使用して現在の時刻を表示する

```
# dtrace -qn 'BEGIN
{
  printf("%Y\n", walltimestamp);
  exit(0)
}'
2010 Aug  5 13:49:31
```

定期的に集計する

- 20 秒毎に結果を出力するスクリプト
  - tick-20sec { trace(“Hello”) }
- 20 秒後からトレースを開始するスクリプト
  - tick-20sec { start = 1 }
  - tick-1sec /start == 1/ { do something }
- 20 秒後に終了するトレース
  - tick-20sec { exit(0) }
- ex.) 10 秒後から 20 秒間、2 秒置きに統計を取る

```
# dtrace -qn 'syscall:::entry { @ = count() }
tick-10sec { start = 1 }
tick-30sec { exit(0) }
tick-2sec / start == 1 / { printa(@); trunc(@) }'
```

スクリプトの開始と終了を補足する

- トレース終了時に処理を行う為のプローブ
  - dtrace:::BEGIN
- トレース開始時に処理を行う為のプローブ
  - dtrace:::END
- ex.) 5 秒後に終了して end! と出力するスクリプト

```
# dtrace -qn 'tick-5sec { exit(0) }
END { printf(“end!”) }'
end!
```

カーネルの変数にアクセスする

- カーネルの mod モジュールの sym 変数の値を参照する
  - mod`sym
- カーネル変数を探す

```
# nm /dev/ksyms | grep OBJT
...
[24399] |     25252716|          4|OBJT |GLOB |0    |ABS    |nthread
```

- ex.) カーネルスレッドの数を毎秒表示する

```
# dtrace -qn 'tick-1sec { printf("%d\n", `nthread) }'
705
705
705
^C
```

プログラムの変数にアクセスする

- ex.) 変数 foo の値を取り出す

```
# cat prog01.c
int foo; int main() { foo = 1; }
# gcc prog01.c
# nm ./a.out | grep foo
[63]    | 134613688|       4|OBJT |GLOB |0    |23     |foo
# dtrace -qn 'pid$target::main:
{ printf("%d\n", *(int *)copyin(134613688, sizeof(int)))}' \
-c ./a.out
0
0
0
0
1
1
1
1
```

関数の引数にアクセスする

- ex.) argN 変数から関数の引数にアクセスする

```
# dtrace -qn 'syscall::write:entry
/pid == $target && arg0 == 1/
{ printf("traced: %s\n", copyinstr(arg1)) }' \
-c '/bin/echo "I can see ya!"'
"I can see ya!"
traced: "I
traced: can
traced: see
traced: ya!"
```

CPU 毎にプローブを実行する

- ex.) スケジューリングキューに溜まっている実行可能スレッドの数を CPU 毎に集計する

```
#!/bin/sh
dscript=`psrinfo | awk '{print $1}' | while read i
  do /usr/ucb/echo '
tick-1sec
{ printf("%d\t%d\n", '${i}',
        \`cpu['${i}']->cpu_disp->disp_nrunnable) }'
  done`
dtrace -qn "${dscript}"
```

ファイルディスクリプタからファイル名を取り出す

- ex.) cat コマンドが読み出したファイルのファイル名を調べる
  - read システムコールの第 1 引数はファイルディスクリプタ

```
# dtrace -qn 'syscall::read:entry
/execname == "cat"/
{ printf("%s\n", stringof(fds[arg0].fi_pathname)) }'
/etc/default/init
/etc/default/init
/etc/X11
/etc/X11
/etc/acct
/etc/acct
```

DTrace の活用例

- CPU の負荷が高い場合の調査方法
- ディスク I/O の負荷が高い場合の調査方法
- アプリケーション (MySQL) を調査する
- File System I/O を調査する
- ユーザランドのロック を調査する
- ネットワーク を調査する

CPU 負荷が高いとき...

- mpstat コマンドを使用して、CPU が律速要因になっていると確認出来る場合
  - idl が 0 だったり、sys が 20% を超えていたり、icsw や syscl の値が多い場合は要注意です

```
% mpstat 1
...
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0   42   0    0  5493  105 20448 5717 1850  792    0 64786   63  30   0   7
  1   27   0    0  4872   10 20024 5355 1728  823    0 65545   63  29   0   8
  2   42   0    0  4971   19 20767 5444 1781  758    0 63488   60  28   0  12
  3   17   0    0  4836    9 19881 5253 1749  757    0 60687   60  29   0  11
  4   32   0    0  5292   12 21156 6040 1873  804    0 66033   62  30   0   8
  5   16   0    0  5170   18 20603 5845 1809  794    0 65853   63  28   0   9
  6   52   0    0  5104    8 21172 5521 1813  751    0 64467   59  29   0  12
  7   41   0    0  5492   24 21896 5918 1821  773    0 65941   60  29   0  11
```

CPU 負荷が高いとき...

- prstat コマンドを使用して、プロセスの消費している CPU 時間が多い場合
  - “CPU の値” == 100 / “CPU 数” の場合は要注意

```
% prstat -Ln 10 -p `pgrep -x mysqld` 1 | while read i; do echo ${i}; done
PID USERNAME  SIZE   RSS STATE  PRI NICE      TIME  CPU PROCESS/LWPID
6124 mysql    2513M 1594M cpu4     7    5   0:04:08 4.7% mysqld/19
6124 mysql    2513M 1594M cpu5     7    5   0:04:10 4.6% mysqld/14
6124 mysql    2513M 1594M cpu0     6    5   0:04:10 4.5% mysqld/21
6124 mysql    2513M 1594M cpu7     6    5   0:04:09 4.5% mysqld/16
6124 mysql    2513M 1594M sleep    7    5   0:04:08 4.5% mysqld/20
6124 mysql    2513M 1594M run      6    5   0:04:07 4.5% mysqld/22
6124 mysql    2513M 1594M run      6    5   0:04:08 4.5% mysqld/23
6124 mysql    2513M 1594M cpu3     7    5   0:04:09 4.5% mysqld/13
6124 mysql    2513M 1594M run      6    5   0:04:08 4.5% mysqld/25
6124 mysql    2513M 1594M run      6    5   0:04:06 4.4% mysqld/18
Total: 1 processes, 26 lwps, load averages: 12.64, 11.31, 6.66
```

CPU 負荷の主な要因

- システムコール
  - (mpstat で sys が高く、syscl の回数が多い場合)
  - システムコールの発生源を特定する
  - それぞれのシステムコールが何回発生したか
  - システムコールの処理に掛かった時間を調べる
- ユーザ関数
  - (mpstat で usr が高い場合)
  - 最も呼び出し回数の多い関数を調べる
  - 最も時間を消費した関数を見つける
    - inclusive CPU time
    - exclusive CPU time
- カーネルサービス

syscall プロバイダ

- システムコールの解析を行いたい場合には syscall プロバイダを利用します
- 用意されているプローブ
  - entry : システムコールの呼び出しを捕捉します
  - return : システムコールからの復帰を捕捉します
  - 全てのシステムコールに関して entry/return のプローブが用意されており、syscall::システムコール名:entry の形式でプローブを呼び出す事が出来ます
- 主な変数
  - probefunc : syscall プロバイダの probefunc にはシステムコール名が入ります
  - argN : entry プローブの場合はシステムコールの引数
  - arg0, arg1 : return プローブの場合は返り値

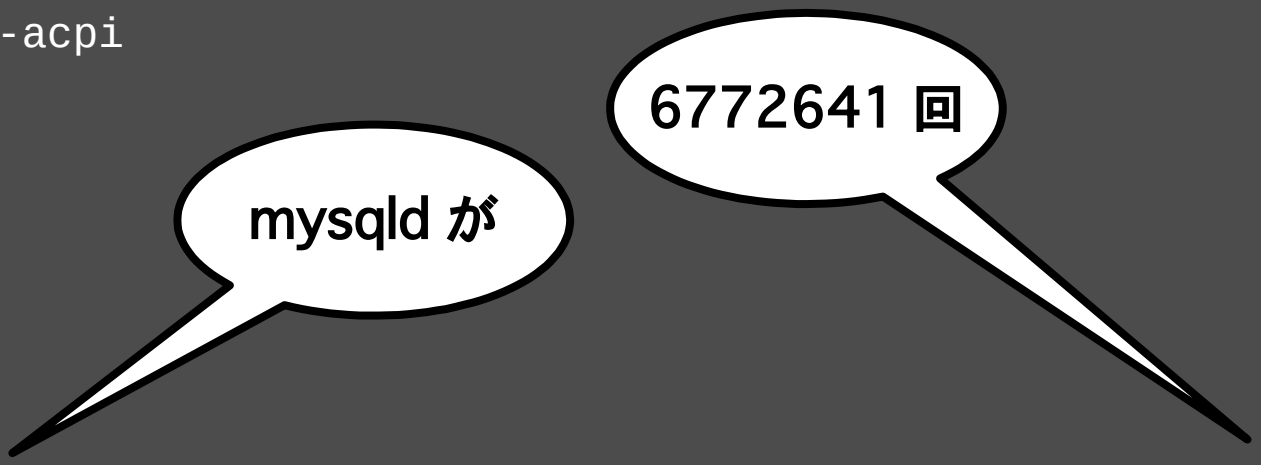
pid プロバイダ

- プロセスの関数呼び出しを調査したい場合は pid プロバイダを使用します
- 主なプローブ
  - entry : 関数の呼び出しを捕捉します
  - return : 関数からの復帰を捕捉します
  - 殆どの関数に関して entry/return のプローブが自動的に作成され、pidプロセス ID:ファイル名:関数名:entry の形式でプローブを呼び出す事が出来ます
- 主な変数
  - probefunc : pid プロバイダの probefunc には関数名が入ります
  - argN : entry プローブの場合は、関数の引数
  - arg0 : return プローブの場合は、関数がどこから return したか
  - arg1 : return プローブの場合は、関数の返り値

システムコールの発生源を特定する

- syscall プロバイダで execname 毎に回数を集計

```
# dtrace -qn 'syscall:::entry {@[execname]=count()}
tick-10sec {printa(@); exit(0)}'
...
  svc.configd                                                     1
  svc.startd                                                      1
  sshd                                                           15
  hald-addon-acpi                                                17
  devfsadm                                                       19
  hald                                                           19
  sendmail                                                       39
  Xorg                                                          199
  dtgreet                                                       199
  java                                                          554
  dtrace                                                       5894
  tpcc_start                                                2733793
  mysqld                                                    6772641
```



それぞれのシステムコールが何回発生したか

- syscall:::entry プローブの中で probefunc 毎に集計

```
# dtrace -qn 'syscall:::entry /pid == $target /
{ @[probefunc] = count() }
tick-10sec{printa(@); exit(0)}' -p `pgrep -x mysqld`
  lwp_mutex_timedlock                                               2
  lwp_mutex_wakeup                                                  2
  llseek                                                           10
  pollsys                                                          30
  pread64                                                          94
  fdsync                                                        11830
  pwrite64                                                      11891
  gtime                                                        206722
  yield                                                        218410
  write                                                        349125
  lwp_park                                                     467665
  fcntl                                                        544531
  priocntlsys                                                  697836
  read                                                         970104
```

read が

970104 回

システムコールの処理に掛かった時間

- syscall プロバイダの中で、timestamp の差分を probefunc 毎に集計します

```
# dtrace -qn 'syscall:::entry
/pid == $target /
{
  self->ts = timestamp
}
syscall:::return
/pid == $target && self->ts /
{
  @time[probefunc] = sum(timestamp - self->ts);
  self->ts = 0
}
tick-10sec
{
  printa(@time);  exit(0)
}
' -p `pgrep -x mysqld`
```

システムコールの処理時間（実行結果）

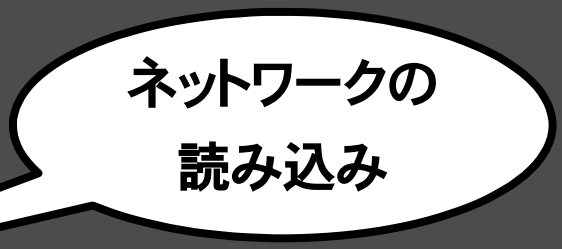
- read, pollsys, lwp_park が多く時間を消費

```
# dtrace -qn 'syscall:::entry /pid == $target /{ self->ts = timestamp }
syscall:::return/pid == $target && self->ts /
{ @time[probefunc] = sum(timestamp - self->ts); self->ts = 0 }
tick-10sec{printa(@time);  exit(0)}' -p `pgrep -x mysqld`
  llseek                                                        51117
  pread64                                                    12084510
  fdsync                                                     21598767
  pwrite64                                                  161030018
  gtime                                                     286315955
  fcntl                                                     903350411
  priocntlsys                                              1801018557
  write                                                    3396810348
  yield                                                    6607125711
  read                                                  15937469500
  pollsys                                               27244084443
  lwp_park                                              65937475852
```

read システムコールの原因は?

- read システムコール呼び出し時に ustack() でスタックのダンプを取得する

```
# dtrace -qn 'syscall::read:entry / pid == $target /
{ @[ustack(20)] = count() }
tick-10sec
{ trunc(@, 5); exit(0) }' -p `pgrep -x
mysqld` | c++filt
              libc.so.1`__read+0x15
              mysqld`vio_read+0x24
              mysqld`unsigned long my_real_read(st_net*,unsigned*)+0xb2
              mysqld`my_net_read+0x209
              mysqld`bool do_command(THD*)+0x89
              mysqld`handle_one_connection+0x3b6
              libc.so.1`_thrp_setup+0x9b
              libc.so.1`_lwp_start
            958546
```



ここまでのまとめ

- DTrace の syscall プロバイダを使用すると、システムコールの発生源、システムコール毎の発生回数、システムコールの処理時間等を見る事が出来ます
- DTrace を使用すると、実行中のプログラムのスタックやレジスタの中身を参照する事が出来ます
- システムコールの引数や返り値にもアクセス出来るので、どのファイルディスクリプタに read が発行されたのか、read されたサイズは何 bytes なのか、といった情報も調べる事が可能です

最も呼び出し回数の多い関数を調べる

- pid プロバイダの中で、probefunc 毎に集計します

```
# dtrace -qn 'pid$target:::entry { @[probemod, probefunc] = count() }
tick-10sec {trunc(@, 20); printa("%15s\t%30s\t\t%@d\n", @); exit(0)}' \
-p `pgrep -x mysqld`
...
        libc.so.1                      clear_lockbyte        44424
        libc.so.1                  mutex_unlock_queue        44424
           mysqld                     dfield_get_ty
           mysqld               dtuple_get_nth_fi
           mysqld                     buf_frame_alig
           mysqld                   dict_table_is_comp
           mysqld                        ut_align_down        59863
           mysqld                    mach_read_from_1      1073076
           mysqld                   dict_field_get_col      1636696
           mysqld         dict_index_get_nth_field      1681825
           mysqld                    mach_read_from_2      1750883
```

mach_read_from_2() が 1750883 回

mach_read_from_2 の原因は?

- pid プロバイダで mach_read_from_2() 関数の呼び出しを補足して、ustack() でスタックをダンプ

```
# dtrace -qn '
pid$target::mach_read_from_2:entry
{
  @[ustack(20)] = count()
}
tick-10sec
{
  trunc(@, 3);
  exit(0)
}' -p `pgrep -x mysqld` | c++filt
```

mach_read_from_2 の原因は?

- 実行結果...

```
# dtrace -qn 'pid$target::mach_read_from_2:entry { @[ustack(20)] = count() }
tick-10sec { trunc(@, 3); exit(0) }' -p `pgrep -x mysqld` | c++filt
...
              mysqld`mach_read_from_2
              mysqld`page_rec_get_n_recs_before+0x11a
              mysqld`btr_cur_search_to_nth_level+0xc17
              mysqld`btr_estimate_n_rows_in_range+0x439
              mysqld`unsigned long long
ha_innobase::records_in_range(unsigned,st_key_range*,st_key_range*)+0x2c3
              mysqld`unsigned long long
check_quick_keys(PARAM*,unsigned,SEL_ARG*,unsigned char*,unsigned,int,unsigned char*,unsigned,int)+0x7c8
              mysqld`TRP_RANGE*get_key_scans_params(PARAM*,SEL_TREE*,bool,bool,double)+0x398
              mysqld`int SQL_SELECT::test_quick_select
(THD*,Bitmap<64U>,unsignedlong long,unsigned long long,bool)+0x10dc
              mysqld`int mysql_update
(THD*,TABLE_LIST*,List<Item>&,List<Item>&,Item*,unsigned,st_order*,unsigned long long,enum_duplicates,bool)+0x5ef
              mysqld`int mysql_execute_command(THD*)+0x1c58
              mysqld`bool Prepared_statement::execute(String*,bool)+0x2a0
              mysqld`bool Prepared_statement::execute_loop(String*,bool,unsignedchar*,unsigned char*)+0x4f2
              mysqld`void mysql_stmt_execute(THD*,char*,unsigned)+0xd9
              mysqld`bool dispatch_command(enum_server_command,THD*,char*,unsigned)+0xe36
              mysqld`bool do_command(THD*)+0x105
              mysqld`handle_one_connection+0x3b6
              libc.so.1`_thrp_setup+0x9b
              libc.so.1`_lwp_start
```

mach_read_from_2 の原因は?

```
# dtrace -qn 'pid$target::mach_read_from_2:entry
{ @[ustack(20)] = count() }
tick-10sec { trunc(@, 3); exit(0) }' -p `pgrep                    t+filt
...
              mysqld`mach_read_from_2
              mysqld`page_rec_get_n_recs_befo
              mysqld`btr_cur_search_to_nth_le
              mysqld`btr_estimate_n_rows_in_ra
              mysqld`unsigned long long ha_i                        ()
              mysqld`unsigned long long ch    _quick_
              mysqld`TRP_RANGE*get_key_scans_params()
              mysqld`int SQL_SELECT::test_quick_select()
              mysqld`int mysql_update()
              mysqld`int mysql_execute_command()
              mysqld`bool Prepared_statement::execute()
              mysqld`bool Prepared_statement::execute_loop()
              mysqld`void mysql_stmt_execute()
              mysqld`bool dispatch_command()
              mysqld`bool do_command()
              mysqld`handle_one_connection+0x3b6
              libc.so.1`_thrp_setup+0x9b
              libc.so.1`_lwp_start
```

prepared statement の UPDATE からの SELECT

最も時間を消費した関数を見つける

- inclusive CPU time
  - 関数内で呼び出している他の関数（子関数）の実行時間も含んだ (inclusive) 実行時間
  - foo() が bar() を呼び出している場合に、bar() の実行時間も foo() の実行時間の一部とする
  - main() の実行時間が一番大きく出る
- exclusive CPU time
  - 関数内で呼び出している他の関数（子関数）の実行時間を除いた (exclusive) 実行時間
  - foo() が bar() を呼び出している場合に、bar() の実行時間は foo() の実行時間に含めない
  - 小さな関数に分割された処理が埋もれ易い

inclusive CPU time

- pid プロバイダの entry/return で timestamp の差分を取ります
  - 再帰関数やコンパイラの最適化に影響を受けます

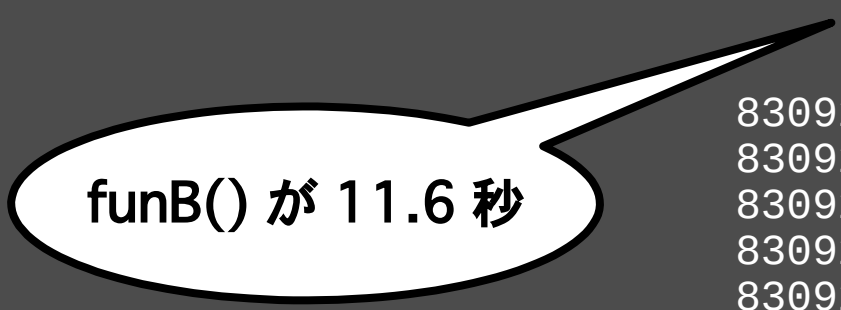
```
dtrace -qn '
pid$target:::entry
{
  self->ts[probefunc] = timestamp;
}
pid$target:::return
{
  @time[probefunc] = sum(timestamp - self->ts[probefunc]);
}
END
{
  printa(@time);
}' -p `pgrep -n mysqld`
```

inclusive CPU time（実行結果）

- 単位はナノ秒です

```
# dtrace -qn '
pid$target:::entry{ self->ts[probefunc] = timestamp;}
pid$target:::return{
  @time[probefunc] = sum(timestamp - self->ts[probefunc]);
}
END{ trunc(@time, 10); printa(@time);}' -c ./a.out

  sleep                                                 10023117940
  funB                                                  11636520954
  funC                                                  11636544440
  rtld_db_preinit                                   8309240860342215
  rd_event                                          8309240861228652
  memmove                                           8309240861722816
  rtld_db_postinit                                  8309240862390474
  setup                                             8309240862429679
  _setup                                            8309240862441209
  main                                              8309252499334308
```



exclusive CPU time

- ustackdepth 変数を利用して、スタックの深さ毎に CPU の消費時間を集計します

```
# dtrace -qn '

pid$target:::entry {
  self->ts[ustackdepth] = timestamp;
  self->el[ustackdepth] = 0;        /* elapsed time */
  self->ad[ustackdepth] = 0;        /* adjustment */
}

pid$target:::return {
  self->el[ustackdepth +1] = timestamp - self->ts[ustackdepth +1];
  self->ad[ustackdepth] -= self->el[ustackdepth +1];
  @time[probefunc] = sum(self->el[ustackdepth +1] + self->ad[ustackdepth +1]);
}

END { printa(@time); }' -p `pgrep -n mysqld`
```

exclusive CPU time（実行例）

- 単位はナノ秒です

```
# dtrace -qn '
pid$target:::entry {
  self->ts[ustackdepth] = timestamp;
  self->el[ustackdepth] = 0;  /* elapsed time */
  self->ad[ustackdepth] = 0;  /* adjustment */
}
pid$target:::return {
  self->el[ustackdepth +1] = timestamp - self->ts[ustackdept    1];
  self->ad[ustackdepth] -= self->el[ustackdepth +1];
  @time[probefunc] = sum(self->el[ustackdepth +1] + self->ad    stackdepth +1]);
}
END { trunc(@time, 5); printa(@time); }' -c ./a.out

  funB                                                656253
  sleep                                              1016151
  funA                                            1601471452
  ___nanosleep                                   10007146550
  rtld_db_preinit                             8308230445500287
```

CPU の解析に関するまとめ

- DTrace を使用すると、CPU を誰がどの様に使用しているか、詳細にドリルダウンする事が出来ます
- DTrace はシステムコール、カーネルサービス、ユーザプロセス等、CPU を消費する処理の全てを解析する事が出来ます
- システムコールの解析には DTrace の syscall プロバイダを使用します
- ユーザプロセスの解析には DTrace の pid プロバイダを使用します

ディスク I/O の負荷が高い場合の調査方法

- io:::start プローブ
- 誰がどこにどれだけ書き込みをしているのか
- I/O のサイズを調べる
- I/O の多重度を調べる

ディスク I/O の負荷が高い場合とは...

- iostat コマンドを使用して、Disk I/O に問題が発生している場合
  - %b が 100 に達している、%w が 0 以外になっている、IOPS やスループットが予想より出ていない、等

```
# iostat -xnCzmp 1
...
                    extended device statistics
    r/s    w/s   kr/s   kw/s wait actv wsvc_t asvc_t  %w  %b device
  163.2   63.1 14093.3 7048.7  0.0 10.4    0.0   46.1   0 100 c0
  163.2   63.1 14093.3 7048.7  0.0 10.4    0.0   46.1   0 100 c0t1d0
  163.2   63.1 14093.3 7048.7  0.0 10.4    0.0   46.1   0 100 c0t1d0s0 (/data)
                    extended device statistics
    r/s    w/s   kr/s   kw/s wait actv wsvc_t asvc_t  %w  %b device
  186.1   84.1 12499.5 1484.4  0.0 10.2    0.0   37.8   0 100 c0
  186.1   84.1 12499.5 1484.4  0.0 10.2    0.0   37.8   0 100 c0t1d0
  186.1   84.1 12499.5 1484.4  0.0 10.2    0.0   37.8   0 100 c0t1d0s0 (/data)
^C
```

ディスク I/O の負荷が高い場合とは...

- ZFS を使用している場合は zpool iostat というコマンドもあります

```
# zpool iostat -v 1
               capacity     operations    bandwidth
pool        alloc   free   read  write   read  write
----------  -----  -----  -----  -----  -----  -----
rpool       58.7G  77.3G      1      1   131K   195K
  c0t0d0s0  58.7G  77.3G      1      1   131K   195K
----------  -----  -----  -----  -----  -----  -----

               capacity     operations    bandwidth
pool        alloc   free   read  write   read  write
----------  -----  -----  -----  -----  -----  -----
rpool       58.7G  77.3G     19      9  2.38M   138K
  c0t0d0s0  58.7G  77.3G     19      9  2.38M   138K
----------  -----  -----  -----  -----  -----  -----
...
```

io:::start プローブ

- I/O を調査する際は io:::start プローブを使用します
- io:::start プローブから参照出来る変数には以下の様な便利なデータが格納されています
  - args[0] : 発行された I/O に関するバッファ情報
  - args[0]->b_bcount : I/O のサイズ (bytes)
  - args[1] : I/O に関するデバイス情報
  - args[1]->dev_name : I/O が発行されたデバイス名
  - args[2] : I/O に関するファイル情報
  - args[2]->fi_name : ファイル名
  - args[2]->fi_pathname : フルパスのファイル名
  - args[2]->fi_mount : ファイルシステムのマウントポイント

I/O を発行しているプロセスの特定

- io:::start プローブが発動した時の execname を調べます
  - args[2]->fi_pathname はフルパスのファイル名です

```
#!/usr/sbin/dtrace -qs
io:::start
/ args[0]->b_flags & B_WRITE/       /* write */
{
  @[execname, args[2]->fi_pathname] = sum(args[0]->b_bcount);
}
tick-10sec
{
  printf(" bytes written\tprocess name\tfile name\t\t\n");
}
tick-10sec
{
  printa(" %@d\t\t%s\t\t%s\n", @); trunc(@)
}
```

I/O を発行しているプロセスの特定

- 実行結果

```
# ./whowhereio.d
 bytes written  process name    file name
 63488          mysqld          <none>
 118784         mysqld          /data/wh100/ib_logfile0
 339968         mysqld          /data/wh100/ib_logfile1
 109740032      mysqld          /data/wh100/ibdata1
 bytes written  process name    file name
 62464          mysqld          <none>
 122880         mysqld          /data/wh100/ib_logfile0
 344064         mysqld          /data/wh100/ib_logfile1
 107118592      mysqld          /data/wh100/ibdata1
 bytes written  process name    file name
 67584          mysqld          <none>
 126976         mysqld          /data/wh100/ib_logfile0
 211456         zpool-rpool             <none>
 319488         mysqld          /data/wh100/ib_logfile1
 114589696      mysqld          /data/wh100/ibdata1
```

I/O のサイズを調べる

- args[0]->b_bcount が I/O サイズです

```
#!/usr/sbin/dtrace -qs
io:::start
/ execname == "mysqld" && args[0]->b_flags & B_WRITE /
{
  @[args[0]->b_bcount, args[2]->fi_pathname] = count();
}
tick-1sec { printf(" bytes written\tntimes\tfile name\n") }
tick-1sec { printa(" %d\t\t%@d\t%s\n", @); trunc(@) }
```

I/O サイズを調べる

- 実行結果

```
# ./iosize.d
...
 bytes written  ntimes  file name
 16384          1       /data/wh100/mysql-bin.000017
 65536          1       /data/wh100/ibdata1
 9216           2       <none>
 32768          2       h100/ibdata1
 262144         7       100/ibdata1
 8192           9       n100/ib_logfile0
 8192           1       /data/wh100/ibdata1
 4096           34      /data/wh100/ib_logfile0
 512            35      <none>
 16384          109     /data/wh100/ibdata1
^C
```

16KB

I/O の多重度を調べる

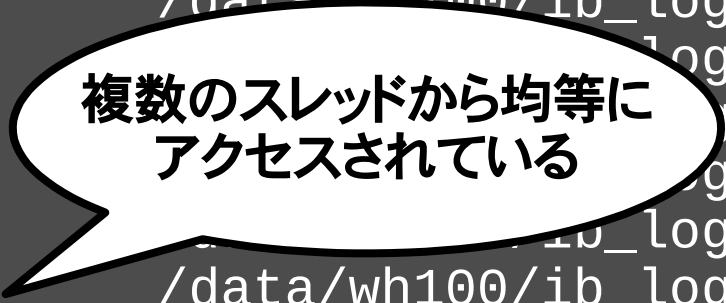
- 特定のファイルに対して I/O を発行しているスレッドの数を調べます

```
#!/usr/sbin/dtrace -qs
io:::start
/ execname == "mysqld" && args[0]->b_flags & B_WRITE
  && args[2]->fi_pathname == "/data/wh100/ib_logfile0" /
{
  @[pid, tid, args[2]->fi_pathname] = count();
}
tick-1sec { printf(" pid\ttid\tntimes\tfile name\n") }
tick-1sec { printa(" %d\t%d\t%@d\t%s\n", @); trunc(@) }
```

I/O の多重度を調べる

- 実行結果

```
# ./iothr.d
...
 pid    tid      ntimes  file name
 1567   3        1       /data/wh100/ib_logfile0
 1567   21       4       /data/wh100/ib_logfile0
 1567   20       6       /data/wh100/ib_logfile0
 1567   17       8       /data/wh100/ib_logfile0
 1567   19       10      /data/wh100/ib_logfile0
 1567   12       11      /data/wh100/ib_logfile0
 1567   18       11      /data/wh100/ib_logfile0
 1567   26       11      /data/wh100/ib_logfile0
 1567   15       14      /data/wh100/ib_logfile0
 1567   25       19      /data/wh100/ib_logfile0
...
```



ここまでのまとめ

- DTrace を使用すると、ディスクを誰がどの様に使用しているか、詳細にドリルダウンする事が出来ます
- DTrace の io:::start を使用すると、I/O の詳細情報にアクセスする事が出来ます

MySQL を調査する

- mysql プロバイダ
- SQL の種類別の発行回数を調べる
- 回数の多い SELECT 文のクエリー内容
- クライアントからの通信量を調べる

MySQL プロバイダ

- MySQL のソースコードに組み込まれたプロバイダです
- ネットワーク、コマンド、クエリー、行アクセス、ロック、ファイルソート、キャッシュ等、56 個のプローブが定義されています
  - insert-done : INSERT 文が終了した時のプローブ
  - read-row-start : 行の読み取りが開始された時のプローブ
  - handler-rdlock-done : read ロック完了時のプローブ
  - keycache-read-miss : MyISAM の INDEX キャッシュミス
- プローブに関連するデータにアクセス出来ます
  - query-start プローブの arg0 => SQL 文全体、arg1 => 接続 ID、arg2 => スキーマ名、arg3 => ユーザ名、arg4 => クライアントのホスト名

SQL の種類別の発行回数を調べる

- mysql プロバイダの *start プローブを使用すると SQL クエリーの実行を補足する事が出来ます

```
# dtrace -qn '
mysql$target:::select-start,
mysql$target:::insert-start,
mysql$target:::insert-select-start,
mysql$target:::update-start,
mysql$target:::multi-update-start
{
  @[probename] = count()
}
tick-1sec
{
  printa(@); trunc(@)
}' -p `pgrep -x mysqld`
```

観察したいクエリーの種類を全て指定

SQL の種類別の発行回数を調べる

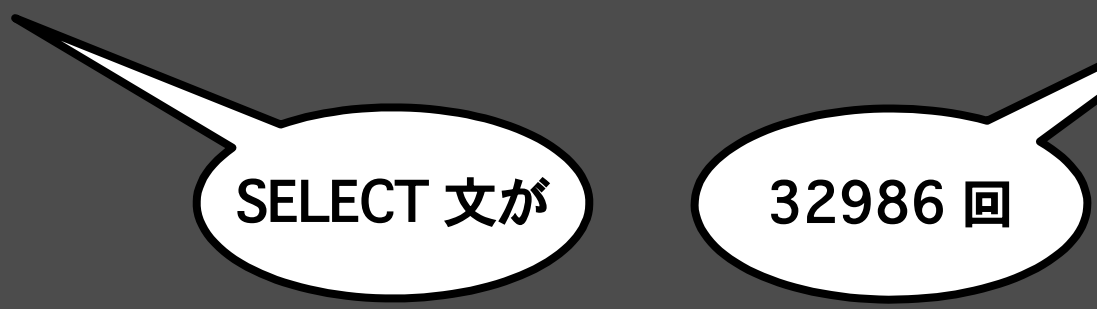
- 以下の出力から SELECT 文が多い事が分かります

```
# dtrace -qn 'mysql$target:::select-start,mysql$target:::insert-start,
mysql$target:::insert-select-start,mysql$target:::update-start,
mysql$target:::multi-update-start { @[probename] = count() }
tick-1sec{ printa(@); trunc(@)}' -p `pgrep -x mysqld`

  insert-start                                                    5414
  update-start                                                    7042
  select-start                                                   32986

  insert-start                                                    5359
  update-start                                                    6994
  select-start                                                   33236

  insert-start                                                    5205
  update-start                                                    6821
  select-start                                                   32612
```



回数の多い SELECT 文のクエリー内容

- mysql プロバイダの *start プローブの arg0 には SQL クエリーが格納されています

```
# dtrace -qn 'mysql$target:::select-start
{
  @[copyinstr(arg0)] = count()
}
tick-1sec
{
  trunc(@, 3);
  printa(@);
  trunc(@)
}' -p `pgrep -x mysqld`
```

arg0 を copyinstr() で
クエリー内容を抽出

回数の多い SELECT 文のクエリー内容

- 以下の様に SQL 文と実行回数を簡単に取り出せます

```
# dtrace -qn 'mysql$target:::select-start
{ @[copyinstr(arg0)] = count() }
tick-1sec{ printa(@); trunc(@)}'

  SELECT w_street_1, w_street_2, w_city, w_sta
warehouse WHERE w_id = 7                  100
  SELECT w_street_1, w_street_2, w_city, w_state
warehouse WHERE w_id = 4                  102
  SELECT w_street_1, w_street_2, w_city, w_state, w_zip, w_name FROM
warehouse WHERE w_id = 6                  116

  SELECT w_street_1, w_street_2, w_city, w_state, w_zip, w_name FROM
warehouse WHERE w_id = 10                  88
  SELECT w_street_1, w_street_2, w_city, w_state, w_zip, w_name FROM
warehouse WHERE w_id = 3                   90
  SELECT w_street_1, w_street_2, w_city, w_state, w_zip, w_name FROM
warehouse WHERE w_id = 7                   90
```

SELECT ...

この SQL は
秒間 116 回
実行されました

クライアントからの通信量を調べる

- net-read-done プローブの arg1 を調べると、通信量が分かります

```
# dtrace -qn 'mysql$target:::net-read-done
{
  @[tid] = sum(arg1)
}
tick-1sec
{
  printa(@);
  trunc(@)
}' -p `pgrep -x mysqld`
```

クライアントからの通信量を調べる

- 16 session, それぞれの受信バイト数が分かりました

```
# dtrace -qn 'mysql$target:::net-read-done { @[tid] = sum(arg1) }
tick-1sec{printa(@); trunc(@)}' -p `pgrep -x mysqld`

        11              63618
        24              68877
        12              69213
        18              73665
        22
        20
        23
        26              78784
        14              79706
        17              91161
        25              93902
        19              98420
        13             100018
        16             102255
        15             103610
        21             105007
```

mysql プロバイダのまとめ

- MySQL の様々な処理に割り込んで、解析に必要なデータを事細かに集める事が出来ます
  - SQL のチューニング
  - MySQL のパラメータチューニング
  - MySQL の障害解析
- MySQL 5.4 から使用可能です
  - MySQL を停止させる必要はありません
  - MySQL に特殊な設定は必要ありません
  - OS の再起動も必要ありません
  - OS の設定も不要です

File System I/O を調査する

- fsinfo プロバイダ
- どのファイルに I/O が発生しているか
- ファイル I/O のサイズ
- ファイルシステム I/O の多重度
- ZFS の ARC のヒット率を見る

fsinfo プロバイダ

- ファイルシステムレベルのアクティビティをトレースしたい場合は fsinfo プロバイダを使用すると便利です
- 用意されているプローブ
  - VNODE 操作に対応
    - vnevent, shrlock, getsecattr, setsecattr, dispose, dumpctl, pageio, pathconf, dump, poll, delmap, addmap, map, putpage, getpage, realvp, space, frlock, cmp, seek, rwunlock, rwlock, fid, inactive, fsync, readlink, symlink, readdir, rmdir, mkdir, rename, link, remove, create, lookup, access, setattr, getattr, setfl, ioctl, write, read, close, open
- 主な変数
  - args[0] : io プロバイダの args[2] と同様の、ファイルシステム情報
  - args[0]->fi_pathname : フルパスのファイル名
  - args[1] : 返り値

DTrace を使用する前に...

- iostat, zpool iostat, fsstat コマンドを使用して、I/O に問題がある事を確認して下さい

```
# fsstat zfs 1
 new  name   name  attr  attr lookup rddir  read read  write write
 file remov  chng   get   set    ops   ops   ops bytes   ops bytes
36.0K   919 2.85K 4.82M  115K  28.4M  425K  224M 43.3G  112M  124G zfs
    0     0     0     0     0      0     0   122  356K    68 26.7K zfs
    0     0     0     0     0      0     0   762 1.96M   345 8.11M zfs
    0     0     0     0     0      0     0   163  486K    78 30.3K zfs
    0     0     0     2     0      4     0   828 2.02M   365  104K zfs
    0     0     0     0     0      0     0   585 1.62M   254 62.9K zfs
    0     0     0     0     0      0     0   465  942K   213 43.5K zfs
    0     0     0     0     0      0     0   210  636K    95 42.6K zfs
    0     0     0     0     0      0     0   182  506K    95 38.4K zfs
    0     0     0     0     0      0     0   341  755K   161 72.7K zfs
    0     0     0    51     0    331     6   339  896K   154 47.0K zfs
```

どのファイルに I/O が発生しているか

- fsinfo プロバイダの args[0]->fi_pathname がファイル名です

```
#!/usr/sbin/dtrace -qs
fsinfo:::write
{
  @[execname, args[0]->fi_pathname] = sum(args[1]);
}
  tick-10sec { printf(" bytes written\tprocess
  name\tfile name\t\t\n"); }
  tick-10sec { printa(" %@d\t\t%s\t\t%s\n", @); trunc(@)
}
```

どのファイルに I/O が発生しているか

- 実行結果

```
# ./fsww10.d
 bytes written  process name    file name
 96 sshd <unknown>
 1200 dtgre
 1333 mysq                      data/edo01-slow.log
 36420 tpcc_s
 84922 mysqld /export/home/mysq data/mysql-bin.000019
 186498 mysqld <unknown>
 294912 mysqld /export/home/mysql/data/ib_logfile0
 8749056 mysqld /export/home/mysql/data/ibdata1
```

ファイル I/O のサイズ

- fsinfo:::write プローブの args[1] は I/O サイズ

```
#!/usr/sbin/dtrace -qs
fsinfo:::write
/ execname == "mysqld"
 && args[0]-> fi_pathname != "<unknown>" /
{
  @[args[1], args[0]->fi_pathname] = count();
}
tick-1sec { printf(" bytes written\tntimes\tfile name\n") }
tick-1sec { printa(" %d\t\t%@d\t%s\n", @); trunc(@) }
```

ファイル I/O のサイズ

- 実行結果

```
# ./fsiosize.d
 bytes written  ntimes  file name
 173            1       /export/home/mysql/data/edo01-slow.log
 1024           1       /export/home/mysql/data/ib_logfile0
 1522           1       /export/home/mysql/data/mysql-bin.000019
 2048           1       /export/home/mysql/data/ib_logfile0
 4257           1       /export/home/mysql/data/mysql-bin.000019
 5346           1       /export/home/mysql/data/mysql-bin.000019
 49152          1       /export/home/mysql/data/ibdata1
 81920          2       /export/home/mysql/data/ibdata1
 512            3       /export/home/mysql/data/ib_logfile0
 32768          4       /export/home/mysql/data/ibdata1
 16384          8       /export/home/mysql/data/ibdata1
```

32K, 16K, etc.

ファイルシステム I/O の多重度

- スレッド (tid) とファイル (args[0]->fi_pathname) の組み合わせ毎に I/O を count() します

```
#!/usr/sbin/dtrace -qs
fsinfo:::write
/ execname == "mysqld"
&& args[0]->fi_pathname != "<unknown>" /
{
  @[pid, tid, args[0]->fi_pathname] = count();
}
  tick-10sec { printf(" pid\ttid\tntimes\tfile name\n") }
  tick-10sec { printa(" %d\t%d\t%@d\t%s\n", @); trunc(@)
}
```

ファイルシステム I/O の多重度

- 実行結果

```
# ./fsiothr10.d
   pid    tid ntimes  file name
...
   9109    15  4    /export/home/mysql/data/ib_logfile0
   9109    17  4    /export/home/mysql/data/ib_logfile0
   9109    26  4    /export/home/mysql/data/mysql-bin.000019
   9109    11  5    /export/home/mysql/data/mysql-bin.000019
   9109    12  5    /export/home/mysql/data/ib_logfile0
   9109    19  6    /export/home/mysql/data/ib_logfile0
   9109    14  8    /export/home/mysql/data/mysql-bin.000019
   9109    26  8    /export/home/mysql/data/ib_logfile0
   9109    25  9    /export/home/mysql/data/mysql-bin.000019
   9109    11  10   /export/home/mysql/data/ib_logfile0
   9109    14  15   /export/home/mysql/data/ib_logfile0
   9109    25  18   /export/home/mysql/data/ib_logfile0
```

複数のスレッドから
均等にアクセスがある

ファイルシステム I/O の多重度

- 実行結果 (ftpd)

```
# ./fsiothr10.d
 pid   tid      nti                                    tar.bz2
 5265    1                                             tar.bz2
 pid     tid      ntimes   file name
 5265    1        17640                                h100.tar.bz2
 pid     tid      ntimes   file name
 5265    1        17608    /root/tmp/wh100.tar.bz2
 pid     tid      ntimes   file name
 5265    1        17663    /root/tmp/wh100.tar.bz2
 pid     tid      ntimes   file name
 5265    1        17502    /root/tmp/wh100.tar.bz2
^C
```

tid 1 しか無いので、
並列度も 1
（当たり前ですが…）

ZFS の ARC のヒット率を見る

- sdt:::arc-hit, sdt:::arc-miss プローブを使用すると、ZFS のキャッシュのヒット率を調べられます

```
# dtrace -qn 'BEGIN{printf("hit\tmiss\n")}
sdt:::arc-hit{hit++}   sdt:::arc-miss{miss++}
tick-1sec{printf("%d\t%d\n", hit,miss);hit = miss = 0}
tick-10sec{exit(0)}'
hit miss
390 4
351 4
276 2
582 1
344 17
309 14
234 0
390 3
610 10
289 7
```

ユーザランドのロックを調査する

- plockstat プロバイダ
- mutex でブロックされている時間を調査
- 条件変数でブロックされている時間
- どのロックを待っているのか

plockstat プロバイダ

- ユーザランドのロックを解析したい場合は plockstat プロバイダを使用すると便利です
- 用意されているプローブ
  - mutex lock
    - mutex-acquire, mutex-block, mutex-blocked, mutex-error, mutex-release, mutex-spin, mutex-spun
  - rw lock
    - rw-acquire, rw-block, rw-blocked, rw-error, rw-release
- plockstat コマンドも内部的に plockstat プロバイダを呼び出して全ての情報を入手しています

DTrace を使用する前に...

- prstat -mL の LCK の値から、ロック待ちが発生しているかを確認して下さい

```
# prstat -mLn 10 -p `pgrep -x mysqld` 1 | while read i; do echo ${i}; done
PID USERNAME USR SYS TRP TFL DFL LCK SLP LAT VCX ICX SCL SIG PROCESS/LWPID
5597 mysql    1.3 0.4 0.0 0.0 0.0 92 5.1 0.3 150  13 871   0 mysqld/15
5597 mysql    1.3 0.4 0.0 0.0 0.0 92 5.0 0.3 146  13 832   0 mysqld/16
5597 mysql    1.3 0.4 0.0 0.0 0.0 91 6.5 0.3 150  12 873   0 mysqld/11
5597 mysql    1.3 0.4 0.0 0.0 0.0 92 5.7 0.3 149  12 876   0 mysqld/17
5597 mysql    1.3 0.4 0.0 0.0 0.0 92 5.5 0.3 148  13 849   0 mysqld/22
5597 mysql    1.3 0.4 0.0 0.0 0.0 90 6.9 0.3 147  12 860   0 mysqld/12
5597 mysql    1.3 0.4 0.0 0.0 0.0 91 6.6 0.3 148  13 852   0 mysqld/23
5597 mysql    1.3 0.4 0.0 0.0 0.0 90 7.5 0.3 147  12 855   0 mysqld/18
5597 mysql    1.3 0.4 0.0 0.0 0.0 90 7.4 0.3 148  14 854   0 mysqld/25
5597 mysql    1.3 0.4 0.0 0.0 0.0 91 6.0 0.3 147  11 849   0 mysqld/19
Total: 1 processes, 26 lwps, load averages: 0.41, 0.42, 0.40
```

DTrace を使用する前に...

- plockstat コマンドを使用して、ロック待ちの状態を確認し、詳細情報を調査して下さい

```
# plockstat -Cve 30 -n 10 -p `pgrep -x mysqld`
plockstat: tracing enabled for pid 5597
        0
Mutex block

Count     nsec Lock                         Caller
-------------------------------------------------------------------------------
  824    99236 0x8ab46c0                    mysqld`os_fast_mutex_init+0x6c
  784    93008 0x8ab46c0                    mysqld`os_event_create+0x73
  799    88820 0x8ab46c0                    mysqld`os_fast_mutex_free+0x89
  648    89664 0x8ab46c0                    mysqld`os_event_free+0x212
  702    80237 0x8ab46c0                    mysqld`os_event_create+0x1ee
  706    78765 0x8ab46c0                    mysqld`os_event_free+0xb7
  257    66047 0x8ab4750                    mysqld`os_mutex_enter+0x16
  174    87324 0x8ab4750                    mysqld`os_mutex_enter+0x16
  100   122029 mysqld`LOCK_open             mysqld`__1cKopen_table6FpnDTHD_
pnKTABLE_LIST_pnLst_mem_root_pbI_pnIst_table__+0x57e
  129    71392 0x8ab4750                    mysqld`os_mutex_enter+0x16
```

mutex でブロックされている時間を調査

- plockstat:::mutex-blocked プローブの arg1 が 0 以外の場合はロックが取得された事を示しています

```
#!/usr/sbin/dtrace -qs
plockstat$target:::mutex-block
{
  self->mtxblock = timestamp;
}
plockstat$target:::mutex-blocked
/self->mtxblock && arg1 != 0/  /* arg1 !0 => lock is acquired */
{
  @[tid]= sum(timestamp - self->mtxblock);
}
plockstat$target:::mutex-blocked
{
  self->mtxblock = 0;
}
tick-10sec
{
  printa(@); exit(0);
}
```

mutex でブロックされている時間 (結果)

```
# ./mutexblocktime.d -p `pgrep -x mysqld`
         4           4955850
         9          13528495
        13         527714017
        21         533647518
        23         541418924
        22         549676275
        11         552772686
        20         556545422
        12         556793898
        26         558359809
        16         566052331
        15         571898235
        14         572705162
        17         573313916
        18         573558782
        24         587398092
        19         593550610
        25         607546895
```

25 607546895

条件変数でブロックされている時間

- pthread_cond_wait() の entry/return を監視します

```
#!/usr/sbin/dtrace -qs
pid$target::pthread_cond_wait:entry
{
  self->start = timestamp;
}
pid$target::pthread_cond_wait:return
/self->start/
{
  @time[tid] = sum(timestamp - self->start);
  self->start = 0;
}
tick-10sec
{
  printf("\ntid\ttime(nsec)\n");
  printa("%d\t\t%@u\n", @time);
  exit(0);
}
```

条件変数でブロックされていた時間

```
# ./condblocktime.d -p `pgrep -x mysqld`
tid     time(nsec)
4       3506887
5       1686408492
9       1982580867
13      7584006973
25      8597461654
12      8650112651
11      8678880959
17      8748242551
26      8851618727
24      8862518382
16      8924451609
19      8972682278
15      8979531996
22      9019878514
18      9051317967
21      9218505048
14      9255378091
20      9391305263
23      9494815995
```

23 9494815995

どのロックを待っているのか

- ustack() アクションを使用する事で、どの関数でロックが頻繁に取得されているかを調べられます

```
#!/usr/sbin/dtrace -qs
pid$target::pthread_cond_wait:entry
{
  @[ustack(20)] = count();
}
tick-10sec
{
  trunc(@, 5);
  printa(@);
  exit(0);
}
```

どのロックを待っているのか（結果）

- log_write_up_to() とあるのでログの書き込み?

```
# ./condustack.d -p `pgrep -x mysqld` | c++filt
...
          libc.so.1`pthread_cond_wait
          mysqld`os_event_wait_low+0x52
          mysqld`log_write_up_to+0xca
          mysqld`trx_commit_for_mysql+0x44f
          mysqld`int innobase_commit(handlerton*,THD*,bool)+0x1dc
          mysqld`int ha_commit_trans(THD*,bool)+0x37d
          mysqld`int end_trans(THD*,enum_mysql_completiontype)+0xb2
          mysqld`int mysql_execute_command(THD*)+0x38a6
          mysqld`bool dispatch_command
                 (enum_server_command,THD*,char*,unsigned)+0x26b3
          mysqld`bool do_command(THD*)+0x105
          mysqld`handle_one_connection+0x3
          libc.so.1`_thrp_setup+0x9b
          libc.so.1`_lwp_start
          542
```

log_write_up_to()

DTrace でネットワークを調査する

- Solaris 10
  - fbt プロバイダや sdt プロバイダ等、汎用のプロバイダに用意されたプローブを使用
    - fbt:ip:tcp_connect:entry
    - fbt:ip:tcp_send_data:entry
    - fbt:sockfs:sotpi_accept:entry
- OpenSolaris, SunOS 5.11 以降
  - ネットワークに特化した専用のプロバイダ
    - IP プロバイダ
    - TCP プロバイダ
    - UDP プロバイダ

DTrace を使用する前に...

- dladm コマンドを使用して、NIC の帯域が飽和しているかどうか確認して下さい
  - 加えて、mpstat や intrastat コマンドで、ネットワークのドライバが CPU を使い切っていないか確認して下さい

```
# dladm show-link -s -i 1
LINK            IPACKETS  RBYTES   IERRORS   OPACKETS    OBYTES      OERRORS
e1000g0         6104982   7741079437 0        1892263     165870086   0
e1000g0         3         192      0         2           420         0
e1000g0         2         128      0         1           170         0
e1000g0         3         192      0         1           170         0
e1000g0         3         192      0         1           170         0
e1000g0         2         128      0         1           170         0
e1000g0         3         192      0         1           170         0
e1000g0         2         128      0         1           170         0
e1000g0         1         64       0         1           170         0
e1000g0         2         128      0         1           170         0
^C
```

IP プロバイダを使用する

- IP プロバイダを使用すると snoop や tcpdump の様なコマンドを簡単に作成する事が出来ます
- ip:ip::send プローブがパケットの送信、ip:ip::receive がパケットの受信に対応しています

```
# dtrace -qn '
BEGIN{printf("NIC\tdir\tproto\tfunc\t\tsource\t\tdestination\tbytes\n")}
ip:ip::send,ip:ip::receive {printf("%s\t%s\t%s\t%s\t%s\t%s\t%d\n",
args[3]->if_name, (probename == "send") ? "S" : "R",
args[4]->ipv4_protostr, probefunc, args[2]->ip_saddr,
args[2]->ip_daddr, args[2]->ip_plength)}'
NIC     dir     proto   func            source          destination     bytes
bge0    S       TCP     tcp_send_data   10.16.62.6      10.16.100.16    148
bge0    R       TCP     ip_input        10.16.100.16    10.16.62.6      20
bge0    S       TCP     tcp_send_data   10.16.62.6      10.16.100.16    132
bge0    S       TCP     tcp_send_data   10.16.62.6      10.16.100.16    132
bge0    R       TCP     ip_input        10.16.100.16    10.16.62.6      20
^C
```

パケット情報抜き出し

まとめ

- DTrace には様々なプロバイダが既に用意されており、それらを上手く使いこなす事で、今まで入手し辛かった情報に簡単にアクセスする事が出来る様になっています
  - 物理ディスク I/O : io プロバイダ
  - ファイルシステム : fsinfo プロバイダ
  - ロック : plockstat, lockstat プロバイダ
  - CPU : syscall, pid, fbt プロバイダ
  - ネットワーク : ip プロバイダ
  - MySQL : mysql プロバイダ

DTrace の広がり

- 言語やアプリケーションに特化したプロバイダも多数存在しています
  - Java : jstack(), hotspot, hotspot_jni プロバイダ
  - PHP : php プロバイダ
  - Perl : perl プロバイダ
  - Ruby : ruby プロバイダ
  - Python : python プロバイダ
  - sh : sh プロバイダ
  - Tcl : tcl プロバイダ
  - Apache : apache プロバイダ
  - Firefox : javascript プロバイダ
  - MySQL : mysql プロバイダ
  - PostgreSQL : postgres プロバイダ

現在も活発に開発が続いています

- PSARC Project からの抜粋

PSARC/2005/589 *DTrace Developer Kit*
PSARC/2006/054 *DTrace JNI Binding*
PSARC/2006/196 *DTrace Filesystem Info Provider*
PSARC/2006/609 *Xserver provider for DTrace*
PSARC/2007/665 *DTrace NFS v4 Provider*
PSARC/2007/036 *sysevent DTrace provider*
PSARC/2008/008 *DTrace Provider for Bourne Shell*
PSARC/2008/050 *DTrace NFS v3 Provider*
PSARC/2008/174 *Enhancing Ruby with DTrace probes*
PSARC/2008/302 *DTrace IP Provider*
PSARC/2008/480 *DTrace CPC Provider*
PSARC/2009/337 *DTrace COMSTAR SCSI RDMA Protocol Target Provider*
PSARC/2009/291 *DTrace COMSTAR Fibre Channel Target Provider*
PSARC/2009/318 *DTrace COMSTAR iSCSI Target Provider*
PSARC/2010/106 *DTrace TCP and UDP providers*
PSARC/2010/194 *DTrace Kerberos Protocol Provider*

SOFTWARE. HARDWARE. COMPLETE.

ORACLE


ORACLE®