ulm university universität ulm

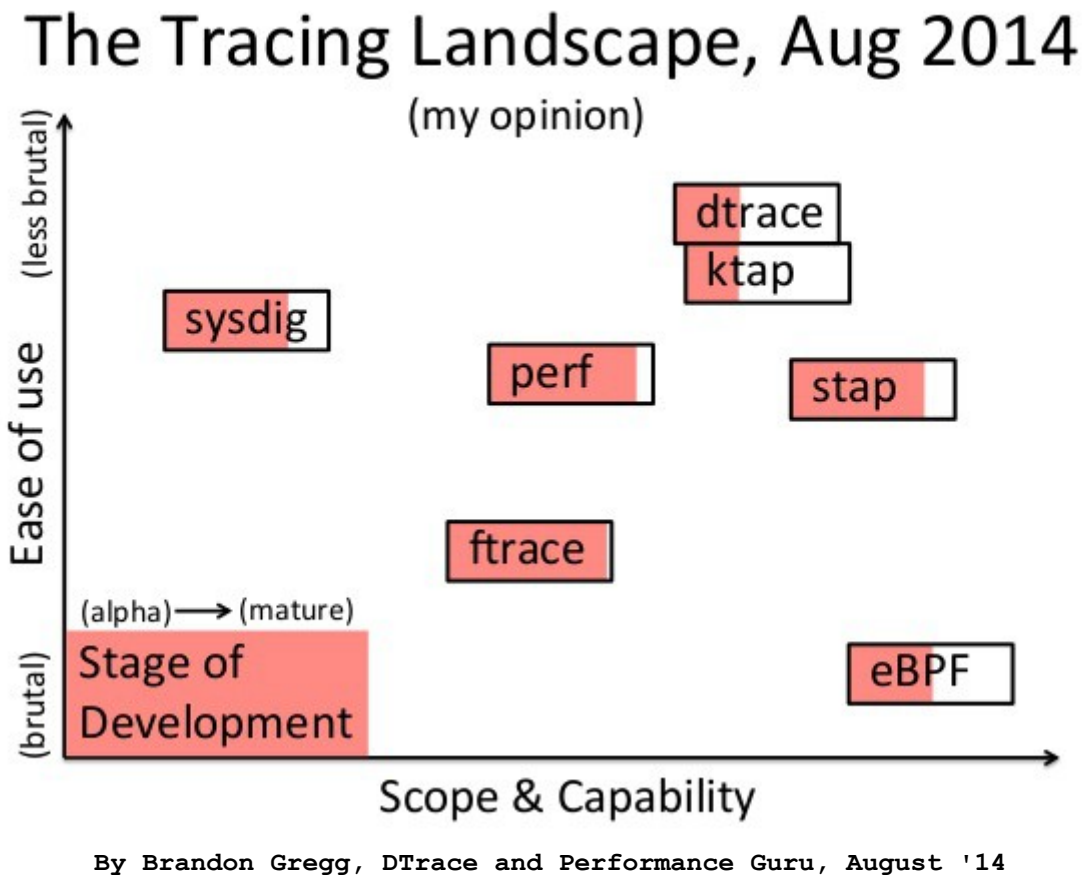
Performance: The OS matters too
What DTrace can do for you

Thomas Nau, kiz (Thomas.Nau@uni-ulm.de)

Not all codes are CPU centric!

There's memory, IO, ...

The Linux tracing landscape



By Brandon Gregg, DTrace and Performance Guru, August '14

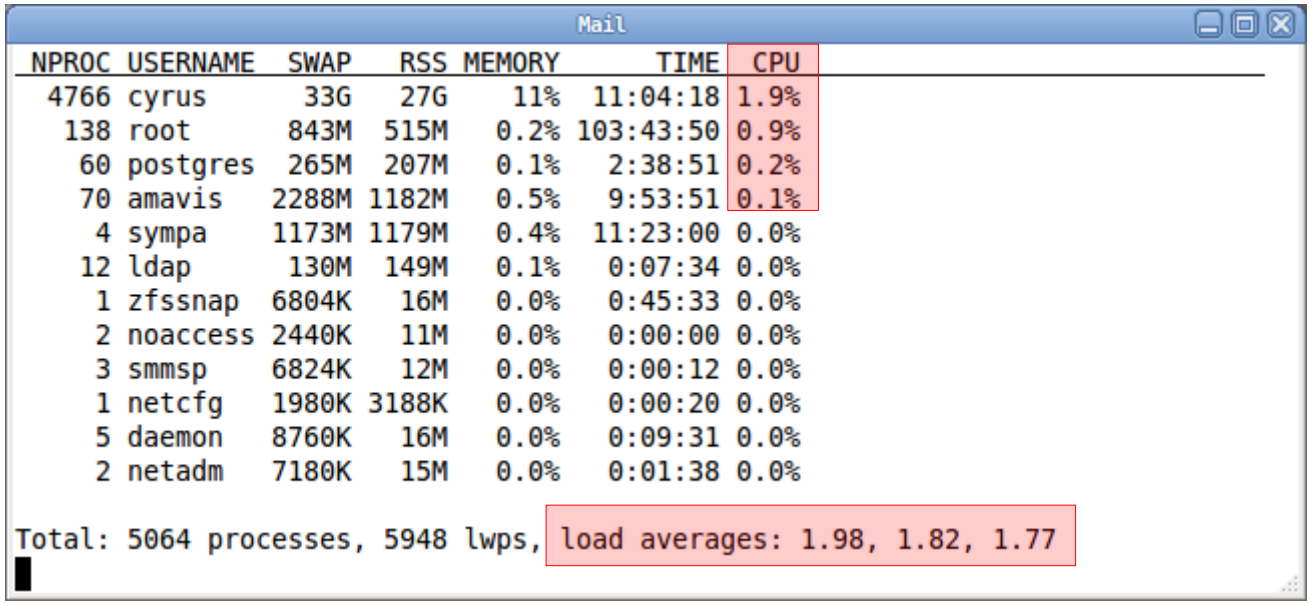
Tools using a statistical approach

- collect statistical data about kernel threads, IO, CPU, virtual memory, ...
  - *vmstat(1m), netstat(1m), kstat(1m), ...*
- help to get a rough idea about lingering problems
  - to high/low value for parameter #1
  - what is a bad value and how bad is bad (IO busy time)?

```
jedi# vmstat 5
 kthr      memory            page            disk          faults      cpu
 r b w   swap  free  re  mf pi po fr de sr m0 m1 m3 m1   in   sy   cs us sy id
 0 0 0 8620144 4617688 67 69 1  1  1  0  0  0  1  0  0 6147 1254 5529  0  1 99
 0 0 0 8702320 4730536 6  5  0  0  0  0  0  0  0  0  0 8010  532 7370  0  2 98
 0 0 0 8788856 4816776 6  1  0  0  0  0  0  0  0  0  0 7990  534 7428  0  2 98
^C
```

Tools using a sampling approach

- *e.g. top(1)* and *prstat(1m)* provides lots of information about running processes and threads
  - starting point if applications are suspected being the trouble maker
  - limited by Nyquist-Shannon-Theorem
    - you will miss short-lived events or spikes like load vs. CPU time; however load is a bad/broken metric anyhow



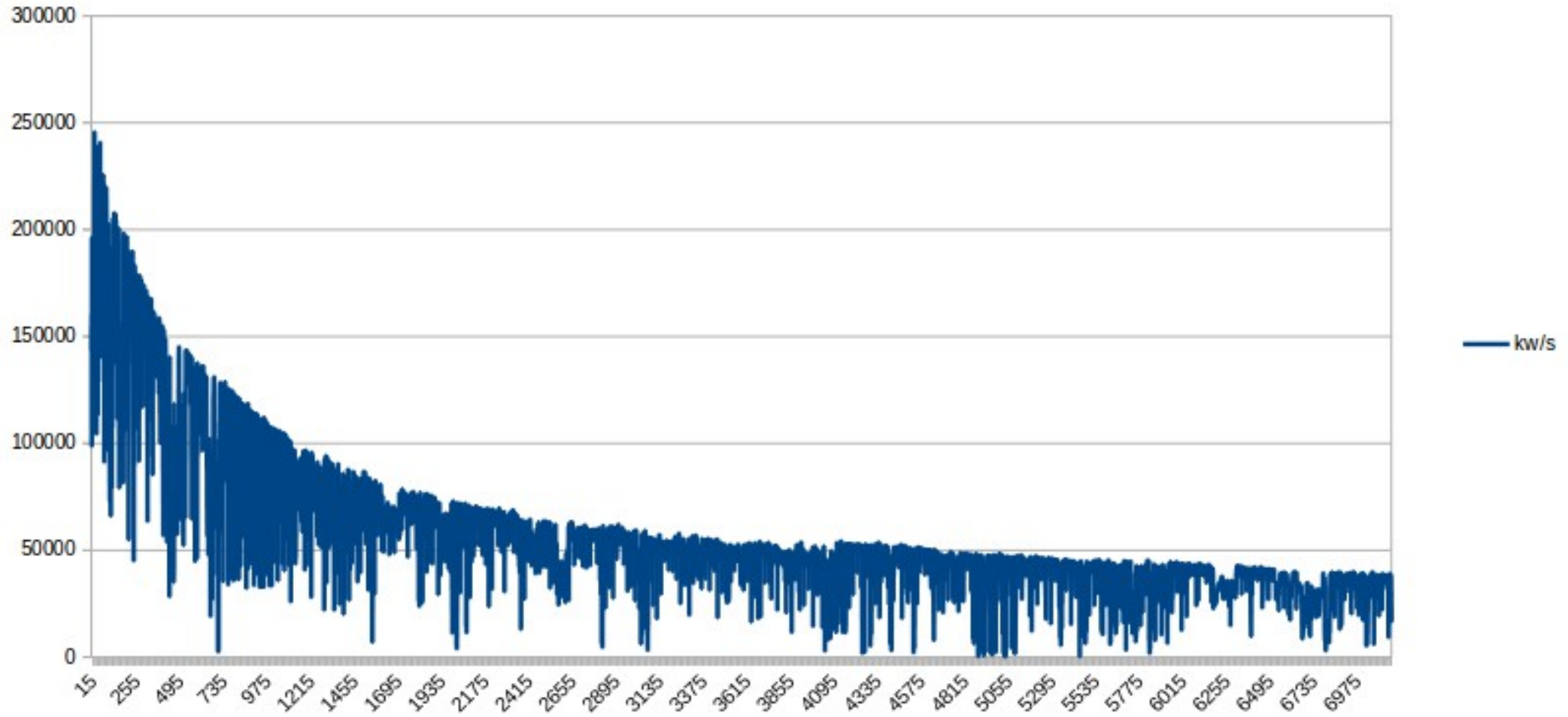
Tools using an event triggered approach

- examples are *truss(1), gdb(1)*
- these tools are not dynamic by nature
  - may heavily impact application timing as process gets stopped to collect data; some problems actually disappear because of it
- hard if not impossible to collect and evaluate data spanning multiple processes such as a “user login process chain”
- mostly user-land centric so no details from within the kernel available
  - noticeable exceptions are kernel-debuggers such as *mdb(1m)*

Showcase: intro

- set the ground: define your expectations for different problem sizes and runtimes
  - sustained IO rate in the 250 .. 500 MB/s range
- feedback loop: check your expectations
  - measured 61 MB/s only for 30 hour run
- determine were differences may come from
  - use DTrace or similar tool of your choice
  - USE method: check utilization, saturation and errors for all resources
- fix, adapt or change and repeat loop

Showcase: checking



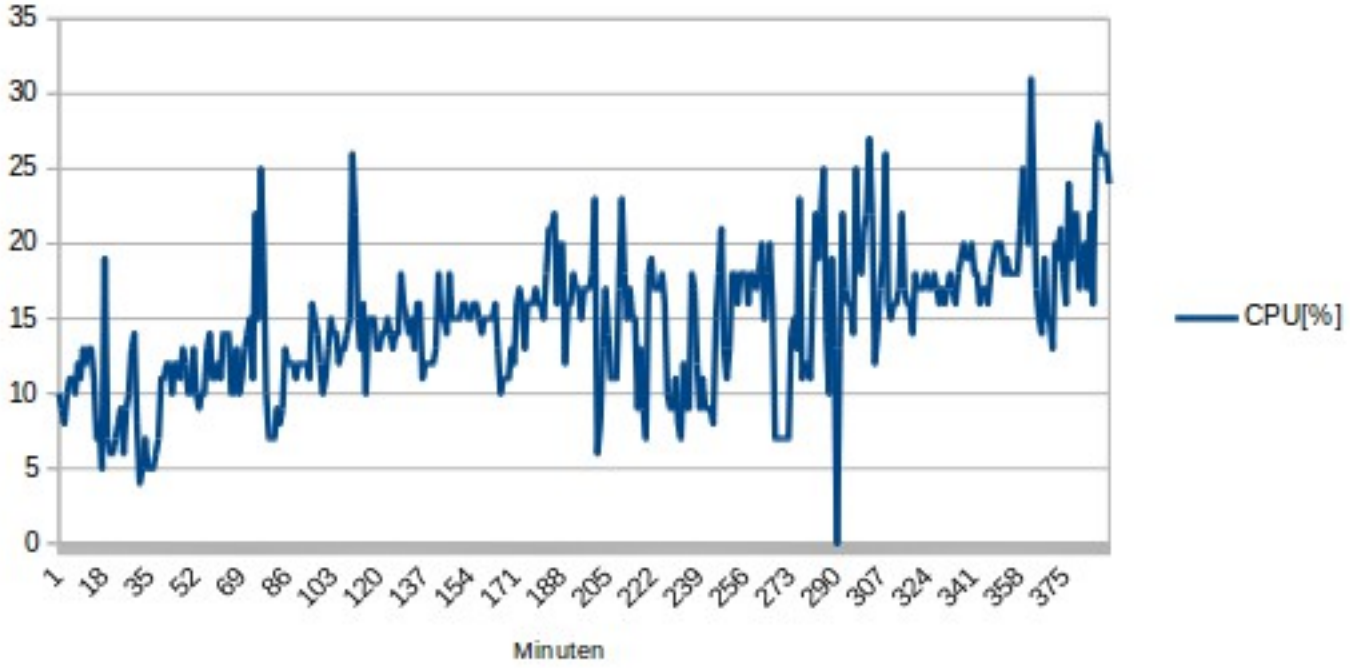
Showcase: looking for the root cause

- get the overall picture using *hotkernel* DTrace script

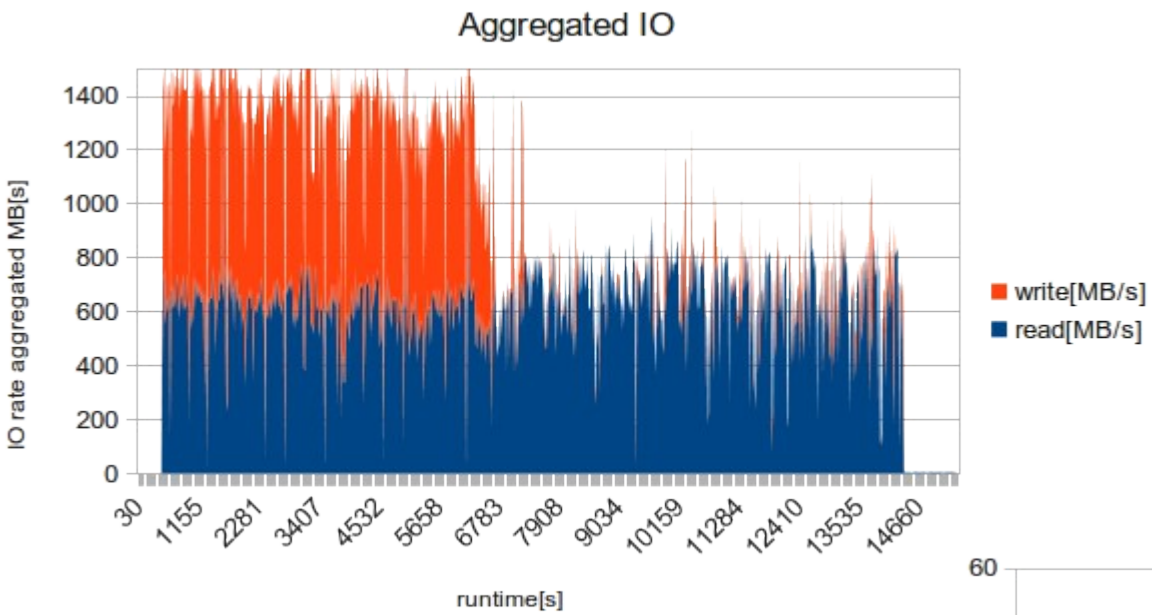
```
Routine                                  calls   prct.
SPARC-T4`copyin                           8630    0.2%
zfs`fletcher_4_native                    10902    0.3%
zfs`vdev_raidz_generate_parity_pq        19815    0.5%
zfs`lzjb_compress                       234676    5.9%
unix`cpu_halt                          3670636   91.8%
```

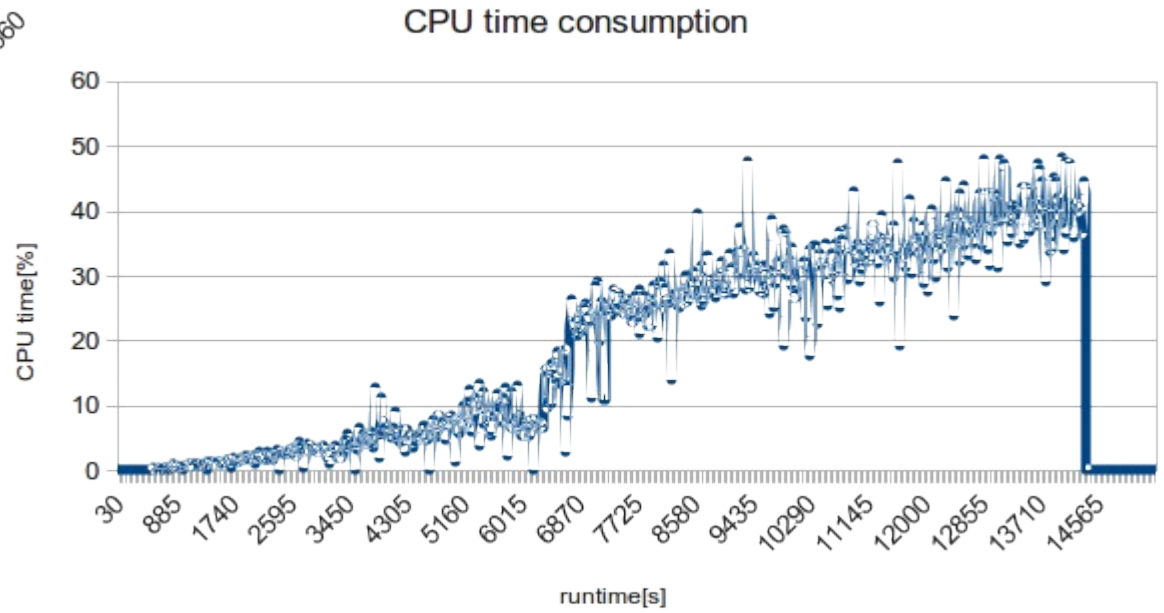
Showcase: looking for the root cause

- drilling down reveals the “bad guy” plus call stacks
  - routine already doubled CPU time consumption after just 6 hours



Showcase: routine fix applied





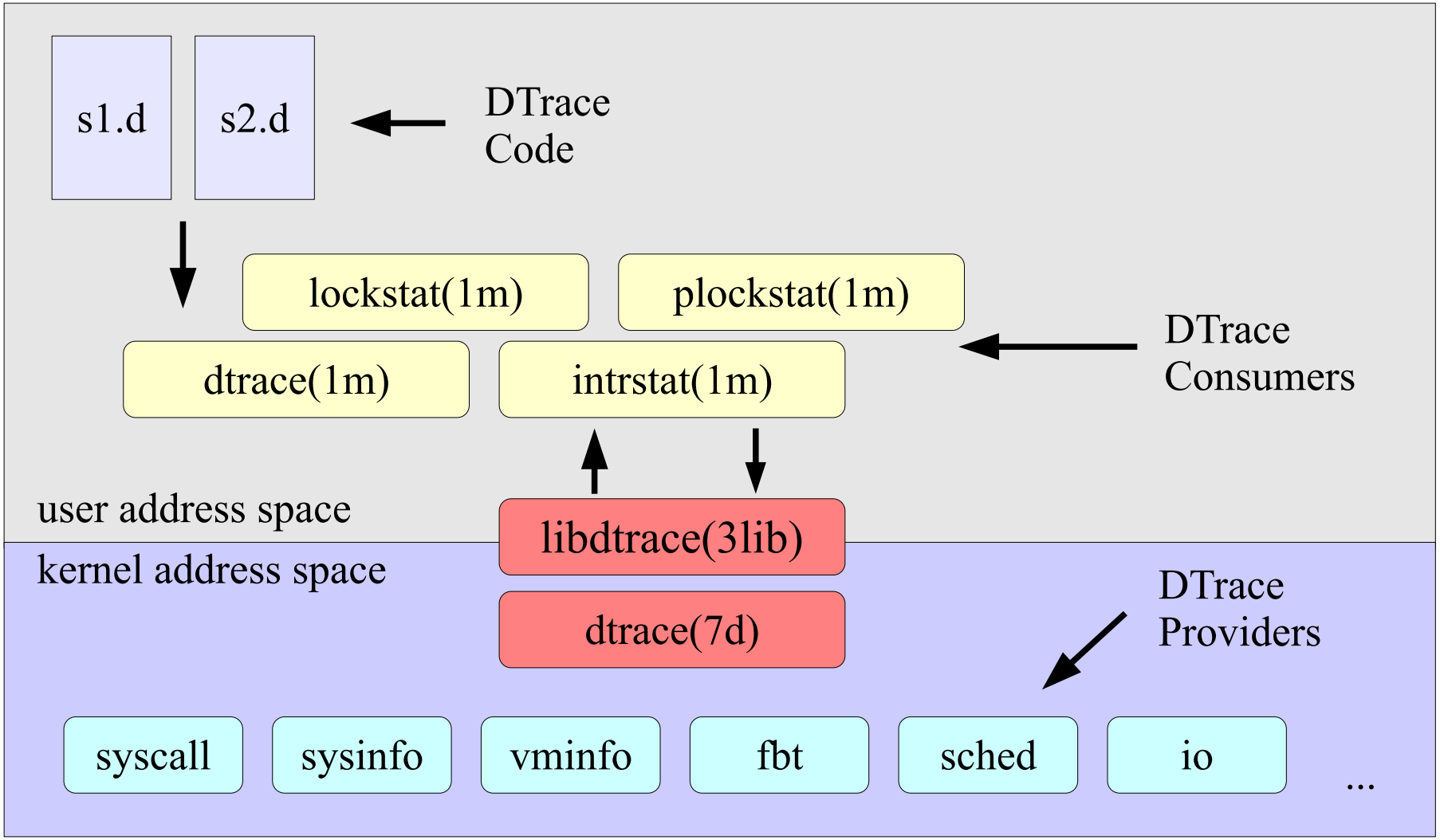
“A process cannot be understood by stopping it. Understanding must move with the flow of the process, must join it and flow with it.”

“First Law of Mentat“ taken from Frank Herberts “Dune“

DTrace, the “Dynamic Tracing Facility”

- dynamically instruments kernel, library and application code in an efficient manner
- about 100.000+ probes present in Solaris 11.3
  - limited availability in FreeBSD, MacOS and Linux
- each single probe can be enabled/disabled at any time
- runs in kernel context
  - collects and pre-processes data
- “D” scripting language

DTrace diagram



“D” program structure

- a “D” program lacks a main routine and acts more like a library/collection of routines called by triggering events
  - multiple ones might be in flight as triggers work per CPU
- basic structure, predicates are optional

```
probe description
/ predicate /
{ action statements }
```

- data types and structures, operators and functions very often look and feel like “C” without loop constructs
  - lots of predefined variables available
  - no need to declare a variable before assigning a value; assigning zero/null frees memory

“D” program structure example

```
obi-wan# cat ./reads.d
#!/usr/sbin/dtrace -s

#pragma D option quiet

syscall::read:entry
/ execname == "bacula-fd" || execname == "nscd" /
{
    printf("%-16s %10s %s\n",
        execname, probefunc, fds[arg0].fi_pathname
    );
}
```

```
obi-wan# ./reads.d
bacula-fd    read       /backup/mail/imap/l/user/PRIVACY/53065.
bacula-fd    read       /backup/mail/imap/l/user/PRIVACY/53065.
nscd         read       /etc/passwd
```

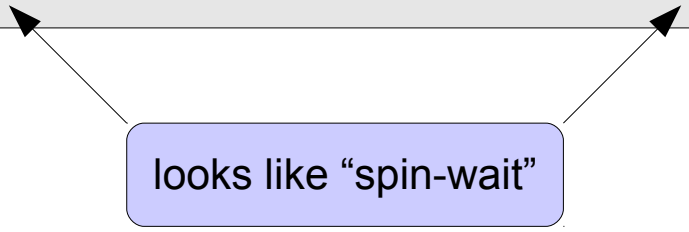
DTrace for developers

```
/* CPU time when calling a routine which matches
 * patterns passed on the command line
 */
pid$target:$1:$2:entry {
    self->ts = vtimestamp;
}

/* do the bookkeeping if we stored data on entry
 */
pid$target:$1:$2:return
/ self->ts /
{
    self->delta = vtimestamp -self->ts;
    @total[probemod, probefunc]        =
        sum(self->delta);
    @thread[tid, probemod, probefunc] =
        sum(self->delta);
    self->ts = 0;
}
```

DTrace for developers

```
obi-wan# OMP_NUM_THREADS=2 ./lib_timing.d -c './partest 100
10' libmtsk ''
...
Timing total, top-10 only
 time [us]                 module:function
      3837          libmtsk.so.1:thread_cancel_point
      3923          libmtsk.so.1:getfsr
      3959          libmtsk.so.1:push_context
      4431          libmtsk.so.1:barrier_reset_nthreads
      4532          libmtsk.so.1:getpsr
      4693          libmtsk.so.1:package_a_task
      5122          libmtsk.so.1:pop_context
      7433          libmtsk.so.1:_omp_affinity_mode
     35567          libmtsk.so.1:ready_to_work
     60326          libmtsk.so.1:sleep_at_barrier
```



DTrace for developers

```
obi-wan# OMP_NUM_THREADS=4 ./lib_timing.d -c ./transpose
transpose ''
...
Timing per thread, top-10 only
 time [us]   TID        module:function
         5     1    transpose:_start
       269     1    transpose:init_misaligned_data_trap_handler
    459892     1    transpose:_$d1A17.MAIN_
    462289     4    transpose:_$d1A17.MAIN_
    464534     3    transpose:_$d1A17.MAIN_
    466945     2    transpose:_$d1A17.MAIN_
   1515176     4    transpose:_$d1B27.MAIN_
   4265414     3    transpose:_$d1B27.MAIN_
   6830205     2    transpose:_$d1B27.MAIN_
   9938128     1    transpose:_$d1B27.MAIN_
...
```

"load distribution" for the 4 threads

44 : 30 : 19 : 7

Fortran-90 code courtesy Dieter an Mey, RWTH Aachen

DTrace for developers

```
obi-wan# OMP_NUM_THREADS=2 ./threads.d -c './partest 10 10'
TID=1              0 CPU   5(2) created
TID=1              0 CPU   5(2) restarted on same CPU
...
TID=1      849879447 sleeping on 'cond var'
TID=1      849879447 CPU   5(2) taken from CPU
TID=1      859891848 CPU   5(2) restarted on same CPU
TID=2              0 CPU   3(4) created
TID=2              0 CPU   3(4) restarted on same CPU
TID=2      859891848 sleeping on 'cond var'
TID=2      859891848 CPU   3(4) taken from CPU
TID=2      859891848 CPU   3(4) restarted on same CPU

TID=2     1029851518 sleeping on 'cond var'
TID=2     1029851518 CPU   3(4) taken from CPU
TID=2     1396340970 from-CPU 7(4) to-CPU 3(4) migration
TID=2     1396340166 CPU   3(4) restarted on same CPU
TID=2     1569516040 sleeping on 'cond var'
```

DTrace for developers

```
TID=2     4469899209 CPU   0(1) taken from CPU
TID=2     4479899053 CPU   0(1) restarted on same CPU
TID=2     4649894853 sleeping on 'cond var'

              libc.so.1`__lwp_park+0xb
              libc.so.1`cond_wait_queue+0x3b
              libc.so.1`_cond_wait+0x66
              libc.so.1`cond_wait+0x21
              libc.so.1`pthread_cond_wait+0x1b
              libmtsk.so.1`sleep_at_barrier+0xc1
              libmtsk.so.1`__mt_EndOfTask_Barrier_+0x87
              partest`_$d1B18.main
              libc.so.1`_thr_setup+0x4e
              libc.so.1`_lwp_start
```

Example: physical IO – latency

```
/* keep start time per device and also requested block
 * as more than one IO per device might be in flight
 */
io:::start {
    start[args[0]->b_edev, args[0]->b_blkno] =
        timestamp;
}

/* tick-provider fires on single CPU to trigger
 * output of data; makes use of first command line
 * argument to set rate
 */
tick-$1 {
    printa("%@d\n", @q);
    clear(@q);
}
```

Example: physical IO – latency

```
/* only handle IOs for which we have a start time */
io:::done
/start[args[0]->b_edev, args[0]->b_blkno]/
{
        /* use efficient "this" variables to set
         * read/write flag and convert nano- into
         * microseconds
         */
    this->rwf = args[0]->b_flags & B_WRITE ? "W" :
                (args[0]->b_flags & B_READ  ? "R" : "*");
    this->us  = (timestamp - start[args[0]->b_edev,
                 args[0]->b_blkno]);
    this->us /= 1000;
    @q[this->rwf] = quantize(this->us);

        /* clear start time flag */
    start[args[0]->b_edev, args[0]->b_blkno] = 0;
}
```

Example: physical IO – latency

```
jedi# ./demo_iolat.d 60s
...
R
           value  ------------- Distribution ------------- count
              32 |                                         0
              64 |                                         1
             128 |@@@@@@@@@@@@@@@@@@@                      759
             256 |@@@@@                                    242
             512 |                                         2
            1024 |                                         0
            2048 |                                         2
            4096 |@                                        33
            8192 |@@@@@@@@                                 340
           16384 |@@@@@@                                   272
           32768 |@@                                       73
           65536 |@                                        25
          131072 |                                         8
          262144 |                                         5
          524288 |                                         4
         1048576 |                                         0
```

„Modern times“ performance issues

- DTrace has access to CPU performance counters e.g through PAPI (Performance Application Programming Interface)

```
T4-System # dtrace -n 'cpc:::DTLB_fill_trap-all-10000 \
                        { @[execname] = count(); }'
dtrace: description ' cpc:::DTLB_fill_trap-all-10000 ' matched
1 probe
^C
  sshd                                                            2
  ldmd                                                            6
  gzip                                                           27
  zpool-ssd                                                      35
  postgres                                                       36
  sched                                                          70
  tar                                                           106
```

Thank you!