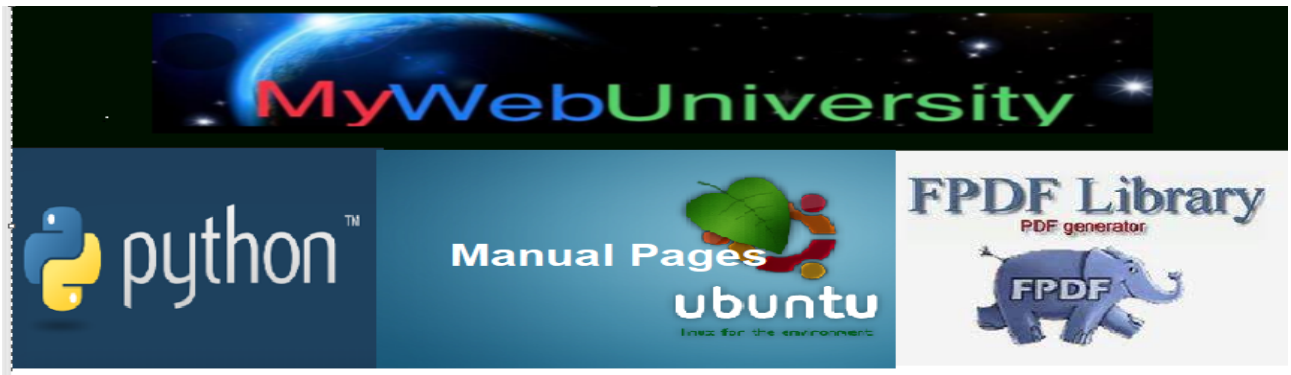

Full credit is given to the above companies including the OS that this PDF file was generated!

Linux Ubuntu 26.04 Manual Pages on command 'perf-script-python.1'

$ man perf-script-python.1

PERF-SCRIPT-PYTHON(1) perf Manual PERF-SCRIPT-PYTHON(1)

NAME

perf-script-python - Process trace data with a Python script

SYNOPSIS

perf script [-s [Python]:script[.py]]

DESCRIPTION

This perf script option is used to process perf script data using perf?s built-in Python interpreter. It reads and processes the input file and displays the results of the trace analysis implemented in the given Python script, if any.

A QUICK EXAMPLE

This section shows the process, start to finish, of creating a working Python script that aggregates and extracts useful information from a raw perf script stream. You can avoid reading the rest of this document if an example is enough for you; the rest of the document provides more details on each step and lists the library functions available to script writers.

This example actually details the steps that were used to create the syscall-counts script you see when you list the available perf script scripts via perf script -l. As such, this script also shows how to integrate your script into the list of general-purpose perf script scripts listed by that command.

The syscall-counts script is a simple script, but demonstrates all the basic ideas necessary to create a useful script. Here?s an example of its output (syscall names are not yet supported, they will appear as numbers):

.ft C

```
syscall events:

event                                count
---------------------------------------  -----------
sys_write                            455067
sys_getdents                           4072
sys_close                            3037
sys_swapoff                            1769
sys_read                             923
sys_sched_setparam                         826
sys_open                             331
sys_newfstat                           326
sys_mmap                             217
sys_munmap                             216
sys_futex                            141
sys_select                           102
sys_poll                             84
sys_setitimer                          12
sys_writev                             8
15                                   8
sys_lseek                              7
sys_rt_sigprocmask                         6
sys_wait4                              3
sys_ioctl                            3
sys_set_robust_list                        1
sys_exit                             1
56                                   1
sys_access                             1
.ft
```

Basically our task is to keep a per-syscall tally that gets updated every time a system call occurs in the system. Our script will do that, but first we need to record the data that will be processed by that script. Theoretically, there are a couple of ways we could do that:

? we could enable every event under the tracing/events/syscalls directory, but this is

over 600 syscalls, well beyond the number allowable by perf. These individual syscall events will however be useful if we want to later use the guidance we get from the general-purpose scripts to drill down and get more detail about individual syscalls of interest.

? we can enable the sys_enter and/or sys_exit syscalls found under tracing/events/raw_syscalls. These are called for all syscalls; the id field can be used to distinguish between individual syscall numbers.

For this script, we only need to know that a syscall was entered; we don?t care how it exited, so we?ll use perf record to record only the sys_enter events:

.ft C

perf record -a -e raw_syscalls:sys_enter

^C[perf record: Woken up 1 times to write data]

[perf record: Captured and wrote 56.545 MB perf.data (~2470503 samples)]

.ft

The options basically say to collect data for every syscall event system-wide and multiplex the per-cpu output into a single stream. That single stream will be recorded in a file in the current directory called perf.data.

Once we have a perf.data file containing our data, we can use the -g perf script option to generate a Python script that will contain a callback handler for each event type found in the perf.data trace stream (for more details, see the STARTER SCRIPTS section).

.ft C

perf script -g python

generated Python script: perf-script.py

The output file created also in the current directory is named perf-script.py. Here's the file in its entirety:

perf script event handlers, generated by perf script -g python

Licensed under the terms of the GNU GPL License version 2

The common_* event handler fields are the most useful fields common to

all events. They don't necessarily correspond to the 'common_*' fields

in the format files. Those fields not available as handler params can

be retrieved using Python functions of the form common_*(context).

See the perf-script-python Documentation for the list of available functions.

import os

```
import sys
sys.path.append(os.environ['PERF_EXEC_PATH'] + \
        '/scripts/python/Perf-Trace-Util/lib/Perf/Trace')
from perf_trace_context import *
from Core import *
def trace_begin():
        print "in trace_begin"
def trace_end():
        print "in trace_end"
def raw_syscalls__sys_enter(event_name, context, common_cpu,
        common_secs, common_nsecs, common_pid, common_comm,
        id, args):
                print_header(event_name, common_cpu, common_secs, common_nsecs,
                        common_pid, common_comm)
                print "id=%d, args=%s\n" % \
                (id, args),
def trace_unhandled(event_name, context, event_fields_dict):
                print ' '.join(['%s=%s'%(k,str(v))for k,v in sorted(event_fields_dict.items())])
def print_header(event_name, cpu, secs, nsecs, pid, comm):
        print "%-20s %5u %05u.%09u %8u %-20s " % \
        (event_name, cpu, secs, nsecs, pid, comm),
.ft
```

At the top is a comment block followed by some import statements and a path append which every perf script script should include.

Following that are a couple generated functions, trace_begin() and trace_end(), which are called at the beginning and the end of the script respectively (for more details, see the SCRIPT_LAYOUT section below).

Following those are the event handler functions generated one for every event in the perf record output. The handler functions take the form subsystem__event_name, and contain named parameters, one for each field in the event; in this case, there?s only one event, raw_syscalls__sys_enter(). (see the EVENT HANDLERS section below for more info on event handlers).

The final couple of functions are, like the begin and end functions, generated for every

script. The first, trace_unhandled(), is called every time the script finds an event in the perf.data file that doesn?t correspond to any event handler in the script. This could mean either that the record step recorded event types that it wasn?t really interested in, or the script was run against a trace file that doesn?t correspond to the script.

The script generated by -g option simply prints a line for each event found in the trace stream i.e. it basically just dumps the event and its parameter values to stdout. The print_header() function is simply a utility function used for that purpose. Let?s rename the script and run it to see the default output:

```
.ft C
# mv perf-script.py syscall-counts.py
# perf script -s syscall-counts.py
raw_syscalls__sys_enter    1 00840.847582083    7506 perf              id=1, args=
raw_syscalls__sys_enter    1 00840.847595764    7506 perf              id=1, args=
raw_syscalls__sys_enter    1 00840.847620860    7506 perf              id=1, args=
raw_syscalls__sys_enter    1 00840.847710478    6533 npviewer.bin      id=78, args=
raw_syscalls__sys_enter    1 00840.847719204    6533 npviewer.bin      id=142, args=
raw_syscalls__sys_enter    1 00840.847755445    6533 npviewer.bin      id=3, args=
raw_syscalls__sys_enter    1 00840.847775601    6533 npviewer.bin      id=3, args=
raw_syscalls__sys_enter    1 00840.847781820    6533 npviewer.bin      id=3, args=
.
.
.
.ft
```

Of course, for this script, we?re not interested in printing every trace event, but rather aggregating it in a useful way. So we?ll get rid of everything to do with printing as well as the trace_begin() and trace_unhandled() functions, which we won?t be using. That leaves us with this minimalistic skeleton:

```
.ft C
import os
import sys
sys.path.append(os.environ['PERF_EXEC_PATH'] + \
        '/scripts/python/Perf-Trace-Util/lib/Perf/Trace')
from perf_trace_context import *
```

```
from Core import *
def trace_end():
        print "in trace_end"
def raw_syscalls__sys_enter(event_name, context, common_cpu,
        common_secs, common_nsecs, common_pid, common_comm,
        id, args):
.ft
```

In trace_end(), we?ll simply print the results, but first we need to generate some results to print. To do that we need to have our sys_enter() handler do the necessary tallying until all events have been counted. A hash table indexed by syscall id is a good way to store that information; every time the sys_enter() handler is called, we simply increment a count associated with that hash entry indexed by that syscall id:

```
.ft C
  syscalls = autodict()
  try:
    syscalls[id] += 1
  except TypeError:
    syscalls[id] = 1
.ft
```

The syscalls autodict object is a special kind of Python dictionary (implemented in Core.py) that implements Perl?s autovivifying hashes in Python i.e. with autovivifying hashes, you can assign nested hash values without having to go to the trouble of creating intermediate levels if they don?t exist e.g syscalls[comm][pid][id] = 1 will create the intermediate hash levels and finally assign the value 1 to the hash entry for id (because the value being assigned isn?t a hash object itself, the initial value is assigned in the TypeError exception. Well, there may be a better way to do this in Python but that?s what works for now).

Putting that code into the raw_syscalls__sys_enter() handler, we effectively end up with a single-level dictionary keyed on syscall id and having the counts we?ve tallied as values. The print_syscall_totals() function iterates over the entries in the dictionary and displays a line for each entry containing the syscall name (the dictionary keys contain the syscall ids, which are passed to the Util function syscall_name(), which translates the raw syscall numbers to the corresponding syscall name strings). The output is displayed after all the

events in the trace have been processed, by calling the print_syscall_totals() function from the trace_end() handler called at the end of script processing.

The final script producing the output shown above is shown in its entirety below (syscall_name() helper is not yet available, you can only deal with id?s for now):

```
    .ft C
    import os
    import sys
    sys.path.append(os.environ['PERF_EXEC_PATH'] + \
            '/scripts/python/Perf-Trace-Util/lib/Perf/Trace')
    from perf_trace_context import *
    from Core import *
    from Util import *
    syscalls = autodict()
    def trace_end():
            print_syscall_totals()
    def raw_syscalls__sys_enter(event_name, context, common_cpu,
            common_secs, common_nsecs, common_pid, common_comm,
            id, args):
            try:
                    syscalls[id] += 1
            except TypeError:
                    syscalls[id] = 1
    def print_syscall_totals():
        if for_comm is not None:
                print "\nsyscall events for %s:\n\n" % (for_comm),
        else:
                print "\nsyscall events:\n\n",
        print "%-40s  %10s\n" % ("event", "count"),
        print "%-40s  %10s\n" % ("----------------------------------------", \
                                 "-----------"),
        for id, val in sorted(syscalls.iteritems(), key = lambda(k, v): (v, k), \
                                  reverse = True):
                print "%-40s  %10d\n" % (syscall_name(id), val),
```

.ft

The script can be run just as before:

perf script -s syscall-counts.py

So those are the essential steps in writing and running a script. The process can be generalized to any tracepoint or set of tracepoints you?re interested in - basically find the tracepoint(s) you?re interested in by looking at the list of available events shown by perf list and/or look in /sys/kernel/tracing/events/ for detailed event and field info, record the corresponding trace data using perf record, passing it the list of interesting events, generate a skeleton script using perf script -g python and modify the code to aggregate and display it for your particular needs.

After you?ve done that you may end up with a general-purpose script that you want to keep around and have available for future use. By writing a couple of very simple shell scripts and putting them in the right place, you can have your script listed alongside the other scripts listed by the perf script -l command e.g.:

.ft C

perf script -l

List of available trace scripts:

wakeup-latency system-wide min/max/avg wakeup latency

rw-by-file <comm> r/w activity for a program, by file

rw-by-pid system-wide r/w activity

.ft

A nice side effect of doing this is that you also then capture the probably lengthy perf record command needed to record the events for the script.

To have the script appear as a built-in script, you write two simple scripts, one for recording and one for reporting.

The record script is a shell script with the same base name as your script, but with -record appended. The shell script should be put into the perf/scripts/python/bin directory in the kernel source tree. In that script, you write the perf record command-line needed for your script:

.ft C

cat kernel-source/tools/perf/scripts/python/bin/syscall-counts-record

#!/bin/bash

perf record -a -e raw_syscalls:sys_enter

.ft

The report script is also a shell script with the same base name as your script, but with -report appended. It should also be located in the perf/scripts/python/bin directory. In that script, you write the perf script -s command-line needed for running your script:

.ft C

cat kernel-source/tools/perf/scripts/python/bin/syscall-counts-report

#!/bin/bash

description: system-wide syscall counts

perf script -s ~/libexec/perf-core/scripts/python/syscall-counts.py

.ft

Note that the location of the Python script given in the shell script is in the libexec/perf-core/scripts/python directory - this is where the script will be copied by make install when you install perf. For the installation to install your script there, your script needs to be located in the perf/scripts/python directory in the kernel source tree:

.ft C

ls -al kernel-source/tools/perf/scripts/python

total 32

drwxr-xr-x 4 trz trz 4096 2010-01-26 22:30 .

drwxr-xr-x 4 trz trz 4096 2010-01-26 22:29 ..

drwxr-xr-x 2 trz trz 4096 2010-01-26 22:29 bin

-rw-r--r-- 1 trz trz 2548 2010-01-26 22:29 check-perf-script.py

drwxr-xr-x 3 trz trz 4096 2010-01-26 22:49 Perf-Trace-Util

-rw-r--r-- 1 trz trz 1462 2010-01-26 22:30 syscall-counts.py

.ft

Once you?ve done that (don?t forget to do a new make install, otherwise your script won?t show up at run-time), perf script -l should show a new entry for your script:

.ft C

perf script -l

List of available trace scripts:

wakeup-latency system-wide min/max/avg wakeup latency

rw-by-file <comm> r/w activity for a program, by file

rw-by-pid system-wide r/w activity

syscall-counts system-wide syscall counts

.ft

You can now perform the record step via perf script record:

perf script record syscall-counts

and display the output using perf script report:

perf script report syscall-counts

STARTER SCRIPTS

You can quickly get started writing a script for a particular set of trace data by generating a skeleton script using perf script -g python in the same directory as an existing perf.data trace file. That will generate a starter script containing a handler for each of the event types in the trace file; it simply prints every available field for each event in the trace file.

You can also look at the existing scripts in ~/libexec/perf-core/scripts/python for typical examples showing how to do basic things like aggregate event data, print results, etc. Also, the check-perf-script.py script, while not interesting for its results, attempts to exercise all of the main scripting features.

EVENT HANDLERS

When perf script is invoked using a trace script, a user-defined handler function is called for each event in the trace. If there?s no handler function defined for a given event type, the event is ignored (or passed to a trace_unhandled function, see below) and the next event is processed.

Most of the event?s field values are passed as arguments to the handler function; some of the less common ones aren?t - those are available as calls back into the perf executable (see below).

As an example, the following perf record command can be used to record all sched_wakeup events in the system:

perf record -a -e sched:sched_wakeup

Traces meant to be processed using a script should be recorded with the above option: -a to enable system-wide collection.

The format file for the sched_wakeup event defines the following fields (see /sys/kernel/tracing/events/sched/sched_wakeup/format):

.ft C

format:

field:unsigned short common_type;

```
        field:unsigned char common_flags;
        field:unsigned char common_preempt_count;
        field:int common_pid;
        field:char comm[TASK_COMM_LEN];
        field:pid_t pid;
        field:int prio;
        field:int success;
        field:int target_cpu;
```

.ft

The handler function for this event would be defined as:

```
.ft C
def sched__sched_wakeup(event_name, context, common_cpu, common_secs,
        common_nsecs, common_pid, common_comm,
        comm, pid, prio, success, target_cpu):
        pass
.ft
```

The handler function takes the form subsystem__event_name.

The common_* arguments in the handler?s argument list are the set of arguments passed to all event handlers; some of the fields correspond to the common_* fields in the format file, but some are synthesized, and some of the common_* fields aren?t common enough to to be passed to every event as arguments but are available as library functions.

Here?s a brief description of each of the invariant event args:

```
    event_name              the name of the event as text
    context                 an opaque 'cookie' used in calls back into perf
    common_cpu              the cpu the event occurred on
    common_secs             the secs portion of the event timestamp
    common_nsecs            the nsecs portion of the event timestamp
    common_pid              the pid of the current task
    common_comm             the name of the current process
```

All of the remaining fields in the event?s format file have counterparts as handler function arguments of the same name, as can be seen in the example above.

The above provides the basics needed to directly access every field of every event in a trace, which covers 90% of what you need to know to write a useful trace script. The

sections below cover the rest.

SCRIPT LAYOUT

Every perf script Python script should start by setting up a Python module search path and 'import?ing a few support modules (see module descriptions below):

```
.ft C
import os
import sys
sys.path.append(os.environ['PERF_EXEC_PATH'] + \
        '/scripts/python/Perf-Trace-Util/lib/Perf/Trace')
from perf_trace_context import *
from Core import *
.ft
```

The rest of the script can contain handler functions and support functions in any order.

Aside from the event handler functions discussed above, every script can implement a set of optional functions:

trace_begin, if defined, is called before any event is processed and gives scripts a chance to do setup tasks:

```
.ft C
def trace_begin():
    pass
.ft
```

trace_end, if defined, is called after all events have been processed and gives scripts a chance to do end-of-script tasks, such as display results:

```
.ft C
def trace_end():
    pass
.ft
```

trace_unhandled, if defined, is called after for any event that doesn?t have a handler explicitly defined for it. The standard set of common arguments are passed into it:

```
.ft C
def trace_unhandled(event_name, context, event_fields_dict):
    pass
.ft
```

process_event, if defined, is called for any non-tracepoint event

```
.ft C
def process_event(param_dict):
    pass
.ft
```

context_switch, if defined, is called for any context switch

```
.ft C
def context_switch(ts, cpu, pid, tid, np_pid, np_tid, machine_pid, out, out_preempt, *x):
    pass
.ft
```

auxtrace_error, if defined, is called for any AUX area tracing error

```
.ft C
def auxtrace_error(typ, code, cpu, pid, tid, ip, ts, msg, cpumode, *x):
    pass
.ft
```

The remaining sections provide descriptions of each of the available built-in perf script Python modules and their associated functions.

AVAILABLE MODULES AND FUNCTIONS

The following sections describe the functions and variables available via the various perf script Python modules. To use the functions and variables from the given module, add the corresponding from XXXX import line to your perf script script.

Core.py Module

These functions provide some essential functions to user scripts.

The flag_str and symbol_str functions provide human-readable strings for flag and symbolic fields. These correspond to the strings and values parsed from the print fmt fields of the event format files:

flag_str(event_name, field_name, field_value) - returns the string representation corresponding to field_value for the flag field field_name of event event_name

symbol_str(event_name, field_name, field_value) - returns the string representation corresponding to field_value for the symbolic field field_name of event event_name

The autodict function returns a special kind of Python dictionary that implements Perl?s autovivifying hashes in Python i.e. with autovivifying hashes, you can assign nested hash values without having to go to the trouble of creating intermediate levels if they don?t

exist.

autodict() - returns an autovivifying dictionary instance

perf_trace_context Module

Some of the common fields in the event format file aren?t all that common, but need to be made accessible to user scripts nonetheless.

perf_trace_context defines a set of functions that can be used to access this data in the context of the current event. Each of these functions expects a context variable, which is the same as the context variable passed into every tracepoint event handler as the second argument. For non-tracepoint events, the context variable is also present as perf_trace_context.perf_script_context .

common_pc(context) - returns common_preempt count for the current event

common_flags(context) - returns common_flags for the current event

common_lock_depth(context) - returns common_lock_depth for the current event

perf_sample_insn(context) - returns the machine code instruction

perf_set_itrace_options(context, itrace_options) - set --itrace options if they have not been set already

perf_sample_srcline(context) - returns source_file_name, line_number

perf_sample_srccode(context) - returns source_file_name, line_number, source_line

perf_config_get(config_name) - returns the value of the named config item, or None if unset

Util.py Module

Various utility functions for use with perf script:

nsecs(secs, nsecs) - returns total nsecs given secs/nsecs pair

nsecs_secs(nsecs) - returns whole secs portion given nsecs

nsecs_nsecs(nsecs) - returns nsecs remainder given nsecs

nsecs_str(nsecs) - returns printable string in the form secs.nsecs

avg(total, n) - returns average given a sum and a total number of values

SUPPORTED FIELDS

Currently supported fields:

ev_name, comm, id, stream_id, pid, tid, cpu, ip, time, period, phys_addr, addr, symbol, symoff, dso, time_enabled, time_running, values, callchain, brstack, brstacksym, datasrc, datasrc_decode, iregs, uregs, weight, transaction, raw_buf, attr, cpumode.

Fields that may also be present:

flags - sample flags

flags_disp - sample flags display

insn_cnt - instruction count for determining instructions-per-cycle (IPC)

cyc_cnt - cycle count for determining IPC

addr_correlates_sym - addr can correlate to a symbol

addr_dso - addr dso

addr_symbol - addr symbol

addr_symoff - addr symbol offset

Some fields have sub items:

brstack: from, to, from_dsoname, to_dsoname, mispred, predicted, in_tx, abort, cycles.

brstacksym: items: from, to, pred, in_tx, abort (converted string)

For example, We can use this code to print brstack "from", "to", "cycles".

if brstack in dict: for entry in dict[brstack]: print "from %s, to %s, cycles %s" %

(entry["from"], entry["to"], entry["cycles"])

SEE ALSO

perf-script(1)